A Real-time Computational Substrate for Embedded Intelligent Systems

by

Dale Edward Parson

A Dissertation Presented to the Graduate Committee

of Lehigh University in Candidacy for the Degree of

Doctor of Philosophy in Computer Science

Lehigh University

1990

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

_____

Dr. Glenn Blank Date
Committee Chairperson and Advisor
Department of Computer Science and Electrical Engineering

_____

Dr. Robert Barnes Date
Committee Member
Department of Computer Science and Electrical Engineering

_____

Dr. Susan Barrett Date
Committee Member
Department of Psychology

_____

Professor Samuel Gulden Date
Committee Member
Department of Computer Science and Electrical Engineering

ACKNOWLEDGMENTS

Let me begin by thanking my wife, Linda, and our daughter and son, Sierra and Jeremy, for their patience in letting me work on this research at times when I might have been interacting with them. I believe that setting an example is one of the most powerful methods of instruction that an adult can use with children, and I feel certain that the diligence with which I have attacked this doctoral work has not been lost on our kids.

My thanks to Dr. Glenn Blank, advisor, for encouraging this work and for providing sound suggestions and criticisms. Dr. Blank contributed important help in getting the results of this work into the world of AI research publication. I thank the committee members, Professors Robert Barnes, Susan Barrett, and Samuel Gulden, for directing me to useful information sources and giving other good advice. I also want to thank two unofficial committee members. Dr. Herbert Rubenstein, now retired from Lehigh, convincingly urged me to go on for my Ph.D. when I took his Human Information Processing course in 1984 as part of my master's program. He gave me early advice when I started back to Lehigh. And Dr. Leon Levy of AT&T Bell Laboratories gave me excellent editorial feedback and valuable assistance in pursuing publication.

Finally, I thank AT&T, whose Tuition Assistance Program paid my way through half of my undergraduate program and all of my graduate work. Besides

providing monetary support, the company has given me time to work during critical periods in my doctoral program.

CONTENTS

FIGURES

EXAMPLE LISTINGS

A Real-time Computational Substrate for Embedded Intelligent 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 thesis presents requirements, psychological

inspiration, architectural basis, implementation, and application of PRIOPS. A major contribution is the design of time-constrained, priority-driven match scheduling for the augmented Rete algorithm.

1. INTRODUCTION

1.1 Introduction to Reactive, Embedded Intelligence

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.

Traditional artificial intelligence processing architectures are predominately complex organizations designed to address the needs of multifaceted cognitive activities. Many AI researchers have considered questions of computational complexity and run-time responsiveness to be secondary issues in the higher-order quest for powerful, general-purpose mechanisms. Using sophisticated architectures such as augmented transition nets for language recognition [96] and predicate calculus for planning [97],* researchers have tackled tough problems in symbol manipulation, but have largely ignored the basic problems of an intelligent physical entity interacting with an often dangerous environment. During interactions with the world, an intelligent physical system must not only decide what do to about a condition in which it is situated, but then must act in time. Many of the problems not only do not require elaborate architectures, they cannot be solved using them. The architectures are too sluggish.

Levesque and Brachman have presented the tradeoff between expressiveness and tractability as fundamental [57]. Pseudo-solutions to unbounded computational problems, such as acquiring faster computing systems or prematurely terminating search, do not address the problem of intractability. Moreover, in many artificial

---

* Even the programming substrate upon which many of these systems are built - LISP - is noted for its generality, power, and execution-time inefficiency.

intelligence applications the expressiveness traded for tractability may be unnecessary expressiveness. As a result of performance problems, there is an increasing effort to improve tractability in AI systems. Recent work on natural language processing [11] and planning [47,56] has focused on reducing computational complexity and improving performance predictability while providing adequate functionality. Research into knowledge representation has sought to transform complex descriptions into simple atomic predicates (a relational database) for fast query performance [23]. Environmental conditions trigger lookup of simple reactive functions in the Pengi system [2,19]. In many cases human performance of time-constrained tasks provides hints and suggestions for realistic computing architectures. Potential applications of real-time inference mechanisms to embedded computing problems abound [48].

This thesis presents a two-tiered architecture for performing constant-time-constrained inference in embedded systems: the Prioritized Production System *(PRIOPS)* [72,73,74]. The architecture is inspired by an existing model of human activity, that of *controlled* and *automatic* human information processing. Controlled processing is complex, flexible, and usually time-consuming behavior analogous to the types of machine processing normally proposed by artificial intelligence architectures. In contrast, the notion of a *habit* approximates the concept of automatic processing; it trades flexibility and generality for efficiency and predictability in dealing with expected environmental situations. The PRIOPS

system uses its *automatic partition* of productions to handle time-restricted, stereotyped interactions with an external environment, reserving use of costly declarative memory and search-based strategies for the more versatile (and slower) *controlled partition.* I propose that PRIOPS is a viable software architecture for implementing reactive knowledge-based systems. It is not primarily a tool for cognitive modelling, but a tool for writing software systems.*

The next section defines some terminology for use in this thesis. Chapter 2 examines the underlying psychological theory of human controlled-automatic processing. Chapter 3 introduces the two-tiered PRIOPS computing architecture. Chapter 4 expands this introduction, presenting PRIOPS notation and augmented Rete pattern matching by way of a simple example. Chapter 5 continues in this vein, using code excerpts and execution statistics from a 940 line PRIOPS demonstration program to show application of PRIOPS constructs.

Chapter 6 is the most technical of the thesis, examining PRIOPS matching algorithms in detail. The chapter is important for rigor in specifying exactly how PRIOPS' implementation accomplishes its goals. There is enough information in Chapter 6 to guide writing of PRIOPS Rete matching operations. Non-programming readers may wish to skip all but the introductory section of this

---

* I will discuss several existing software approaches to cognitive modelling of controlled and automatic processing.

chapter.

Chapter 7 summarizes the work, discusses related research, and points the way to future work with PRIOPS. Following Chapter 7 is an annotated bibliography. Last come two appendices that document the current PRIOPS program.

1.2 Definitions for Real-time, Embedded Computing

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 to 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.

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 I 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.

Some definitions of *real-time* responsiveness are intuitive, while others are formal. Wirth provides this definition [98, p. 577]:

> "If we depart from this rule (execution time independence) and let our programs' validity depend on the execution speed of the utilized processors, we enter the field commonly called 'real-time' programming."

Application requirements determine the dependence on execution speed and speed predictability. PRIOPS is designed to achieve constant-time-bound reactivity to predictable environmental events and event combinations. Consequently, 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.

2. CONTROLLED AND AUTOMATIC HUMAN INFORMATION PROCESSING

The lack of an artificial intelligence programming approach for designing time-constrained, reactive systems led me to look at the fundamental requirements of these systems. I concentrated on studies of the most successful embedded intelligent systems presently in operation: people. Humans operate in a wide variety of environments. They can learn to respond to significant, and particularly to dangerous environmental phenomena without engaging in excessive, time-consuming mentation. Practice of consistent activities enhances performance. Finally, humans can usually perform high-level, symbolic processing simultaneously with habitual procedures. Literature search led me to the theory of *controlled* and *automatic* human information processing.

Schneider and Shiffrin first proposed their theory of *controlled* and *automatic* human information processing in 1977 [81,89]. 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, cognitive, and motor 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 [84]. An overview and general definition

of automatism is found in Shiffrin and Dumais [90]. Schneider and Shiffrin originally enumerated the following characteristics for controlled processing [89, p. 159-160]:

> 1) Control processes are limited-capacity processes requiring attention.
>
> 2) The limitations of short-term memory cause the limitations of controlled processing.
>
> 3) Humans adopt control processes quickly, without extensive training, and modify them fairly easily.
>
> 4) Control processes show a rapid development of asymptotic performance.
>
> 5) Control processes direct the flow of information between short-term and long-term memory.

These characteristics are in direct contrast to those originally enumerated for automatic processing [89, p. 160-161]:

> 1) The capacity limitations of short-term memory do not hinder automatic processes. These processes do not require attention.
>
> 2) A person may initiate some automatic processes, but once initiated all automatic processes run to completion.
>
> 3) Their speed and automaticity will usually keep their constituent elements hidden from conscious perception.
>
> 4) These processes require considerable training to develop and are most difficult to modify, once learned.
>
> 5) They do not directly cause learning in long-term memory, although they can indirectly affect learning through forced allocation of controlled processing.

Automaticity fulfills the requirements for reliable reactive processing among humans. It is predictable, efficient, takes priority over controlled processing, and does not consume short-term memory. These characteristics prove to be important features of time and memory restricted, reactive computer processing as well.

Features 1 through 3 of automatic processing are reminiscent of interrupt handlers in conventional computing systems. Interrupt handlers do not normally require the attention of the processes they interrupt. Interrupt handling can be largely transparent to higher level processing that is occurring. Alternatively, because interrupt processing priorities are typically greater than other processing priorities in a system, an interrupt handler has the capability of diverting or preempting the higher level processing when the interrupting event calls for such actions.

Features 1 and 2 of controlled processing, along with feature 1 of automatic processing, relate to the use of *attention* and *memory*. The next two characteristics of automatic processing have to do with *priorities* and *interruption* of competing processes. The remaining features relate to *practice* and *learning.* I will treat these topics in turn.

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. Conflicts may arise over use of the sensorimotor system. Two automatic processes that require a turn of the head and a redirection of vision, but in opposite directions, obviously cannot successfully operate concurrently. Non-competing automatic processes can operate in parallel, while controlled processing is by nature competitively serial: only one, centralized controlled system exists, and all controlled processing uses it.

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 Priorities and Interruption

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 [90]. 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 [89, p. 160]. 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 she can practice these responses *in situ* [83]. Consistency is important for O(1) processing 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? I 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 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, exhaustive 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 [24]. 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.

2.4 The Controlled-Automatic Interface in Skilled Performance

Application of the controlled-automatic theory to skilled activities more complex than simple recognition and recall tests is important to PRIOPS, because an embedded knowledge-based system may be required to perform skilled activities. In discussing skilled performance, Schneider and Fisk concentrate on interaction of the two types of processing [84]. They make three points, the first relating to controlled enabling of groups of automatic productions [84, p. 135]:

> *"The first function of controlled processing is the maintenance of strategy information in short-term store to enable sets of automatic productions.* Skilled performers exhibit a great deal of flexibility. A performer can rapidly change strategies that substantially alter performance ... The subject cannot change the productions quickly, but can rapidly change the enabling conditions. For example, in a tennis game, a player may switch from trying to tire an opponent to forcing the opponent to the rear of the court. Such a strategy shift would be presumed to change the contents of short-term store, and thus enable or tune different classes of automatic productions. In the same sense that external stimulus conditions, such as the speed of the ball, should determine how the resulting production is executed, internal conditions such as strategy nodes should also determine which productions are executed."

Controlled processing thus allows automatic productions to use short-term memory by providing enabling or *gating* triggers for automatic productions.*

---

* Note that at the time of this paper (1983) [84], Schneider and Fisk were using the word *production* to label the form of controlled and automatic processing. Schneider has since switched to a *connectionist* implementation model. I discuss pre-PRIOPS computer-based models of the controlled-automatic theory in the next section of this chapter.

Automatic processing takes much consistent practice to build, but when automatic reactions are built in the presence of enabling triggers from controlled processing's short-term memory, these memory-based triggers become part of the stimulus situation that activates the automatic reactions. Consequently controlled processing can rapidly enable and disable sets of automatic productions, requiring short-term memory to do so.

Schneider and Fisk also discuss a second way in which controlled processing uses short-term memory to assist automatic tasks [84, p.137]:

> *"A second function of controlled processing in skilled performance is the maintenance of time varying information in short-term store.* Automatic processing may activate information in short-term memory, but, without additional controlled processing, that information will decay in several seconds. In sports, for example, the player may have to maintain information not currently available to the sensory system such as the positions of key players who are not visible. Automatic processes may determine what information is encoded and in what form, but controlled processing resources must be used to maintain that information."

Here controlled processing is performing an adjunct service for automatic processing, providing the latter with short-term records of recently sensed information. With the additional level of indirection through short-term store, automatic productions that rely on controlled processing for this service are less responsive than automatic productions triggering directly off of incoming sensory information.

The third point about skilled performance discusses the overall function of

controlled processing [84, p. 137-138]:

> *"A third function of controlled processing is skilled behavior in problem solving and strategy planning.* Problem solving is an extensive area of psychology which cannot be covered in any detail here. We wish only to make three points. First, the skilled performer must solve problems such as 'what is the strategy of my opponent and what is my best counter strategy?' Second, that such problem solving requires extensive controlled processing resources. Certain performance situations are often novel and hence, are unlikely to evoke automatic productions. And third, that effective strategic planning occurs either when not engaged in the task (e.g., between plays in football), or when the task can be performed almost entirely by automatic productions alone."

Note that in at least some performance situations, strategic controlled processing must wait for lulls in automatic interaction with the environment. Maintenance of short-term store for controlled-automatic interaction - the first two major operations of controlled processing in skilled performance - can occupy a substantial percentage of controlled processing's time, necessitating the deferment of strategic controlled planning until breaks in activity.

2.5 Computer-based Modelling of Controlled and Automatic Processing

2.5.1 Production System Modelling of Controlled and Automatic Processing

As seen in the last section, Schneider's and Fisk's work on modelling controlled and automatic activity used a form of production system [84, p. 120-121]:

> "Practice leads to the development of a large vocabulary of *automatic productions* which perform consistent stimulus to

> response transformations. We are using the term 'productions' in the Newell sense [68] of a generalized condition-action rule that, when its appropriate stimulus conditions are satisfied, performs a given action. You might think of this as a generalized stimulus-response mechanism. The terms stimulus and response are *not* interpreted in the limited sense of a physical stimulus and motor response. Rather, the stimuli and responses can be either internal or external and may refer to classes of conditions and responses as well as individual instances."

Production systems were popular notations for cognitive modelling at the time. Schneider and Fisk did not construct a computer-based model of their theory of skilled behavior, nor did they refine the production system concept to elaborate the fine detail of their theory. My research shares this general notion of reactive stimulus-response productions, while exploring detailed mechanisms for supporting time- and memory-constrained implementation as a reactive computer programming notation.

Hunt and Lansman actually implemented a production system model of controlled-automatic processing [40]. Sensory and short-term memory records consist of simple feature-weight fields. Each field in a record represents a simple feature, and a field's weight determines the degree of presence or absence of the corresponding feature. The system determines productions to fire based on production activation levels that are set, in turn, by feature strengths in records examined by the production left-hand-sides. Automatic productions are hand-coded into a semantic net. Production activation determines the extent of spreading activation in the net. Production notation allows performance of right-hand-side

actions, in addition to semantic net activation propagation. This model does not address details of controlled-automatic interaction in skilled performance discussed in the last section.

PRIOPS shares the Hunt and Lansman model of memory-less automatic productions, but does not use spreading activation for production selection. PRIOPS sensory and working memory fields consist of arbitrary numbers, symbols, or user-defined sets that allow more complex operations than simple feature-weight sampling. The Hunt and Lansman model does not address questions of constant-time restrictions or priority-based production scheduling. The differences in the two architectures comes largely from the difference in their purposes: Hunt and Lansman set out to hand-code a model for simulating some results of choice reaction time experiments. With PRIOPS I am providing an architecture to support symbolic, knowledge-based computation for software development.

With its use of feature weights and spreading activation, the Hunt and Lansman model is actually closer to the *connectionist* model currently used by Schneider and Detweiler than it is to PRIOPS. The next subsection discusses this connectionist model.

2.5.2 Connectionist Modelling of Controlled and Automatic Processing

Schneider and Detweiler use the Controlled Automatic Processing Model 1 *(CAP1)* program to simulate controlled and automatic processing [85,86,87]. The

model, illustrated in Figure 1, is composed of a number of *processing modules*, each module structured as a *neural network.* There are *visual, auditory, tactile, spatial, speech, motor, semantic,* and *context* (a form of short-term memory) modules in the current simulation. Like the Hunt and Lansman model, each module receives input in the form of a vector of feature weights. Like other connectionist models, each module maps input weight vectors to output weight vectors by passing the input into a network of simple, interconnected processing nodes. Learning determines the degree of attenuation performed by node interconnections. Nodes themselves perform summation of input and input threshold determination, *firing* and passing output signals down connections to subsequent nodes when input exceeds the firing threshold. Neural nets share a stimulus-response mode of operation with production systems, but use a non-symbolic, signal processing model for elementary operations. Interested readers should consult Rumelhart and McClelland [79] for an introduction to connectionism.

Schneider's and Detweiler's neural model has two distinctive organizational features. First, modules communicate by passing output vectors of weights across an *inner communications loop* to receiving modules. For example, if the spatial module recognizes a dangerous incoming stimulus - an input vector representing a rapidly approaching object - the spatial module might attempt to send a vector for rapid avoidance along this inner loop to the motor module. The motor module

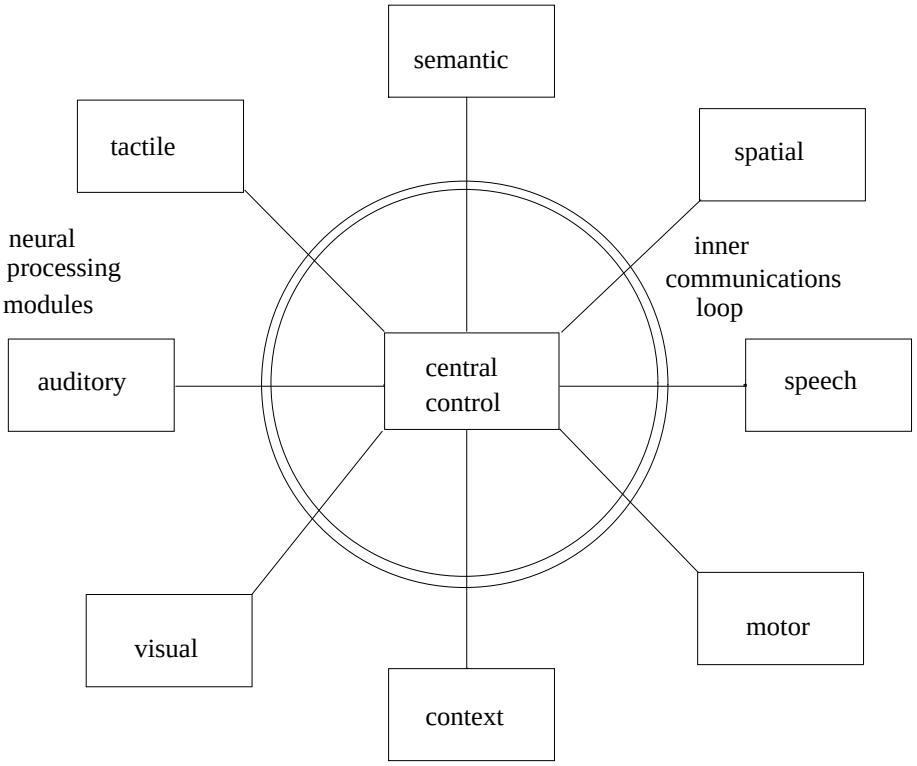


Figure 1 - The CAP1 system level structure

would internally transform this input vector into a vector capable of driving motor neurons connected to muscles. The inner loop is parallel in that it can carry complete vectors, but is serial in that a portion of the loop can carry only one message vector at a time. During intervals when neural modules are transforming information internally, they do not use the communication loop, and can act in parallel. The loop introduces serial delays and dual-task interference for inter-module communications.

A similarity of this model and PRIOPS is that *priorities* help determine scheduling of *automatic* activities. Priorities are a component of the information learned within each neural module. If the spatial-to-motor reaction described in the above paragraph were learned with sufficient consistency, and therefore priority, within the spatial module, the priority of the spatial output would override any contending modules and send the message over the inner loop. Sufficiently high learned priority establishes automaticity, and consistency of successful stimulus-response reactions determines learned priorities. When intra-module priorities are low - representing novel situations or other reasons for uncertain responses - hardware inhibition on module output suppresses message transmission until *controlled processing* can select appropriate module messages for transmission. Controlled processing is the second distinctive organizational feature of this model. In addition to the eight modules mentioned above, a *central control structure* has connections to all modules and the inner loop. This hardware structure performs conflict resolution when no module priority is high enough to override output inhibition. The central control also determines controlled message sequencing to avoid message collisions (and resulting interference) on the loop.

As with the Hunt and Lansman model, PRIOPS' biggest difference from this connectionist model is its use of symbol manipulation as the basis of computation. The CAP1 simulation's use of priorities is analogous to PRIOPS' use of priorities (as we shall see) for scheduling production matching and actions. The work of

Schneider and Detweiler is an application of neural network concepts, but they do not address the issue of O(1) stimulus-to-response mapping.

When beginning the research that has led to PRIOPS, the stimulus-response characteristics of both production systems and neural networks led me to consider both as candidates for reactive real-time architectures. After some exploration I chose to pursue research into a reactive production system architecture for a number of reasons. Production system matching concepts are reasonably mature, allowing me to concentrate on computational complexity-related *enhancements* to an existing body of knowledge, rather than performing fundamental research that connectionism would have entailed. I wanted to focus on real-time specific problems. Nevertheless, research into constant-time-bound input-to-output vector mapping for neural nets could certainly provide the basis for a useful real-time, embedded connectionist architecture. An *O(1) implementation* of a neural stimulus-response network would have to consider the mapping of the abstract neural net onto processing hardware. It is not enough to assure a constant-length signal path through a network, if the network is simulated using non-O(1) time-sharing of available processors across simulated network nodes.

Perhaps a hybrid architecture, one that applies production system mechanisms to symbol-based subproblems and neural mechanisms to signal-based subproblems, could be amenable to practical application. This is a potential area for future research.

3. THE CONTROLLED-AUTOMATIC PRIOPS ARCHITECTURE

The preceding chapter presents how humans have applied division of labor to complex tasks, factoring out important but consistent activities for automatic processing. The PRIOPS notation allows the production system programmer - be it a human programmer or a machine learning algorithm - to do the same for a symbol manipulation program. *Automatic partition* activities include time-constrained tasks that react to the environment and controlled enabling information. The *controlled partition* houses complex operations typically studied in artificial intelligence research.

Figure 2 illustrates data flow in the PRIOPS controlled-automatic 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.

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.* The automatic partition does not store information about its previous state configurations.* 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 2 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.

---

* *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.*

** Hard-wired reactive circuits correspond to *reflexes* in humans. Acquired automatic processes are more complex than reflexes, and they may augment built-in reflexive processing.

3.1.1 O(1) Time Complexity

Time and memory requirements for code in the automatic partition never grow beyond a predetermined constant limit. The compiler 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. While constructing non-iterative execution paths, the code generator can calculate worst-case time bounds. The limitations inherent in non-iterative code restrict the range of size and complexity across which input data to the code may vary.

The compiler disallows iterative composition of automatic 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 2, 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.

3.1.2 O(1) Space Complexity

Lack of persistent memory means 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.* Where data are combined by intersecting paths (e.g., *sensor fusion)*, 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.

3.1.3 Automatic Priorities

Until now I 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,

---

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

although work on multiprocessor production systems [31,32,33,92] 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.

3.1.4 Varieties of Automatic Code

Figure 3 shows 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, I refer to this as an *instantaneous real-time* requirement. This is the variety of automatic code with the tightest response time requirements.

Because 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. 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 automatic reaction to incoming information. There is no degradation of responsiveness through arbitrary-length buffering in the automatic partition. 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 automatic partition reacts to the immediate environmental situation. Besides simplifying memory management and timing analysis, this arrangement is attractive because it corresponds to the memory-less character of

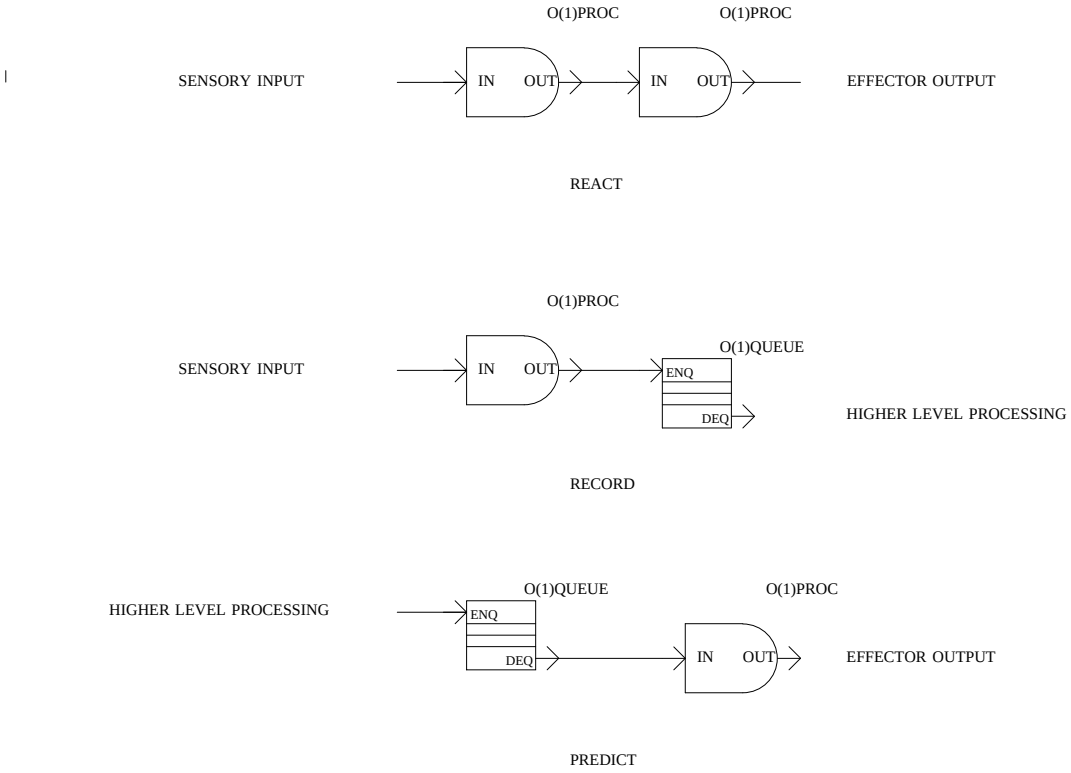


Figure 3 - Three varieties of constant-time-constrained processing

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 2 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 2 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, the compiler 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

Forward-chaining production systems have a stimulus-response organization. They are a form of knowledge representation that is closer to the idea of reactive, 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.

This stimulus-response structure led me to investigate using a forward-chaining production system for the controlled-automatic embedded architecture. Acyclic composition of productions appeared promising for reactive, automatic processing. Compiler-driven inspection of dataflow promised to be straightforward. Also, interactions of executing productions can achieve controlled partition behavior, such as goal-directed search and learning. Production system working memory supports controlled data buffering with minimal explicit programming. The

stylized syntactic format of individual productions make them convenient for programmed manipulation in planning and learning [7,52,55,59,67,71,77].

Unfortunately, the algorithms implementing the run-time environments for existing production systems do not guarantee constant-time bounded responsiveness. The Rete pattern matching algorithm [25,27] is a popular and efficient approach for matching 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 [15,21,26] with novel, time-constrained semantics. Support for this augmentation comes in the form of an enhanced Rete matching algorithm. Enhancements include the use of time- and memory-constrained matching for automatic productions, and priority-based scheduling and preemption of production matching, conflict resolution and actions.

This chapter introduces PRIOPS notation and matching by way of a hypothetical example: an embedded system that monitors environmental temperatures and drives a wheeled vehicle in response to these temperatures. I discuss PRIOPS declarations, controlled productions, controlled Rete, automatic productions, and automatic Rete. A discussion of the internal structure of PRIOPS and its use of augmented Rete matching follows in the next chapter.

4.1 An Application of Controlled-Automatic Processing

Figure 4 shows the data flow for a system that monitors temperature sensors. One automatic behavior reacts to dangerously high readings by triggering effector motions - turning wheels - 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.

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

4.2 PRIOPS Preliminaries

The syntax and much of the semantics of PRIOPS derive from OPS5 [15,21,26]. 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. Listing 1 declares set *all-sensors* with a universe of *t1, t2, t3, t4, tremote1,* and *tremote2.* These six temperature sensors provide the input which drives this 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.

```
(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

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, and fields are strongly typed; unlike OPS5, a field name may not be used for a working memory element type that does not declare that field.* PRIOPS supports typed vector fields similar to singly dimensioned Pascal and C arrays.

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.

4.3 PRIOPS Controlled Productions

Listing 2 shows the four controlled productions that correspond to the controlled partition activity of Figure 4. The first production, *attend-to-hitemp*, responds to temperature sensor readings that exceed some *hi-interest* limit. Readers

---

* 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.

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.

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 *ˆsensor-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 *ˆtarget* field value, a value of type *set all-sensors.*

```
(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 trace-goal-inspect -1        ; trace inspect goals for debugging
        (goal ^action inspect ^target <inspecting>)
        -->
        (write Goal: Inspect <inspecting> (crlf))
)

(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

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* and *trace-goal-inspect.* The former 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. Trace-goal-inspect is a simple debugging production that reports the assertion of an inspect goal.

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. It has 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.*

4.3.1 Rete Network for Controlled Productions

Early production systems were extremely inefficient because they performed redundant matching. With each new modification to working memory, a production system selected all productions whose tests might be satisfied by the working memory change, and performed *all* tests in those productions' left hand sides. These tests included condition elements *not* related to the current memory change, because these unrelated condition elements appeared in productions with other condition elements related to the change. Other redundant tests were *identical* tests required for different condition elements [25,27].

The Rete algorithm eliminates redundant matching. PRIOPS, like OPS5, compiles productions in source code form into a *Rete network.* Rete uses two devices to gain efficiency. The first is a form of common subexpression elimination. When distinct condition elements share identical leading tests, the compiler generates a single test code sequence for the shared tests. Distinct test

---

* The next two chapters discuss, in turn, explicit memory element garbage collection (as in report-alarm) and garbage collection internal to PRIOPS.

code is necessary only when condition element tests diverge. This sharing is without regard to the position of a condition element in a production.

For example, the *goal* condition elements of productions move-to-inspect-target and trace-goal-inspect in Listing 2 both test for an *^action* of *inspect.* Since they share an identical leading test, the compiler generates code to test *^action = inspect* only once.

Code sharing can extend across several condition elements. If two productions share identical leading condition elements, then the compiler generates one set of tests for the shared elements. In addition to tests for a single memory element, the productions share tests across multiple memory elements - *join* tests. In move-to-inspect-target, variable <*interesting*> joins an inspect goal memory element to a sensor memory element when their respective *^target* and *^sensor-id* fields are equal. Since trace-goal-inspect does not have a sensor condition element, it does not require this join test.

Rete also improves efficiency by saving intermediate results of match testing, rather than recalculating intermediate results with each working memory change. For example, assume several *limit* working memory elements match the second condition element of attend-to-hitemp. Rete saves references to these partial matches, so that if a subsequent memory change matches another condition element - perhaps asserting a sensor memory element - there is no need to repeat the limit element tests before doing the sensor-limit join tests.

The Rete net of Figure 5 has five types of nodes, distinguished graphically and processed differently. Circles are *pre-join* nodes, which are tests that apply to just one memory element. Each such test compares an element field value against a constant, or compares two fields within a single working memory element. The latter test occurs when a variable appears multiple times within a single condition element. Ovals are *join* nodes, which compare fields in multiple memory elements. The appearance of a single variable in more than one condition element in a production produces a join test across the corresponding memory elements (e.g., move-to-inspect-target's <interesting>). An oval with a backslash is a *not* node, the inverse of a join. A not succeeds only when its inter-condition element comparisons *fail*. Boxes represent *memory* nodes that hold the results of condition element and join successful matches. Finally, production names at the bottom of Figure 5 signify *instance* nodes. When a combination of working memory elements consistently match all condition elements on a production's left hand side, the production's instance node contributes an *instantiation* of the production - the production paired with the matched working memory token - to the conflict set of triggered productions.

Figure 5 shows the *limit ^limit-type = hi-interest* constant test from attend-to-hitemp, and the shared *goal ^action = inspect* constant test from move-to-inspect-target and trace-goal-inspect. Dynamic behavior flows through a Rete net when a production or device driver produces new data - asserts or retracts a working

memory element. A reference to the altered element propagates down the path for its type. A limit element, for example, advances to the pre-join ^limit-type test node of Figure 5. Other production condition element tests that test limit objects differently, would cause branching in this test path. A production testing *limit ^limit-type = hi-danger* would generate an alternative limit path that 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.

Memory element assertions and retractions propagate identically through the pre-join paths. In Figure 5, no pre-join testing of sensor elements occurs because none appear in the controlled productions.

At the end of a pre-join path these memory element references (called *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 5. 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, or *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 the 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 5 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.

4.3.2 Worst case times for Standard (and Controlled) Rete.

As Haley [34] has pointed out, a compiler can compute weak worst-case

execution times for a single memory change 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. The compiler can detect such exclusions by scanning the successors to a node in a pre-join chain. It must look 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 5, 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.

4.4 PRIOPS Automatic Productions

Listing 3 shows the three automatic productions that correspond to the automatic partition activity of Figure 4. 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.

Production *react-to-overtemp,* with a priority of 10, recognizes a dangerous temperature condition and responds by initiating several activities. The *ˆsensor-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 *ˆsensor-id t1,* production *react-to-overtemp-t2* will test *ˆsensor-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.

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

```
(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

macro, the selected macro expansion replaces the macro tag wherever the latter

```
(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

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 6. 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 *ˆsensor-id t1* constant test node in Figure 6 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.

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 or controlled priority level. Controlled matching proceeds only 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 6 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 [15, p. 230]. In PRIOPS, right hand side assertions to and retractions from working memory immediately trigger matching only when the condition elements (Rete nodes) for the changing memory elements are of greater priority than the production making the changes. Lower priority

assertions and retractions defer while the right hand side continues. 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; I will not discuss these details here.

Besides the use of priority matching queues, the second major distinction in automatic partition Rete is the appearance of the *register* nodes of Figure 6 in place of the *memory* nodes of Figure 5. 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 [68]. 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 join test code for macro generated matching *is shared.* The nodes are distinct, but parallel expanded join 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

---

* Controlled Rete therefore supports a form of *universal instantiation*, providing production firings and variable bindings for all working memory element combinations consistent with a production's condition element tests. Automatic Rete, in contrast, provides a form of *existential instantiation*, giving only one firing and set of variable bindings for a production's condition element tests, based on working memory element *recency*.

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 matching queue is emptied, without considering lower priority pending computations. When no automatic productions are instantiated, matching of controlled productions ensues.

Returning to Figure 6, 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 input 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. THE MAZE DEMONSTRATION PROGRAM

This chapter illustrates a demonstration program compiled and executed using the current PRIOPS implementation. In the spirit of Pavlov, we shall observe the behavior of a PRIOPS program in a simulated maze. Figure 7 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 are two mobile entities: the PRIOPS **PROGRAM** and the **HUMAN** (letters **P** and **H** in Figure 7 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.

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.
- Explicit 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.**

5.1 An Overview of Maze Processing

Figure 8 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.

Controlled productions maintain memory of visited maze locations. 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 controlled partition buffers.

5) Learning.

The overall flow of the maze program as seen in Figure 8 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 8, from sensory readings and memory, to search behaviors in the controlled partition, to instinctual behaviors in the automatic partition, to garbage collection at the controlled-automatic interface, to learning in the

controlled partition, and finally to resulting learned behaviors in the automatic partition. The final subsection discusses execution results for the maze of Figure 7.

5.2 Sensa, Memory, and Search in the Controlled Partition

In Listing 5 are three declarations for working memory element classes. 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.

```
(structure sensors              ; 4 pseudo-sensors 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
)
```

Listing 5 - Declarations for PROGRAM sensors and a maze location

Listing 6 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.

The PROGRAM uses this heuristic strategy rather than simple depth-first search, because the HUMAN may drive it into known dead-ends or down novel paths. The PROGRAM does not expend time recording intermediate state information - its path - while fleeing from the HUMAN. A strict depth-first, backtracking approach would need the intermediate state information. In this situation the heuristic search is more robust than depth-first search, allowing the PROGRAM to recover its search processing context from the immediate environment and memory, and resume exploration at the end of a HUMAN avoidance session.

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.

```
(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 6 - Controlled productions for a new maze location

Production *look-out* triggers when a new sensors element is made at some visited location. 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), 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.) 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 Listing 6 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 Listing 6 then runs again.

The tests of *newly-arrived* and *look-out* contain only inter-condition element (join and not) tests. In *look-out,* no intra-condition constraints appear for sensors, visited, or inspect elements. The result is that Rete matching will perform the join tests, comparing the *^x* and *^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. As we have seen, 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 [34].

5.3 Reactive Movement in the Automatic Partition

Listing 7 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.*

With only one condition element, *panic-left-up* 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.

```
(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 7 - Three automatic productions for emergency reactions

Recall that 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 7, *panic-left-right,* share the initial test *sensors ^left-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* nullifies 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.

5.4 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 controlled partition. 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 [28]. 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 [28,60,94]. 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. Collection waits while a burst of automatic activity completes, and it begins to work upon resumption of controlled activity. In this section I examine an explicit garbage collection production. In the next chapter I resume this discussion by examining garbage collection in internal PRIOPS mechanisms.

The maze demonstration program contains several explicit garbage collection productions. Explicit garbage collection productions recover buffered memory elements that, through testing of contents and time stamps, are found to be redundant or outdated. Listing 8 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 7 will trigger only on the most recently matched move element, so this garbage collection is necessary only for the controlled partition. 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.

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

Listing 8 - A priority 0, controlled garbage collection production

5.5 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 [52]. SOAR uses both domain knowledge and generic weak search methods to search problem spaces, and automatically builds chunk productions that avoid repeated searches [49,53]. 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 9 shows learning production *learn-pass-up,* followed by the learned escape production *auto-24-1-up.*

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 6. 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

```
(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")
        (write learnfile "          {(sensors ^up-sense <> human ^x "
                <endx> " ^y " <endy> ") <sensing>}")
        (write learnfile "          {(lastmove) <oldmove>}")
        (write learnfile "          ->")
        (write learnfile "          (remove <sensing>)")
        (write learnfile "          (modify <oldmove> ^direction up)")
        (write learnfile "          (call move up))")
        ; 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 9 - Maze escape learning and a learned escape automatic production

tests match the *^x* and *^y* fields of the joined memory elements. The *^visits < 10000* test prunes locations that are part of dead-end paths. Dead-end detection code marks visited records with a ^visits 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 9. *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 *^up-sense* test with all other *auto-X-Y-up* learned productions. In addition, this production shares the *^up-sense <> human ^x 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 [93]. 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.

5.6 Maze Statistics

I 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. Both controlled and automatic execution use the PRIOPS intermediate run-time code, so execution time increases caused by op code interpretation penalizes both partitions. The PRIOPS compiler 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 8, 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 9 for the maze of Figure 7.

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 7. 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 6 and 8 help illustrate the source of this delay. Note that the two controlled productions of Listing 6 (and many other controlled productions as well) trigger on *sensors* elements. Now examine *auto-24-1-up* in Listing 9; 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 controlled partition, the escape statistics change. Now learned escape, acting in the controlled partition, 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 controlled partition 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. The most recent token arriving at an automatic Rete node input supplants earlier tokens, eliminating their match time. With the storage capacity of an automatic register limited to one, automatic matching need only consider the most recent working memory change that arrives at or beyond the first register node in an automatic path. When several memory change tokens await processing at a single automatic node input, all but the last are discarded. Note the <*oldmove*> operations of *auto-24-1-up* in Listing 9. A *modify* operation in standard Rete works by emitting two messages: a *delete* for the old working memory element (the former last move in this production), and an *add* for the modified copy of the deleted element (the current last move). Standard Rete propagates both messages down matching paths as far as successful tests allow. When *auto-24-1-up* is priority 0, both modify-generated messages traverse full priority 0 Rete paths. When *auto-24-1-up* is priority 127, the replacing *add* message supplants the *delete* message at the first register node. With no pre-join tests for lastmove, that register is the first Rete node target in the automatic network. No matching or propagation occurs for the delete message.

This elimination of *modify oldmove's* delete propagation through automatic partition matching reduces learned escape processing from 66 seconds to 55 seconds, a reduction of 16.6%. The automatic partition 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 automatic partition provides not only O(1) matching with preemptive priorities, but also provides opportunities for additional speed enhancements beyond the capabilities of standard Rete.

Forgy gives the best-case effect of working memory size on number of memory change tokens as O(1), and the worst-case effect as $O(W^C)$ (where W is the number of elements in working memory and C is the number of condition elements in a production). Best-case effect of working memory size on time for one firing is O(1), and worst-case effect is $O(W^{2C-1})$ [25,27]. With maximum C known for a non-learning production system at compilation time, worst case effects of working memory size on the number of memory change tokens and firing time is polynomial. For automatic Rete, PRIOPS reduces the worst-case compile-time complexities to the best-case, O(1). While there are many problems that automatic matching cannot address, it does not waste computational time on anticipated reactions. We can expect that the more readily automatic processing meets a problem's demands, the more drastic performance improvements will be.

6. PRIOPS INTERNALS

Previous chapters used examples to illustrate PRIOPS' application of various Rete node types. This chapter gives an overview of PRIOPS implementation internals. Focus is on mechanisms important to run-time support of PRIOPS programs, starting with key data structures. Non-programmers may wish to limit their reading of this chapter to this introduction. Skipping subsequent sections will result in loss of implementation detail.

Section 6.1 discusses *working memory.* Working memory is the short-term store of production systems. This section examines composition of individual working memory elements, and aggregation of elements into working memory. Restrictions on working memory operations in automatic productions assist in guaranteeing constant-time-bound execution.

The following sections concern themselves with (long-term) production memory. The most expensive (and therefore from an implementor's perspective, important) production-oriented activity is matching working memory changes to left-hand-side tests in the Rete network. Because PRIOPS is based on an underlying uniprocessor machine, concurrent execution of pending Rete activities is not possible. While a single working memory change may require testing at several Rete nodes, only one node can test at any time. Other test activities must wait, so Section 6.2 discusses priority queues where deferred Rete activities wait.

Section 6.3 treats conflict set priority queues, where production instantiations wait for execution or retraction. In the automatic partition, both sets of queues support O(1) insertions and deletions.

The next sections illustrate internal actions and external interactions of the now-familiar PRIOPS Rete node types. Pseudo-code outlines the behavior of each Rete node type. Forgy provides a similar discussion for the LISP-based Rete node types of OPS2 [25, p. 81-87]. OPS2 nodes correspond to PRIOPS controlled nodes, while the automatic nodes of this chapter are unique to PRIOPS. For this reason I begin with controlled Rete. Controlled Rete has all of the power of standard OPS Rete, while supporting additional operations such as lexicographical symbol comparisons. Automatic Rete must support O(1) reactivity down to the bottommost level of code. It is not enough to design an elegant, apparently O(1) algorithm, if its implementation inadvertently uses some non-O(1) routine. For this reason I supply detailed pseudo-code descriptions of node behaviors. The individual sections and node types are:

- 6.4 The Rete Network - this section discusses interconnection of node types into a net, and the accompanying interactions.

- 6.5 The Pre-join Test Node - this node type tests conditions within a single working memory element.

- 6.6 Controlled Memory Node Types - these node types store variable numbers of working memory combinations that satisfy sequences of condition elements.

- 6.7 The Controlled Join Node - this node type tests conditions across multiple working memory elements and builds memory element combinations.

- 6.8 The Controlled Not Node - this node type also tests conditions across multiple working memory elements, but it inhibits processing of working memory combinations that satisfy its tests.

- 6.9 The Controlled Instance Node - this node type manages insertion (and deletion) of production instantiations into (from) the conflict set.

- 6.10 Automatic Register Node Types - unit-size counterparts to controlled memory nodes.

- 6.11 The Automatic Join Node - the O(1) counterpart to the controlled join node. It is O(1) because it tests a single combination of working memory elements for a single working memory change.

- 6.12 The Automatic Not Node - the O(1) counterpart to the controlled not node.

- 6.13 The Automatic Instance Node - the O(1) counterpart to the controlled instance node. It allows only one instantiation of its production.

Section 6.14 resumes discussion of garbage collection of working memory elements begun in the maze chapter. Here the focus is on time-constrained recovery of structures internal to PRIOPS. Once again, the thrust is deferral of garbage collection during automatic processing, with resumption of this activity upon return to controlled processing. Section 6.15 discusses two inference drivers, the *controlled inference driver* and the *automatic inference driver*. These procedures initiate and manage activity in the Rete network and conflict set. Like automatic Rete nodes, the automatic inference driver must exhibit O(1) behavior. Both drivers must deal with the potential for critical section problems that comes

with interrupt handling and interleaved process execution. The chapter ends with Section 6.16, which discusses the PRIOPS compiler and generated code. Appendix A documents the syntax and semantics of the implementation of PRIOPS written as part of this research. Appendix B gives an overview of source code organization for this implementation.

6.1 Working Memory

The structure declarations of previous chapters provide much information about working memory element formats. A PRIOPS working memory element occupies a contiguous area of memory, similar to a C *struct* or a Pascal *record.* PRIOPS accesses *int* fields as 4-byte C *long* integers, float fields as 4-byte C single precision *floats*, and symbol fields as 4-byte C *pointers* to structures that describe corresponding symbols. Symbol construction stores each distinct symbol uniquely as in LISP [4], so symbol pointer comparisons for equality and inequality consume constant time. While PRIOPS supports symbolic computing from the programmer's point of view, compilation transforms symbol manipulation down to non-symbolic computation wherever possible, especially for automatic processing.

Strong typing of working memory element fields avoids run-time testing of field type tags. While run-time type tag testing would be an O(1) activity, it would still consume time. Run-time testing would allow type mismatches to go undetected during compilation. A future version of PRIOPS might allow untyped

fields (with all of the accompanying penalties) for controlled productions. It is doubtful that automatic productions, which are by nature special-purpose, will require this flexibility.

Chapter 4 also mentioned mapping user-defined symbol sets into constant-space-bound bit maps at compile time. The size of bit maps, implemented as arrays of C *unsigned short* integers, depends on the number of symbols in the PRIOPS *set* declaration. Using bit maps, machine operations accomplish *intersection* (bitwise AND of the operands), *union* (bitwise inclusive-OR of the operands), *set difference* (bitwise AND of the left operand with the one's complement of the right operand), and *membership test* (bitwise AND with a bit mask and test for non-zero) in constant time. Examination of OPS5 and other LISP-based programs led me to discover that much symbol usage can be mapped into set based operations. An individual bit map stores information more compactly than a list of equivalent symbol pointers. Operations over a *dense* bit set (i.e., one that contains most members of the underlying universe) are faster than operations that iterate over a list of symbols. For *sparse* sets and sets intended to hold only one symbol at a time, symbol arrays and individual symbol fields are available.

Each working memory element contains a) fields specified by the programmer in a structure declaration, b) a *time stamp,* and c) links to the rest of working memory. The time stamp (a long integer) is the basis for the *recency* comparisons of PRIOPS' conflict resolution. Each PRIOPS *make* operation increments the

global time stamp counter and assigns this value to the new working memory element it makes.

Working memory is a doubly-linked list, chained together by the two links in each working memory element. Even though this is a simple sequential list, remember that Rete matching does not search working memory. Instead, each working memory change traverses the Rete net. Assertions (at the head of the list) and retractions (from any location in the doubly-linked list) take constant-bound time to adjust a maximum of four pointers (two for retractions). The only time PRIOPS traverses working memory as a sequential list is during user debugging queries, queries that do not occur at production run-time.

Initialization of a new working memory element sets all numeric fields to zero, all set fields to empty, and all symbol fields to the unique *nil* value. Because this element initialization consumes time, the controlled inference driver maintains a *ready list* of initialized elements potentially needed during bursts of automatic activity. I discuss ready list maintenance later in this chapter.

Restrictions on tests and operations allowed in automatic productions keep these productions constant-time-bound. Constant-time machine instructions support numeric type conversion and arithmetic operations. Automatic productions may therefore freely intermix integers and floats, applying all relational and arithmetic operations to them. All set operations work in constant time, and may therefore appear in automatic productions. Set operations include both those already

discussed, and tests including *equality, inequality, superset* (>=), *proper superset* (>), *subset* (<=), and *proper subset* (<).

Tests for *symbol equality* and *inequality* rely on the fact that PRIOPS stores each distinct symbol only once. These tests use constant-time machine pointer comparisons, and can appear in any production. Working memory field operations that depend on symbol string length may not appear in automatic productions. Most such operations are O(n) on the length of symbol strings, and include lexicographical order tests, coercion of symbol values to values of other types, coercion of values of other types to symbol values, and string concatenation. Restricting maximum symbol string size would make these operations O(1) in a sense - the compiler would always assume a maximum string length - but such restriction would provide only weak, worst case limits. PRIOPS allows the programmer to write these operations in the controlled partition, staging information for the automatic partition. The automatic partition may *use* symbols, it simply may not *acquire* them.

Built-in right hand side text file input-output is restricted to controlled productions, both because of arbitrary text length, and because the operating system that provides file I/O for current PRIOPS (Microsoft® MSDOS 3.1) is not a time-constrained system. *Time-constrained application specific I/O* is much of the *purpose* of the automatic partition. O(1) custom C input drivers supply data to the automatic partition using PRIOPS *make, remove* and *modify* procedure calls.

Automatic productions supply data to O(1) custom C output drivers using the *call* operation from production right hand sides. C drivers have access to the same memory element and field operations that automatic productions have. In most cases these drivers will be short routines, with automatic production sequences performing the bulk of time constrained work.

The symbol table entry for each *structure* declaration maintains additional information needed at run-time. This information includes offsets of symbol fields (for initialization to the *nil* value); ready list and garbage list information (the latter is for *removed* elements whose remove matching is not yet complete); priority of elements of the structured type (inherited from the highest priority production matching the structured type); and a pointer to the Rete node where matching for this structured type commences.

6.2 Matching Priority Queues

PRIOPS' uniprocessor implementation requires storing the state of deferred processes in a data structure until these processes execute. Even multiprocessor Rete systems, which support more processes than hardware processors, must record deferred process states [31]. When a working memory change propagates down Rete paths, it usually arrives at several Rete nodes, often in parallel. PRIOPS keeps waiting match processing in a series of *matching queues.* Queues allow O(1) insertion and deletion of entries. First-in, first-out (FIFO) queuing discipline

ensures that more recent working memory changes arrive at Rete nodes later than earlier changes. Order of arrival is important in the automatic partition, since the most recently arriving memory change token at or beyond the first register node in a path, replaces any earlier arrivals. FIFO queuing ensures that the most recent arrival represents the most recent memory change affecting a Rete node.

PRIOPS maintains separate sets of queues for automatic and controlled matching. Each set is an array of queues, with one queue per priority value. Supported priorities are 1 to 127 for automatic productions, and -128 to 0 for controlled productions.* The automatic inference engine uses the former, and the controlled inference engine uses the latter. Each queue is a first-in, first-out singly-linked list. Partitioning queued matching activity by priority allows fast access to the highest priority pending process; the priority of this activity resides in a global variable, which when used as an index, provides rapid O(1) access to the correct queue. When the highest priority queue becomes empty, it may be necessary to iterate over lower priority queues in search of deferred, lower-priority activity. With the number of queues constant bounded at compile time, this iteration is O(1).

---

* Implementation considerations determined the range of priority values: a priority fits compactly into an eight-bit byte.

Each queue entry constitutes a *message* to a Rete node. A working memory change enqueues a message for the Rete node at the start of test paths for the memory element's structure type. Each Rete node likewise enqueues messages for descendant nodes. Each message represents deferred processing of the working memory change for a single Rete node. Deferred processing becomes active when a message arrives at its Rete node. A message contains three pieces of information:

- A pointer to the *target* (destination) Rete node.
- A pointer to the working memory *token.*
- The *action (add* or *delete)* to perform on the token at the target.

The target is the Rete node that tests or stores the working memory change. The node may be controlled or automatic. The priority of the target Rete node determines the priority queue when a enqueuing a new message. The structured type of a changed memory element determines its initial target Rete node.

The token represents the working memory change to be tested. The token pointer is a direct pointer to a changing working memory element for *pre-join* Rete nodes. Each pre-join node tests a field within a single memory element against constants or against other fields in the same element. When its test succeeds, a pre-join node passes the token to descendant pre-join and memory nodes. Starting with an initial *join*, tests examine joined combinations of memory elements. A single post-join token represents a combination of working memory elements that

satisfies the most recent join's tests and all preceding tests. From the first join onward a token pointer points to an array of (joined) pointers to contributing working memory elements. The controlled memory or automatic register node following a *join* or *not* node stores these arrays of direct pointers. A join of two pre-join tokens, for example, will create an array of two pointers to contributing memory elements; the memory (or register) node descendant from the join node stores this array. The memory elements of this token array satisfy all of the pre-join tests of their respective condition elements; in addition, they satisfy the inter-condition element tests of the two condition elements. The token pointer for the message out of this join node (to descendant Rete nodes) after join completion is a pointer to the array of pointers to memory elements.

Each Rete node reacts to *adds* and *deletes* of tokens. An *add* results from a working memory element *make,* a *delete* from a *remove.* The right hand side *modify* operation sends both *delete* (for the original) and *add* (for the modified copy) messages. All PRIOPS *add* messages require execution of pre-join and join tests to determine successfully matched tokens. Post-join *deletes* can examine memory (or register) nodes for departing tokens, avoiding redundant execution of matching tests.

Both *join* and *not* nodes are *two-input nodes* - they join the messages of predecessor memory and register nodes. A message to a two-input node must tell it the incoming direction of the message. Consequently two-input nodes require

four action types: *add* and *delete* (from the left input), and *radd* and *rdelete* (from the right input). The memory node sending a message to a join node sets the action type when it enqueues the message.

6.3 Conflict Set Priority Queues

A satisfied production's execution consists of its right-hand-side actions. OPS5 has a serial model of production execution. Although productions can share matching tests, and several production instantiations* can enter or leave the conflict set during a single inference cycle, conflict resolution selects only one instantiation for execution in a single inference cycle. PRIOPS goes further, allowing matching *and execution* of a higher-priority automatic instantiation to interrupt lower priority execution. Nonetheless, some production instantiations must wait in the conflict set while others run. Only one instantiation at a given priority executes at a time, and only one controlled instantiation executes at a time. PRIOPS uses queues to hold conflict set instantiations.

Like matching queues, the automatic and controlled conflict sets are arrays of queues. Again there is one linked queue per priority. Access to the highest-priority conflict set is through fast indexing of a global variable. Each queue entry

---

* An instantiation is a production plus a combination of working memory elements that satisfy the production's condition elements.

represents an instantiation that is ready for execution. A single queue entry contains the following information:

- Linkage to information for the satisfied *production.*
- A pointer to the working memory *token* that satisfies the production.
- Sorted *time stamps* for the token.

Production linkage provides access to several important pieces of information. Foremost is a pointer to *right hand side code* for execution when the production runs. This executable code receives two arguments, a pointer to the instantiation token, and a pointer to storage for *variables bound on the rule's right hand side.* The production linkage provides access to this second pointer. Whereas variables bound on a rule's left hand side represent fields in working memory elements that exist prior to rule triggering, variables bound on a rule's right hand side require temporary storage during rule execution. Only one instantiation of a given rule can execute at a given time.* Consequently, for RHS variables the compiler allocates static storage, which each execution of a rule reuses.

Additional pointers link an instantiation to its contributing *instance Rete node.* This node manages changes to the set of instantiations for a given production

---

* Automatic preemption can allow one rule's execution to interrupt another rule's execution, resulting in interleaved right hand side processing. However, the interrupting rule will always be of higher priority, and therefore a different rule.

caused by working memory changes. We will examine this node type in the section on Rete nodes.

The sorted time stamps are simply copies of time stamps in the working memory elements that compose the token. Time stamps are sorted according to the current conflict resolution strategy for fast token-to-token comparisons. Each conflict set priority level maintains its instantiations in sorted order. As stated in previous chapters, automatic conflict resolution uses memory element age to resolve conflict among instantiations at a single priority, so time stamps are sorted with the oldest first. Controlled *lex* (lexical) strategy (as in OPS5) uses recency of memory elements, while the *mea* (means-ends analysis) strategy gives special weight to the recency of a memory element matching a first condition element. Controlled time stamps are sorted according to strategy.

6.4 The Rete Network

The introduction to this chapter enumerates and summarizes the Rete node types. While each node type exhibits distinctive behavior, all types perform several common activities. Fields supporting generic node processing are:

- The *priority* of the node.
- The node *type tag.*
- Location and size information for *executable test code.*

- Information about *contributing productions.*
- Linkage to *sibling and descendant nodes.*

Figure 6 of Chapter 4 showed priority values attached to Rete nodes. The type tag is an enumerated value that shows a node's type (e.g., *pre-join test node* or *automatic not node).* The type tag serves as an index into a table of *node handler procedures*, defined in this chapter. When the inference engine sends a dequeued message to a Rete node, it calls the node's handler by way of the type tag.

Executable test code applies only to test nodes: *pre-join, join,* and *not* nodes. This code performs tests compiled from source condition elements. It returns a success or failure flag that determines the fate of token propagation to node descendents.

Information about productions whose left hand sides contribute to a node, is presently useful for debugging. Currently PRIOPS does not support the *excise* command for dynamically eliminating productions; production information at a Rete node will allow a future implementation of excise to identify and recover storage for Rete nodes associated only with departing excised productions. OPS5 does not recover Rete nodes for excised productions. Instead, it continues to perform all matching operations for all excised productions before noting that they should not fire [15, p. 242-243].

Inter-node linkage allows organization of Rete nodes into a network. All nodes contain *sibling* and *descendent* links. Suppose we compile the two productions of

```
(p high 127
        (type1 ^a 1 ^b 2 ^c <C>)
        (type2 ^x 3 ^y 4 ^z <C>)
        ->
        RIGHT HAND SIDE
)

(p low 1
        (type2 ^x 3 ^y 9 ^z <Z>)
        (type2 ^x 3 ^y 4 ^q > <Z>)
        ->
        RIGHT HAND SIDE
)
```

Listing 10 - Two productions with shared tests

Listing 10. We wind up with a Rete net that looks like Figure 9. The leftmost pre-join chain performs constant tests for condition element one of production *high.* The *type2* tests diverge to perform constant tests for the two second condition elements (^y 4) and the first condition element of *low* (^y 9). Priority numbers decorate the nodes of Figure 9.

Figure 10 shows the network of Figure 9, redrawn to show actual implementation links. Each Rete node has a minimum of two linkage pointers - a descendant pointer and a sibling pointer. When alternative paths descend from a node such as the first *type2* node of Figure 9, the descendent link points to the descendent node with the *highest priority.* Matching finds other direct descendents by traversing the sibling chain, which is *sorted by priority in descending order.*

Figure 10 shows the (priority 1) *^y 9* test for *type2* as the right sibling of the higher priority (127) *^y 4* test. After performing the *^y 4* test, matching will enqueue the lower priority message and continue to dequeue priority 127 messages and send these messages to their targets, until priority 127 matching completes. If any priority 127 production is then instantiated, the priority 1 messages will remain in their queues during conflict resolution and execution.

Each two-input node has two parents, one supplying messages to its left input and one supplying messages to its right input. The *join* nodes of Figure 9 are examples. Consequently, register nodes (and memory nodes in the controlled partition) may have two descendants, requiring left and right descendant pointers. The two-input nodes themselves contain left and right sibling pointers to connect to other descendants of the respective left and right parent registers (memories). The center register of Figure 9 has two descendents. Figure 10 shows the resulting right descendent / right sibling chain. Again the sibling chain is sorted by priority to enable deferral of lower priority matching. Two-input nodes also have left and right parent pointers (not shown), because upon receiving a message from one input, join tests must consult the opposing register (memory).

In addition to core fields just discussed, each Rete node type supports fields specific to its functionality. The following sections examine the assortment of node types. A small section of pseudo-code that outlines the behavior of a node type's handler accompanies each description.

6.5 The Pre-join Test Node

6.5.1 Basic Pre-join Behavior

The pre-join node type contains no extra fields. It links to executable code that tests a single field of a single memory element token against constants and other fields within the memory element. Pre-join is the only type of node that controlled and automatic partitions may share. Because the compiler will not allow non-O(1) tests for automatic productions, all pre-join tests in the automatic partition run in constant-bound time. Here is the pre-join handler:

1. Pass the memory element reference to the compiled code.
   The latter returns success or failure.

2. If tests succeed

   Enqueue the current token and action for transmission
   to the descendant node.

3. Enqueue the current token and action for transmission
   to the sibling.

The enqueuing procedure identifies the correct queues for descendants and siblings, based on their priorities. All Rete nodes enqueue messages for their siblings regardless of test results, because siblings represent *alternative* test paths.

The pre-join node does not distinguish between *add* and *delete* actions. Matching occurs for both cases. A pre-join node retains no local memory of added tokens that it has previously passed to descendents, so it must use matching to

determine whether to propagate a delete message. Returning to Figure 9, the first *type2* node will pass added tokens satisfying *ˆx 3.* The node could retain local memory of the added tokens it has propagated. Delete matching would then require a search of this local memory. The node would send delete messages to descendents only for tokens in its local memory. However, pre-join tests are simple. In many cases they operate more quickly than a scan of local memory. While not necessarily improving performance, local memory would consume storage. Finally, join matching time dominates the performance of Rete. Therefore, pre-join nodes perform their simple matching tests for both *add* and *delete* actions, retransmitting successful matches to descendants.

6.5.2 Pre-join Message Queuing

Message queue entries for automatic pre-join nodes show an important difference from other automatic node types. Because of the possibility of multiple instantiations for a single *controlled* production, there may be multiple messages in queue for a single controlled Rete node at one time. Conversely, any new memory element that completely matches the pre-join tests of an *automatic* condition element supersedes all previous matches for that condition element. The maximum number of queued messages for automatic nodes including and following the initial register node in a chain is therefore *one per node input.* A new automatic message always *replaces* any older queued message for these automatic node inputs.

At first glance it might appear that by not allowing controlled and automatic productions to share individual pre-join nodes - that is, by compiling completely distinct controlled and automatic Rete nets - the maximum number of messages in queue for an automatic pre-join node might reduce to one. This reduction is not possible. To see why, consult Listing 10 and Figure 9 again. Suppose the following three *make* actions occur:

(make type2 ^x 3 ^y 4 ^z -1) ← call this element 1

(make type2 ^x 3 ^y 9 ^z 5) ← element 2

(make type2 ^x 3 ^y 4 ^z 11) ← element 3

All three messages enter the priority 127 queue with an identical target: the first *type2* pre-join node. It is not possible to reduce these three to one at the first pre-join node, because element 2 will propagate to the rightmost register, while elements 1 and 3 will propagate to the center register, with element 3 replacing 1. Elements 1 and 3 satisfy *all* of the pre-join tests for the condition elements associated with the center register node.

We see that automatic pre-join nodes do allow queuing delays. In this context I repeat a statement from subsection 3.1.3 on automatic priorities: 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.* The latter clause concerns us here. Suppose a reactive processing chain can respond to a triggering environmental event in 5 milliseconds. If the event in question actually occurs once per millisecond, then the constant-time-bound reactive processing is simply not fast enough for the environmental problem. PRIOPS can guarantee the processing time for automatic reactions, as long as the arrival rate for events does not saturate that processing time. Stated simply, automatic matching can guarantee response times only if the system designer can guarantee maximum event arrival rates. This condition is true for any system that takes non-zero time to respond to an event. Given that the system requires finite time, one can always propose arrival intervals shorter than that time.

For critical pre-join paths, the PRIOPS programmer should use distinct *structure* types for controlled and automatic condition elements. Distinct types will keep memory changes intended purely for controlled processing from forming queues at automatic pre-join nodes. Some redundant node generation will occur because controlled nodes may repeat the tests of automatic nodes, but controlled information will not impede automatic data flow.

While we have not yet examined other automatic node types in detail, it is worthwhile to continue tracing the three *make* operations through the network of Figure 10. The make operations might occur outside of any rule - either from an initialization file or from user keyboard input when PRIOPS is not in production

run mode. These make operations might also occur in the right hand side of an executing production whose priority is greater than or equal to the priority of the Rete node targets of the make operations (priority 127 in this example). In such cases all three makes enter the priority 127 queue with add messages for the first *type2* node. Because they were made together without interruptions, they arrive together at the head of the queue without intervening messages.

Now consult Figure 10. The first *type2* node processes the messages in turn, finds that they satisfy its test, and enqueues them for its priority 127 descendant. Again the messages reach the head of the queue together. Element 1 satisfies the ^ *y 4* test, so the second pre-join node enqueues an add message for its descendent register node, then enqueues an add message for its sibling (priority 1) pre-join node. Element 2 does not satisfy the second node's test, so this element enters only the priority 1 queue with the sibling as a target. Finally, element 3 satisfies the second pre-join node test, and the act of enqueuing element 3 for the register node overwrites the element 1 message with the element 3 message. Having satisfied all pre-join tests for a condition element, element 3 cancels matching for other elements satisfying that automatic condition element.

In the absence of *type1* memory elements, priority 1 matching will eventually advance memory element 2 to the rightmost register node. Elements 1 and 3 fail to traverse this path. The join test fails (^q, with a default value of 0, is not greater than ^z's value of 11), so propagation ends at the first-level register nodes.

Now suppose that the three make operations occur in the right hand side of a production whose priority is lower than that of the first target Rete node for the makes - lower than 127 in this example. The right hand side make actions might enqueue all three makes together, as in the case we just examined. Alternatively, finding the make matching to be higher priority than that of the current production, the first make might suspend the making right hand side and proceed with priority 127 matching for element 1. Possible design alternatives are:

> Make execution of a rule's right hand side atomic. Working memory operations merely enter their queues until the rule finishes.
>
> Suspend the current right hand side execution when the change has a target node of greater priority than the currently executing production. Perform right hand side working memory changes in the order specified by the programmer.
>
> Suspend the current right hand side execution when the change has a target node of greater priority than the currently executing production. Allow the compiler to rearrange run-time order of right hand side working memory changes so that the highest priority change occurs first.

Atomicity of right hand side operations argues for the first alternative, responsiveness to priorities argues for the others. In chosing the second alternative for PRIOPS, I have opted for responsiveness to priorities and flexibility for the PRIOPS programmer. PRIOPS performs right hand side working memory changes in the order specified by the programmer. This choice is in keeping with the OPS5 tradition of performing condition element tests in programmer-specified order, rather than rearranging them in attempts at optimization.* Making all right hand

side executions atomic would disable the programmer from intentionally ordering memory change actions for sequencing effects. Automatically reordering memory change actions would likewise take sequence control away from the programmer. The programmer is free to achieve the effects of the other design alternatives through appropriate programming. Right hand side suspension occurs only when the memory change invokes higher priority *automatic* matching. Other memory changes wait in appropriate matching queues.

Recall from the chapter on human controlled-automatic processing, that a key function of strategic controlled productions is rapid enabling and disabling of sets of related automatic productions. The PRIOPS programmer achieves atomicity of right hand side actions by disabling higher-priority automatic productions while sending non-enabling memory change messages to them. When all non-enabling memory change actions are done, the running production *makes* the enabling working memory element, and higher priority matching commences.**

With this programming flexibility comes responsibility. Let us return to Figure 10 and examine the effects of our three *make* examples when they occur in an

---

* OPS5 also performs right hand side actions in programmer-specified order, which is immaterial since OPS5 does not provide preemption or priority-based match scheduling.

** For example, if automatic production *react* contains condition element *(enable ^production react)*, the lower priority right hand side does not *(make enable ^production react)* until it has made other messages for *react.*

executing right hand side with a priority of 0. The make of memory element 1 triggers matching immediately, suspending the lower-priority production. Element 1 propagates by itself to the center register node. With no higher priority rules instantiated, the interrupted production resumes execution, makes element 2, and again waits for matching. Element 2 propagates to the right register node in Figure 10. This time the join test *succeeds* (^q, with a default value of 0, is greater than ^z's value of -1), so production *low* (at priority 1) enters the automatic conflict set. Being automatic and more salient than the suspended production, *low* executes. Eventually the (priority 0) suspended production *will* resume - all executing productions complete unless an interrupting production *halts* inference - and the make of memory element 3 will supplant the token from element 1, emptying register nodes from the failed join onwards. But by this time, the make of element 1 has had an effect - the execution of rule *low*. Obviously, judicious use of enabling and disabling working memory changes is necessary when the effects of lower priority actions on higher priority matching must be serialized.

The discussion of pre-join matching has required detailed examination of the actions of PRIOPS priority matching queues. Priority-based match scheduling is unique to PRIOPS. Standard OPS5 Rete matching makes no use of queues. Given that all standard Rete matching completes before conflict resolution, inter-node communication occurs through simple procedure calls. A satisfied pre-join node, for example, will directly call its descendants with the message. Likewise, message

passing to siblings is through direct procedure calls. PRIOPS requires priority queues both because of its ability to defer portions of matching, and because more recent matches to pre-join chains of automatic condition elements displace earlier matches. The first-in first-out queuing discipline helps support this latter mechanism.

Multi-processor Rete implementations *do* require queues of matching processes [31], but these implementations are intended to mirror the functionality of standard Rete at higher speeds. They do not use production priority as a key for scheduling and deferring matching. Priority-based matching and restricted automatic memory are the distinguishing features of PRIOPS Rete, and we are seeing that these features have important effects.

6.6 Controlled Memory Node Types

6.6.1 Alpha and Beta Memory

I have used the term *memory node* for the controlled partition's unbounded storage mechanism, and *register node* for the automatic partition's single-place storage mechanism. In fact, each partition uses two types of storage nodes. This discussion focuses on the simplified Rete net of Figure 11. On the left side is the standard organization of pre-join tests and join/not nodes. Earlier illustrations show that any chain of pre-join test nodes terminates in a memory node (register for automatic productions), and a memory node follows each join or not node. The

left side of Figure 11 emphasizes the fact that the a production's initial join (brought about by the abutment of its first two condition elements) is the descendant of two pre-join test chains, while subsequent joins (due to subsequent condition elements) *always* have post-join left parent memories, and pre-join right parent memories. Standard Rete supports two memory types as a result. *Alpha* memory nodes terminate pre-join chains. Alpha memory has two distinguishing characteristics: a) each token it stores is always a direct pointer to a working memory element, and b) it may serve as a left input to some joins nodes (in cases where it represents the first condition element in production(s)), and may serve as the right input to others. In addition to core Rete node fields, an alpha node has token storage fields. It also has a right descendent pointer for child joins where the alpha memory contributes the join's right input. Figure 11's left side shows that all pre-join chains except those chains representing first condition elements, will advance their messages to right descendants (i.e., right join inputs). Only the first pre-join chain's alpha memory in the figure sends messages to a join's left input.

The descendants of join and not nodes are called *beta* memory in Rete literature. Beta memory always stores joined tokens, that is, arrays of pointers to working memory elements. A beta memory always supplies its tokens to join and not left inputs, so it needs no right child pointer. Storage fields for joined tokens (pointer arrays) and the absence of right descendants differentiate beta from alpha

memory. The beta memory has an integer field that records the number of working memory element pointers in the joined tokens it stores. Both controlled memory types can store dynamically varying numbers of tokens.

6.6.2 Standard versus Balanced Rete Networks

The right side of Figure 11 gives a viable alternative compilation of condition element tests into a *balanced* or *binary* Rete network. Part of the purpose for a balanced net is avoidance of the *long-chain effect* of standard Rete. Assume that the pre-join tests labelled *B, C,* and *D* in Figure 11 have been satisfied. None of the joins of standard Rete *(AB, ABC,* and *ABCD* in the figure) have been tested, because the join tests require *A.* In contrast, the *CD* join of the balanced network has completed, reducing (in this example by 1) the number of joins that must execute when *A* finally arrives. The problem becomes more severe with longer chains.

Unfortunately, balanced Rete is not always a good solution to the long-chain problem. In fact, OPS5 encourages programmers to make use of the effect. OPS5's *mea* strategy uses a production's first condition element to select instantiations related to the current *goal* for execution. The first condition element enables productions related to the goal expressed in its tests, and mea conflict resolution selects the instantiation with the most recent memory element matching the first condition element for execution (additional conflict resolution criteria break ties for

first element recency). Consequently, OPS5 programmers use the first condition element to *disable* join matching not related to an active goal. Memory changes may remove right input messages from disabled join nodes, so many join tests may be avoided. This is a simple, standard OPS5 mechanism for deferring and possibly avoiding join test processing for joins unrelated to active goals. The tradeoff is between longer time to enable large sets of goal-related productions (making element *A* in Figure 11's left side), and more time spent processing joins unrelated to the current goal (joining *CD* in Figure 11's right side, possibly to have *C* or *D* retracted before *A* ever appears). Gupta has measured assorted applications of OPS5 and SOAR [31, Sections 5.2.6 and 8.6], and found that neither approach of Figure 11 is a clear-cut winner. Hand coded OPS5 programs fared better with standard Rete, because join chains were short and programmers took advantage of the join enabling/disabling strategy. SOAR programs that learned productions performed better with balanced Rete, because learned productions contained many condition elements (long join chains) and SOAR programs did not take as much advantage of deferring joins.

PRIOPS uses the standard structure on the left side of Figure 11. Long-chain time is not a severe problem for automatic Rete because the work of automatic joins is greatly simplified from controlled joins, to the point of O(1) complexity. Automatic conflict resolution does not give special purpose to the first condition element, allowing automatic productions to have their enabling/disabling condition

elements appear anywhere in their left hand sides (i.e., supply any join node in Figure 11's left side). The programmer can opt for long-chain enable time by placing the enabling condition element early in the production left hand side. Conversely, the programmer can opt for reduced enable time at the cost of unnecessary joins, by placing the enabling condition element late in the production left hand side. Given these advantages of the standard network structure, and the fact that the standard structure is often better for hand-coded programs, PRIOPS sticks with the standard structure.

6.6.3 Alpha Memory Handling

Next comes the pseudo-code for the (two-output) alpha memory handler:

1. If action is add token

   Store the token in local memory.

   Set the "right action" to "radd."

   Else (delete)

   Find & remove the token from local memory.

   Set the "right action" to "rdelete."

2. Send the current token with the incoming action to the left child.

3. Send the current token with the "right action" to the right child (thereby informing the join node of the message input direction).

4. Send the current token with the incoming action to the sibling.

PRIOPS does not *enqueue* controlled messages once they reach controlled memory, because all right siblings and all descendants of controlled alpha memory nodes are controlled nodes. All controlled matching completes before controlled conflict resolution proceeds; controlled matching need not consider individual node priorities. As a result, controlled nodes pass messages by direct procedure call to siblings and children. Controlled matching uses the procedure call approach of standard Rete to avoid unnecessary queue overhead. Controlled matching operates only when there is no automatic matching to perform. If an incoming interrupt triggers automatic matching during procedure-call based controlled matching, the latter waits. Automatic matching resulting from a working memory change in a running controlled production will not occur until after controlled matching and conflict resolution has selected the production to run.

6.6.4 Beta Memory Handling and Token Hashing

Controlled beta node handling is simpler than alpha node handling. A beta memory node in standard Rete has one child - a join, not, or instance node (the latter follows the last join/not at the point of entrance into the conflict set). There is no "right action." A beta memory node has no siblings. The beta node stores the output of the parent join or not. While that parent node may have siblings (e.g., the left automatic join node of Figure 10), the beta memory, as a logical extension to the parent node, is without siblings. Its handler is very simple:

1. If action is add token

    Store the joined token in local memory.

  Else (delete)

    Remove the joined token from local memory.

2. Send the current token with the incoming action to the child.

Some implementations of Rete place the left and right (input) memories for a join or not node into the node. Join/not nodes in these implementations do not share memory nodes. In Figure 10, the left join node would contain two register nodes. The right join node, which in the figure shares its right (input) memory with the other join node, would contain distinct internal right and left memories. The reason for this change is that standard Rete memory nodes store their tokens in linear lists, so search time for join and not matches is O(n) on the number of tokens in the memory opposite the incoming message.* With memory nodes internalized, join (and not) nodes store tokens in a *global hash table* [31, p. 62-66]. A token's hash key is derived from the unique I.D. of the join node and, more importantly, the value of fields in the token that are tested for *equality* in the join (not) tests. It is because the hash key is unique to a join node's use of the token,

* For each distinct token-pair to be join tested, test time is usually O(1), since field tests are identical to pre-join tests, the difference being that fields of interest span multiple (but constant) memory elements. A few controlled field tests, such as lexicographical comparison, are not O(1).

that the join (not) nodes keep private memories. The effect is that, for a join that performs equality tests, search for tokens likely to satisfy the tests is fast. Speedup depends on the quality of hashing and the hashing overhead. The join node must still perform all of its tests, but it inspects only likely candidates. Hashing offers no improvement for joins that perform no equality tests, because non-equality tests are satisfied by more than one value (and therefore tokens in more than one hash bucket). Join/not hashing uses a global hash table, because many local join memories may contain few elements.

PRIOPS controlled memories presently use the traditional linear lists. This implementation is due to constraints on program development time - controlled PRIOPS memory could undoubtedly benefit from hash-based retrieval. Note that automatic registers would not benefit from hashing, because automatic registers store at most one token. No search for candidate tokens takes place.

6.6.5 Critical Sections for Memory Access

Due to the pseudo-concurrency caused by allowing interruption and resumption of matching, conflict resolution, and right hand side execution, there is potential for critical section problems with memory nodes. In his multi-processor Rete architecture, Gupta *locks* memory nodes during access so that concurrent node activations cannot interfere with each other while reading and modifying memory [31, p. 69].

Concurrent access of memory nodes turns out to be less of a problem with PRIOPS. The reason is that each node (including memory and registers) has a priority, that only processes with the same priority will access a node, and only automatic processes with higher priorities will interrupt a process accessing a node. These interrupting processes will not access the target node of the interrupted process.* There *is* potential for critical section problems elsewhere in PRIOPS. All processes that work with global ready lists, garbage lists, working memory linkage, the scheduling queue and the conflict resolution queue - essentially all global run-time data - must be careful to avoid the problem. Most involved data structures are stacks and queues, so the solution in the current uniprocessor PRIOPS is to briefly disable interrupts inside the functions that add elements to and remove elements from these stacks and queues. The intervals during which interrupts are disabled are, of course, constant-time-bound. Operations at the ends of affected queues and stacks are O(1). Empirical research has shown that simple locks are preferable to more complex synchronization schemes for multi-processor Rete systems [32]. The majority of multi-processing contentions occur during very brief, occasional intervals, over task queues and multiple access to join/not node memories. PRIOPS simple access-based interrupt disabling scheme is likewise

* The interrupting processes *may* examine the node's priority field to enqueue messages to it, but the node's priority does not change during execution.

sufficient for uniprocessor PRIOPS.

6.7 The Controlled Join Node

6.7.1 Join Node Handling

In discussing Figure 10 we have seen that a join node contains a right sibling link field. It also contains backward pointers, allowing it to examine input memory opposite an incoming token. Join tests, like pre-join tests, are in the form of compiled code. The handler for the controlled join type follows:

1. If action is radd token (add from right input)

    1.1 Determine memory type of left input (alpha or beta).

    1.2 For each token in left input memory

        1.2.1 If the right input token (memory element) is NOT part of the left token (deja vu test)

            Pass both tokens to the compiled join test code.

            If tests succeed

                Construct a composite token.

                Send the composite token with an add action to the child beta memory.

Else if action is add token (add from left input)

    For each token in right input memory

        Pass both tokens to the compiled join test code.

If tests succeed

Construct a composite token.

Send the composite token with an
add action to the child beta memory.

Else (delete or rdelete)

For each joined token in the child memory node of this join

If the incoming token is a component of the joined token

Send the composite token with a delete action
to the child beta memory.

2. Send the incoming token and action to the sibling node.

Again controlled child and sibling messages propagate using procedure calls. Procedure calls assure that descendant nodes receiving a controlled delete action will finish using the propagated join token before the join's child beta memory node recovers storage for the joined token.

The delete actions do not perform pattern matching. Instead they compare an incoming left token against leading components of previously joined tokens, and an incoming right token against trailing components of previously joined tokens. Original OPS5 Rete performed redundant pattern matching for delete actions, while PRIOPS and several other Rete variants [10,80] employ faster delete tests using token pointer comparisons.

6.7.2 Working Memory Self-join Testing

Use of priorities to schedule join node activations forces PRIOPS to handle join testing somewhat differently from standard Rete. The join node needs to test for cases where a single working memory change contributes messages to both its left and right inputs. Line 1.2.1 of the above pseudo-code has a special test I call *deja vu* (after the fact that the token arriving at the right input "has been here before" in the left input). Line 1.2.1 restricts a right-input token - which is always a reference to a single working memory element - from joining any left token that contains a reference to the right token's working memory element. Figure 12 will help explain the problem. The left side of Figure 12 shows the Rete net for production *thing2.* The production simply tests for the presence of a *thing* structure, without regards to values for the *thing's* fields. Thing2 contains two condition elements that perform this test.

Below the production text in Figure 12 is the net, consisting of an alpha (two-output) memory, a join for the two condition elements, and a beta memory to receive the output of the join. This beta memory leads to the conflict set. Suppose the action *(make thing)* occurs during program execution. Assume that this is the only element in working memory. The alpha memory first stores the token, then advances it to *join A* (the left child). Join A joins the left token to all right tokens (there are no join test restrictions), advancing thing-thing to the beta memory, and instantiating thing2. Control returns back to the alpha memory, which then

advances the thing token to its *right child*, in this case join A. This new invocation of join A joins the right token to all left tokens, advancing *another* thing-thing token to beta memory, and *another* thing2 instantiation proceeds to the conflict set. This situation is an error. Rete should not create two instantiations with *identical* working memory elements bound to the same condition elements for a single production.*

Forgy's solution in the initial Rete document [25, p. 83] is for the alpha memory node to send the token to its left child *before* storing the token. When procedure call machinery returns control to the alpha node, the node *then* stores the token, and finally sends the token to its right descendents. As a result, join A in Figure 12 will form thing-thing only when it receives the right message. The left incoming message will not find thing in the right memory.

Even in standard uniprocessor Rete, Forgy's solution is not complete as stated. Examine the right side of Figure 12. Rule *thing3* joins thing three times. When *(make thing)* advances the token to alpha memory, alpha memory sends the add message to its left child without first storing the token. Join B fails (thing is not yet in its right memory), and control returns to alpha memory. Alpha memory

---

* Here *identical* means *with identical time stamps.* Two working memory elements may have identical field values and still be distinguishable by virtue of their *recency*. Each distinct memory element has a distinct time stamp.

stores the token and send messages to its right descendents. Suppose the message goes first to join B. Join B creates thing-thing, the first beta memory saves thing-thing, join C joins thing-thing to thing in its right memory (thing is now stored there), the bottom beta memory stores thing-thing-thing, and *thing3* enters the conflict set. Control now sends the thing token into the right input of join C. Join C constructs another thing-thing-thing when triggered by this right input, and a redundant instantiation enters the conflict set. Forgy's solution for standard Rete apparently assumes that right output messages from an alpha memory node go first to join nodes *joining the largest number of tokens*, and work backward to join nodes joining the smallest number of tokens. The absence of this requirement constitutes a bug in standard Rete documentation.

Even this solution does not work for PRIOPS. Presently PRIOPS controlled nodes ignore priorities at matching time in their use of procedure calls. But suppose the networks of Figure 12 are in the automatic partition. Because queue-based control never *returns* to the activation of the alpha memory node, it cannot store the token after calling join B through its left input. Suppose *thing2* and *thing3* are *both* compiled, thing2 with a priority of 2, thing3 with a priority of 1. Compilation yields a Rete net identical to the right side of Figure 12, with nodes from the alpha memory through the first beta memory having priority 2, and with thing2's instantiation emanating from this beta memory. In such a case join B would run before join C, again resulting in redundant instantiation of thing3.

Line 1.2.1 (the *deja vu* test) of the join handler pseudo-code gives the solution for both controlled and automatic join nodes. Self-join occurs only for incoming left tokens. Right messages can avoid self-joins, since by the time a joined token containing the right token reference arrives at the join's left input, the token reference will already have been stored in the join's right-input memory. This ordering of arrival would not be so deterministic in a balanced Rete network, giving another reason to avoid it in PRIOPS. I assume that multi-processor Rete implementations use a comparable solution. Order of message arrival in these systems is nondeterministic - variations in inter-processor communication speeds may even cause delete actions to precede their counterpart add actions [31, section 5.2.3]. Nonetheless, I have not seen the problem discussed.

6.8 The Controlled Not Node

Not nodes share many properties with join nodes. They are two-input nodes with child beta memories. A not field records each node's token size. Unlike the join, the not does not create new tokens. Instead it *passes* incoming left tokens when they are not *inhibited* by matching right tokens. To accomplish this task, the join node keeps an internal left memory. Each entry in a not left memory contains two fields: a link back to a corresponding token in the left-input memory, and an *inhibition count* for that token. Not's handler code shows that it *adds* left tokens to its descendents whenever the inhibition count reaches zero, and *deletes* tokens from

its descendents whenever the inhibition count goes from zero to non-zero:

1. If action is radd token (add from right input)

    For each token in left token memory

        If the right input token (memory element) is
        NOT part of the left token (deja vu test)

            Pass both tokens to the compiled not test code.

            If tests succeed

                Increment the left token's inhibitor count.

                If the left token's inhibitor count equals 1

                    Send the child memory a delete message
                    for the left token.

Else if action is add token (add from left input)

    Add the token to the local left memory with an inhibitor
    count of zero.

    For each token in right input memory

        Pass both tokens to the compiled not test code.

        If tests succeed

            Increment the new left token's inhibitor count.

    If the left token's inhibitor count equals 0

        Send the child memory an add message for the left token.

Else if action is rdelete (delete an inhibitor from the right)

    For each token in left token memory

Pass both tokens to the compiled not test code.

If tests succeed

Decrement the left token's inhibitor count.

If the left token's inhibitor count equals 0

Send the child memory an add message for the left token.

Else (a delete of a left input token)

Find the left token in the local left memory.

If the left token's inhibitor count equals 0

Send the child memory a delete message for the left token.

Recover storage from the local left memory.

Send the current token and action to the sibling (in all cases).

The deja vu test appears again for the avoidance of redundant self-matches. Not's rdelete action is the only non-pre-join delete action to require matching. This is because the not node does not build joined tokens that include a reference to matching right tokens - not does not *join* tokens, but uses right tokens to inhibit left token propagation. Not does not keep individual pairings of left tokens and inhibitors, but rather keeps an inhibitor count (the number of inhibiting right tokens) for each left token. Memory could be traded for eliminated rdelete matching by storing a list of inhibitors with each inhibition count, but sequential search of this inhibitor list could itself become expensive.

6.9 The Controlled Instance Node

The controlled instance node is the final controlled node type. This node manages additions and deletions of production instantiations to and from the conflict set. An instance node is always the child of a memory node that stores the tokens representing complete matches to the condition elements of this node's production. The compiler, after generating the memory node for the final condition element of a production, generates the production's instance node.

In addition to core Rete fields, the instance node keeps a link to additional information about its production (e.g., right hand side executable code and right hand side variable storage area). The controlled instance node also maintains instantiations it has added to the conflict set on a private list. It searches this list and deletes an instantiation in response to a delete action.

```
If action is add token

        Build an instantiation structure, filling it in with
        token and production right hand side information.

        Add the instantiation to the instance node's list
        of private instantiations.

        Using insertion sort, add the instantiation to the
        conflict set for the rule's priority.

Else (delete)

        Find the instantiation with the departing token in the
        instance node's private list.
```

Remove the instantiation from the conflict set,
recover storage.

Send the current token and action to the sibling.

The present implementation of PRIOPS uses insertion sort to keep the conflict set in sorted order, avoiding sorting at conflict resolution time. An alternative would be to perform an O(n log n) sort such as quicksort at conflict resolution time. I used insertion sort because, in a properly designed production system program - one that does not generate large conflict sets, which usually result from unnecessary join successes that are later retracted - the conflict set is typically small. This is especially true for PRIOPS, since the conflict set is spread across 256 individual conflict set queues. Using quicksort on small conflict sets would not always be worth the extra overhead of re-sorting the individual-priority conflict set each time conflict resolution selects that set. Also, each dynamically varying conflict set is maintained as a doubly linked list to allow varying growth, and for quick deletion. Insertion sort is more straightforward than quicksort or heapsort for such a list. Finally, conflict set insertion time is only a small component of the match time along a successful path through the Rete net to the conflict set. Only the tokens that have passed all pre-join and join tests, and failed all not tests, make their way to the conflict set. Test loops performed before reaching a controlled instance node dominate the time complexity of its complete path.

I will consider automatic conflict set sorting in the section on automatic

instance nodes.

Data flows from an instance node into the conflict set, so a controlled instance node has no Rete node descendants. Because an instance node is a child of a memory node, it can have join and not node siblings. If another production has condition elements identical to an instance node's production, then that other production will contribute a sibling instance node.

6.10 Automatic Register Node Types

The pre-join node type serves both controlled and automatic matching functions. Each controlled-specific node type has an automatic counterpart. Much of the preceding discussion explains inter-node relationships for both partitions.

Like controlled memory nodes, the automatic partition uses two storage node types: a dual-output alpha register type, and a single-output beta register type. The former terminates automatic pre-join chains, while the latter follows automatic join and not node types. Both register node types have the capacity to store only one token. This limited capacity is one of the key features of the automatic partition, one that determines the form of all automatic pseudo-code that follows.* An important characteristic of all of the upcoming automatic node handlers is that *they*

---

* The other key feature of PRIOPS Rete, *deferred matching*, affects both partitions. Only automatic activity defers lower priority (controlled or automatic) activity.

do not iterate. Previously examined controlled join and not nodes were seen to iterate over opposing memories for incoming messages. With the restriction of unit size registers, automatic handler code does not exhibit this iteration. It is simple straight-line code. The O(1) requirement for automatic code suggests this straight-line form, and each section of pseudo-code exhibits it.

6.10.1 The Automatic Alpha Register Node

In addition to core Rete fields, an automatic alpha register node contains space for storage of the current token. The current token for an alpha node is a pointer to a single working memory element. This internal storage contrasts with controlled memory nodes, which must maintain dynamically varying lists of stored tokens. The compiler statically allocates automatic register nodes with internal token storage.

Each automatic alpha node contains an *occupied* field that shows whether the token field holds a valid pointer. Like the controlled alpha node, the automatic alpha has right child linkage for join and not nodes requiring its output in their right inputs. Here is the node handler:

```
Set temporary boolean "advance" to true.
If action is add token
    Store the joined token in the node's local storage.
    Set node's "occupied" flag to true.
```

Set the "right action" to "radd."

Else (delete)

If node is occupied and incoming token to delete
is the same as the token currently in the node

Set the node's "occupied" flag to false.

Else

Set boolean "advance" to false.

Set the "right action" to "rdelete."

If "advance" is true

Enqueue the incoming token and action for the left child.

Enqueue the incoming token and "right action" for the right child.

All actions are basic O(1) field read and write operations (static token size is known at compile time), and no iteration exists in the handler, so this handler is O(1). The handler logic reflects the fact that match message scheduling will abort outdated messages at the head of an automatic chain when a newer message arrives. An add message simply reuses the register. The handler stores the token without consulting the *occupied* field, overwriting any older token that may inhabit the node. Delete handling is more complicated because a delete action is valid only for the corresponding token formerly added to the node. If a subsequent add message has replaced the delete's token, then the delete must have no effect. In setting the temporary "advance" switch to false, the alpha node terminates

processing of the delete message.

For example, suppose working memory element *A* has satisfied all pre-join tests for an automatic condition element. *A's* token arrives at empty alpha register node *R*. *R* stores and advances *A's* add message. Later a different memory element *B* also satisfies the same pre-join test chain as *A*, so the alpha register node overwrites *A* with *B* and advances the latter token. *(B's* advancement will cause replacement of *A* in all descendants of the alpha node.) Eventually *A's* delete message arrives at the alpha node. When the handler sees that the token to delete does not match the token already in a register node, it terminates the outdated delete message.

6.10.2 Automatic Message Queuing

A final field common to all automatic node types is the *lqueue* pointer field. When not null, this pointer links the node to its current entry in the matching queue. A non-null value signifies that a message for this node is in queue. If a newer message arrives at the matching queue before this waiting message arrives at the node, the enqueuing procedure overwrites the older message. (Each *two-input* automatic node can have *two* current messages in queue, one for each input. The *rqueue* pointer for an automatic join or not node indicates the node's right input queued message.)

Overwriting a message queue entry presents a problem for automatic alpha

registers. If the newer message replaces an outdated queue message for an automatic node, then the automatic node will not be able to forward the lost message to siblings and descendants. This course of action is appropriate for automatic descendants and siblings, for whom the lost message is also outdated. But an automatic alpha register node may have pre-join siblings at lower priority, where the pre-join chain leading to the alpha register is a prefix of the pre-join chains continuing at these siblings. We have seen that all pre-join nodes must allow multiple messages to remain queued. An alpha register may also have a controlled alpha memory sibling, for a case where automatic and controlled productions share identical pre-join components of condition elements. Messages that are outdated for an alpha register are *not* outdated for its pre-join and alpha memory siblings, so the enqueuing procedure must give the alpha register special treatment. Here is the code for the enqueuing procedure *putpq:*

```
If the target Rete node pointer is null
        Return (occurs for messages to empty descendant or sibling chains)
Disable interrupts.
Set "newq" local queue entry pointer to null.
Set local "newpriority" to the priority of the target Rete node.
If the target is automatic and is not a pre-join node
        If action is radd or rdelete (message to a right input)
```

Set "newq" queue entry pointer to the target's "rqueue" value.

Else

Set "newq" queue entry pointer to the target's "lqueue" value.

If "newq" is null (no existing automatic entry)

If we are currently running in the automatic partition
or if the target node is in the automatic partition

Get storage for a queue entry from the queue entry
ready list, if none is available then issue run-time
error & abort inference.

Else (controlled partition)

Get storage for a queue entry from dynamic memory manager
(interrupts are enabled during this non-O(1) activity).

Put this new queue entry at the tail of the priority queue.

Else ("newq" is not null, meaning putpq is overwriting an earlier
automatic message in the queue)

If the target is an alpha register and the message is delete
and the older queued action is add and the current message's
token is not the same as the older message's token (i.e.,
a subsequent add has made this delete superfluous, as in the
alpha register code)

Recursively enqueue the sibling of the alpha register.

Enable interrupts & return.

(At this point the queue entry is enqueued, but its fields are empty).

Set the target, action, and token fields to putpq's input arguments.

If the queue entry's target is a non-pre-join automatic node

If action is radd or rdelete (message to a right input)

Set the target's "rqueue" field to point to its unique queue entry.

Else

Set the target's "lqueue" field to point to its unique queue entry.

If the target's priority is greater than current priority

Set a flag signaling increase in priority.

If the target is an alpha register

Recursively enqueue the sibling of the alpha register.

Enable interrupts & return.

Putpq is O(1) for calls during automatic processing. During controlled processing it will call non-O(1) storage allocation procedures. The listing shows interrupt disabling during a critical section (access of a global message queue), use of a *ready list* for obtaining pre-allocated storage during automatic processing, and use of the automatic node "lqueue" and "rqueue" fields. Explicit handling of messages to the automatic alpha register appears twice. The previous section discussed how the register itself deals with receiving an outdated *delete A* message after receiving a displacing *add B. Putpq* must not allow *delete A* to replace *add B* in the register's unique queue entry, so *putpq* duplicates the delete test seen in the alpha register node handler. The alpha register is the only node type that must consider the possibility of a replacing add preceding a delete of an automatic

node's old contents. The automatic register is the ancestor of all other automatic node types in the Rete network. By filtering outdated delete actions at this node type, PRIOPS ensures that outdated delete actions cannot reach other node types.

Putpq also shows the recursive call for the alpha register's first sibling. The recursive call enables the alpha register's sibling chain to receive messages overwritten for the register node. This call occurs at most once for an alpha register node message (an alpha register is never the sibling of another alpha register). Because this is a tail recursive call, it is implemented as a simple jump to the start of *putpq* with new procedure arguments, avoiding recursive call overhead.

Getpq, the procedure for dequeuing an entry from the head of a match message queue, is considerably simpler:

```
Disable interrupts.
If there is an entry at the head of the queue
    Take the front entry out of the queue.
    Pass the action, Rete target node, and token pointer to
    the calling procedure via reference parameters.
    If the target is a non-pre-join automatic node
        If action is radd or rdelete (message to a right input)
            Set the target's "rqueue" pointer to null.
        Else
```

Set the target's "lqueue" pointer to null.

If we are currently running in the automatic partition
or the queue entry ready list needs more ready structures

Put the newly freed queue structure on the ready list
(the ready list is a LIFO, O(1) singly linked list).

Enable interrupts.

Else

Enable interrupts.

Return the free storage to (non-O(1)) memory management.

Else

Report illegal read of an empty queue and abort.

O(1) concerns extend to the smallest detail of PRIOPS internals. Library procedures and functions supplied with the PRIOPS implementation compiler (a C compiler for the present PRIOPS) are not usable for automatic partition activities, because no assumptions about their time complexity are possible. Most library operations, such as input-output and memory management, will not be O(1).

6.10.3 The Automatic Beta Register Node

The automatic beta register type is the simplest node type. It need not manage the varying length storage of the controlled beta memory, since the register can hold at most one token:

If action is add token

Store the joined token in local register.

Else (delete)

Remove the joined token from local register.

Enqueue the incoming token and action for the child.

All actions are O(1) (static token size is known at compile time), and no iteration exists in the handler, so this handler is O(1). This node type contains a field to record token size (a token is an array of pointers to working memory elements), storage for a single token, and an *occupied* flag.

6.11 The Automatic Join Node

Like its controlled partition counterpart, the automatic join node contains a right sibling link and back pointers to its two parent nodes. It uses executable code to perform tests.

If action is add (from left input) or radd (from right input)

If action is add

Sibling is left sibling.

Left token is incoming token.

Consult right input register for right token.

Else (radd)

Sibling is right sibling.

Right token is incoming token.

Consult left input register for left token.

If input registers are occupied and (action is add or
the right token is not in the left token - deja vu test)

Pass both tokens to the compiled join test code.

If tests succeed

Construct the composite token inside the child
beta register node.

Enqueue the add action with the composite token
for the child beta register node.

Else if the child beta register is occupied
(the join has failed, any old join is outdated)

Enqueue a delete action with the old stored token
for the child beta register node.

Else (action is delete (from left) or rdelete (from right)

If action is delete

Sibling is left sibling.

Else

Sibling is right sibling.

If the child beta register is occupied

Enqueue a delete action for the child beta register.

Enqueue incoming action for sibling.

All actions are O(1) (static token size is known at compile time), and no

iteration exists in the handler, so this handler is O(1). Whether an automatic join passes or fails, any descendant tokens become outdated, since at least one of the working memory elements contributing to them has become outdated at its alpha register. Consequently a failed add propagates a delete.

With the join node storing its joined output directly in its child beta register, the action of the beta register handler reduces to propagating incoming messages to its descendants.

6.12 The Automatic Not Node

The automatic not node contains a right sibling link for right input, a right parent link, storage for a single *left token*, a record of its token size, and an *inhibited* flag for its current left token. Whereas the controlled not node maintains a list of left tokens and inhibitor counts, the automatic not retains only the most recent left token. Therefore it avoids the variable length token list and inhibitor counts.

```
If action is add (from left input) or radd (from right input)
    If action is add
        Sibling is left sibling.
        Left token is incoming token.
        Store reference to token in local left register,
        set occupied flag to true.
```

Consult right input register for right token.

Else (radd)

Sibling is right sibling.

Right token is incoming token.

Consult left input register for left token.

If input registers are occupied and (action is add or
the right token is not in the left token - deja vu test)

Pass both tokens to the compiled not test code.

If tests succeed

Set local inhibited flag to true.

If child beta register is occupied

Enqueue delete message for child beta register.

Else if the incoming action is add (the new left token
is not inhibited) OR
the old left token is occupied and inhibited

Set local inhibited flag to false.

Copy the left token to the beta register.

Enqueue an add action with the left token
for the child beta register node.

Else if action is delete (left token)

Sibling is left sibling.

Set local occupied & inhibited flags to false.

If child beta register is occupied

Enqueue delete message for child beta register.

Else (action is rdelete - delete the inhibitor)

Sibling is right sibling.

If not node left token is occupied and inhibited

Copy left token to the beta register.

Enqueue an add action with the left token
for the child beta register node.

Set local inhibited flag to false.

Enqueue incoming action for sibling.

All actions are O(1) (static token size is known at compile time), and no iteration exists in the handler, so this handler is O(1).

6.13 The Automatic Instance Node

The automatic instance node is the final node type. Because each automatic production allows at most one instantiation, the one triggered by the most recent working memory elements, the automatic instance stores all instantiation fields statically and locally. Like the controlled instance node it has a link to production right hand side code and variable storage. It also maintains a *live* flag that signals whether it presently holds an instantiation.

If the "live" flag is true (there is an outdated instantiation)

Remove the old instantiation from the conflict set

and set "live" to false.

If action is add

Set "live" flag to true.

Copy token information into the locally stored instantiation (compiler copies in invariant production information).

Sort token time stamps for conflict resolution, link instantiation into conflict set.

Else

Clean up outdated instantiation information.

Enqueue incoming action for sibling.

An incoming add replaces any old instantiation, while an incoming delete removes it. Hence both cases remove any live instantiation. Much of the information for any single instantiation is known at compile time, so only token specific information is copied into the instantiation at run-time.

Again insertion sort links the instantiation into the conflict set doubly linked list. Because the maximum conflict set size for an automatic priority is constrained by the constant number of automatic productions of that priority, there is a constant worst-case bound on insertion time. Insertion into the conflict set is therefore O(1). Deletion from the conflict set requires adjusting two links as for controlled instantiations, again an O(1) operation. All automatic Rete matching is O(1), from pre-join testing to automatic instantiation and including queue operations. Therefore any automatic partition Rete path is O(1). The only way to

achieve time complexity greater than O(1) is through cyclic composition of automatic productions. The next chapter will discuss future work on detection of non-O(1), cyclic composition of automatic productions.

6.14 Ready Lists and Garbage Collection

The maze example showed how a priority 0 production could explicitly reclaim outdated working memory elements - garbage. This collection does not penalize automatic processing. It takes place upon resumption of controlled processing, before lower-priority controlled activities have an opportunity to generate additional garbage.

Internally PRIOPS uses *ready lists* to supply initialized working memory elements to automatic productions, and *garbage lists* to collect working memory elements discarded by automatic processing. Ready lists of initialized working memory elements are available for automatic productions that perform *make* operations. There is a distinct ready list for each working memory type made in the automatic partition. The compiler also determines the number of auxiliary object structures, such as matching queue structures seen in the *putpq* and *getpq* listings. All ready lists are stacks with O(1) push and pop operations. During automatic execution productions remove initialized elements from ready lists and enqueue messages for Rete in O(1) time. When automatic execution completes, the controlled matcher replenishes ready lists to prepare for the next burst of automatic

activity.

Right hand side *remove* operations cannot recover working memory elements immediately, because the matching for remove operations must wait in queues. At remove initiation time, a reference to a removed element enters a *garbage list* for the element type. After all matching is complete, the controlled matcher can recover garbage storage. Garbage lists are stacks with O(1) push and pop operations. Garbage collection and ready list maintenance work in concert to reuse structures with minimum overhead. For example, rather than return garbage working memory elements to global memory management, only to have ready list maintenance request new element storage from memory management, garbage collection reclaims removed working memory structures, reinitializes them and places them on ready lists without calling global memory management. *Getpq's* listing showed an example of this, recovering queue entry structures into a ready list.

Replenishing ready lists places an average real-time requirement on controlled storage management; each ready list must contain sufficient elements to satisfy the most demanding burst of automatic activity. The compiler can determine the correct size of some ready lists, those constrained by the number of automatic nodes. The PRIOPS programmer must presently estimate the number of initialized working memory elements to keep on element type ready lists. Dynamic adjustment of ready list sizes is an area for future work.

The controlled and automatic inference drivers in the next two sections show the location of ready list maintenance and garbage collection calls.

6.15 Inferences Drivers

A Rete-based inference driver is a routine that oversees the match-fire-execute inference cycle. Matching occurs in the Rete network. The inference driver selects a production instantiation to execute from the conflict set, and starts execution by calling the production's RHS executable code. RHS working memory changes trigger new matching, and hence new inference.

PRIOPS has two inference drivers because of differences between controlled and automatic inference. Controlled inference completes all matching before conflict resolution, never interrupts other processing, and performs storage reclamation. Automatic inference performs highest-priority conflict resolution and RHS execution immediately after highest-priority matching. It interrupts lower priority activities, and does not collect garbage. The next two subsections examine these distinctive inference drivers.

6.15.1 The Controlled Inference Driver

The three major components of the controlled partition inference engine are the controlled Rete network, right hand side executable code for controlled rules, and the controlled inference driver. Conflict resolution work appears in both the

controlled instance node handler and in the inference driver.

The controlled inference driver scans controlled matching queues, dispatching messages to Rete nodes. When all controlled matching completes, the driver selects an instantiation for execution.

```
While in PRIOPS "run" mode

    If deferred exception handling for interrupt handler is posted

        Report exception by type and abort inference.

    If ready list "pump" flag is set (set by automatic inference
    driver that was entered from an interrupt handler)

        Pump up ready lists.

    While automatic activity is pending (only satisfied when
    user's initiation of PRIOPS triggers automatic activity)

        Call the automatic inference driver.

    For the topmost to the bottommost controlled priority

        Current global priority is controlled priority.

        While controlled priority queue has pending matching

            Dequeue match message, send it to target node.

            Note that matching occurred in this "for" loop.

    If matching occurred in above "for" loop

        Restart controlled inference driver.

    Garbage collect all removed working memory elements
    whose delete matching in both partitions is now complete.
```

```
For the topmost to the bottommost controlled priority
    If controlled priority conflict set has instantiation
        Current global priority is instantiation's priority.
        Disable interrupts.
        Remove instantiation from conflict set.
        Enable interrupts.
        Pass instantiation token and right hand side variable
        space pointer to right hand side executable code.
        Free dynamic storage of controlled instantiation.
        If right hand side requests "exit" or user specific
        "run count" is exhausted
            Report end of run and return to user.
        Restart controlled inference driver.
    Return to user when there is no pending activity.
```

The basic structure of controlled inference consists of one loop to find and execute matches, and a second loop to execute a right hand side. The match loop repeats until it runs a full cycle with no matches. Descending once from top to bottom is not enough, because interrupt-triggered automatic activity may enqueue controlled matching after the above matching loop has passed the appropriate controlled priority. In fact, it is always possible for an automatic interrupt handler to enqueue controlled activity after completion of the controlled matching loop. Such controlled activity defers until the next pass through the controlled inference

driver.

Removed working memory elements are marked as *safe* only after all of their matching is finished, which is to say after controlled matching completes.* Procedures to *pump* up the ready lists do so only in the controlled partition, and recover garbage lists (rather than calling global memory management's *malloc)* where possible.** The pump code runs upon PRIOPS "run" invocation, upon deferred request from the automatic partition (seen above), and upon return from the automatic to controlled partition (next section).

One reason for completing all controlled matching before controlled conflict resolution is garbage collection. Indefinite postponement of low priority controlled matching could cause indefinite delay in collection of removed memory elements, possibly resulting in dynamic storage exhaustion.

6.15.2 The Automatic Inference Driver

The automatic inference driver can be entered in four ways. The user can

---

* Actually a *location* on each LIFO garbage list is marked as *safe.* Working memory elements beyond this location are available for reclamation.

** The word *pump* comes from the analogy of a ready list as a *cistern* with high and low *water marks.* Pumping fills the cistern. The pump does not put water into the cistern until the water level reaches the low mark; it then pumps to the high mark. Making the two marks identical would cause unnecessary overhead - each small request for water (a ready item) from the cistern (ready list) would incur pumping overhead. Separate water marks introduce *hysteresis* into the pumping process, incurring pumping overhead only at occasional intervals.

make working memory changes that enqueue automatic activity before running PRIOPS. The right hand side of a controlled production can make an automatic element change, resulting in an immediate call to this driver. The right hand side of an automatic production can make a higher priority automatic element change, resulting in a recursive call to this driver. Finally, an interrupt handler can make a higher priority automatic element change, calling this handler asynchronously from either partition.

Each invocation of the automatic driver iterates over automatic matching queue and conflict set priorities ranging from the increased priority that triggered the driver call, to the priority of the calling activity. This driver never acts below priority 1, the lowest automatic priority.

```
If deferred exception handling for interrupt handler is posted
    If currently in an interrupt handler
        Return.
    Else
        Report exception by type and abort inference.
For the triggering automatic priority down to the caller's priority
(adjust global priority, but never down to a controlled priority)
    While automatic priority queue has pending matching
        Current global priority is matching priority.
        Dequeue match message, send it to target node.
```

If automatic priority conflict set has instantiation

Current global priority is instantiation’s priority.

Disable interrupts.

Remove instantiation from conflict set.

Enable interrupts.

Pass instantiation token and right hand side variable space pointer to right hand side executable code.

If right hand side requests "exit" or user specific "run count" is exhausted

Restore global priority to caller’s priority.

If currently in an interrupt handler

Post deferred halt and return.

Else

Report end of run and return to user.

Restart automatic inference driver.

Restore global priority to caller’s priority (may be controlled).

If caller was controlled partition activity

If currently in an interrupt handler

Post a ready list "pump" request flag.

Else

Pump up ready lists.

Return to caller.

The automatic driver inspects the conflict set for a priority immediately after draining that priority's matching queue. If the conflict set yields an instantiation, lower priority matching waits. After execution, matching restarts at the execution priority, because the executed right hand side (as well as interrupt handlers) may have generated working memory changes at the execution priority. Right hand side changes and interrupting changes cause recursive driver calls only when they contribute changes *higher* in priority than the current global level. Each call to the automatic driver restores the global priority to the caller's level before returning. The result is that each driver invocation iterates over a unique range of automatic priorities, avoiding critical section problems.

Upon returning to the controlled partition, a non-interrupting automatic driver calls procedures to pump the ready lists as a controlled activity. This code belongs here rather than in the controlled driver, because the controlled driver never knows when a controlled right hand side will cause a call to automatic inference. The automatic driver, on the other hand, can easily detect an immanent return to the controlled partition. An interrupting driver cannot pump the ready lists, because it may have interrupted ready list pump operations. Ready list pump operations are not reentrant; they are not O(1), and so cannot hide behind disabled interrupts. Instead an interrupting automatic driver returning to the controlled partition sets a pump request flag.

The two drivers interact to react to heightened priority activity, recover storage, and recover from exceptions at the earliest possible opportunities, without causing non-O(1) automatic processing or critical section problems. From a software engineering viewpoint, this experimental architecture promises to scale nicely to a supportable production system design tool.

6.16 The Compiler and Generated Code

I will not treat auxiliary code modules such as unique symbol maintenance or the C language interface, other than to say that non-O(1) activities are restricted to the controlled partition. The production compiler itself warrants a few words. It uses a simple recursive descent, hand-coded parser. Appendix A gives the lexical rules and LL(1) grammar for PRIOPS. The compiler's symbol table code share's the run-time code's use of unique symbol-attribute structures and operations.

Instead of generating native machine code, the compiler produces an intermediate code for run-time interpretation. This code consists of enumerated *op codes.* The run-time interpreter is a simple switch statement on the value of the current op code. The interpreter acts by fetching the next op code, advancing the op code pointer, using the op code as an index into a jump table, and jumping to the op code handler. Op code indexed jumping appears as a C switch construct. Each op code handler retrieves in-line arguments by way of the op code pointer. PRIOPS also maintains a value stack at run-time, for use in argument and return

value passing.

I used this virtual run-time machine model in order to simplify compilation. The initial PRIOPS architecture is experimental, and I did not wish to spend a lot of time focusing on machine-specific code. Operations in automatic Rete nodes and right hand sides are O(1), but they are not as fast as they would be if compiled to native machine code. Before using PRIOPS for a time-constrained industrial application, the compiler should generate native machine code. One option is to generate equivalent C code and pass the latter through an optimizing C compiler.

The present PRIOPS compiler generates O(1) code for automatic activities, but it does not calculate actual execution times. Once again, I did not wish to spend many months hand tracing the output of the C compiler and hand summing operation times for a specific processor, nor did I wish to attempt to write a processor-specific machine code analyzer. A full PRIOPS compiler should report machine times. For research purposes I am satisfied generating useful O(1) code, code whose time complexity is constant-bound at compile time. The present compiler comprises about 12,000 lines of C code.

7. RELATED WORK AND FUTURE DIRECTIONS

PRIOPS is unique in applying constant-time-constrained, priority-scheduled processing to a production system architecture. The first section of this chapter examines several non-O(1) applications of production systems to embedded systems. Subsequent sections discuss possible enhancements to the present PRIOPS compiler, and open ended research directions. Concluding remarks end the chapter.

7.1 Related Work

Designers of an expert system that assists computer operators overseeing an IBM operating system make claims of *a continuous real-time expert system* [22]. While making several enhancements to standard Rete for more efficient embedded operation, the system does not address constant-time-constrained issues. Modifications include: compilation of rule right hand sides to machine code; a *timed-make* for creating working memory elements at specified times or after specified intervals; a *remote-make* command for introducing communications from other processors into Rete; and the addition of a *communication phase* to the standard *recognize, conflict resolution* and *act* inference cycle. The communication phase operates to handle *remote-makes* just before conflict resolution. A final modification to standard OPS5 is the use of *explicit production priorities,* but only

at *conflict resolution* time. Priority use is similar to PRIOPS controlled partition priorities, augmenting standard conflict resolution strategies. This expert system is an example of a successful embedded production system.

PAMELA is another Rete-based production system for embedded applications [10]. Like PRIOPS, PAMELA avoids redundant matching during *delete* processing by making pointer comparisons between stored tokens and delete messages. PAMELA makes several modifications designed to reduce the number of Rete nodes for a program. One alteration is automatic reordering of condition element tests, by field, to allow increased node sharing among condition elements. Another change places some *non-shared* pre-join nodes *after* some of the two-input test nodes, allowing increased sharing of the latter. Reducing the number of test nodes does not guarantee faster execution. In some OPS5 programs intentional test ordering may help impose limiting constraints early in a matching chain, reducing potential join times. For such programs the PAMELA approach may actually *increase* execution time, by disabling programmer-supplied test ordering and bringing joins closer to the start of test chains. In fact, *unique-attribute* tests intended to avoid unnecessary joins in PRIOPS and recent SOAR work [93], often contribute non-shared pre-join test nodes. By moving these unshared nodes to after join nodes to increase join node sharing, PAMELA would nullify the intended purpose of such tests. In preserving programmer-specified ordering of tests and right hand side actions, the traditional OPS5 approach assumes that the

programmer makes intelligent use of knowledge of the internals of Rete processing. PRIOPS continues this tradition, making knowledge of its enhanced Rete essential to the PRIOPS programmer.

A more important PAMELA contribution from the PRIOPS point of view is the introduction of *interrupt-based modifications to working memory*. Procedural interrupt handlers may search and modify working memory. Rete matching for interrupt-initiated memory changes does not commence until the current rule right hand side (and matches associated with its memory changes) completes. A rule may explicitly allow interrupt-based memory change matching to commence after the completion of any of its right hand side actions. Such memory changes contribute to a distinct *demon conflict set.* A *demon* is a rule triggered by interrupt handling that is capable of interrupting execution of another rule. Demon conflict resolution and firing occurs after any changes to the demon conflict set, but only after completion of the current right hand side action, which may include time consuming matching. Each working memory change in PAMELA produces temporally atomic Rete matching activity; a demon can interrupt a RHS, but not individual RHS memory change matching. There is no notion equivalent to PRIOPS' deferral of low-priority match messages, nor its O(1) automatic matching. The right hand side action gives the level of granularity for interruption in demon matching. Complex Rete matching for the interrupted RHS action can cause demon matching to wait for long periods while less important matching finishes. In

PRIOPS, the Rete node gives the level of granularity for scheduling, and higher-priority matching preempts lower-priority activities immediately. PRIOPS never makes higher-priority activities wait for lower-priority ones, and it defers lower-priority components of matching for a single working memory change.

PAMELA makes synchronization difficult: An interruptible (by demons) rule that refers to memory elements bound in its LHS must test for deletion of these elements (by interrupting demons). PAMELA requires the programmer to place explicit tests for this situation in interruptible right hand sides. Such tests are prone to timing dependent errors. PRIOPS defers the actual recovery of removed working memory elements using *garbage lists,* so avoiding premature loss of needed memory elements. When an interrupting automatic production removes a memory element being used by an interrupted rule, deferred removal allows the interrupted rule to continue using the element upon resumption. PRIOPS requires no special code for the right hand side of interrupted rules.

Despite some deficiencies, PAMELA is the first production system to successfully apply a primitive two-tiered architecture to an embedded application. However, in not restricting joins or other matching activities to O(1) complexity, PAMELA violates a constraint fundamental to PRIOPS' automatic partition. Temporal atomicity of matching for each working memory change also reduces responsiveness.

There has been some use of the SOAR production system in embedded system experiments [54,69]. Reports are partial and inconclusive. Emphasis is on learning from both experience and instruction. SOAR passes sensory input through encoding productions, and motor output through decoding productions, that are loosely analogous to the *record* and *predictive* processing of Section 3.1.4 of this thesis. SOAR imposes no constant-time constraints on these or any other processes, and there is no equivalent to automatic processing and reactive data flow. All processing interacts with SOAR's goal-oriented, memory-based central cognitive component. Despite claims of responsiveness, SOAR allows interrupt-based redirection of processing only during the cognitive *decision phase* of its inference cycle, typically requiring reaction to interruption to wait across *multiple* memory change matches and production firings [69, p. 119].* This waiting results in even more delay than PAMELA's deferral of demon production matching until completion of an interrupted RHS action. A further claim that "(approximately) constant-time access to the whole of memory is responsive to the real-time requirement," [69, p. 114] is misleading. Using an unconstrained variant of

---

* SOAR's inference cycle consists of two phases: an *elaboration phase*, during which inference assembles information, and a *decision phase*, during which inference directs problem solving processing. SOAR performs no conflict resolution, allowing all instantiations to run to completion, possibly triggering other instantiations during a single phase. Each phase can thus include matching and execution for multiple instantiations, resulting in unresponsive real-time behavior for interrupt-based processing redirection.

standard Rete, best-case time to match a working memory change is constant-bound, but worst-case time varies polynomially with the size of working memory, and exponentially with the number of condition elements in a production [25,27]. Based on questions I posed to John Laird at the Applications of Artificial Intelligence VII conference, it appears that SOAR's insistence on hierarchical, goal-directed processing may impede its ability to respond to events unrelated to its current stack-based goal activations. I suspect that SOAR's monolithic goal-oriented organization, with its lack of support for autonomous reactive processing, will require extension along lines suggested by PRIOPS research before it can successfully function in a demanding embedded application. I await further reports on SOAR's embedded applications.

The above systems are OPS-based production systems that augment Rete matching to improve embedded performance. There are other current projects that do not involve OPS or Rete, but nonetheless use some form of reactive processing to respond to environmental conditions. They are systems that may be amenable to implementation using PRIOPS. To some degree they have begun to reconstruct the controlled versus automatic processing distinction.

The first is *Phoenix*, a software system for planning and directing reactions to forest fires [20]. Current Phoenix reacts to simulated fires in Yellowstone National Park. The focus of research is the system's *cognitive component*, which is responsible for planning and directing reactions to fires. In addition to this

component, Phoenix contains a *reflexive component* that reacts to immediate environmental difficulties. This component attracts attention to sensed fires, and avoids dangerous actions such as driving a bulldozer into a nearby blaze. Both components read sensors and drive effectors. The reflexive component uses a subset of the capabilities of PRIOPS automatic partition. It has very loose coupling to the cognitive component, setting simple flags to inform the latter that the reflexive component has acted. Cohen, et. al. [20] emphasize organization of planning in the cognitive component.

Brooks' robot control system uses multiple levels of control to guide a mobile robot through a cluttered office environment [14]. Bottom levels are reactive, and could be implemented as automatic productions. They have simple responsibilities based on the immediate environment, such as avoiding running into obstacles. Higher levels subsume the responsibilities of lower levels, hence the name *subsumption architecture.* If a higher level of control breaks down due to hardware or software failure, its inhibition of lower, reactive levels likewise fails. These levels are then free to react to immediate situations, providing graceful degradation of capabilities when higher levels fail.

Another system that is thoroughly reactive is Pengi [2]. It simulates some aspects of reactive human processing in an environment by playing the Pengo video game. All processing is lookup and execution of reactive functions, based strictly on the environmental situation. The system retains no memory of episodes,

and does not support variable binding. All reactions are hand coded.

The above three programs use operations equivalent to some subset of PRIOPS controlled and automatic processing capabilities. None addresses issues of O(1) implementation or priority-based match deferral. I believe that PRIOPS provides a solid base for the construction of numerous types of embedded reactive system architectures such as these.

7.2 Enhancements to Current PRIOPS

7.2.1 Machine Code Generation

An essential improvement to the current compiler for use in a realistic embedded application is the replacement of interpreted intermediate code generation with native machine code generation. One possibility geared toward portability is the generation of C language source as an intermediate language for compilation. This change would result in measurable speed improvements, but would not change the fundamental issues of run-time complexity addressed in this thesis.

7.2.2 Run-time Learning

An enhancement somewhat at odds with the notion of compiled code is support for run-time incremental learning in PRIOPS. Though present in the PRIOPS grammar, current PRIOPS does not support run-time equivalents to OPS5's *build*

and *excise* commands for acquiring and discarding productions. Learning for the maze program in the last chapter worked by saving learned productions in an external file, and later recompiling the complete program and the entire Rete network.

A design difficulty unique to PRIOPS is the problem of modifying the automatic network on the fly, as the result of run-time learning, while high priority run-time messages are traveling through the very net to be modified. Network restructuring due to changes in priorities for existing nodes could easily lead to synchronization problems with time-constrained matching. Other Rete-based learning production systems do not have this problem, because all pending matching completes before any right hand side learning action changes the Rete network. In addition, while learning for other Rete systems may lead to additional nodes at the ends of descendant and sibling chains, learning does not modify existing inter-node relationships. PRIOPS' priority-based ordering (and learning-initiated reordering) of sibling Rete nodes is unique.

The most promising prospect is to build distinct, non-shared Rete chains for learned productions at learning time. While such chains do not reap the time and space benefits of node sharing, they do provide short term executable code without learning synchronization overhead. Learning would support unshared, O(1) automatic chains. Subsequent executions could compile the complete program, including learned productions, into the shared Rete net (as in the maze example).

A conflict between compiled machine code and run-time learning arises because machine code compilation is typically a multi-pass operation involving object files. For example, having the PRIOPS compiler generate C code for compilation, provides no mechanism for incremental production compilation at run time. One alternative is to continue to generate interpreted intermediate code for learned productions at learning time, with the performance deficits of such code. A different alternative is the learning-time generation of C code, followed by immediate compilation (through separate execution of the C compiler) and run-time loading of the compiled object file. The tradeoff is between compilation speed (learning speed) and execution speed. Incremental loading of object modules is not widely supported at present.

7.3 Future Research Directions

PRIOPS in its present form supports a base for embedded production system programming. PRIOPS programmers may use this base directly. Research directions for PRIOPS emphasize software capabilities built atop this base. Designing PRIOPS production systems with such tools would be a combination of direct production programming and higher level specification.

7.3.1 Rete Data Flow Analysis

The only way to achieve time indeterminacy in the automatic partition is

through cyclic composition of productions. Cyclic composition occurs when one production makes working memory elements that enable (or removes memory elements that disable) a production that, directly or indirectly, so enables the first production. A production can cyclically enable itself.

A production must be allowed to remove memory elements that trigger it, in order to remove stimuli that it has successfully handled; maze production *panic-up-left* (Listing 7) is an example of an automatic production that removes a triggering *sensors* element. An automatic production can also safely assert self-inhibitors. Both types of actions serve to disable self-triggering.

A rough first approximation of cyclic composition detection would look for cycles without regard to field tests. Any right hand side *make* of a memory element matched by a non-negated condition element, or any *remove* matched by a negated condition element, would offer a candidate path for search for a cycle in the Rete / right hand side network. This simple approximation is too general. For example, automatic production *automove* (also Listing 7) eliminates its triggering *move* memory element stimulus, not by removing the memory element, but by *modifying* one of its fields (in *automove* by setting *direction* to *nil).* Modify is equivalent to *remove, followed by add.* Enabling cycle detection must therefore consider individual field tests.

Current PRIOPS does not provide facilities for traversing the Rete net in search of cyclic composition. The Rete network does contain most of the declarative

information needed for such search. PRIOPS presently maintains only executable code for right hand side operations, but the compiler could retain details of right hand side memory change actions. Cycle search would require compile-time propagation of field constraints through the Rete network, eliminating candidate paths when field constraints are found to be mutually exclusive. Field tests not related to compile-time constants (i.e., comparisons of fields not tested or set with compile-time constants) would not contribute to compile-time elimination of candidate paths. Design of *efficient* algorithms for searching Rete in this way is an area for future work.

7.3.2 Planning and Learning

Temporal logic may provide a basis for time-constrained planning [3]. Planning might occur in processes distinct from PRIOPS processes, generating PRIOPS programs as their output. Alternatively, controlled partition activities might include generation and refinement of plans during interaction with the environment. Plan refinement would improve high-level plan statements, conceivably in a planning language rather than as PRIOPS productions. In both the batch and interactive scenarios, PRIOPS serves as a target language for the planner.

Given the emphasis of human automatic processing research on *practice*, it is natural to explore mechanisms for *learning* PRIOPS automatic reactions through

repetition of consistent interactions with stimulating phenomena. The explicit *chunking* technique employed in the maze program gives one approach for converting the results of controlled search into automatic reactions. Whereas the maze program used explicit learning productions, SOAR's chunking operates implicitly within the production system architecture [52,53]. Incorporation of learning machinery directly into the PRIOPS base is an area for research.

Another important consideration for learning, whether through explicit learning rules or internal apparatus, is the achievement and maintenance of correct O(1) stimulus recognition and response generation across production learning transformations such as *designation* (novel rule creation), *composition, chunking, specialization* and *generalization.* Transformation rules for generating and preserving correct O(1) stimulus-response behavior across production restructuring would be the outcome of such research. Given a learning algorithm restricted to producing correct O(1) reactions, it would be unnecessary to search for problems such as cyclic composition that may occur in hand-written code.

Explanation-based learning (EBL) is an approach for transforming a general theory of a domain into an *operational* description of some part of that domain [44]. The transformation uses one or several instances of domain situations as triggers for the operationalization of domain knowledge. Operationality is defined in terms of *usability* and *utility.* Usable means that a description is in a form accessible by the performance system, and utile means that a description is worth

using as judged by some performance criteria (e.g., time and space complexity). EBL constitutes a search from non-operational to at least sufficiently operational descriptions for a given concept, the end result being a description in a usable format.

Some learning of reactive automatic PRIOPS productions is transformation of information that is already known, but is not in a form capable of being utilized within necessary time bounds. Operationality for real-time code is correct processing within the necessary constant time constraints. Explicit programming, planning, and practice yield transformation of non-operational information contained within the system and processing environment into information that is both usable and utile at performance time. The literature for explanation based learning may provide insights readily adaptable to PRIOPS learning.

7.4 Conclusions

PRIOPS as defined in this thesis provides a solid base for designing time-constrained, embedded knowledge-based systems. The success of OPS5 and the Rete algorithm in producing useful knowledge-based systems encouraged incremental enhancement of their capabilities, while keeping the fundamental model of production system programming. Augmentations to OPS5/Rete for O(1) matching and priority-based scheduling build nicely on previous work, and fit readily into the programming notation.

So far three published papers have come out of this research. I delivered the first at the Applications of Artificial Intelligence VII conference in Orlando, Florida on March 30, 1989. It appeared in the conference proceedings [72]. That paper generated an invitation to submit a paper to the *International Journal of Expert Systems: Research and Applications.* In November, 1989 the journal accepted an expanded version of the conference paper [73]. The paper has not yet been published (March, 1990). Finally, the conference paper was one of sixteen selected from the proceedings by the 1989 Program Committee for contribution to a special issue of the *International Journal of Pattern Recognition and Artificial Intelligence* entitled "New Developments in Expert System Issues." I submitted a paper on the maze search and learning problem [74] that has just been accepted (March, 1990) for publication this upcoming summer.

Research directions indicate *controlled activities* - data flow analysis, planning, and learning - as areas for future work. The automatic partition provides a substrate upon which to build. Its structure, and the contributing theory of human automatic processing, suggest approaches to these controlled activities. I feel that this initial PRIOPS research has opened up several interesting avenues for exploration. I look forward to productive exploring.

ANNOTATED BIBLIOGRAPHY

1. Agha, Gul, *Actors - A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA: MIT Press, 1986.

Actors a is fine grain concurrent computational model out of the Message Passing Semantics Group at MIT. Actors are computational agents located at mail addresses who process information via their asynchronous communications. Arrival of messages is guaranteed, but not preservation of order of messages, a fact which presents difficulties for real-time processing. A single global time cannot be defined over a system of interacting actors, since each refers to a local clock. This contrasts with the dissertation research into embedded systems, where part of the definition of an embedded system is that a consistent system clock can be maintained within a margin of error acceptable by the application. Actor messages are identified by a tag, a target mail address, and a communication. Being driven by incoming communications, Actors are data-driven. Actors are of interest to this research because their internal behavior is non-iterative. Each Actor acts by processing its incoming communications, which involves sending messages to Actors and Actor communities which implement more primitive processing; the exception is primitive or "rock-bottom Actors," who process communications directly. An Actor can specify a "proxy" Actor to handle messages which the former cannot, providing inheritance and default message handling. Finally the

processing of any single communication handled by an Actor at a mailbox includes the specification of a replacement behavior for that mailbox. Thus the non-iterative nature of Actors is extreme: communications are targeted to mailboxes, but successive replacement behaviors for a mailbox may be distinct behaviors. Unlike dataflow machines, Actor systems thus implement history sensitivity. Unlike production systems which rely on a global working memory, the internals of an Actor are private, with communications occurring via distinct mailboxes. See references by Hewitt and Lieberman.

2. Agre, Philip E. and David Chapman, "Pengi: An Implementation of a Theory of Activity." *AAAI-87, Sixth National Conference on Artificial Intelligence*, Volume 1. Los Altos, CA: Morgan Kaufmann Publishers, Inc., 1987, p. 268-272.

Pengi is a program that plays the video game "Pengo." It is a reactive architecture that performs simple indexed reaction function lookup and execution, based on immediate environmental conditions. Pengi does not perform explicit, top-down planning, and does not record a history of its states. There is no equivalent of the controlled partition, so there is no learning, and none of the issues of the controlled-automatic interface arise. The authors do not discuss internal design of the program. See [19].

3. Allen, James F., "Maintaining Knowledge about Temporal Intervals." *Communications of the ACM*, Vol. 26, No. 11 (November, 1983), p. 832-843.

Time is defined in terms of intervals rather than points on a line. The thirteen mutually exclusive interval relationships are:

center; c c c c l c c l. Relation Symbol Inverse Pictorial Example

X before Y < > XXX YYY

X after Y > < YYY XXX

X equal Y XXX YYY

X meets Y m mi XXXYYY

X met-by Y mi m YYYXXX

X overlaps Y o oi XXX YYY

X overlapped-by oi o XXX YYY

X during Y d di XXX YYYYYY

X contains Y di d XXXXXX YYY

X starts Y s si XXX YYYYYY

X started-by Y si s XXXXXX YYY

X finishes Y f fi XXX YYYYYY

X finished-by Y fi f XXXXXX YYY

A constraint net is maintained; its arcs are labelled with possible relationships of the intervals represented by the nodes connected to the arcs. The net possesses less representational power than some temporal logics; a gain in computational efficiency is the result. Superordinate *reference intervals* can be established to impose hierarchy and time-related inheritance on the net. The correct use of reference intervals can increase efficiency. One potential weakness is that event sequences are assumed to be linear rather than cyclic - Tuesday cannot occur both before and after Wednesday. In the present research, temporal planning and temporal truth maintenance could have bearing on the controlled partition of processing, rather than the reactive, automatic partition. Planning would allow for the construction of preliminary automatic reaction sequences based on expected time relationships of events. Temporal truth maintenance of some sort is necessary for environmentally context-dependent controlled events: when a controlled processing sequence is initiated by an automatic event (which was triggered by an environmental event), and the automatic event is later abandoned (due to subsequent environmental changes), outdated controlled processing should be abandoned in a timely way because the context triggering its execution is no longer in force.

4. Allen, John, *Anatomy of LISP*. New York: McGraw-Hill, 1978.

Background reading on the internal structure of LISP.

5. Allworth, S. T., *Introduction to Real-time Software Design.* New York: Springer-Verlag, 1981.

A text on real-time operating system issues.

6. Anderson, John R., "Practice, Working Memory, and the ACT* Theory of Skill Acquisition: A Comment on Carlson, Sullivan, and Schneider (1989)." *Journal of Experimental Psychology, Learning, Memory, and Cognition*, Vol. 15, No. 3 (May, 1989), p. 527-530.

Anderson replies that the data from [16] does not contradict ACT*'s learning mechanisms (ACT* is a production system cognitive model that learns primarily from production composition). He gives some examples for ACT* productions for their data. See [16,17].

7. Anzai, Yuichiro, "Doing, Understanding, and Learning in Problem Solving." *Production System Models of Learning and Development*, ed. David Klahr, Pat Langley and Robert Neches. Cambridge, MA: MIT Press, 1987, p. 55-97.

Deals with acquiring rules for proceeding through a search space more efficiently by: a) constructing initial problem space, b) collecting bad instances while performing a weak search, c) acquiring productions for avoiding bad instances, d) collecting good instances and building subgoal generation procedures, and e) discovering patterns in subgoal structures (in this sequence). Anzai briefly mentions the need for more work to relate the problem-solving and learning processes to the perceptual/ motor control issues of real-time tasks (p. 92-93).

8. Archer, Jr., Rowland Frank, "Representation and Analysis of Real-Time Control Structures." MIT Laboratory for Computer Science, September, 1978.

This is mostly the development of a notation and corresponding semantics for sequencing, iteration, and preemption in real-time code. One interesting idea is that of codestripping, where preemptive scheduling is replaced by voluntary surrender of the processor via system calls which are inserted into the generated code by the compiler. Such call based context switching should normally involve less overhead and is more predictable than preemption. On the down side, the ability of the compiler to insert context switches into generated code at the correct places may be error prone unless the code is otherwise restricted. If general iteration is allowed, switches must be placed inside of innermost loops, perhaps causing switching too frequently, or must be placed within such loops and surrounded by additional tests (such as testing the trip number through the loop) in order to avoid excessive frequency of context switching. Part of the idea of PRIOPS' non-iterative code is to break the code into non-looping pieces which perform logically complete processing, so that process switching occurs naturally at the end of a non-iterative chunk. Consequently the nature of the source language directs the compiler in its calls for context switches.

9. Baker, Jr., Henry G., "Actor Systems for Real-Time Computation." MIT Laboratory for Computer Science Technical Report 197, March, 1978.

This is an early paper on Actor systems (see [1] for more recent work). It deals with real-time only incidentally. Much of the latter has to do with incremental garbage collection.

10. 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.

Several improvements to Rete are made; the results are not O(1). Remove avoids replicating make's match processing through the storage of join counters and pointers to tokens contributing to a join. The number of nodes is reduced in several ways. One-input tests are ordered by class field. Some non-shared one-input tests (called *special one-input nodes)* are moved to *after* two-input nodes in order to allow generation of fewer two-input nodes. This reduction in number of nodes does not guarantee faster execution. Interrupt handlers are allowed to perform searches of and modifications to working memory. Such modifications are queued until the end of the currently executing rule's right hand side; they are then matched. Rule right hand sides may explicitly allow interrupt data matching to occur between right hand side commands. Such matching contributes to a *demon conflict set* for interrupting data. The demon conflict set is distinct from the normal conflict set, and is used to trigger alarms. Demon conflict resolution and firing is performed immediately after any additions to the demon conflict set (but only after completion of the current right hand side action, which may include time consuming matching). One difficulty is that non-atomic right hand sides that are preempted by demons, may have data that they are using removed by the demons. The solution in this system is to require explicit tests by the preempted productions

to ensure continued existence of necessary data; this method is extremely prone to timing-dependent problems. No attempt to restrict the size or time of join matching is done.

11. Blank, Glenn David, "A Finite and Real-Time Processor for Natural Language." *Communications of the ACM,* Vol. 32, No. 10 (October, 1989), p. 1174-1189.

Register vector grammar uses constrained embedding and boundary backtracking to parse natural language sentences in O(n) time. The restrictions on expressiveness, such as limits to embedding depth, correspond to human language processing limitations.

12. Brinch Hansen, Per, *The Architecture of Concurrent Programs.* Englewood Cliffs, NJ: Prentice-Hall, 1977.

The major item of interest here is the concept of restricting the constructs of a programming language (in this case Concurrent Pascal) in order to constrain run-time characteristics of the generated processes and in order to enhance opportunities for compiler-based error detection. See [13].

13. Brinch Hansen, Per, "Distributed Processes: A Concurrent Programming Concept." *Communications of the ACM,* Vol. 21, No. 11 (November, 1978), p. 934-941.

This article suggests extensions to Concurrent Pascal (see [12]) for real-time multiprocessing. The language is to be augmented with remote procedure calls and guarded regions providing synchronous communications. An assumption of one processor per process is made, with the number of processes being known at

compile time. This assumption is unrealistic, since software modifications which change the number of processes would automatically require hardware reconfiguration.

14. Brooks, Rodney A., "A Robust-Layered Control System for a Mobile Robot." *IEEE Journal of Robotics and Automation,* Vol. RA-2, No. 1 (1986), p. 14-23.

Brooks describes a *subsumption architecture* for controlling a mobile, exploratory robot. The design centers around supporting multiple *levels of competence* for the robot. Each level can accomplish a complete collection of tasks, including complete sensor-to-effector data flow. Lower levels perform simple interactions, such as avoidance of collisions with environmental objects. For each level there is a corresponding *layer of control.* Higher, more complex layers include the activities of lower layers (hence subsumption), and can inhibit lower level reactions. The result is a robust architecture that degrades gracefully. If a higher layer fails, its inhibition of lower layers stops, so these lower layers can still provide reasonable, simpler reactions. The robot does not fail catastrophically with failure of higher levels.

15. 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.

While geared toward OPS5, there is much of interest to any production system programmer or designer.

16. Carlson, Richard A., Marc A. Sullivan and Walter Schneider, "Practice and Working Memory Effects in Building Procedural Skill." *Journal of*

Experimental Psychology, Learning, Memory, and Cognition, Vol. 15, No. 3 (May, 1989), p. 517-526.

This paper questions the notions of Anderson's ACT* system and the SOAR system, that ALL learning (chunking for SOAR, primarily composition for ACT*) result from restructuring of production memory. Carlson, et. al. vote for a distributed short-term memory (rather than a central, production system one), and for speeding up component tasks in their experiment (as opposed to restructuring of component tasks as expected by the 2 production system models of learning), to account for their results.

17. Carlson, Richard A. and Walter Schneider, "Practice Effects and Composition: A Reply to Anderson." *Journal of Experimental Psychology, Learning, Memory, and Cognition*, Vol. 15, No. 3 (May, 1989), p. 531-533.

Carlson and Schneider reply that ACT*'s coverage of their data is ambiguous, with increased time for matching increasingly complex composed production conditions offsetting composition improvements (Anderson's defense) in execution speed. They conclude that "Accepting boundary conditions on composition seems to preserve the strengths of ACT* and to be preferable to the difficulties resulting from explaining the negation effect on the basis of pattern matching time."

18. Chandrasekharan, M., B. Dasarathy and Z. Kishimoto, "Requirements-Based Testing of Real-Time Systems: Modeling for Testability." *IEEE Computer*, Vol. 18, No. 4 (April, 1985), p. 71-80.

This paper proposes using a finite state machine model (FSM) augmented with decision procedures and signal handling and timing capabilities as the basis for a

real-time system specification. Some weaknesses of FSM in comparison to other models (such as Petri nets) include inability to perform certain computations (such as stack-based calculations), verbosity of finite machine specifications, and the sequential mind set of FSM. Parts of the first two problems can be corrected with decision procedure extensions which are distinct from the FSM (and thus more powerful); non-FSM mechanisms are thus restricted in scope. The third limitation of strictly sequential processing meets the authors' application. They propose their augmented FSM for systems in which sequential computations dominate. Expressive power is sacrificed for reliability, since testing requirements for a FSM can be more readily analyzed than for a more powerful mechanism. Methods for automatic test generation and execution are discussed.

19. Chapman, David and Philip E. Agre, "Abstract Reasoning as Emergent from Concrete Activity." *Reasoning about Actions and Plans,* Proceedings of the 1986 Workshop, ed. Michael P. Georgeff and Amy L. Lansky. Los Altos, Ca: Morgan Kaufmann, 1987, p. 411-424.

This paper posits that abstract reasoning is not primitive, but derived phenomenologically, developmentally, and implementationally from concrete activity. Crucial to the process of integration of concrete events is *internalization:* getting control over interactions with the environment by bringing them inside yourself. The paper emphasizes routine activity in *situated* environments; concrete activity is very close to the notion of automatic human processing. Both the emphasis on processor-environmental interaction and the de-emphasis on the

importance of detailed planning are important to PRIOPS research.

20. Cohen, Paul R., Michael L. Greenberg, David M. Hart and Adele R. Howe, "Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments." *AI Magazine,* Vol. 10, No. 3 (Fall, 1989), p. 32-48.

The paper presents an architecture, set in the context of simulated fire-fighting in Yellowstone Park, for interactive plan-based action, replanning, and reflexive reaction among coordinated, autonomous agents. Emphasis is incremental refinement and instantiation of an agent's plan sequence(s), including temporal instantiation, scheduling, and error recovery. Each agent has a two-part architecture: a *reflexive component* (RC) and a *cognitive component* (CC). The former recognizes basic emergency conditions as reported by sensors, and generates immediate effector responses. Like PRIOPS automatic partition, it retains no memory. The CC is responsible for maintenance, selection, and instantiation of plans, event memory, and communications with other agents. The RC is much more loosely coupled to the CC than the automatic partition is to the controlled partition in PRIOPS. The RC can set simple flags to inform the CC of its actions, but it cannot interrupt the CC. There is no equivalent to *record* and *predict* automatic-controlled buffering in PRIOPS, nor is there resource recovery at an automatic-controlled boundary. Learning of RC actions is absent from the current model. Focus is on dynamic planning - a controlled activity from the PRIOPS viewpoint. *"Lazy skeletal refinement* responds to a complex dynamic world by postponing decisions, while grounding potential actions in a framework that

accounts for data, temporal and resource interactions," (p. 43). Plan instantiation is more flexible than classic, static planning.

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

This book concentrates on OPS5 for a range of problems, and includes an excellent chapter on Rete and OPS5 programming for efficiency.

22. Ennis, R. L., J. H. Griesmer, S. J. Hong, M. Karnaugh, J. K. Kastner, D. A. Klein, K. R. Milliken, M. I. Schor and H. M. Van Woerkom, "A Continuous Real-Time Expert System for Computer Operations." *IBM Journal of Research and Development*, Vol. 30, No. 1 (January, 1986), p. 14-28.

The expert system discussed assists computer operators using the IBM Multiple Virtual Storage/System Product - Job Entry Subsystem 3 (MVS/SP-JES3). It is written using a modified version of OPS5 whose modifications include: a) compilation of rule right hand sides; b) the availability of a TIMED-MAKE command for creating working memory elements at specified times or after specified intervals; c) the availability of a REMOTE-MAKE command for communications d) the addition of a "communication phase" to the standard recognize, conflict resolution, act cycle to handle the REMOTE-MAKE; messages are handled just before conflict resolution; and e) the use of explicit rule priorities in conflict resolution. Priorities are superimposed on the MEA and LEX strategies, and an example is given of the use of priorities to implement mutual exclusion by garbage collecting productions. Priorities are considered at conflict resolution rather than at matching time.

23. Etherington, David W., Alex Borgida, and Ronald J. Brachman, "Vivid Knowledge and Tractable Reasoning: Preliminary Report." *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Vol. 2, August, 1989, p. 1146-1152.

This paper outlines an approach for extracting some information from a knowledge base, and instantiating cases in a relational database (the *vivid knowledge base* or VKB) for fast, lookup-based queries. Vivid query time should be sublinear in the size of the knowledge base. The authors concentrate on strategies for converting various knowledge base representation forms, such as universally quantified sentences and several types of disjunctions, into database entries. Some transformations gain efficiency at the loss of information.

24. 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.

The most important thrust of this article is that automatic processing mechanisms are built into long-term memory by controlled processing, but that further storage/learning does not occur during automatic processing. Recollection and learning require attention, which is not in effect during strict automatic processing.

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

Forgy's dissertation and explanation of Rete in the context of OPS2. See [27].

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

The definition of Official Production System 5.

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

An efficient match algorithm which allows production condition tests to be shared among separate conditions whose leading parts are identical; results of partial matches are stored in the matching network. Rete makes refraction tractable. Rete is the standard forward-chaining production system match algorithm against which other match algorithms are compared. Part of the current research is into real-time matching. This effort consists of an attempt to determine what modifications and enhancements can be made to Rete so that a subset of the contributing productions residing in an instantaneous real-time partition of productions (with remaining productions residing in the average real-time partition) can be shown to execute in constant bounded time when their triggering conditions are met. Such time bounding is not guaranteed by standard Rete.

Space and time complexity of standard Rete used in OPS5:

l l l.

Complexity measure Best case Worst case

Effect of working memory size O(1) $O(W^C)$ on number of memory change

tokens

Effect of production memory size O(P) O(P) on number of match processing nodes

Effect of production memory size O(1) O(P) on number of memory change tokens

Effect of working memory size O(1) $O(W^{(2C-1)})$ on time for one firing

Effect of production memory size $O(\log_2 P)$ O(P) on time for one firing.

Where C is the number of patterns in a production, P is the number of productions in production memory, and W is the number of elements in working memory.

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

This article outlines both the standard, interruption collection techniques as well as incremental and non-garbage-generation techniques. Interruption techniques include: mark and sweep; stop and copy (which has better locality of reference than mark and sweep); and reference counting. Incremental techniques include: incremental stop and copy (portions of stop and copy are interleaved with normal execution; hardware dereferencing of multiply indirect pointers is often used; paging overhead can be high due to poor locality of reference); incremental

reference counting (moving objects onto the zero reference queue can be interleaved with execution); generation scavenging (objects are promoted to higher generations based on age, and by implication stability, with younger generations being collected more frequently); and ephemeral collection (a two generation version of generation scavenging / stop and copy hybrid). Garbage collection can be avoided in cases where stack allocation and explicit heap management can be used (e.g., as in Pascal). Garbage collection is important to real-time processing because interruption collection destroys system performance when executing, and incremental collection spreads slowdown more evenly across program execution.

29. Glass, Arnold Lewis and Keith James Holyoak, *Cognition*, Second Edition. New York: Random House, 1986.

Background reading in cognitive psychology.

30. Gupta, Anoop and Charles L. Forgy, "Measurements on Production Systems." Carnegie Mellon Memo CMU-CS-83-167, December, 1983.

Some initial measurements which are expanded in Gupta's dissertation [31].

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

Gupta examines issues in parallel hardware implementation of Rete, basing his analysis on six existing production systems written in OPS5 and SOAR. He comes to the conclusion that the possible speed improvement is limited to roughly a factor of 10 rather than the 100-fold to 1000-fold improvement expected. Reasons cited include the facts that only a small number of productions are affected per

working memory change, there is a large variation in the processing requirements of these productions, and the number of changes made to working memory per recognize-act cycle is very small. Gupta designs a 32 to 64 processor, shared memory MIMD machine with a dedicated hardware scheduler attached to the bus; typical speed improvements are less than a factor of 10. Different levels of granularity of parallelism are explored: "Production parallelism" (the largest grain), where productions are partitioned across the hardware and production matches are performed in parallel; this method suffers due to the loss of inter-production node sharing in Rete; "node parallelism," where activations of different two-input Rete nodes are executed in parallel; "intra-node parallelism," similar to node parallelism, where multiple activations of each two-input Rete is allowed; and "action parallelism," where the effects of multiple modifications to working memory within one inference cycle are allowed to occur concurrently. The combination of intra-node and action parallelism proved to be the most effective. Parallelism in the conflict-resolution and RHS evaluation are briefly mentioned, but since the latter typically account for about 5% each of system execution time, with matching consuming the remaining 90%, Gupta concentrates on matching. Application parallelism, where the nature of the computer application allows straightforward use of multiple processors, is mentioned but not discussed in detail because it is application specific. Gupta simulates execution of his machine.

32. Gupta, Anoop, Charles L. Forgy, Dirk Kalp, Allen Newell and Milind Tambe, "Parallel OPS5 on the Encore Multimax." *Proceedings of the International Conference on Parallel Processing,* Vol. 1 (August, 1988), p. 271-280.

Application of Gupta's and related research to a particular multiprocessor. Mutual exclusion through use of simple locks was found to be more time effective than complex synchronization schemes. Majority of multiprocessing contentions occur over task queues and multiple access to two-input node memories.

33. Gupta, Anoop, Charles Forgy and Allen Newell, "High-Speed Implementations Rule-Based Systems." *ACM Transactions on Computer Systems,* Vol. 7, No. 2 (May, 1989), p. 119-146.

This reiterates some of the points made in Gupta's other papers. The emphasis is on the architecture required for a hardware production system machine. Rete on this architecture is compared to Rete on the DADO and NON-VON machines, with the production machine *simulation* (there is no machine yet) proving superior. All of the forms of production system parallelism in Gupta's dissertation are repeated here.

34. 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.

This paper tackles problems concerning proving guaranteed response times when using Rete based pattern matching. The author shows that joins across working memory elements dominate the matching time complexity, and that join times cannot be predicted without restricting the general Rete join. Proposed

limitations are: 1) Join matching limitations: Establish some finite limit on the number of matches for a join (to use Forgy's terminology, a limit to the number of tokens input/stored by a two-input Rete node); 2) Pattern instantiation restrictions: Establish a limit on the number of instances of a pattern (the number of working memory elements matched to a condition element); 3) Relation instance restrictions: Establish some finite limit on the number of instances of a relation (where a relation is a working memory element with its contents, but not its recency tag considered); 4) Cardinality restrictions: Establish some finite limit on the number of instances of a relation given a set of values for some subset of its arguments (i.e., using the latter as "keys"). My extends this by embedding priority-based, preemptive processing in the Rete net, and by restricting the number of sensor-signal based relation instances to one per sensor.

35. Hayes-Roth, Barbara, "The Blackboard Architecture: A General Framework for Problem Solving?" Heuristic Programming Project Report No. HPP-83-30, Stanford University, May, 1983.

The blackboard architecture is composed of *entries*, *knowledge sources*, the *blackboard*, and the *control mechanism.* Entries are objects of user-determined complexity; associated attribute-value pairs describe an element's semantic content, its relationship to other entries, its history of generation and modification, and any other useful information. Knowledge sources are the cognitive processes that produce entries. Knowledge sources are composed of a condition portion which must be satisfied, followed by an action portion which builds and modifies entries.

Knowledge sources do not communicate directly, but only through the blackboard. The blackboard is a global data base containing all entries generated by all knowledge sources during problem solving; it serves two functions. First, it mediates all knowledge source interactions. Second, it organizes all partial and complete solutions generated for the problem under attack. The blackboard may have user-determined internal structure. Typically knowledge is represented at various levels of abstraction. Finally, a central control mechanism is responsible for maintaining an agenda of satisfied knowledge sources and scheduling these for action. This scheduler is knowledge based and can employ various inference mechanisms and search strategies in solving the problem. Blackboard systems exhibit opportunistic behavior, triggering knowledge sources as they recognize pertinent information. Control may be combinations of top-down, bottom-up, and island expansion and convergence. Multiple sources of knowledge at differing levels are handled well.

Entries, knowledge sources, and the blackboard correspond roughly to working memory elements, productions, and the working memory matching network in OPS-like production systems. The control scheduler has no direct counterpart; control in production systems is distributed throughout the productions. A blackboard system attempts to separate control and domain information to a greater degree than a production system. Only the basic mechanisms of conflict resolution normally exist in a production system; conflict resolution typically has rather

limited flexibility. It is possible, however, to design a more central scheduler into a production system. With the scheduler built using the production system architecture, it would undoubtedly be slower than a scheduler built into the architecture.

36. Hildreth, Ellen C. and John M. Hollerbach, "The Computational Approach to Vision and Motor Control." MIT AI Lab Memo 846, Center for Biological Information Processing Memo 014, August, 1985.

Interesting points about motor control include the following: 1) Because feedback loops within the nervous system operate too slowly, moderately fast to fast arm movements must be controlled in an open-loop manner (p. 48-49). Neural feedback speed is insufficient to support a classic servo-mechanism, but it can support more global or long-term adaptation; the alternative appears to be predictive control. 2) Movement planning involves a hierarchy of trajectory planning, inverse kinematics, inverse dynamics, and torque production. 3) Practice: "One way of compensating for an inability to model the actuation and transmission elements is to tune the output for specific movements through repetition. This approach is very reminiscent of the motor tape idea, in which the output is known only for one particular trajectory. According to this approach, general movements would be made coarsely or suboptimally with an imprecise system model and control, but for frequent movements the control system would modify its output for a new repetition based on errors from the previous repetition," (p. 58). Specialized mechanical feedback systems are discussed. 4) Specially tuned, inflexible

elemental movements are discussed in a tone comparable to automaticity (p. 67).

37. Hoare, C.A.R., "Monitors: An Operating System Structuring Concept." *Communications of the ACM*, Vol. 17, No. 10 (October, 1974), p. 549-557.

An seminal paper on a construct which aids in avoiding the critical section problem between concurrent processes which share resources. Critical sections definitely figure into interrupt handling, and thus into time-dependent processing.

38. Holland, John H., Keith J. Holyoak, Richard E. Nisbett and Paul R. Thagard, *Induction: Processes of Inference, Learning, and Discovery.* Cambridge, MA: MIT Press, 1987.

The authors present a framework for inductive learning and performance directed tuning based on a system of mental models which are homomorphic to the phenomena which they model. Mappings from initial and goal states in the modelled phenomena to initial and goal states in the model are such that transformation operators which transform the modelled initial states through intermediate states to modelled goal states have a direct mapping correspondence to operators which transform the model initial states through corresponding intermediate states to model goal states.

Models are built of frame-like clusters of message passing production rules; the rules are the small-grain substrate of the architecture. Rules are selected for firing based on strength, support, and specificity. Each rule accumulates a fluid strength, based on the degree to which it has been successfully applied in the past. Portions of strengths are back propagated from rules which consume messages (i.e.,

triggered rules) to the rules which sent the messages (i.e., triggering rules). The rules which terminate these inference chains are rewarded with strength increments for success and decrements for failure. This back propagation of strengths (termed the 'bucket brigade') is one of the learning mechanisms in the model. Strengths probabilistically contribute to conflict resolution; a strong rule has a high likelihood of firing when its antecedent conditions are met by an incoming message, but low-strength rules also occasionally fire, allowing them opportunities to prove their abilities and increase their strengths through reward. The degree of support which a triggered rule receives (i.e., the number of rules sending enabling messages to it during the current cycle) also contributes to conflict resolution. Specificity based conflict resolution allows construction of a default hierarchy of rule clusters; more specific rules represent exceptions and special cases, which take precedence over more general, default rules. Finally, a limited degree of parallel rule firing is allowed by conflict resolution. The overall effect is one of spreading activation among related rule clusters within a goal context.

Rules are of two types: 1) Synchronic rules supply definitions, local and hierarchically inherited properties, and general associations of clusters. 2) Diachronic rules define changes and actions, especially temporal transformations. The processing architecture is partitioned across three major levels: 1) Empirical rules deal with empirical phenomena in the environment. 2) Inferential rules comprise the architecture's meta-rules; they deal with acquisition and modification

of rules, and can themselves be acquired and manipulated. 3) Operating principles are built-in and support the rest of the architecture; these include support for the message passing based activation of rules and the bidding system whereby rule satisfaction, strength, support and specificity yield rule bids whose magnitude determine the probability of rule firing.

Learning mechanisms discussed include generalization, specialization, formation of frame-like rule clusters, strength propagation, rule composition and alteration through genetic operations, and analogy. Statistical heuristics such as the 'law of large numbers' are emphasized; central tendencies and variability in attribute values contribute to the definition of these attribute values in rule antecedents. Thus both rule creation and rule firing have stochastic components.

The authors apply the framework to a wide range of processing conditions including animal conditioning, human category hierarchy studies, intuitive physics, social psychology, and scientific discovery. They end by discussing further work necessary for the growth of the framework into an actual theory.

39. Hsu, Ching-Chi and Feng-Hsu Wang, "The Search Ahead Conflict Resolution for Parallel Firing of Production Systems." *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Vol. 1, August, 1989, p. 91-96.

The paper discusses static, compile-time analysis of production test matching and LIFO conflict resolution, such as OPS5's *lex* strategy. The result is a conflict resolution strategy that allows parallel execution of multiple rule instantiations

while preserving the semantics of an equivalent uniprocessor LIFO strategy. The paper does not deal with parallel match processing.

40. Hunt, Earl and Marcy Lansman, "Unified Model of Attention and Problem Solving." *Psychological Review,* Vol. 93, No. 4, p. 446-461.

This a controlled-automatic production system simulation that reproduces results comparable to earlier studies on humans. Unlike PRIOPS, no Rete-based symbol matching occurs. The system determines productions to fire based on activation levels that are set, in turn, by simple feature strengths in sensory input and working memory elements. The automatic processing is really a semantic net of hand-coded interconnections. Production notation is used to allow RHS actions in addition to semantic net activation propagation.

41. Jacob, Robert J. K. and Judith N. Froscher, "Software Engineering for Rule-based Systems." Naval Research Laboratory, Washington, D.C.

This paper discusses a design approach based on partitioning a rule base into groups, where a group is characterized by a large degree of internal coupling via sharing of working memory references. Inter-group communications occur through a small set of shared working memory references. Algorithms which can detect and highlight inherent partitions in existing rule bases are outlined, and such partitioning is advocated as a design method. While object oriented programming is not mentioned per se, this modular approach certainly suggests the possibility of partitioning the knowledge base into individual objects with accompanying production methods. Communications between objects could be based on loosely

coupled, explicit message passing.

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

Object-oriented extensions to an existing embedded system model are proposed. Methods for performing modifications (enhancements in particular) to a system without interfering with ongoing operations are discussed. Modifications include addition of enhancements, modification of object internals, with the object interface appearing unchanged, and modifications of objects causing changes in the object interfaces.

43. Jonides, John, Moshe Naveh-Benjamin, and John Palmer, "Assessing Automaticity." *Acta Psychologica* 60, 1985, p. 157-171.

The authors propose two guidelines in the experimental study of automaticity: 1) The concept of automaticity is best applied to *component processes* of complex behaviors rather than to behaviors as a whole. 2) Criteria chosen for the identification of automaticity should be motivated by the process in question. They leave open the question of whether (in view of their second point) "there is really no unified concept of automaticity that cuts across the particular tasks and paradigms that appear in the literature? It is too early to tell."

44. Keller, Richard M., "Defining Operationality for Explanation-Based Learning." *AAAI-87, Sixth National Conference on Artificial Intelligence,* Volume 2. Los Altos, CA: Morgan Kaufmann Publishers, Inc., 1987, p. 482-487.

First explanation based-learning is contrasted with empirical learning. The

latter performs simple syntactic analysis of similarities and differences among a large number of training instances. Explanation-based generalization (EBG) performs an in depth, knowledge-intensive analysis of a single (and typically positive) training instance, generating an explanation of why the instance is an example of the concept to be learned. The explanation is next generalized to fit a larger class of instances, and a description of the larger class is extracted from the generalized explanation. This description constitutes a generalization of the original instance.

Note that in order to fit the instance to the more general target concept, the learning system must already have some information about the more general, target concept prior to its exposure to the training instance. In fact the author discusses EBG from the position that it is a method for refining existent concepts (as opposed to primarily acquiring new ones) by transforming concept representations into representations that are more functionally useful, based on the training instances and on operationality (functional usefulness) criteria. The paper outlines a three space approach to concepts: the Instance Space contains instances; the Concept Space contains the abstract, implicit (non-represented) concepts; the Concept Description Space contains the explicit (represented) concept descriptions, many of which may map onto a single abstract concept. Concept Description Space is partitioned into non-operational and operational regions, where operationality is defined in terms of usability and utility. Usable means that a description is in a

form accessible by the performance system, and utile means that a description is worth using as judged by some performance criteria (e.g., time and space complexity). The paper characterizes EBG as constituting a search from non-operational to maximally (or at least sufficiently) operational descriptions for a given concept, the end result being a description in a usable format.

The paper goes on to categorize operationality of several learning systems in terms of granularity of utility estimates or measurements (binary or continuous), of certainty of utility estimates, and of variability of operationality criteria (static or dynamic). Continuous estimation allows tuning of performance (as opposed to acceptable/unacceptable), and dynamic variability allows performance criteria to be learned with performance mechanisms (as opposed to having performance criteria hard-coded into the program). The author's experimental system MetaLEX possesses all three desirable qualities (including full certainty), but at large computational cost. The performance system is exercised at every decision point in the path from the non-operational to operational description in order to evaluate potential changes in description, making the learning more like practice than planning; this practice-based evaluation provides support for the conjunction of full certainty and continuous granularity in the evaluation of description changes.

The application to my research-in-progress relates to the fact that the generation of constant-time-bounded matching structures amounts to a transformation of information which is already known but is not in a form capable of being utilized

within the time bounds. Operationality for real-time code is thus defined as processing within the necessary time/space constraints. Explicit programming, planning, and practice amount to a transformation of non-operational information contained within the system and processing environment into information which is both usable and utile at performance time.

45. Kelly, Michael A. and Rudolph E. Seviora, "An Evaluation of DRete on CUPID for OPS5 Matching." *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence,* Vol. 1, August, 1989, p. 84-90.

The authors' DRete generates multiple executions of a join node, one for each token stored in one of its memories, when a token arrives or departs at the opposing memory. These executions run in parallel. The speedup is on the order of Gupta's (or less), around 8 with 64 processors over the speed of 1 processor (32 processors give almost as much speedup, so improvement has levelled off by 64).

46. Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language,* Second Edition. Englewood Cliffs, NJ: Prentice Hall, 1988.

PRIOPS is written in a version of C that *almost* matches the proposed ANSI standard (Microsoft® C 5.1).

47. Korf, Richard E., "Planning as Search: A Quantitative Approach." *Artificial Intelligence* 33 (1987), p. 65-88.

The author applies time and space complexity analysis to planning using subgoals, macro-operators, and abstraction as knowledge sources. Planning is discussed in terms of problem solving search.

48. Laffey, Thomas J., 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.

This is a survey of real-time expert system applications, tools, and theoretic issues. It addresses possible definitions of "real-time" and the increased difficulties of real-time systems over traditional expert systems. Specific problems in using OPS5's Rete Algorithm are mentioned. Application areas discussed include aerospace, communications, financial, medical, process control, and robotic applications. A point is made that none of the systems examined adequately addresses the fundamental problem of guaranteed response times. Proposed research issues include basic performance, guaranteed response times, production systems, and real-time derivations of the Rete Algorithm. The article provides an excellent list of references for applications and tools discussed.

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

The Universal Weak Method is one of the key aspects of the SOAR architecture. It is essentially the weakest, most knowledge deprived search method possible - directionless trial-and-error with the ability to perform backtracking upon failure and avoid repetitious looping through identical states. The authors propose that incremental additions of domain specific knowledge to the Universal Weak Methods automatically give rise to the variety of standard weak methods (e.g., hill climbing or means-ends-analysis).

50. Laird, John E., "Universal Subgoaling." Memo CMU-CS-84-129, Carnegie-Mellon University, May, 1984.

Universal Subgoaling is one of the key aspects of the SOAR architecture. It implements automatic generation of subgoal / nested problem space searches when the operator selection and application mechanism of SOAR reaches a processing impasse due to tie, inactivity, preference conflicts or invalidation of a context role withing the current processing context. Chunking tracks the processing of subgoals and chunks over them with single recognize/act productions.

51. Laird, John E., Paul S. Rosenbloom and Allen Newell, "Towards Chunking as a General Learning Mechanism." Part of Memo CMU-CS-85-110, Carnegie-Mellon University, January, 1985.

A short look at SOAR's chunking. See [52].

52. Laird, John E., Paul S. Rosenbloom and Allen Newell, "Chunking in SOAR: The Anatomy of a General Learning Mechanism." *Machine Learning* 1 (1986), p. 11-46.

Chunking is one of the key aspects of the SOAR architecture. When a subgoal is initiated, a trace is made of the information the subgoal draws from its supergoals' environments during the useful parts of its searching; return results are also traced, and the processing dependencies of this input to output information flow is bound as the conditions and actions of a chunk production. Thereafter whenever the conditions arise which would have given rise to the subgoal search processing, the chunk production fires, short-circuiting the search. Chunking comes in two flavors, terminal and general. With terminal chunking only subgoals

whose solution does not require nested subgoal processing are chunked; with general chunking all subgoal processing is chunked. Terminal processing builds hierarchies of chunks in a bottom-up order. Weaknesses of SOAR's chunking include severe memory consumption (SOAR assumes unlimited memory), insensitivity to data which almost triggers subgoal processes and insensitivity to updates to subgoal problem spaces after chunking is complete.

53. Laird, John E., Allen Newell and Paul S. Rosenbloom, "SOAR: An Architecture for General Intelligence." *Artificial Intelligence* Vol. 33, No. 1 (1987), p. 1-64.

SOAR (State Operator And Result) is a processing architecture based on searching through a set of problem spaces in order to solve goals. Processing contexts are stacked; each context consists of goal, problem space, state and operator roles. Augmentations are linked declarative additions to the basic roles; preferences are votes cast (veto, acceptable, and relative or absolute preferences) for changes to role contents in a context. The machine cycle consists of 2 parts: Elaboration, where productions firing in parallel add prospective role objects, augmentations and preferences to working memory; and Decision, where preferences are used to order potential actions. Either a change action is accepted, or a subgoal process is generated when processing reaches an impasse. See other SOAR papers by these authors. SOAR is implemented as a derivative of OPS5.

54. Laird, John E., "Learning from External Environments using SOAR." *Proceedings of Applications of Artificial Intelligence VII*, Volume 1095, Part 1, ed. Mohan M Trivedi. Bellingham, Washington: Society of Photo-Optical

Instrumentation Engineers, 1989, p. 575-576.

The paper is an abstract for the talk given at the conference on March 30, 1989. Laird emphasized SOAR as a system which combines the important characteristics of I) Interactive systems (e.g., control systems); II) Knowledge-based and general systems (e.g., expert systems and planning systems respectively); and III) Learning systems (e.g., explanation based learning). Constraints for type I systems include: 1) real-time operation; 2) accept sensory data upon arrival; 3) integrate sensory data into world model; 4) permit sequential and parallel motor commands; 5) accept help from other intelligent agents. Laird emphasized the latter and the importance of learning both from the environment or examples and from interaction with other intelligent agents. For type II activities in SOAR, Laird emphasized Universal Subgoaling as the method for dynamically directing hierarchical problem solving and planning (the plans being stored as chunks). Laird listed the activities of a type III (learning) system as: 1) improving speed/performance; 2) solving new problems; 3) correcting errors; 4) learning (new) relevant features from the environment; and 5) predicting behavior of the environment. Both *advice* and *experience* should contribute to learning. Learning should not impair performance, and learning should have immediate benefit. Problems with deductive learning are that the real world domain theory may be incomplete, and the system must fix incorrect knowledge. Methods for accomplishing the latter might include removing errors, modifying them, lowering

rule strengths, masking them with new knowledge, or adding new knowledge to correct the effects of errors. SOAR's chunking does not remove erroneous knowledge, but builds new chunks which mask the erroneous knowledge (the error generates an impasse which leads to solution of the error-generated problem and generation of the masking chunk).

Laird gave examples in terms of controlling a simple robot vehicle which he was holding. At question and answer time, I posed a question about whether SOAR could respond to an emergency sensor (e.g., dangerous temperature) if the sensor was unrelated to the current stack of subgoals. I suspected that it would always be necessary to keep an artificial goal in memory (e.g., goal: don't let the temperature get too high) in order to enable chunks for handling the sensor. He said that the sensor would be responded to correctly, but that the system would immediately drop AND FORGET the current goal processing in reacting to the sensor. He did not elaborate, but this leads me to believe that the artificial goal for monitoring the temperature sensor is maintained deep in the stack, and when it is dealt with (and satisfied), all intermediate goals are dropped (as is the case).

55. Langley, Pat, "A General Theory of Discrimination Learning." *Production System Models of Learning and Development*, ed. David Klahr, Pat Langley and Robert Neches. Cambridge, MA: MIT Press, 1987, p. 99-161.

Deals with a model of learning that starts out with missing rules that first present errors of omission, leading to creation of overly general rules which present errors of commission, followed by increased discrimination based on

pairwise comparisons of correctly and incorrectly applied rule instantiations. The theory covers conjunctive and disjunctive concept learning, search improvements, learning in noisy environments, and learning information which is subject to change. Conflict resolution is: a) refraction, b) *production strength* (important to PRIOPS), c) recency, and d) random selection. The production strength is increased as a rule is repeatedly instantiated (which demonstrates its value). Strengthening helps to weaken noise induced productions and helps to track changing information; strengthening causes productions to be forwarded into activation in a beam search like manner - learned productions are never forgotten (breadth first), but only the currently strong ones are important (beam width). Learning search heuristics is one topic covered. Criteria of a good scientific theory is also an interesting aside.

56. Lesser, Victor R., Jasmina Pavlin and Edmund Durfee, "Approximate Processing in Real-Time Problem Solving." *AI Magazine,* Vol. 9, No. 1 (Spring, 1988), p. 49-61.

This article outlines an approach to planning where a tradeoff is made between estimated execution time and quality of the plan. Estimates of execution times are made, based at least in part on experience. Steps which are in some way redundant or which in some way do not fully contribute to the final problem solution are eliminated when such elimination is needed in order to meet timing constraints. An elimination of such redundant or unnecessary processing is termed a "well-defined approximation"; well-defined approximations are contrasted with various ill-defined

approximations. Three types of approximate reasoning discussed are approximate search strategies, data approximations, and knowledge approximations. Approximate search strategies include eliminating corroborating (redundant) support and eliminating competing interpretations (when the eliminated interpretations are significantly less well supported than their competitors). Data approximations include incomplete event processing (some non-critical information in the data is ignored) and cluster processing (clusters of data are processed as a single unit). Knowledge approximations include approximations that work with data approximations (geared toward translating constraints on data to constraints on data clusters) and approximations that summarize several sources of knowledge into a single, less discriminating knowledge source by eliminating some of the intermediate processing steps. The paper examines situations within which these approximations can be applied in a well-defined fashion. An important assumption about this framework for real-time control is that it is possible to make a reasonably accurate estimate of the time to carry out the steps of the plan and the quality of the expected solution.

57. Levesque, Hector J. 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.

The tradeoff is between expressiveness and tractability, with full first-order logic being intractable. Two pseudo-solutions, speeding up the computing

environment and terminating search at a predetermined time and reporting excessive search time, do not address the fundamental problem of tractability. The authors suggest a continuum of notations and architectures that trade expressiveness for tractability. The PRIOPS automatic partition is a powerful example of trading expressiveness - the automatic partition's capabilities are severely constrained to performing habit-like activities - for tractability - in the PRIOPS case, O(1) complexity.

58. Levinthal, Charles F., *Introduction to Physiological Psychology.* Englewood Cliffs, NJ: Prentice-Hall, 1983.

Background reading in physiological psychology.

59. Lewis, Clayton, "Composition of Productions." *Production System Models of Learning and Development,* ed. David Klahr, Pat Langley and Robert Neches. Cambridge, MA: MIT Press, 1987, p. 329-358.

Lewis performs a formal analysis of composition of productions, where interacting sequences of applicable productions are replaced by a single production representing their composition. Safe composition is discussed, where a safe composition preserves the effects of the original productions. Composition does not rely on trace data but rather on strictly syntactic examination of the productions. Lewis discusses the need to apply similar formal analysis to other production system based models of learning such as proceduralization, discrimination, and generalization.

60. Lieberman, Henry 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.

The article deals with generation scavenging incremental garbage collection. Generation scavenging is like stop and copy with multiple regions, ordered by age. The assumption is that old data is fairly stable; a high degree of all garbage is generated by recent (temporary) data. Therefore regions are maintained and garbage collected according to age, with the youngest being collected more frequently. Garbage collection consists of two parts: condemnation initiates evacuation of an area; scavenging adjusts pointers from outside the area. An explicit requirement necessary for efficient scavenging is that pointers are predominately from newer to older data, i.e., that few circular mutations (e.g., NCONC) are used. Pointers from old to new data require additional levels of indirection. Non-mutative programming of the kind required by this method (a common style of LISP programming) does incur data copy overhead penalties. While generation scavenging may be efficient for programs written in the proper style, it appears to involve a great deal of overhead; the authors give no measurements or empirical evidence in support of their algorithm.

61. Liebowitz, Burt H. and John H. Carson, *Multiple Processor Systems for Real-time Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1985.

This overview text deals with hardware, networking, operating system, database, reliability, queueing theory, and engineering considerations for time-constrained distributed processing. Three types of functional allocation of tasks to

processors is considered: a) dedicated function, in which specific functions are preassigned to processors (large grain MIMD); b) traffic sharing, in which the input data stream is divided (either statically or dynamically) across processors at run-time (more task homogeneous MIMD, possible pipelined and array SIMD); and c) dynamic allocation of tasks themselves (rather than just input data) at run-time, the most complex with the most overhead, but conceivably the most effective in load balancing. Combinations of a and b are most common. The flavor of the text is system engineering.

"Most real-time systems provide a natural four-way split of functions: communications processing, applications processing, file handling (database) and miscellaneous support functions," p. 35.

62. Lindsay, Peter H. and Donald A. Norman, *Human Information Processing.* New York: Academic Press, 1977.

Background reading in cognitive psychology.

63. Ma, Richard Perng-Yi, "A Model to Solve Timing-Critical Application Problems in Distributed Computer Systems." *IEEE Computer,* Vol. 17, No. 1 (January, 1984), p. 62-68.

This paper discusses a heuristic search-based method for allocating tasks to processors. Major components of such a distributed system design include the interconnection network (IN), an application description language (ADL), and software development (SD) which links the ADL to the IN. Software development in turn consists of task partition and task allocation (to processors). The heuristic

search strategy discussed is based on branch-and-bound; it attempts to: 1) minimize interprocessor communication cost, 2) balance the utilization of processors, and 3) meet various engineering application requirements. Contributing factors to 'thread' port-to-port time include task execution time (i.e., processing time within the port-to-port thread of execution), queueing delay time (where processor sharing occurs), and interprocessor communication time. Guidelines for the heuristic are: 1) to reduce task execution time, large-size tasks (long execution times) should be allocated to higher rate processors, 2) to reduce queueing time, large instruction sizes and frequently enabled tasks should be allocated to different processors, 3) to reduce inter-processor communication costs, tasks with high coupling factors should be placed on the same processor. The queue length (number of tasks) of each shared processor is given an upper limit in order to place a guaranteed limit on queueing delays.

64. Mishkin, Mortimer, Barbara Malamut and Jocelyne Bachevalier, "Memories and Habits: Two Neural Systems." *Neurobiology of Learning and Memory,* ed. Gary Lynch, James L. McGaugh and Norman M. Weinberger. New York: Guilford Press, 1984, p. 65-77.

Mishkin, et. al. present evidence that the dual controlled-automatic processing model (see [81,89]) derives from dual underlying neural hardware.

65. Mitchell, Tom M., "Generalization as Search." *Artificial Intelligence* 18(2) (March, 1982), p. 203-226.

This is Mitchell's paper on the *version space* strategy, which is compared to depth first and breadth first generalization strategies. The version space strategy is

like bidirectional breadth-first, where two sets of descriptive generalizations are learned and refined: Set G holds the most general generalizations capable of matching the training instances seen so far; set S holds the most specific generalizations capable of matching the training instances seen so far. Between the two lies the range of version spaces which can match the training data; as the two converge, a precise set of matching descriptions for the data is formed.

For each negative training instance i:

> Retain in S only generalizations not matching i.
>
> Make generalizations in G that match i more specific, only to the extent required so that they no longer match i, and only in such ways that each remains more general than some generalization in S.
>
> Remove from G any element that is more specific than some other element in G.

For each positive training instance i:

> Retain in G only those generalizations that match i.
>
> Generalize members of S that do not match i, only to the extant required to allow them to match i, and only

in such ways that each remains more specific than some generalization in G.

Remove from S any element that is more general than some other element in S.

The main effect of a negative instance is that G is made more specific (a version is overly general when it matches a negative instance). The main effect of a positive instance is that S is made more general (a version is overly specific when it does not match a positive instance). The two effects combine to cause the two spaces to converge. When training instance information is insufficient to allow disambiguation of all test cases, S and G form a space where test cases which might represent the target concept can be matched. When training instances are exhaustive, S and G merge to provide a precise matching description.

66. Mitchell, Tom M., Richard M. Keller and Smadat T. Kedar-Cabelli, "Explanation-Based Generalization: A Unifying View." *Machine Learning* 1(1), 1986, p. 47-80.

See [44] for more on EBG. EBG reorganizes information gleaned from a single training instance in order to generate a concept description which satisfies some operationality criteria. The algorithm is knowledge intensive in that domain information and a concept definition must already exist. EBG's purpose is to map the training instance onto the concept in the context of the domain information by explanation, then use the existing concept to make the explanation generally

applicable; the result is a concept definition which describes a set of instances to which the training instance belongs. Inputs to EBG are: 1) a goal concept, 2) a training example, 3) the domain theory, and 4) operationality criterion. The algorithm generates a generalization of the training example that is a sufficient concept definition for the goal concept and that satisfies the operationality criterion. The two-part EBG algorithm is:

Explain:

> Construct an explanation in terms of the domain theory that proves how the training example satisfies the goal concept definition.
>
> This explanation must be constructed so that each branch of the explanation structure terminates in an expression that satisfies the operationality criterion.

Generalize:

> Determine a set of sufficient conditions under which the explanation structure holds, stated in terms that satisfy the operationality criterion.
>
> This is accomplished by regressing the goal concept through the explanation structure. The conjunction of the resulting

regressed expressions constitutes the desired concept definition.

67. Neches, Robert, "Learning Through Incremental Refinement of Procedures." *Production System Models of Learning and Development*, ed. David Klahr, Pat Langley and Robert Neches. Cambridge, MA: MIT Press, 1987, p. 163-219.

The Heuristic Procedure Modification program (HPM) is built using the PRISM production system. Traditional models of learning in production systems focus on chunking, generalization, and discrimination. HPM focuses on the invention of new actions by examining the procedures executed (productions fired) in its performance component and adding productions using heuristics dealing with efficiency and interestingness criteria. HPM's learning component is therefore heavily oriented towards meta-knowledge. Productions in that component trigger on structures reflecting the processing of goals (the "goal trace") and actual production system cycle execution (the "production trace"); these two traces overlap, with the former concentrating on high level, goal structured solution of problems, and the latter concentrating on implementation data and flow of control. HPM is therefore very dependent on tracing and explicit representation of traced material; as a model of a cognitive system, HPM relies heavily upon detailed knowledge of its own learning mechanisms. The traces are saved in the form of semantic net which is traversed using a context-sensitive spreading activation scheme to prune the search space. Both detailed episodic information and derived semantic information are retained; HPM's execution is very memory intensive.

Several of the heuristics used in the test application compare traces of distinct processing on similar or identical data in order to find the more efficient traced procedures or in order to invent new, equivalent procedures. HPM learns without feedback, examing its own performance post hoc in order to discover effective procedures for dealing with input. The learning mechanism is domain independent.

68. Newell, Allen, "Production Systems: Models of Control Structures." *Visual Information Processing*, ed. William G. Chase. New York: Academic Press, 1973, p. 463-526.

This is one of Newell's earliest papers on production systems, and it sets the stage for his later work. The paper emphasizes the importance of the specification of a control structure in the illustrating and modelling of cognitive processes. Newell feels that a production system of some variety (which will be implemented as SOAR in later research) could be an appropriate model of such processes. The production system discussed here was PSG; conflict resolution depends upon textual order of the productions. Working memory size is presumed to be very limited (Miller's proverbial 7 items, + or - 2, page 466). This limit is significant in relation to extreme memory consumption by later models, and my own requirements about limits on memories representing sensory buffers. Newell mentions perceptual buffers such as the visual icon on page 467, but does not incorporate them (or indeed any sensorimotor processing) into his production system.

"Having gone this far, it is tempting to state a hypothesis about the locus of conscious experience. It is not to be associated with the content of any memory, not even of STM which defines in an operational sense what the subject is momentarily aware of, i.e., to what he can respond to in the next tens of milliseconds. Rather, phenomenal consciousness is to be associated with the *act* of matching, and its content is given by the set of STM items extracted by the matched condition. Thus, it is an ephemeral fleeting thing that never stays quite put and never seems to have clearly defined edges (the never-step-into-the-same-river-twice phenomenon). It seems like an interesting hypothesis. That the hypothesis can be stated in such a precise form is attributable to having a detailed model of the control structure," (Page 508).

Newell notes that deficiencies of the model presented using PSG include lack of sensory/perceptual components, lack of motor components, and lack of any method for learning new productions (LTM). The latter is in his estimation the most serious, and of course much work has been done since then (and much remains to be done).

An important advantage of production systems advanced in the paper is that they represent both the theory and a working simulation of the cognitive process being modelled; such systems are precise as required by their nature as computer programs. However a production system is not a neutral language for stating a theory - the architecture and implementation of the production system language

contribute to any theory stated using that system.

A final note on an effective limit to STM size and its relation to a theory of error is interesting (p. 523-524):

"Take STM as having indefinite length but being sufficiently unreliable so that there is an increasing probability of an element disappearing entirely. Whether this is decay with time, with activity or what not is secondary. The fate of each element is somewhat independent so that early ones can disappear before later ones. This is the primary error source, from which error propagates to all tasks according to the strategy with which the subject operates. Such a strengthening of the unreliability assumption will reinforce the encoding hypothesis, so that all tasks must be dealt with by encoding. The role of STM becomes one of holding a few items after decoding (dumping into STM) to be picked up quickly by coupled productions, and of holding a few items strung out prior to encoding into a new chunk. Thus the short term capacity is not the length (or expected length) of STM, but is composed from the size of codes and the space for their decoding. For example, a short term capacity of seven might occur via a chunk of three and four, with the STM holding four items reliably enough to get them decoded and emitted. Thus, no memory structure exists in the system that has a capacity of seven. In particular the STM would appear to be misnamed."

69. Newell, Allen, Paul S. Rosenbloom, and John E. Laird, "Symbolic Architectures for Cognition." Chapter 3 of *Foundations of Cognitive Science,* ed. Michael I. Posner. Cambridge, Ma: MIT Press, 1989, p. 93-131.

While the purpose of this chapter is introduction to the requirements and functions of any cognitive architecture, examples for ACT* and SOAR appear. The latter includes SOAR in embedded applications. In a section called "Interaction with the External World," the authors list important properties of an embedded cognitive architecture as: a) interfaces that connect sensory and motor devices to the symbol system; b) handling of asynchronous external events, including support for *buffering* and *interruption;* c) real-time reactivity; and d) environmentally time-constrained learning. There is no clean distinction between instantaneous real-time requirements (my interpretation of property c here) and average real-time requirements (such as learning in an environment). SOAR examples claim environmentally responsive mechanisms, but they all involve interaction with the non-O(1), monolithic central cognitive component. Section 7.1 of this thesis discusses claims for embedded SOAR at more length.

70. Newman, I. A., P. P. Stallard and M. C. Woodward, "Performance of Parallel Garbage Collection Algorithms." Computer Studies 166, Department of Computer Studies, Loughborough University of Technology, Loughborough, Leicestershire, U.K., September, 1982.

This article reviews the time and space performance characteristics of several mark and sweep based parallel garbage collection algorithms in the context of multiple processor marking. Two, three and four color (one and two marking bits) algorithms from Dijkstra and Lamport (3 color), Minsky-Knuth-Steele-Muller-Wadler (2 colors plus a stack or, preferably enhanced by replacing the stack with a

circular FIFO), and the authors Newman and Woodward (4 colors with better storage requirements than the stack algorithm, but with the inability to deal with circular structures) are discussed. Both projected costs and the results of actual tests are shown. The stack-based algorithm has the best overall time complexity, but with the expense of a potentially enormous stack.

71. Ohlsson, Stellan, "Truth Versus Appropriateness: Relating Declarative to Procedural Knowledge." *Production System Models of Learning and Development*, ed. David Klahr, Pat Langley and Robert Neches. Cambridge, MA: MIT Press, 1987, p. 287-327.

A rational model of learning is explored, where appropriate declarative knowledge about a domain (represented as implications in logic) is used to construct and tune procedural constructs (represented as recognize/act productions). The method relies neither on detailed traces of internal procedural processing (which is considered to be of questionable psychological validity; contrast with Neches' HPM) nor on feedback from the learning environment. Instead declarative knowledge about the domain is coupled with environmentally activated production based procedures in order to generate new productions which effect the tuning and modification of procedures. Only a single, currently activated procedural production is available for inspection, rather than a complete history offered by procedural traces in other models (e.g., Neches). This approach, intended to model the inaccessibility of much human procedural processing to introspection, trades practice trials for practice time and memory; the latter are required by tracing, but

lack of tracing necessitates repeated practice in order to reactivate existing productions for examination. Much of the learning is composite (several declarative statements and procedural productions may contribute to the formation of a new production), so repeated practice allows access to all of the requisite existing productions, and learning is gradual. An execution example demonstrates evolution of processing from model building (simulation) through manipulation of strictly symbolic information to perceptual recognition based action triggering. Each shift is accompanied by increases in processing speed and more shallow analysis of input data. Processing is shifted from centrally controlled analysis to peripherally based recognition. The domain specific declarative information used in the construction of procedures is hand loaded; Ohlsson does not address mechanisms for the acquisition of this knowledge (which is distinct from the procedural productions), stating that such acquisition is a distinct problem. Contrast the latter with SOAR, where declarative and procedural knowledge are represented more homogeneously, although control in Ohlsson's system is organized around a goal-directed state space search in a manner reminiscent of SOAR. Like Neches and unlike SOAR, learning uses meta-knowledge based productions rather than a transparent mechanism built into the architecture. Ohlsson's model intentionally ignores certain temporal relationships among input in its avoidance of tracing.

72. 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.

This is the first PRIOPS paper. It concentrates on enhancements to standard Rete.

73. Parson, Dale E. and Glenn D. Blank, "Automatic versus controlled processing: an architecture for real-time production systems." *International Journal of Expert Systems: Research and Applications*, Vol. 2, No. 3/4 (1990), p. 393-418.

The paper explains human controlled-automatic processing and its relation to PRIOPS, discusses the two-tiered approach to embedded processing, and examines Rete and a detailed PRIOPS approach to a temperature sensor problem. This is an expanded version of the earlier SPIE paper.

74. Parson, Dale E. and Glenn D. Blank, "PRIOPS: A real-time production system architecture for programming and learning in embedded systems." Invited by and submitted to the *International Journal of Pattern Recognition and Artificial Intelligence* in February, 1990.

The SPIE paper [72] was one of sixteen selected by the Applications of Artificial Intelligence VII program committee for contribution to a special issue of this journal entitled "New Developments in Expert System Issues." The result is this paper, which illustrates PRIOPS use through the maze demonstration program in this thesis. The journal accepted the paper in February, 1990 for publication sometime in the upcoming summer.

75. Quinlan, James, "A Comparative Analysis of Computer Architectures for Production System Machines." Carnegie Mellon Memo CMU-CS-85-178, May, 1985.

Quinlan undertakes a comparative study of the execution characteristics of several existing OPS5 programs on 6 distinct architectures. He comes to the not surprising conclusion that the microcoded CPU architecture specifically designed for Rete is the most efficient, followed by a RISC architecture also geared toward Rete requirements. The paper does discuss Rete as implemented in OPS83, making it the most up-to-date discussion of Rete.

76. Rosenbloom, Paul S., John E. Laird, John McDermott, Allen Newell and Edmund Orciuch, "R1-SOAR: An Experiment in Knowledge-Intensive Programming in a Problem-Solving Architecture." Part of Memo CMU-CS-85-110, Carnegie-Mellon University, January, 1985.

An application of SOAR's chunking to R1's task of configuring VAX™ computers. See [52].

77. Rosenbloom, Paul S. and Allen Newell, "Learning by Chunking: A Production System Model of Practice." *Production System Models of Learning and Development,* ed. David Klahr, Pat Langley and Robert Neches. Cambridge, MA: MIT Press, 1987, p. 221-286.

Provides support for the notion of terminal (i.e., bottom-up) chunking as the mechanism which underlies the 'power law of practice' ('log-log linear learning law'). Assumes that the time to process a chunk < time to process its constituents; chunks are learned at a constant rate on the average; probability of recurrence of an environmental pattern decreases as the pattern size increases. Constraints required by the model include: some form of parallel processing; some form of bottleneck (capacity limitation); constraints on location of the bottleneck; locus of parallelism cannot be constrained to the sensory and motor portions (i.e., higher cognitive

functions can also be chunked). Implemented using Xaps2 production system. A chunk is composed, bottom-up, of 3 component productions: a) a stimulus pattern, b) a response pattern, and c) a connection between the two. It is possible to combine chunked stimulus patterns, for instance, in order to create a superordinate stimulus pattern for a superordinate chunk. The same holds true for response patterns. See [53].

78. Rosenbloom, Paul S., John E. Laird and Allen Newell, "Knowledge Level Learning in SOAR." *AAAI-87, Sixth National Conference on Artificial Intelligence*, Volume 2. Los Altos, CA: Morgan Kaufmann Publishers, Inc., 1987, p. 499-504.

SOAR's application to both *symbol level learning* and *knowledge level learning* is explored. Symbol level learning consists primarily of performance improvement using knowledge already available to the system. Knowledge level learning consists of integration of new information into a system. Most earlier work with SOAR demonstrated the former; recognition and recall tasks discussed in this paper deal with the latter.

79. Rumelhart, David E., James L. McClelland, and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1: Foundations. Cambridge, Ma: MIT Press, 1986.

Background reading in connectionism.

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

Rete Algorithm as applied to SOAR. Some of these enhancements can be applied to PRIOPS.

81. 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.

This paper and the subsequent Part II. [89] are the seminal papers on automatism. See the latter for more details. Part I. describes the experiments, the notions of consistent and varied mappings of target and distractor stimuli, and of automatic and controlled human information processing. Response times indicate that controlled processing consists of serial, terminating search; automatic detection times flatten with practice, and do not vary significantly with load.

82. Schneider, Walter and Arthur D. Fisk, "Dual Task Automatic and Controlled Processing in Visual Search, Can It Be Done Without Cost?", Human Attention Research Laboratory Report No. 8002, University of Illinois, February, 1980.

Summary: Do not waste valuable controlled processing on already automated tasks (it slows them down) when the former is needed for new training, especially new training which may itself be automatizable.

83. 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.

This article suggests that automatic processing is developed in graded steps relative to the degree of consistency of stimuli (a potential log function of the degree of consistency), rather than all or none. Rosenbloom and Newell [77] also discuss log graded skill acquisition curves.

84. 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.

The thrust of this article is that controlled processing can set up enabling information in short-term store that can switch to different sets of automatic productions. Thus control strategy can help direct the execution of automatic mechanisms. "The first function of controlled processing is the maintenance of strategy information in short-term store to enable sets of automatic productions," (p. 135). "A second function of controlled processing in skilled performance is the maintenance of time varying information in short-term store," (p. 137). "A third function of controlled processing in skilled behavior is problem solving and strategy planning," (p. 137).

85. Schneider, Walter, "Short Overview of CAP1 Simulation." personal communication, July, 1987.

See connectionist papers [86,87].

86. Schneider, Walter and Mark Detweiler, "A Connectionist/Control Architecture for Working Memory." *The Psychology of Learning and Motivation*, Vol. 21, ed. G.H. Bower, New York: Academic Press, 1988, p. 54-119

Combination of neural network model into Schneider's previous automatism work. Interesting comments are made about correspondence between fast weight changes and retroactive interference, slow weight changes and proactive interference, and phases of skill acquisition.

87. Schneider, Walter and Mark Detweiler, "The Role of Practice in Dual-Task Performance: Toward Workload Modeling in a Connectionist/Control Architecture." *Human Factors* 30(5) (October, 1988), p. 539-566.

Schneider and Detweiler apply their connectionist model (enhanced from [86])

to dual-task experimental data. The model is a collection of connectionist modules (visual, auditory, tactile, semantic, spatial, speech, motor, and context are shown, where context is a form of medium-short-term memory - not as short as simple vector buffering - that is short-term in terms of having *fast-changing connection weights).* Internally the models use *priority learning,* and inter-module communication involves the *delta learning rule.* Inter-module communication takes place across an *inner ring;* controlled processing moderates message transmissions across this inner loop, specifically, *when priority from modules is insufficient to achieve automatic processing. Like PRIOPS, automatic processing is high priority, with the capability of deferring or bypassing lower-priority controlled processing.* The context module is capable of temporary context saves of lower-priority processing during high-priority interrupts (p. 547-548).

The paper proposes 5 stages of skill acquisition, with the first three dominated by controlled processing, the last two by automatic processing: 1) controlled comparison from buffered memory, 2) context-maintained controlled comparison (using the context module), 3) goal-state-maintained controlled comparison, 4) controlled assist of automatic processing, and 5) automatic processing.

"A message that was transmitted prior to a positive event ... would be associated *within* the module with a high-priority tag ... Automatic processing occurs when a message associated with a high-priority event is transmitted in the absence of attentive input. This takes place when the local circuit of the priority

tag inhibits the attenuation units transmitting the message ... If the priority tag is high enough, the vector is transmitted out of the current module to the next module. This process then cascades through a series of stages," (p. 552).

The paper observes that exhaustive single-task practice for a task (be it consistent - automatic - or not) does not eliminate additional dual-task problems when the problem is approached with another task. The dual-task situation presents resource contention problems, especially timing of use of critical resources such as the inner communication loop. Seven strategies for overcoming specific dual-task contention problems are: 1) shedding, delaying, and preloading tasks (i.e., scheduling), 2) letting go of unnecessary, high-workload strategies (using sub-optimal solutions to save time), 3) utilizing noncompetitive resources (e.g., using spatial visualization for one task where the independent tasks may have both used semantic), 4) multiplexing transmissions over time (time-sharing), 5) shortening transmissions (to reduce collisions), 6) converting interference from concurrent transmissions (i.e., learning to filter *expected* noise from other, irrelevant messages on the inner loop, and 7) chunking transmissions (send small chunks that mean a lot). The emphasis is on avoiding collisions on the inner communications loop.

88. Shastri, Lokendra, "Connectionism and the Computational Effectiveness of Reasoning." To appear in *Theoretical Linguistics*, 1990.

The paper accompanied a talk by Shastri at Lehigh, Fall, 1989. Shastri contrasts *reflexive* inference for supporting certain cognitive behaviors in real-time,

with the slower and more deliberate *reflective* inference. His connectionist architecture performs both. His definition of *real-time* for reflexive inference is that performance can be no *worse* than *sublinear* in the size of the knowledge base. His model shares PRIOPS notion of non-iterative reaction processing: "If we desire computational effectiveness, our representation should map the domain knowledge into a graph with the following property: Portions of the graph that are relevant to the solution of a reflexive inference problem must be trees or DAGs. Notice that the complete graph need not be a tree or a DAG and only appropriate subgraphs need be so," (p. 3).

After the talk I discussed the non-iterative nature of time-constrained reaction processing with Shastri. His model also has a weak correspondence with PRIOPS' notion of restricted short-term memory in reactive processing, but there is no correspondence to the idea of deferring processing by priority. The model assumes unlimited processor nodes, so all activity - high and low priority - can proceed in parallel. I have found the assumption of unlimited hardware processors, particularly unlimited *potential* processors allocated by performance-time *learning*, to be the biggest drawback of many connectionist architectures, including this one. Moreover, this model makes a one-to-one mapping of concepts to hardware processors, rather than distributing concepts as activation patterns across multiple processors. Despite the fact that an inference path through a directed acyclic network of nodes is constant-bound in length, actual *O(1) implementation* of this

model on limited-processor hardware would require some form of processor-sharing. At that point the notion of priority-based scheduling would become important.

89. 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.

This is continued from Part I. [81]. The general theory discusses long-term store, structural levels of store, automatic processes, thresholds of activation, controlled processes, short-term store, learning, retrieval, and forgetting. Important properties of controlled processes include: a) They are limited-capacity processes requiring attention; b) The limitations of control processes are based on those of short-term store (such as the limited-comparison rate and the limited amount of information that can be maintained without loss); c) control processes can be adopted quickly, without extensive training, and modified fairly easily; d) control processes can be used to control the flow of information within and between levels, and between short-term store and long-term store; e) control processes show a rapid development of asymptotic performance. Common examples of control processes include maintenance or rote rehearsal, coding rehearsal, serial search, long-term memory search, and decisions and strategies of all kinds.

Important properties of automatic processes include: a) They are not hindered by capacity limitations of short-term store and do not require attention; thus

automatic processes often appear to act in parallel with one another and sometimes appear to be independent of each other; b) some automatic processes may be initiated under subject control, but once initiated all automatic processes run to completion automatically; c) they require considerable training to develop and are most difficult to modify, once learned; d) their speed and automaticity will usually keep their constituent elements hidden from conscious perception; e) they do not directly cause new learning in long-term store (though they can indirectly affect learning through forced allocation of controlled processing); performance levels will gradually improve over trials as the automatic sequence is learned. See [90].

90. 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.

This is a summary of automatism (see articles by Schneider, et. al). An overall definition for automatism is given by two rules: Rule 1. Any process that does not use general, nonspecific processing resources and does not decrease the general, nonspecific processing capacity available for other processes is automatic. Rule 2: Any process that always utilizes general resources and decreases general processing capacity whenever a given set of external initiating stimuli are presented, regardless of a subject's attempt to ignore or bypass the distraction, is automatic. A process that satisfies either Rule 1 or Rule 2 is automatic, but some automatic processes may not satisfy either rule. A discussion is made of the often intermixed nature of controlled and automatic processing sequences, making them difficult to

disentangle at times.

91. Stillings, Neil A., Mark H. Feinstein, Jay L. Garfield, Edwina L. Rissland, David A. Rosenbaum, Steven E. Weisler and Lynne Baker-Ward, *Cognitive Science, an Introduction.* Cambridge, MA: MIT Press, 1987.

Background reading in cognitive science.

92. 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.

An extension to Gupta's research on parallelism above, this paper suggests that speedups in *learning* production systems may be greater than the 10 to 20 limit suggested earlier by Gupta for non-learning systems. Implementation discussions include examination of the mechanisms for adding productions at run-time, and the update of learned Rete nodes with the contents of working memory immediately after learning.

93. Tambe, Milind 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.

This paper discusses the problems of *large cross-product matching time* resulting from overly general *multi-attribute chunks,* and explores using more specific *unique-attribute chunks* to speed matching at the cost of generality. PRIOPS automatic productions *MUST uniquely match tokens,* since automatic Rete uses unit-size registers to store tokens.

94. Ungar, David 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.

This article discusses problems with existing generation scavenging systems. The major problem involves a tradeoff between early tenuring of objects (promotion to infrequently scavenged generation(s)) in order to reduce scavenging overhead (particularly copying), and late tenuring in order to reduce tenuring of garbage (with the resulting space loss). The paper promotes a dynamic, feedback-based tenuring policy, in contrast to the typical tenuring policy of "tenure any object passing a threshold age." The two-part dynamic policy is:

1) *feedback mediation* - after a scavenge, if few objects in the early generation(s) have survived the scavenge, set tenure age high (or infinite) to avoid tenuring; if many objects survive, set tenure age low to avoid scavenge copying.

2) *demographic information* - after a scavenge, if the aggregate surviving data size exceeds the (copy related) pause time threshold (requiring tenuring age to be reduced), search back through data sizes by age to determine the appropriate new tenure age (i.e., only tenure what NEEDS to be tenured to meet scavenge time constraints).

An additional improvement over standard generation scavenging is storing of large and long-lived data objects (typically display bit maps and text in the test applications) outside of the scavenged area; only relatively small headers for these data are scavenged, reducing overhead.

These modifications show improvement for interactive applications in the authors' empirical tests, but the results for non-interactive applications are

uncertain.

95. White, Stephanie M. and Jonah Z. Lavi, "Embedded Computer System Requirements Workshop." *IEEE Computer,* Vol. 18, No. 4 (April, 1985), p. 67-70.

One conclusion of this workshop is that a finite state machine model generates an excessive number of states, and that structuring / partitioning based on the system environment is necessary. Contrast with [18]. Strict functional hierarchy and dataflow models were also labelled as insufficient, since embedded systems are heavily time- and event-dependent. One machine model selected as appropriate is a set of cooperating finite state machines whose interaction with the environment is explicitly modelled. Fault detection and recovery is important. Where exact performance specifications are lacking, inexact information should be tagged and evaluated as such (see [56] on approximate processing). The authors recommend that an embedded computer system abstract model should be developed. The model should include all properties necessary to describe the embedded computer system, should be consistent across all system life cycle phases, and should be described in a language of objects, relationships, and attributes.

96. Winograd, Terry, *Language as a Cognitive Process,* Volume I: Syntax. Reading, Ma: Addison Wesley, 1983.

Background reading in computational linguistics.

97. Winston, Patrick Henry, *Artificial Intelligence,* Second Edition. Reading, Ma: Addison Wesley, 1984.

Background reading in artificial intelligence.

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

Wirth discusses the progression of sequential programs, multiprograms, and "If we depart from this rule (execution time independence) and let our programs' validity depend on the execution speed of the utilized processors, we enter the field commonly called 'real-time' programming," (p. 577). "The essential point here is that the set of concepts (for reasoning) and facilities (for description) of real-time programming should be small extensions of those governing multiprogramming, which in turn should be small extensions of those used in sequential programming," (p. 578). Wirth recommends confining time-dependent program parts to device processes, which is not possible for time-dependent higher level data transformations. His language of choice is Modula. The article ends by casting a vague vote for multiprocessing to cure the ills of time constrained processor sharing.

99. Zisman, Michael D., "Use of Production Systems for Modeling Asynchronous Concurrent Processes." *Pattern-Directed Inference Systems*, ed. D. A. Waterman and Frederick Hayes-Roth, New York, NY: Academic Press, 1978, p. 53-68.

The major difficulty in using traditional production systems to model asynchronous concurrent processes is that a complex control structure must be embedded in working memory. This paper investigates the possibility of using a separate explicit control structure, based on Petri nets. Dynamic creation of subordinate control nets and associated working memories is proposed as method

to allow structuring of short term memory.

Appendix A: PRIOPS Syntax and Semantics

This appendix gives the LL(1) grammar, lexical and semantic notes for the present implementation of PRIOPS. I assume that the reader has written and executed at least simple OPS5 programs. Given the OPS5 background of PRIOPS, I advise any interested reader to study and work with OPS5 [15,21,26] before beginning programming with PRIOPS. This appendix discusses only changes to OPS5 not discussed elsewhere in the thesis.

PRIOPS LL(1) Grammar ("/\" signifies empty):

```
        START
1               ::=     COMMAND MORE-COMMANDS

        MORE-COMMANDS
2.1             ::=     COMMAND MORE-COMMANDS
2.2             ::=     /\

        COMMAND
3               ::=     ( TOP-LEVEL )

        TOP-LEVEL
4.1             ::=     strategy CONFLICT
4.2             ::=     predicate symbolic-atom BTYPE*
4.3             ::=     function BTYPE symbolic-atom BTYPE*
4.4             ::=     action symbolic-atom BTYPE*
4.5             ::=     set SET-NAME symbolic-atom symbolic-atom*
4.6             ::=     structure symbolic-atom FIELD*
4.7             ::=     make symbolic-atom INITIAL-FIELD*
4.8             ::=     remove TIMETAGS
```

4.9 ::= openfile symbolic-atom symbolic-atom
symbolic-atom
4.10 ::= closefile symbolic-atom symbolic-atom*
4.11 ::= edit symbolic-atom
4.12 ::= shell
4.13 ::= load symbolic-atom
4.14 ::= dump symbolic-atom
4.15 ::= call symbolic-atom CONSTANT*
4.16 ::= run CYCLES
4.17 ::= wm integer*
4.18 ::= ppwm PPWMATCH
4.19 ::= cs
4.20 ::= matches symbolic-atom symbolic-atom*
(matches not implemented)
4.21 ::= watch WLEVEL
4.22 ::= trace symbolic-atom*
4.23 ::= pbreak symbolic-atom*
4.24 ::= excise symbolic-atom symbolic-atom*
4.25 ::= reset
(excise and reset not implemented)
4.26 ::= exit
4.27 ::= stderr symbolic-atom
4.28 ::= p symbolic-atom PRIORITY LHS -> RHS

CONFLICT
5.1 ::= mea
5.2 ::= lex
5.3 ::= /\

BTYPE
6.1 ::= ATYPE
6.2 ::= set SET-NAME

ATYPE
7.1 ::= int
7.2 ::= float
7.3 ::= symbol

SET-NAME
8 ::= symbolic-atom

FIELD

9.1 ::= ATYPE symbolic-atom CONST-SUBSCRIPT
9.2 ::= set SET-NAME symbolic-atom

CONST-SUBSCRIPT

10.1 ::= ^ integer
10.2 ::= /\

INITIAL-FIELD

11 ::= ^ CONST-FIELD-REF CONSTANT

CONSTANT

12.1 ::= symbolic-atom
12.2 ::= number
12.3 ::= [symbolic-atom*]

CONST-FIELD-REF

13 ::= symbolic-atom CONST-SUBSCRIPT

TIMETAGS

14.1 ::= integer integer*
14.2 ::= "*"

CYCLES

15.1 ::= integer
15.2 ::= stop
15.3 ::= continue
15.4 ::= /\

PPWMATCH

16.1 ::= symbolic-atom INITIAL-FIELD*
16.2 ::= INITIAL-FIELD*

WLEVEL

17.1 ::= 0
17.2 ::= 1
17.3 ::= 2
17.4 ::= 3
17.5 ::= 4

PRIORITY

```
18.1            ::=   integer
18.2            ::=   /\

      LHS
19              ::=   POSITIVE-CE CE*

      CE
20.1            ::=   POSITIVE-CE
20.2            ::=   NEGATIVE-CE

      POSITIVE-CE
21.1            ::=   FORM
21.2            ::=   { LABELED-FORM }

      LABELED-FORM
22.1            ::=   variable FORM
22.2            ::=   FORM variable

      NEGATIVE-CE
23              ::=   - FORM

      FORM
24              ::=   ( symbolic-atom LHS-TERM* )

      LHS-TERM
25              ::=   ^ VAR-FIELD-REF LHS-VALUE

      VAR-FIELD-REF
26              ::=   symbolic-atom VAR-SUBSCRIPT

      VAR-SUBSCRIPT
27.1            ::=   ^ VAR-INDEX
27.2            ::=   /\

      VAR-INDEX
28.1            ::=   integer
28.2            ::=   variable

      LHS-VALUE
29.1            ::=   { RESTRICTION RESTRICTION* }
29.2            ::=   RESTRICTION
```

RESTRICTION

30.1 ::= << CONSTANT CONSTANT* >>
30.2 ::= PREDICATE ATOMIC-VALUE
30.3 ::= ATOMIC-VALUE
30.4 ::= (symbolic-atom ATOMIC-VALUE*)

ATOMIC-VALUE

31.1 ::= VAR-OR-CONSTANT
31.2 ::= [SETVAL*]

VAR-OR-CONSTANT

32.1 ::= symbolic-atom
32.2 ::= number
32.3 ::= variable
32.4 ::= SYMBOL-MACRO (not in initial PRIOPS)

SETVAL

33.1 ::= symbolic-atom
33.2 ::= variable (not in initial PRIOPS)
33.3 ::= SYMBOL-MACRO (not in initial PRIOPS)

SYMBOL-MACRO (not in initial PRIOPS)

34 ::= @ SYMBOLS MACRO-TAG @

SYMBOLS

35.1 ::= [symbolic-atom symbolic-atom*]
35.2 ::= set SET-NAME

MACRO-TAG (not in initial PRIOPS)

36.1 ::= variable
36.2 ::= /\

PREDICATE

37.1 ::= =
37.2 ::= <>
37.3 ::= <
37.4 ::= <=
37.5 ::= >=
37.6 ::= >

RHS

```
38              ::=   ACTION ACTION*

      ACTION
39              ::=   ( ACT )

      ACT
40.1            ::=   make symbolic-atom RHS-FIELD*
40.2            ::=   remove ELEMENT-KEY
40.3            ::=   modify ELEMENT-KEY RHS-FIELD*
40.4            ::=   halt
40.5            ::=   bind variable RHS-TERM
40.6            ::=   build RHS-TERM
40.7            ::=   excise RHS-TERM RHS-TERM*
                      (build and excise not implemented)
40.8            ::=   call symbolic-atom RHS-TERM*
40.9            ::=   write RHS-TERM RHS-TERM*
40.10           ::=   writeh RHS-TERM RHS-TERM*
40.11           ::=   writeq RHS-TERM RHS-TERM*
40.12           ::=   openfile RHS-TERM RHS-TERM RHS-TERM
40.13           ::=   closefile RHS-TERM RHS-TERM*

      RHS-FIELD
41              ::=   ^ VAR-FIELD-REF RHS-TERM

      ELEMENT-KEY
42.1            ::=   number
42.2            ::=   variable

      RHS-TERM
43.1            ::=   ATOMIC-VALUE
43.2            ::=   ( FUNCTION )

      FUNCTION
44.1            ::=   genatom
44.2            ::=   read RHS-TERM
44.3            ::=   newline RHS-TERM
44.4            ::=   compute EXPRESSION
44.5            ::=   BTYPE RHS-TERM
44.6            ::=   USER-FUNCTION

      USER-FUNCTION
```

```
45              ::=     symbolic-atom RHS-TERM*

        EXPRESSION
46.1            ::=     number EXPR-REST
46.2            ::=     variable EXPR-REST
46.3            ::=     symbol EXPR-REST
46.4            ::=     ( EXPRESSION )

        EXPR-REST
47.1            ::=     OPERATOR EXPRESSION
47.2            ::=     /\

        OPERATOR
48.1            ::=     +
48.2            ::=     -
48.3            ::=     *
48.4            ::=     /
48.5            ::=     %
48.6            ::=     |
48.7            ::=     &
48.8            ::=     ^
48.9            ::=     <<
48.10           ::=     >>
```

PRIOPS Lexical Considerations:

Basic types for PRIOPS include long integers (32 bits signed), single precision floats (32 bits), and symbols (represented as char *).

An unquoted symbol starts with an alphabetic character and contains alphanumerics, underlines and dashes. All unquoted symbols are converted to lower case. For symbols which contain other characters, use the quoted symbol notation. Quotes (") surround a quoted symbol. It may contain any printing character, and it looks like a C string constant. The symbol may contain an escape

sequence of a *backslash* followed by one of the following characters (as in C): n (newline), r (carriage return), t (horizontal tab), b (backspace), f (form feed), and " (escaped "); these are converted into the corresponding control characters from the source file or on input using "read." Two consecutive backslashes yield a literal backslash. Upper case letters are not converted to lower case in quoted symbols. A backslash at the end of a line within a quoted symbol signals continuation of the string. No control characters except horizontal tab and the bell may appear in a quoted symbol. Unquoted and quoted symbols are both considered lexically as symbols. Each unique symbol is stored only once, so that symbol comparisons involve pointer comparisons.

No control characters except newline, carriage return, horizontal tab and bell (ASCII 7) may appear in a source file.

<Variables> are encased in "<" and ">". Variable names start with alphabetics and may contain alphanumerics, underlines and dashes. Upper and lower cases are distinct and are preserved in variable names.

Integer and real constants must start with a numeric character.

Hexadecimal integer constants start with a "!" (e.g., !A is 10), and may contain 0-9, A-F, a-f. They are internally represented as standard 32 bit integers.

PRIOPS Semantic Considerations:

Numbers at the start of each paragraph refer to the LL(1) grammar above.

1, 2 and 3: The command loop is interpretive. During interactive command interpretation both the controlled and automatic partitions are quiescent (no matching is performed). To get preliminary matches of working memory changes entered from command mode, perform "(run 0)." This completes all pending matching and returns to the command loop.

4.1 and 5: Strategy here applies to controlled partition conflict resolution. Explicit priority is considered first at conflict resolution time, followed by OPS5 mea or lex criteria. The automatic partition uses priorities at match time to defer portions of matching. Conflict resolution within the automatic partition is priority first, age of memory elements second (older elements have higher priority), and specificity last. The default strategy is lex as in OPS5.

4.2: User defined predicates and their parameter types are declared here. The named C function must expect these argument types and return int. C predicates return zero for false, non-zero for true, and are called within LHS tests (see 30.4 below). Argument types must match parameter types declared here. An implicit first argument supplied to the predicate is the field against which the predicate is applied (field test where the predicate is written in 30.4). A float is passed as a 4-byte float, an int as a long int, a symbol as a (void *), and a set as a pointer to an array of unsigned shorts (see 4.5) that represent that set. Arguments are passed on PRIOPS' evaluation stack, and are read using macros defined in file "external.h". Predicates (and user-defined functions and actions as well) must NOT modify their

arguments.

4.3: User defined functions, used in RHS calculations. Similar to user defined C predicates, these C functions return one of double, long int, char * (for symbols), and unsigned short * (for symbol sets, see 4.5). When returns types are symbols or symbol sets (returned as char * and unsigned short * respectively), the actual character and unsigned short arrays to which these pointers point, should be static or heap arrays accessible to the C functions (i.e., not auto arrays in the called functions). Upon return from the functions PRIOPS will copy the arrays into its own data space; consequently the next time such C functions are called, the C arrays are accessible for reuse.

4.4: These are C functions returning void. They accept arguments similar to user defined predicates and functions, and are called for side effects. One use is as output port drivers.

4.5: This is a symbol set declaration. After the set name comes one or more symbols which make up the universe of this set type. Sets are represented as bits vectors contained in arrays of unsigned shorts. Assume that an unsigned short contains 16 bits (the case for the 8086/8088 family). The first 16 symbols go into bits 0 through 15 of the first unsigned short in the array, with the first symbol in bit 0. Likewise the seventeenth symbol goes into bit 0 of the second unsigned short. Since set size is determined from 4.5, all sets of this type take identical storage. When passing sets to or returning them from C functions, the pointer to

this array of unsigned shorts is actually passed. The order of declaration is therefore important. Symbols sets are useful in determining mutual exclusion of test paths. The symbol to bit position mapping is maintained during program execution; symbol variables placed into set fields or compared to sets are converted into the correct set type with the symbol's bit set to 1 implicitly. Furthermore sets can be read and written as [symbol1 symbol2 "QUOTED-SYMBOL3" ...] by the read and write functions; [] represents the empty set.

4.6: Structures correspond to C structures, and are made up of named fields of type int, float, symbol, and user-defined symbol sets. The first three, atomic types may be vector fields up to 1000 in length (as per 9.1). The total size required by a structure may not exceed 10000 bytes.

4.7: Make a structured type working memory element, filling in the named fields with the correct types. As in all cases where needed, integers and floats are coerced automatically where needed. Uninitialized int fields are set to 0, uninitialized float fields to 0.0, uninitialized symbol fields to the symbol "nil", and uninitialized set fields to the empty set [].

4.8: Remove working memory elements with the corresponding time tags, or all for "*". Timetags can be seen using "wm."

4.9 and 40.12: Open a file. As per OPS5, the first argument is a symbolic atom used as the program's name for the file. The second argument is the file system's name for the file. The third argument is "in", "out", or "append".

4.10 and 40.13: Using the program's names for open files, close them.

4.11: If the environment variable PREDIT is set to the path for a valid executable file, the DOS will be invoked to execute that file with the symbolic atom as its argument. This is intended to be used for editing source files. Upon completion of editing control is returned to PRIOPS command loop.

4.12: Use DOS COMSPEC environment variable (if set) to spawn a DOS shell.

4.13: Take command interpreter input from the named file; used to compile source files and execute command files. Load commands may be nested up to 10 deep within files.

4.14: Dump working memory to the named file. Memory is dumped within a sequence of "make" statements so that the named file can restore the memory elements using "load." Working memory elements are dumped in order of creation (i.e., oldest first) so that, upon subsequent loading, the memory element recency relationships are maintained.

4.15: Call the user-defined C procedure for side effects.

4.16: Execute the inference engine. If CYCLES is a positive integer, then execution continues until CYCLES rule firings have completed. Both control and automatic rule firings are counted, and control rules which are preempted and retracted from the conflict set during firing are added to the count, but are not subtracted during retraction. If CYCLES is "stop" (or equivalently is not specified), execution continues until both conflict sets are empty. If CYCLES is

"continue", then control will not be returned to the command interpreter on empty conflict sets. Instead, the inference engine will wait for incoming interrupts to trigger rules. In all cases executing a "halt" instruction will return control to the command interpreter. A control-C will likewise return control after the currently executing rule is completed. Both the automatic and controlled partitions are inactive during top-level command interpretation. When CYCLE exhaustion, halt, or keyboard control-C occur within an interrupt handler, the halt defers until interrupt processing completes; the halt then occurs.

4.17: Display working memory elements.

4.18: Pretty print matching working memory elements.

4.19: Print the conflict set.

4.20: Show intermediate matching information for named productions.

4.21: Establish a watch level. Watch 0 performs no tracing and prints no statistics; this is the default. Watch 1 prints summary statistics for both partitions when returning to the command interpreter. Watch 2 displays changes to the controlled partition’s working and production memories. Watch 3 displays controlled partition conflict set changes in addition to the above. Watch 4 prints traces of automatic partition firings and conflict set changes in addition to the above. Note that any tracing, and automatic tracing in particular, slow execution speed; these traces can affect timing-dependent processing. No watch reporting occurs within interrupt-handler instantiations.

4.22: Trace toggles the trace status of a rule. Trace on for a rule traces when the rule instantiation enters the conflict set, leaves the conflict set, and fires. Trace with no arguments lists names of all rules with trace toggled on; default is trace off. Both controlled and automatic rules can be traced (but see the warning in 4.21 above). No tracing occurs within interrupt-handler instantiations.

4.23: Pbreak toggles the break status of the rule. With pbreak on control returns to the command interpreter when the rule is about to fire. Pbreak with no arguments lists names of all rules with pbreak on. Pbreak processing applies to both controlled and automatic rules. Pbreak within interrupt-handler instantiations is deferred until all interrupt handling completes

4.24: Excise removes rules from production memory. Unlike OPS5, rule storage is reclaimed.

4.25: Remove contents of working and production memory for both partitions. All data type declarations are removed and open files are closed. Executing "reset" is equivalent to starting a new PRIOPS session.

4.26: Exit PRIOPS, returning to the invoking process. Interrupt vectors are restored to values held when PRIOPS was entered.

4.27: Send stderr messages (including all trace information) to the named file; the name is the DOS file system path name, not an internal name. The command (stderr stderr) returns standard error output to the console.

4.28: Compile a production. Priority may vary from -128 to 127; if not stated, priority defaults to 0. All priorities > 0 place the production in the automatic partition, where all matching memory nodes are unit size and statically allocated, and production priorities defer portions of matching.

12.3: Constant symbol set.

16.1 and 16.2: Optional constant restrictions define the wme's to be pretty printed. When the structure type name is missing, field matching is performed across all structure types. No restrictions (i.e., "(ppwm)") prints all wme's. Set fields may only be specified when a structure type name follows ppwm; matching for the set field succeeds if the intersection of the set constant and matched field is non-empty (i.e., if at least 1 element from the ppwm pattern is in the set), OR if both sets are empty.

The LHS of rules corresponds closely to OPS5 syntax. Variables and element variables occupy the same name space, so an entire memory element cannot have the same variable tag as a field variable name. Element variables can be used as integers in the RHS of rules; when so used, they represent the time tag of the working memory element to which they are bound.

30.4: Call to a user-defined C predicate. There is always an implicit first argument, the working memory field in which the predicate call appears.

31.2 and 33: A set value may consist of the empty set [], or any combination of [symbolic-constant <variable>], where the variables are bound to symbolic

atoms contained in the universe of that set type. The symbol macro of 33.3 is expanded to a symbol constant as discussed next. A symbol <variable> may NOT be contained in an automatic production's symbol set test. Such variables should be resolved into symbols sets in controlled productions, which pass the resolved symbols sets in symbol set variables to the automatic productions which use them.

32.4, 33.3, 34, 35, and 36: Symbol macros cause expansion of multiple productions. The enclosing textual production is known as the "base production." When a symbol macro is encountered, the base production is cloned into a number of macro productions whose priorities and Rete nets are identical to the point of macro expansion. Each macro production gets exactly one CONSTANT symbol at the place of macro expansion; this symbol may be inside a symbol set (31.2), or may be in a symbol field (32.4 when expanded is equivalent to 32.1). The base production becomes undefined, and the names of the new productions become the base name with -SYMBOL appended, where SYMBOL is the constant symbolic-atom chosen for that production from the list. For example, if production "read-sensor" contains the test "^ sensorid @ [a b c] @", production read-sensor becomes undefined, and productions read-sensor-a, read-sensor-b, and read-sensor-c are defined and compiled in parallel. The first contains the test "^ sensorid a" in the place of the above symbol macro. Note that symbol macros may be expanded inside of symbol set field tests. Testing "^ family-set [brother sister <relation> @ [mother father] @]", gives two productions, one testing "^ family-set [brother

sister <relation> mother]", and the other "^ family-set [brother sister <relation> father]". Within the @ @ macro enclosure, the symbols supplied to the macro may be the universe of a symbol set type instead of enumerated symbols (35.2 instead of 35.1). In the latter case a production is generated for each symbol in the universe of that set type. If the optional MACRO-TAG is included (34) using variable notation, that variable represents the expanded symbol constant throughout the rest of that production. If a test with production "p1" is "^sensorid @ [a b c] <id>@", then for "p1-a" variable "<id>" represents the constant "a" wherever used. It is important to note that these tags represent constants, not variables whose contents are unknown until run-time. Macro tags may be used any place that a symbol constant can be used (for example, inside an automatic production's test of a symbol set, where symbol variables cannot be used). If multiple @ @ macro expansions are encountered, the cloning process is repeated for each of the already cloned productions. At each expansion, the number of productions being compiled is multiplied by the number of constant symbols within the macro call, and all base productions become undefined.

37: Ints and floats may be intermixed, with the former coerced to the latter when needed. For symbols and symbol sets, = and <> are self-explanatory; they can be used in any production. For symbols, <, <=, >= and > may be used to test lexicographic order WITHIN CONTROLLED PRODUCTIONS ONLY. For symbol sets, <= tests "is a subset", < tests "is a proper subset", >= tests "is a

superset", and > tests "is a proper superset". These set tests can be used in any production.

40: THE FOLLOWING ACTIONS MAY NOT BE USED IN AUTOMATIC PRODUCTIONS: build, excise, write, writeh, writeq, openfile, and closefile. Output from the automatic partition is accomplished through device drivers as explained later. Modify is equivalent to remove followed by make as in OPS5. 40.12 and 40.13 correspond to 4.9 and 4.10 in the command interpreter.

40.5: Bind uses the program context to figure out the type of the bound object, but this is not possible for situations such as "(bind <myvar> (read <myfile>)". These cases default to type symbol, but can be explicitly coerced: "(bind <myvar> (int (read <myfile>)))" OR "(bind <myvar> (set animals (read <myfile>)))".

40.6: Build is very different from its OPS5 namesake. The most important distinction is that a production’s building may be dispersed across multiple firings through string concatenation. Build itself is passed one argument, a string (symbol) representing an entire production source text, including the opening "(p" and closing ")". No macro expansion may be used in a built production. When a string variable is supplied as the argument to build, its value will be substituted. See (48) below for using "+" for string concatenation. Only controlled productions may build new productions; the current contents of working memory are matched against the new production immediately upon compilation (unlike OPS5).

40.7: Excise removes the named production(s) and reclaims their storage. An excise, like a build, may result in modifications to priorities of the Rete nets.

40.8: Call a user-defined action.

40.9, 40.10, and 40.11: The standard write expands quoted symbol escape sequences such as backslash-n into their control characters. Writeq surrounds all symbols in quotes, and does not expand escape sequences; writeq is useful for writing files to be read by PRIOPS. Writeh is identical to write, except that integers are written in hexadecimal, preceded by a "!". Symbol sets may be read and written using the "[symbol1 symbol2 "QUOTED.SYMBOL"]" notation discussed in 4.5.

44: THE FOLLOWING FUNCTIONS MAY NOT BE USED IN AUTOMATIC PRODUCTIONS: read and newline. Input in the automatic partition is accomplished through device drivers as explained later.

44.1: Generate a new symbol as in OPS5 genatom.

44.2: Read one field value from the named, open file. A type check is performed upon input.

44.3: Read all input as though a quoted string up to the next end of line, and discard the end of line. If end of line is the next character, a single space is returned as the symbol. No analysis of the input line characters is performed; the characters are stored verbatim.

44.4, 46, and 47: Compute, as in OPS5, is right-associative with no precedence. Precedence can be forced through the use of parentheses.

44.5: This is explicit type coercion. Ints and floats may be freely coerced. Symbols may be coerced to and from ints and floats ONLY IN CONTROLLED PRODUCTIONS. A number is coerced to a string representation using "sprintf." A symbol is coerced to a number using "sscanf," but if some part of the symbol does not represent a number, an error is reported. A symbol beginning with a "!" is taken to contain a hexadecimal integer in the remaining characters. The only time SET COERCION is appropriate is in RHS expressions with constant sets, where the type of the expected set is not obvious. For example, in *"(bind <myset> [dog cat rat]),"* the type of [dog cat rat] may be indeterminate; several universes may contain these set members. To disambiguate constant sets in RHS's, PRIOPS requires *"(bind <myset> (set animal [dog cat rat])),"* Explicit set identification is not needed where the type of set is already determined by a field reference in a working memory element change.

44.6 and 45: A call to a user-defined C function.

48.: Compute operators are standard for floats and ints, with type coercion between the two being implicit. Modulo (%), bitwise inclusive or (|), and (&), exclusive or (^), shift left (<<) and shift right (>>) are binary operators allowed only on integers; floats may not be used. For symbol sets, + signifies union, * is intersection, - is set difference; these may be used in any production. Symbols may

use + for concatenation. Set and symbol compute operations apply to CONTROLLED PRODUCTIONS ONLY.

C Language Interface:

The current version of PRIOPS defines its own main, and PRIOPS cannot be called as a subtask from an enclosing C program. C functions and device drivers can, however, be called from PRIOPS. When compiling PRIOPS, the void function "map_externals()" must be supplied; it takes no arguments. Within "map_externals()", calls to PRIOPS functions c_pred(name,funcptr), c_action(name,funcptr), and c_func(name,funcptr) must be made for all user-defined predicates, actions, and functions (respectively). These are declared in "external.h" and are defined in "external.c"; name is a char *, pointing to the name used in the PRIOPS predicate (4.2), function (4.3), or action (4.4) declaration. funcptr is a pointer to the function associated with that name. "map_externals" is called from "main" before the PRIOPS compilation process begins. A default empty "map_externals()" is supplied in the PRIOPS library for when no external C functions are called.

User defined device drivers must also be initialized before PRIOPS is begun. At present all direct I/O for the automatic partition is through device drivers; OS and BIOS calls cannot be used by the automatic partition, since interrupts can cause automatic processing to preempt the controlled partition while it is in an OS

critical section. No memory management occurs in the automatic partition. Output can be controlled through action calls from automatic productions.

Consult "external.h" and the maze demo C file "primaze.c" for details and examples.

Appendix B: PRIOPS Source File Organization

This appendix identifies source file names, lengths (in number of non-blank lines), and functions for the current PRIOPS compiler. Two makefiles exist, "priops.mk" and "maze.mk". The former is for generic PRIOPS, the latter for the maze program. All compilation was done using Microsoft® C 5.1. PRIOPS comprises about 12,000 lines of C code. The PRIOPS executable file consumes about 200,000 bytes of storage when loaded, not counting the considerable dynamic storage needed to compile and run production system programs.

Header files:

compile.h (52): Command interpreter & production compiler declarations. See compile.c

declare.h (12): Top-level data type function declarations for "predicate," "function," "action," "set," & "structure." See declare.c.

external.h (171): External C function coordination. See declare.c for c_pred, c_action & c_func. See runtime.c for utility functions used by user C functions. Include this in application C code I/O driver files (e.g., drivers.c & primaze.c).

rete.h (533): Rete network node structure declarations and associated priority queue scheduling declarations. See runtime.c and runrete.c.

runtime.h (82): Global declarations needed at run time. Also top-level "run" &

Rete traversal support. See runtime.c.

scanner.h (89): Lexical analyzer header file. Tokens and related definitions. Compiler error logger. See scanner.c.

symbol.h (148): Symbolic atom manager header file. See symbol.c.

wme.h (81): Working memory element (structure) declarations and top-level interpreter wm operation declarations. See wme.c.

Total header file lines: 1168.

C source files:

compile.c (3476): Recursive descent production compiler.

declare.c (717): PRIOPS top-level data declaration symbol table manipulation.

priops.c (688): Program entry, initialization, and top-level interactive command interpretation.

runrete.c (1472): Run-time interpreter for intermediate code.

runtime.c (2393): Run-time Rete traversal and queue handling code, controlled and automatic inference drivers, miscellaneous run-time support.

scanner.c (531): Lexical analyzer for compiler and PRIOPS symbols at run-time.

symbol.c (261): Unique symbol handling functions.

test87.c (34): 8087 math processor test code.

wme.c (1134): working memory element manipulation.

Total lines of ".c" C code: 10706.

priop86.asm (66): assembler, saves/restores 8087 state

Maze files:

maze.pri (940): The PRIOPS part of the demo application. 2 initialization productions, 9 garbage collection, 2 declarative memory update, 31 maze search, 1 controlled move, 1 automatic move, 24 automatic panics, 5 exit reactors, 14 learning productions = 89 productions.

primaze.c (768): The C part of the demo application. 1708 lines of code for the demo.

TOTAL C code lines from above (.h & .c files): 12642.

TOTAL C code lines + assembler + PRIOPS source: 13648

maze.mk (54): Make file for maze demo.

priops.mk (53): Make file for generic PRIOPS.

runmaze.c (2400): A slight variant of runtime.c for the maze demo.

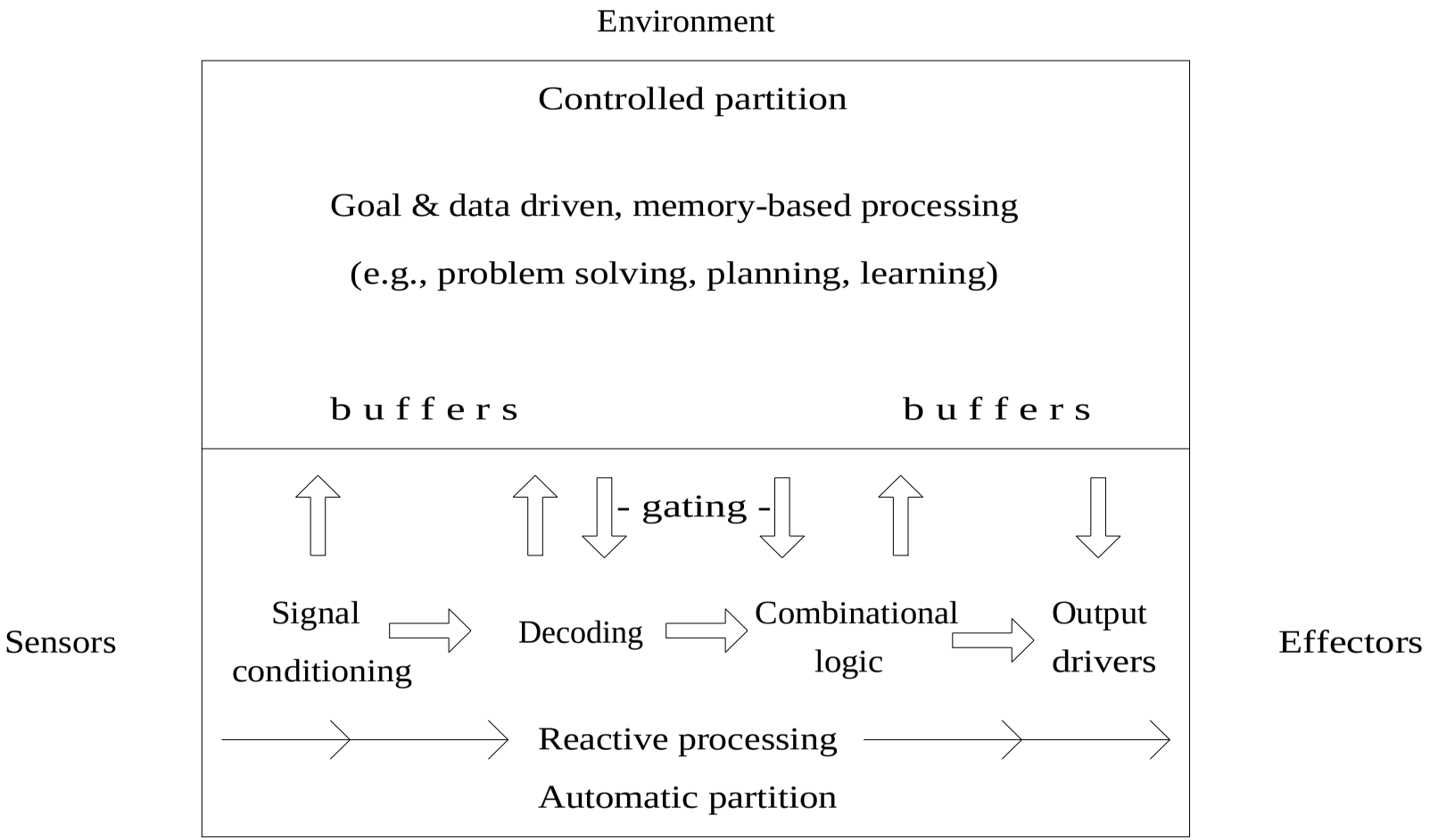


Figure 2 - The two-tiered PRIOPS architecture

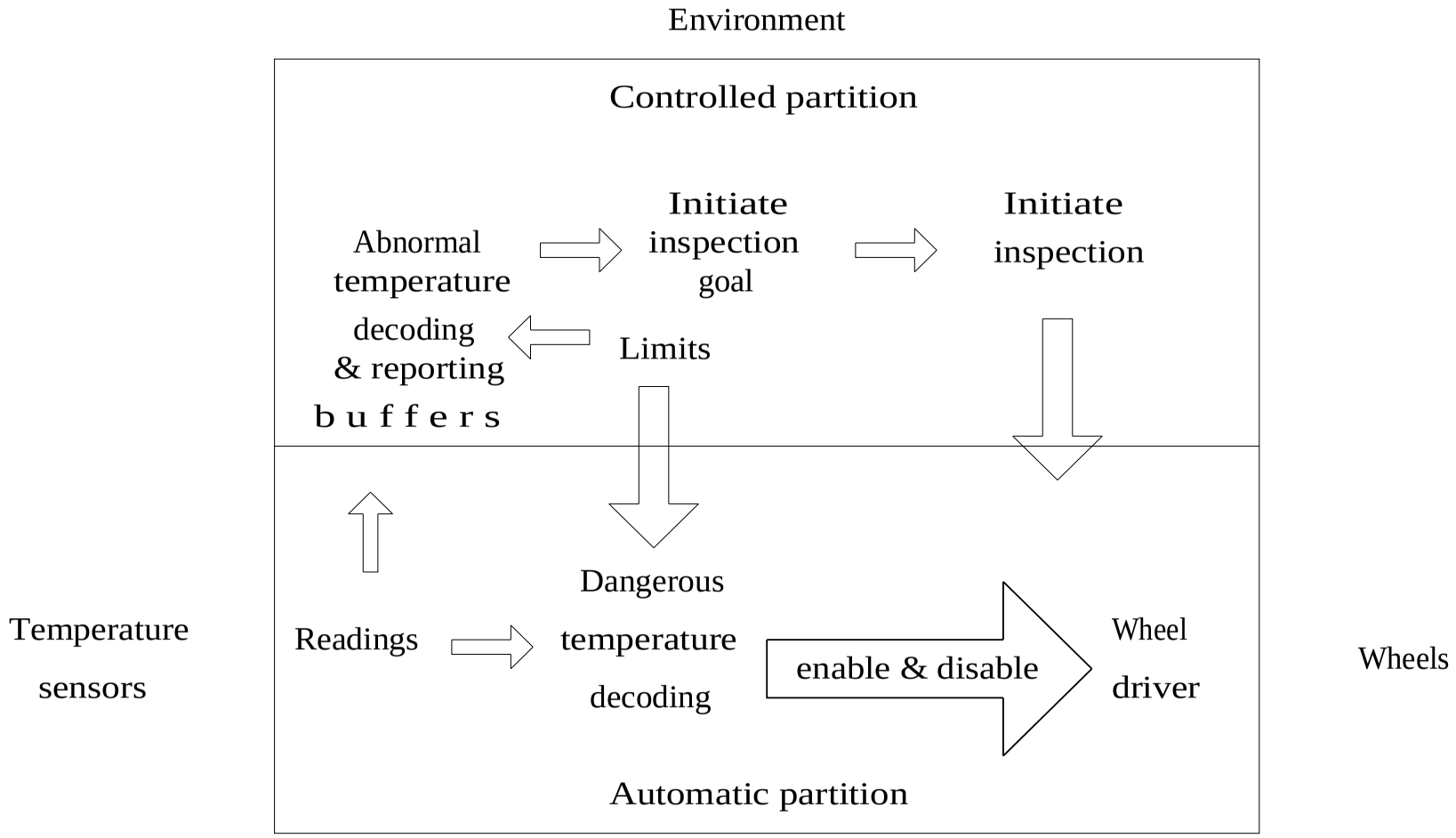


Figure 4 - A temperature sensor driven example

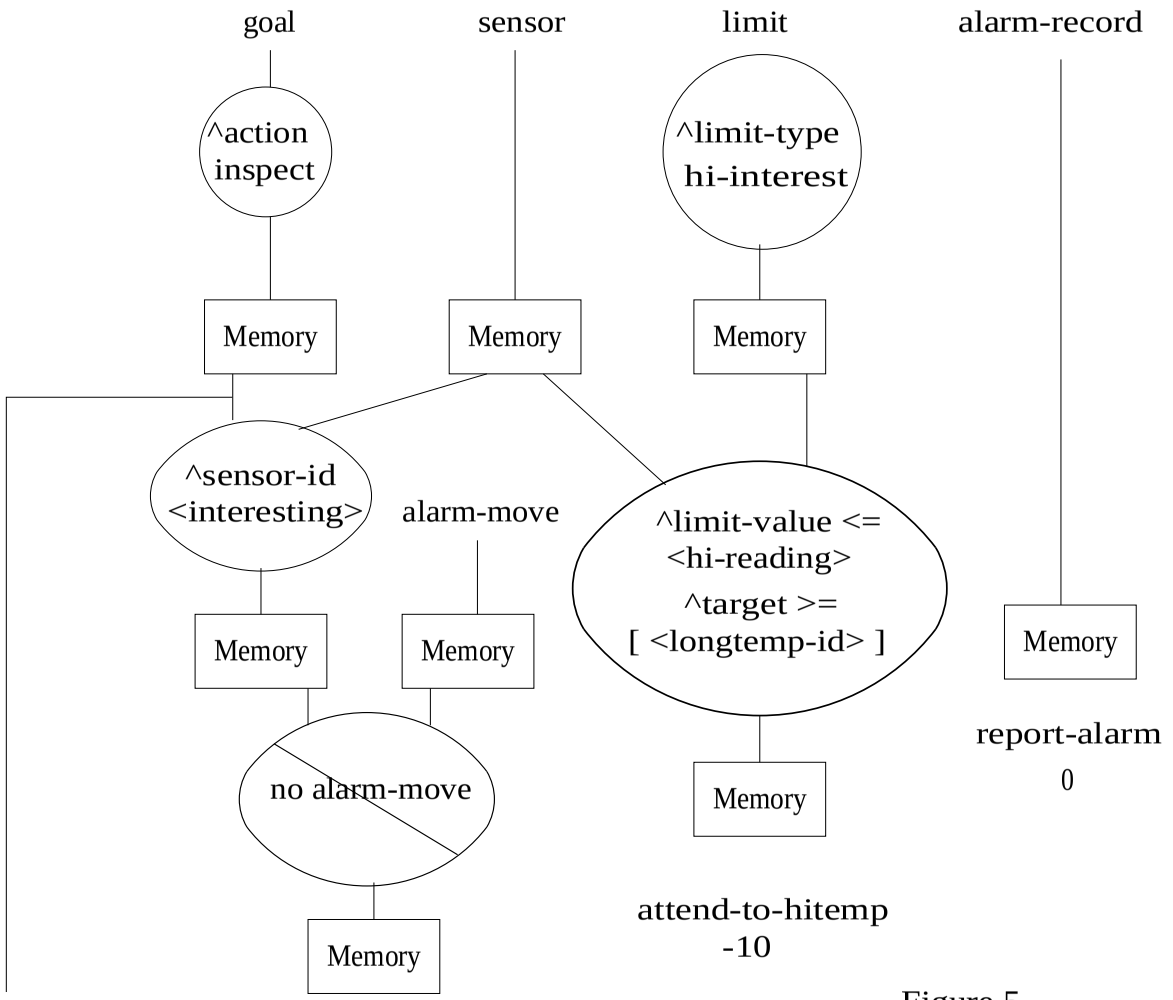


Figure 5

Controlled partition Rete net

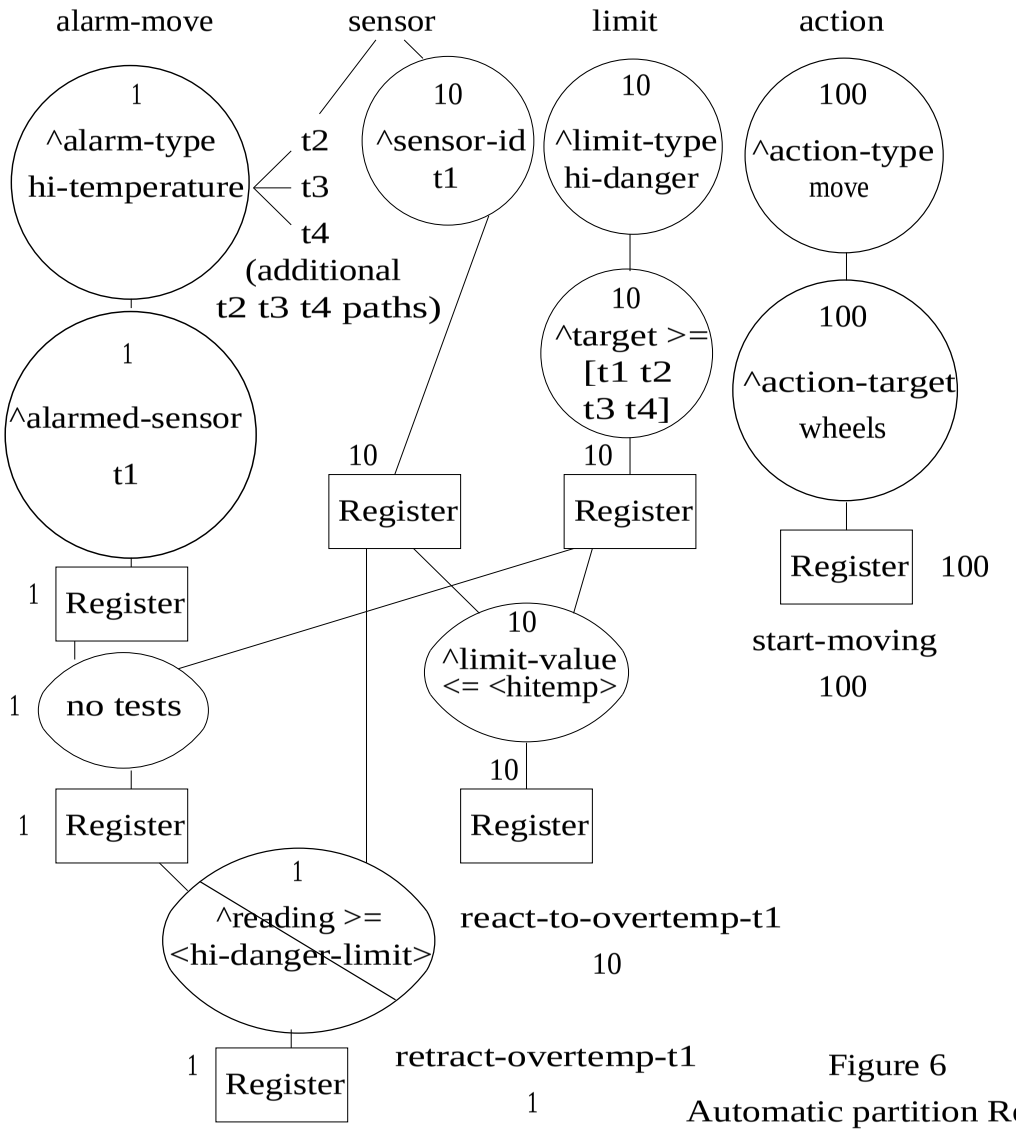


Figure 6
Automatic partition Rete net

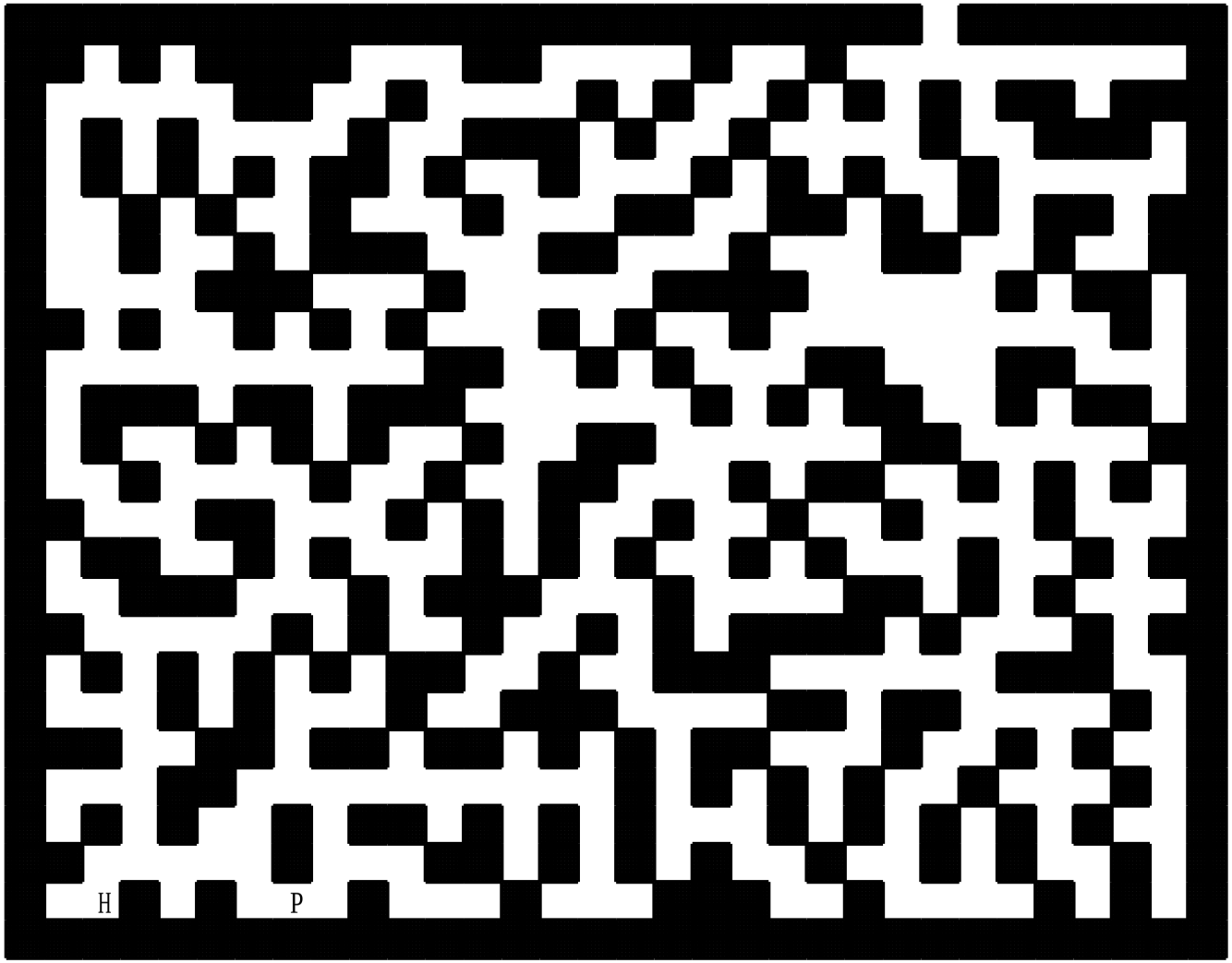


Figure 7 - The maze

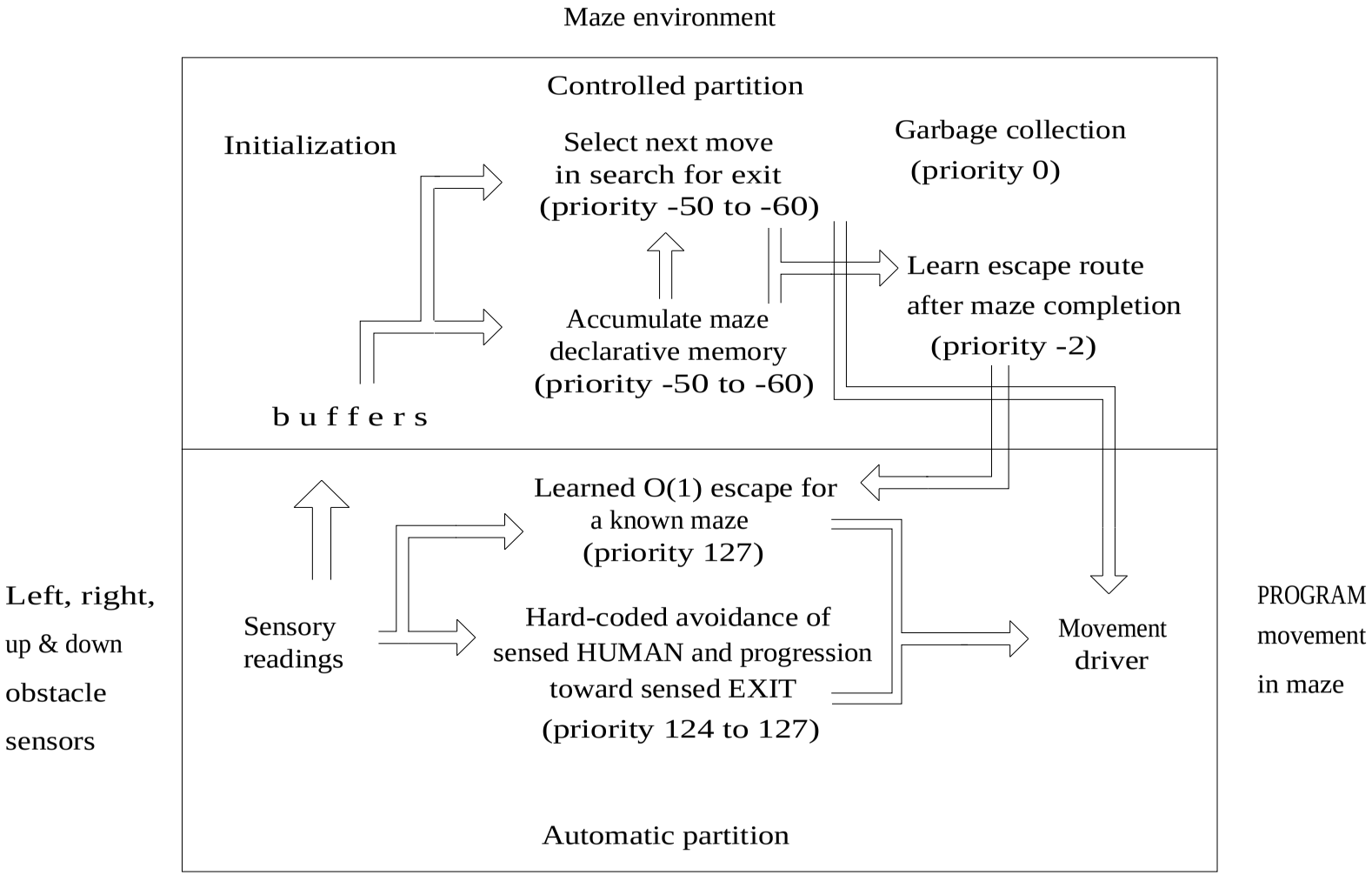


Figure 8 - PROGRAM data flow during maze traversal

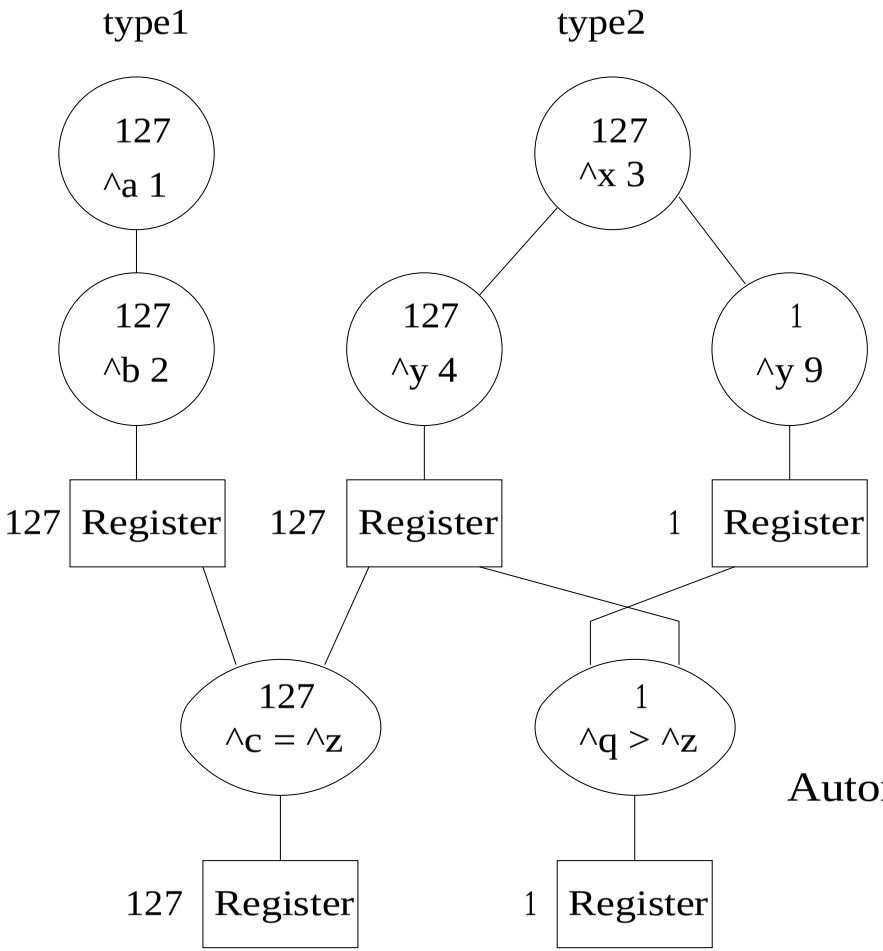


Figure 9

Automatic Rete logical view

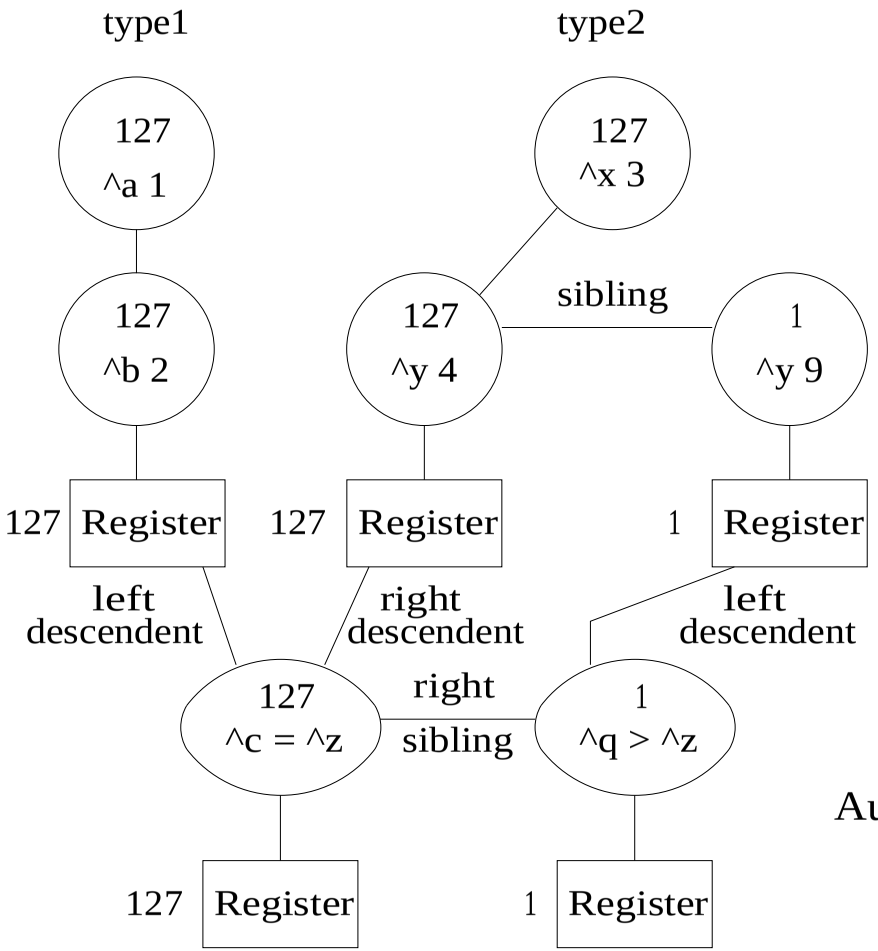
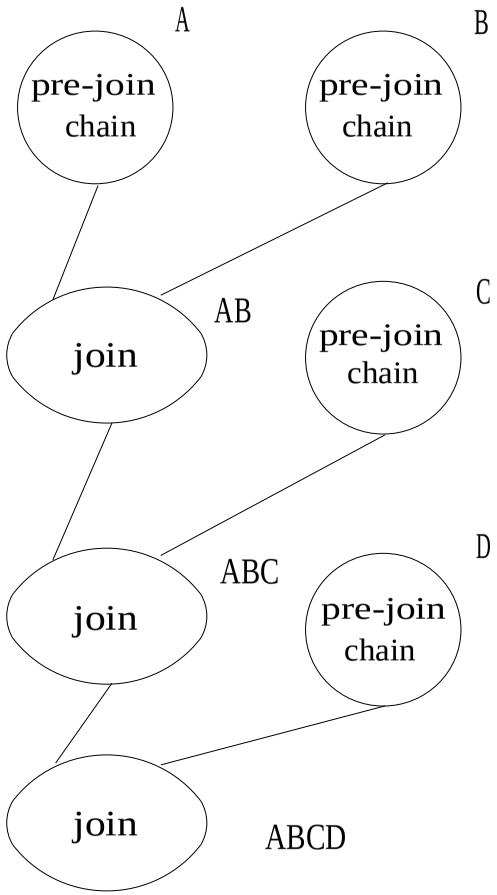


Figure 10

Automatic node linkage



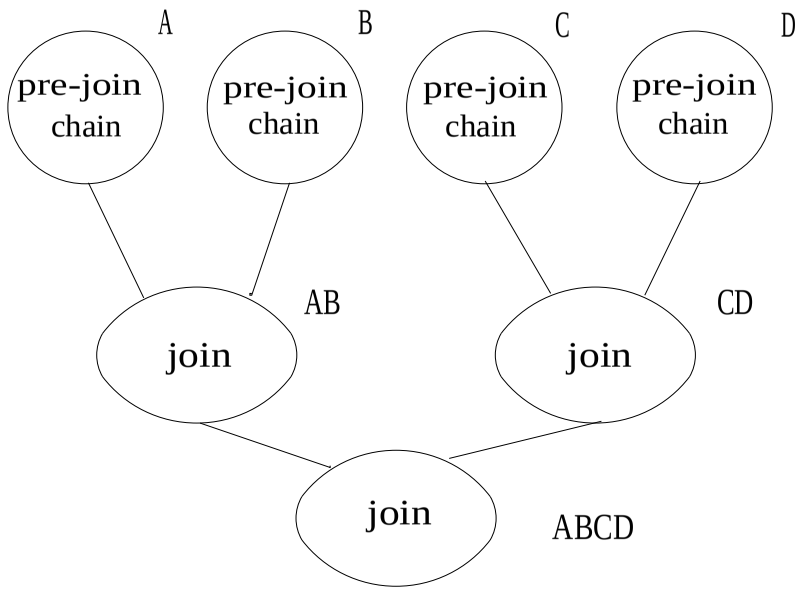
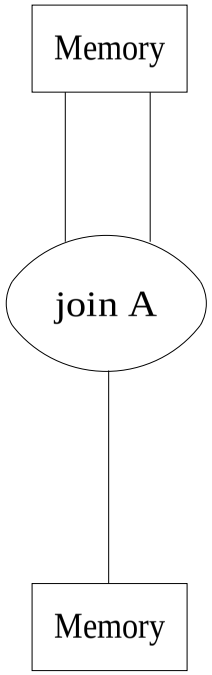


Figure 11

Standard and balanced Rete nets

(p thing2 (thing) (thing) -> RHS)



(p thing3 (thing) (thing) (thing) -> RHS)

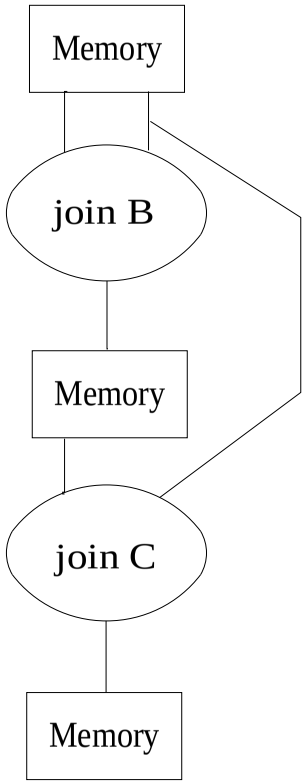


Figure 12

Working memory element self-joins