Contents

Chapter 1

Introduction and Language Overview

OPS5 is a well-known language for implementing rule-based systems. Rule-based languages have been widely used for implementing expert systems, however the performance of such systems has usually left something to be desired. It has been suggested [Gupta, 1987, Ishida and Stolfo, 1985] that the performance limitations of rule-based systems could be overcome by the appropriate use of parallelism. This approach has become increasingly practical with the advent of symmetric multiprocessors and concurrent programming languages. This report describes the second release of UMPOPS, a Lisp-based version of OPS5 developed at the University of Massachusetts which has been modified to support a number of levels of parallel activity: *rule* parallelism which allows rules to be fired concurrently, *matching* parallelism which allows the pattern matching to be performed in parallel, and *action* parallelism which allows individual working memory changes to be made in parallel. Lock-based mechanisms for (partially) ensuring program correctness are provided. A number of language features have been added to the basic OPS5 syntax to allow the implementation of more sophisticated heuristic control functions and constructs for expressing iteration over sets of rule instances which support parallelism. A mechanism for expressing search states as instances of multiple worlds has been provided in order to allow search of multiple spaces to be performed concurrently.

The first chapter of this report gives an overview of the OPS5 language and concepts specific to the parallel implementation. This is followed by a user's manual which describes the invocation and use of the current version of UMPOPS and the modifications to the syntax required for parallelism, set functions, and heuristic control. Although the first part of the second chapter will be of interest only to those readers actually programming in UMass parallel OPS5, the description of new language features incorporated into the language may be of more general interest. The third chapter discusses restrictions on parallel rule firing and continues with tutorial on programming rule-based systems in parallel in the form of an extended analysis of the design and performance of two OPS5 programs. The final section of this report is devoted to discussing the low-level issues involved in implementing a parallel production system based on the Rete network.

1.1 Language Overview

UMPOPS was constructed from a public domain version of OPS5 written by C. Forgy and the syntax of UMPOPS is in many respects identical to that described in the OPS5 Technical Report [Forgy, 1981]. There are, however, a number of exceptions and extensions to the language, both to provide parallel operators and to increase the general expressiveness of the language. This report assumes a familiarity with the use of OPS5, but the following definitions may be useful for those less familiar with the language and its implementation.

1.1.1 Definitions

Working Memory: A production system consists of a set of productions examining a set of facts which describe the current state of the system. In OPS5, this set of facts is called *working memory*. Each fact is represented by a single *working memory element* which consists of a *class* followed by a list of attributes and values. For example, a typical working memory element might have the form

```
(cat ^name Socrates ^color orange ^size large ^weight heavy)
```

Each working memory element is assigned a *timetag* which describes the order in which the working memory elements were created, and serves to uniquely identify each element. Different elements may contain identical values but will be assigned different timetags. Despite their name, timetags do not usually record the actual creation time of a working memory element, however, because creation time is occasionally of interest to the experimenter, UMPOPS has been modified to report the time at which elements are created.

Productions: A production consists of a lefthand side (LHS) which contains a list of patterns to be matched against working memory and a righthand side (RHS) which contains a list of instructions to be executed in the event that the production is fired.

The Lefthand Side: The lefthand side contains a list of *condition elements*. Each condition element consists of a pattern which can match one or more elements in working memory. There must be at least one corresponding working memory element for every condition element in order for the rule to be instantiated (that is, for it to be entered into the conflict set). Condition elements may be negated, in which case the rule only matches if there is *no* working memory element which satisfies the negated condtion element. Condition elements may contain variables; a rule may only fire if there is a set of working memory elements which can generate a consistent set of variable bindings.

The Righthand Side: The righthand side of a production contains the operations to be performed if the rule is fired. This can contain any combination of changes to working memory, input/output statements, or function calls. The execution time of a production is equal to the amount of time required to execute all the statements in the righthand side. In general, studies of parallelism in production systems attempt to reduce this execution time by increasing the speed of the working memory changes.

The Matching Process: *Matching* is the process by which a new or modified working memory element is compared against the lefthand side of all the productions in the system in order to see if any of them are enabled by the latest change to working memory. This matching process is considered to be the most time-consuming aspect of executing a production system and a considerable amount of research has been done to determine if match time can be significantly reduced by performing the match process in parallel [Gupta, 1987].

In OPS5, the matching process takes place when working memory elements are added to, or deleted from, memory. This means that the match process actually takes place at the same time as the righthand execution phase. The implication of this is that the match and execution phase are not actually separate as the conventional description of the production system execution cycle indicates. When operating in parallel, it is important to remember that working memory may still be changing while the match process is taking place.

Conflict Set: The conflict set is the list of all the rules which are eligible to fire. A conflict set entry contains the name of the production, copies of the working memory elements which caused the production to match, a binding list which contains the values of variables bound in the lefthand side of the production, and rating information which may or may not be used when performing conflict resolution.

1.1.2 Levels of Parallelism

The parallel OPS5 currently supports rule-, action-, and match-level parallelism. These levels of parallelism are described below.

Production Parallelism In serial OPS5, only one production can be executed at a time. If more than one rule is eligible to fire, then a single one must be chosen. Parallel OPS5 allows multiple rules from the conflict set to be executed simultaneously.

Match Parallelism: When match parallelism is invoked, the matching process which determines which productions are enabled by a working memory change is carried out in parallel. This reduces the amount of time required for a single working memory change to take place.

Action Parallelism: If a production contains multiple actions in its righthand side, it is possible that they may be able to be executed concurrently, thus reducing the execution time of the production by a factor proportional to the number of actions (assuming all actions take approximately the same amount of time to execute).

1.2 Language Features

Version 2.0 of UMass Parallel OPS5 has the following differences from the version described in version 1.0 of this manual.

- Version 2.0 supports rule-,action-,and match-level parallelism. However there are new language constructs for action- and match-level parallelism and new synchronizing mechanisms to prevent spurious rule executions when running asynchronously.
- All parallel operations are now carried out by *rule demons* rather than by individual *thread* activations; this architecture allows a greater control over when rules, actions, and matches are executed.
- Interference between rule executions is now prevented by means of a simple locking protocol in the case of interactions due to positive condition elements.
- Four rule-firing paradigms are currently supported: parallel asynchronous, parallel synchronous, serial, and parallel asynchronous tasks with optional internal conflict resolution.
- A scheduler/controller package has been added which provides the ability to control rule firings either by pruning or by specifying priorities. The scheduler is heavily instrumented to record timing statistics for rule firings, queue latencies, and lock acquisitions.
- Set-oriented rules are now supported at a simple level. The implications of asynchronous rule execution on set-oriented rule semantics is discussed later in this manual.
- A lefthand side notation for expressing "meta-level" control information is provided.
- Several new righthand side actions, some experimental in nature, have been added in order to support both parallelism and control activities.
- A "multiple worlds" representation has been added to facilitate parallel search.

Other changes to the language which are largely transparent to the programmer include the addition of hash tables to node memories, partial compilation of righthand sides of rules, and multiple changes to the internal representations of various structures such as working memory elements and the conflict set. These and other efficiency changes have yielded approximately a five-fold increase in performance over version 1.0, both running in parallel and serially. Although these modifications are not directly relevant to parallel activity, they are described in appendix A for the benefit of other researchers experimenting with lisp-based versions of OPS5.

1.3 The Rule-firing Architecture

The rule-firing architecture of the system is implemented as shown in Figure 1.1. As each rule instantiation becomes enabled, it is placed on a queue of eligible rules (which replaces the conflict set). The instance is rated according to a rule-specific (and possibly situation-specific) function prior to being placed on the queue. The main process takes each rule instance off the queue in turn and attempts to acquire locks associated with the working

memory elements positively referenced or modified by the instance. If the locks can be acquired, the rule is scheduled using one or a combination of its rating and a rule-specific priority assigned by the programmer at compile-time. The rule instance is placed by the scheduler in one of a series of queues, ordered according to priority. Thus, certain rules can be guaranteed higher priority. Each individual queue may also be declared to be a priority queue so that rules can be prioritized according to their ratings. The rule instances are removed from the execution queues by *rule demons* which then proceed to execute the rules. If heuristic pruning is enabled, the rule demons will first execute a control function attached to the rule type in order to determine whether the rule instance should be pruned. This is necessary to maintain the responsiveness of the system, as the state of the system may change dramatically between the time when a rule becomes eligible to fire and when a processor becomes free to execute it. This is particularly true if rules are prioritized – less important rules may remain on the execution queues for significant lengths of time. The rule-demon approach was adopted over the previous approach of forking off rule executions using the thread construct when it became clear that the thread mechanism did not allow sufficient flexibility in ordering and pruning rule executions in response to changes within the system. The rule demon system also makes it easier to instrument and measure the behavior of the scheduling queues. The queue-based server architecture was inspired in part by the method in which match-level parallelism is implemented in CParaOPS5 [Kalp *et al.*, 1988].

Rule Demons

Although the queue servers are called "rule demons", this is actually a misnomer as they are actually responsible for filling requests for action and match-level parallel activities as well. The control structure of the rule demons is structured according to the model that low-level activities should be served before higher granularity activities. Therefore, the central loop of each rule demon operates in the following steps:

1. While any match level operations are on the match queues, remove one match operation and process it.

2. If no match level operation is enqueued, while any action level operations are on the action queues, remove and process one action.

3. While no match or action requests are extant, monitor the rule queues, beginning from the lowest numbered (highest priority) and scanning towards the highest (lowest priority). If a rule is found, it is removed from the queue and executed.

4. Go to step 1 and repeat.

The dynamic behavior of the system can be modified by changing the control behavior of the rule demons; the order in which rule queues are traversed by the demons can be specified at invocation time. Because there may be increased contention for queues at low-levels of granularity, UMPOPS by default provides two queues each for action or match level parallelism; this number may be increased by the user if monitoring indicates that contention for the scheduling queues has become a bottleneck.

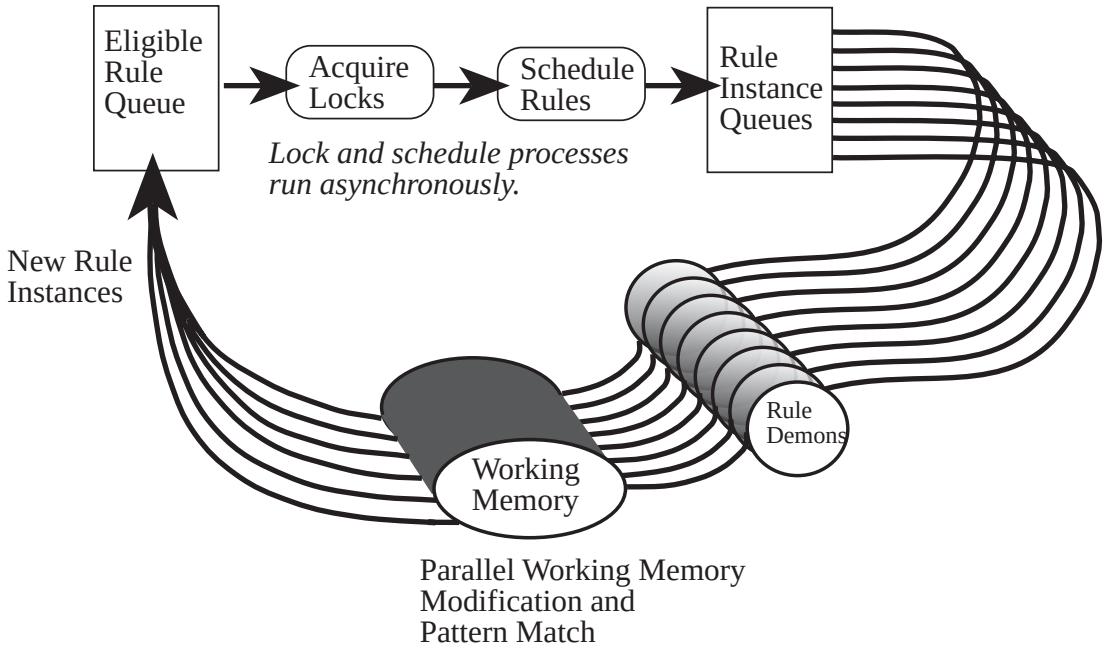

Figure 1.1: The architecture of the parallel rule-firing system.

1.3.1 Run Time Statistics:

UMPOPS was implemented to serve as an experimental tool for exploring the characteristics of parallel rule-based systems and the tradeoffs imposed by various architectural choices. The rule-firing architecture is therefore instrumented to return timing information, both for the overall statistics of each run and individual statistics relating to each rule firing.

Execution Statistics The following statistics are gathered each time the inference engine is invoked.

- Total run time: The total time taken to run a program.
- Individual processor times: The total time spent by each processor in executing rules, as well as average, minimum and maximum times spent by each processor in executing rules.
- The total number of rules scheduled, executed, locked out, or deleted by heuristic functions.

Rule Execution Statistics: For each *rule*, the system keeps a record of the number of executions, the time to execute the righthand side, the time to "pre-evaluate" the rule, the time spent in the 'conflict set', the time spent waiting in the execution queue, and the number of attempts and time required to acquire working memory locks. For each of these statistics, the system records average, minimum, and maximum figures as well as standard deviations. These statistics are also available for specific rule instances.

Queue statistics: For each scheduling queue, the system records the time at which each addition or deletion takes place; this allows the size of each of the priority queues to be charted over the course of a run.

Critical Regions: Mechanisms are available to measure the amount of time which is spent waiting to acquire locks on each critical region, however it was found that the presence of these mechanisms affected the timing of the system, so they should only be employed when it is suspected that there is serious contention for a resource.

Working Memory Elements: For each working memory element, the system records the time of creation and the time required to perform the match for that element.

1.4 A Locking Scheme for Ensuring Partial Correctness of Working Memory

When rules are executed in parallel, their righthand side actions may interact, causing inconsistencies or errors in working memory. These interactions may take two forms, either *clashing* or *disabling*. Disabling occurs when two or more rules which are mutually disabling execute concurrently (see Figure 1.2). In a serial rule-firing system, only one of these rules would fire and the others would be disabled. The working memory state achieved by the concurrent firing of these rules is therefore not achievable by any serial rule-firing order. Clashing behavior occurs when two rules perform competing modifications to the same working memory element. Because OPS5 does not enforce a unique representation for working memory elements, clashing behavior *per se* cannot take place. Instead, clashing rules may create multiple derivations of working memory elements leading to spurious or redundant rule firings and subsequent explosive growth of working memory size.

A number of techniques have been developed for detecting rule interactions [Ishida and Stolfo, 1985, Ishida, 1990, Schmolze, 1989]. These algorithms usually consist of a static analysis phase which is performed at compile time and a runtime component which dynamically examines all eligible rules and selects a co-executable set. The dynamic component, which cannot be performed at compile time because variables in rules do not become bound until the rules are instantiated, is relatively expensive, both because of the synchronization cost of accessing all eligible rules, and because it must perform unification of variables in order to precisely identify the rule interactions.

In the programs which have been developed for UMPOPS, rule interactions have been observed to occur only rarely. Rather than accept the synchronization delays associated with a full analysis of rule interactions[1], it was chosen to enforce only a subset of the correctness criteria using a scheme of read/write locks on working memory elements. The implementation of the locking mechanism is as follows: Each working memory element is assigned a structure which contains a read counter and a write flag. As each rule instantiation enters the conflict set, each working memory element that appears on the lefthand

[1] The overhead due to analysis of rule interactions has been estimated to limit effective speedup due to rule parallelism to between 5 and 10 [Schmolze and Neiman, 1992].

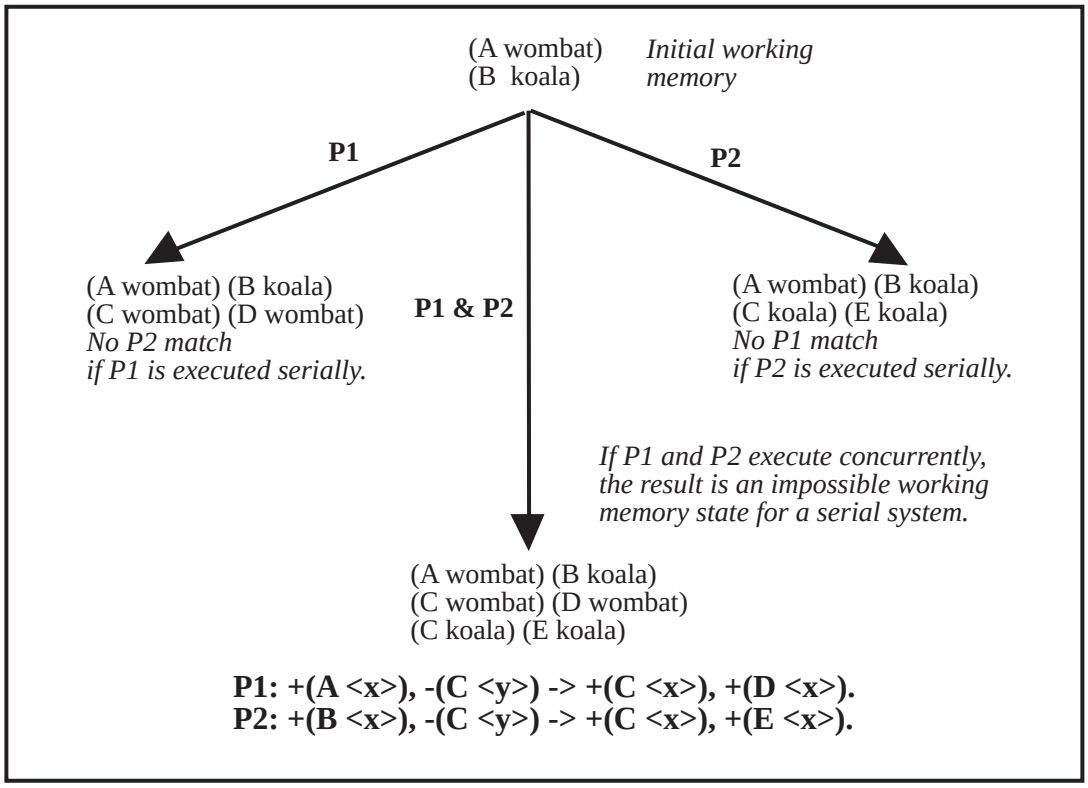

Figure 1.2: When mutually disabling rules are allowed to fire concurrently, the result may be a working memory state which could not be produced by any sequential rule firing.

side and which is modified in the righthand side is placed on that instantiation's *write* list. Each working memory element which is referenced in the LHS but not modified is placed on the instantiation's *read* list.

Before the rule instantiation is executed, each list is examined. If any of the working memory elements on the read list have their write flag set, then another rule currently executing is about to modify that working memory element. Because the rule instantiation will eventually be disabled by the removal of the element being modified, it is removed from the conflict set but not executed. Similarly, if any of the working memory elements on the write list have their write flag set, the instantiation is also removed from the conflict set, otherwise, if other rules are referencing the working memory element, the rule instantiation is not executed, but is placed back on the conflict set. All read and write privileges associated with working memory elements must be obtained before rule execution, thus the righthand side of a production may be considered atomic; either all of the actions will be executed, or none will be.

Although the creation of a working memory element is certainly a write operation, it is treated as a special case. When a rule creates a working memory element, it actually acquires a read lock (increments the read counter) for that element. The reason for the special treatment is that when asynchronous rule firing is enabled, a rule might be stimulated by the addition of a working memory element and become eligible to fire even while the working memory element is still being added. If a write lock were obtained on the working memory element, any rules stimulated by that element would be unable to fire (and, if fact, would be discarded from the eligibility queue). If a read lock were not obtained on the working memory element, then the rule stimulated by the element could theoretically delete it, causing competing match operations and a possible race condition within the Rete net.

The primary disadvantage of the working memory locking scheme is that it only guarantees partial correctness. Interactions involving rules which contain negated condition elements cannot be detected or prevented because it is not generally possible to acquire a lock on an element, or set of elements, which does not exist. An experimental version of UMPOPS has been developed which guarantees full serializability, but with an associated performance penalty; details of this implementation can be found in [Schmolze and Neiman, 1992].

The concept of locking elements to prevent interactions due to concurrent modifications is widely used in database systems and a similar scheme to the one just described was implemented in a DBMS-based production system by Sellis,et al. [Sellis *et al.*, 1987]. This implementation uses *region locks* to prevent interactions due to negative conditions. A region lock typically prohibits access to a class of working memory elements, possibly restricted by value and, depending on the precision with which the region can be identified, may prove unduly pessimistic in restricting access to working memory.

UMPOPS provides a mechanism similar in nature to region locking which allows a single working memory element to be locked, even before that element has been created. This mechanism, called *make-unique* allows the programmer to define a working memory element and certain key fields to be unique, that is, only one attempt to create an element with the class and key values will succeed, all other attempts will be locked out. As will be seen later in this report, such a mechanism is crucial for implementing common programming idioms

such as merging the results of a parallel search.

The working memory locking scheme presents a number of advantages and disadvantages as opposed to the serializability guarantees provided by Schmolze or Ishida. The advantages are listed below:

Low Overhead: The overhead of the UMPOPS locking scheme is limited to the generation of the read and write lists and the actual acquisition of the locks, both of which incur minimal costs, approximately 1-2% of rule execution times as opposed to approximately 10% for full guaranteeing of serializability (see [Schmolze and Neiman, 1992] for the argument behind this assertion).

Because locks are acquired independently of other executing instantiations, the lock acquisition time is O(N) where N is the number of elements referenced by the instantiation. In contrast, the overhead associated with the scheme proposed by Schmolze [Schmolze, 1991] is at best $O(N^2)$ where N is the number of instantiations in the conflict set.

Rules can execute asynchronously: It is not necessary to compare each eligible rule against all others, so synchronization is not required and rule-firing may take place asynchronously[2].

No compile-time analysis is required: Because all working memory elements read or written by a rule instantiation are automatically determined at run-time, no compile-time analysis is required. This is particularly significant in systems in which the RHS syntax is complex or continually changing; as will be seen in the following section, both of these are true of UMPOPS.

As mentioned previously, the most significant disadvantage of the locking scheme is that it does not guarantee serializable programs; it merely allows serializable programs to be designed and constructed. The burden falls upon the programmer to ensure the correctness of the program. When executing rules asynchronously, the locking scheme accepts and schedules rules in the order in which they arrive, thus opportunities for heuristic control are not available. As reported elsewhere [Neiman, 1991], this is compensated for by the increased throughput provided by the asynchronous rule execution policy. A final disadvantage is that the current algorithm requires a central scheduler and lock acquisition takes place serially, thus keeping lock overhead to a minimum is critical. Lock acquisition could be performed in parallel, however this would increase the complexity of the lock acquisition code and the bottleneck due to serial scheduling does not appear to decrease performance to the point where this would be necessary.

1.5 Heuristic control

The scheduler of UMPOPS has been modified to support heuristic control, that is, rules can be either pruned or ordered according to heuristic evaluation functions. To allow rule firings

[2]Interaction detection does not necessarily imply synchronization, for example, Schmolze has developed an asynchronous version of his interaction detection algorithm. Asynchronous algorithms for detecting rule interactions may not produce the maximal parallel sets, however.

to be prioritized, multiple execution queues are provided and each is assigned a priority. Rules assigned to a low priority queue are executed first. This is implemented by insuring that the rule "demons" which monitor the queues first look for tasks to perform in the lower numbered queues. Each rule *type* is assigned a static priority at compile time and each instantiation of that rule is placed in the appropriate queue. In order to allow prioritization within rules of the same type, individual rule queues can themselves be declared to be priority queues. Rule *instances* are rated by rule-specific evaluation functions and inserted into the appropriate priority queue. Much of this control information is specified within the rule using the `meta` construct; the exact syntax will be described in the following section.

There are several points in the match-schedule-execute cycle in which heuristic control can be performed. First, heuristic control is usually performed *de facto* during the rule matching phase; that is, if the rules were not heuristic in nature, there would be no need to implement the program using a production system architecture. Writing rules in such a way as to eliminate unnecessary rule instantiations will minimize the number of rules which must be scheduled and processed by the control demon.

The next opportunity for pruning rules is during the "pre-execution" phase. This phase needs some explanation. After the rule is placed on the eligibility queue, it is scheduled by a dedicated process. The heuristic scheduling functions need information in order to rate the rule, and much of this information is carried in the variables bound by the rule instantiation. In OPS5, this information would normally not be available without executing the righthand side, however, executing the righthand sides of some rules is just what we're trying to avoid by the incorporation of heuristics. So UMPOPS does a "pre-evaluation" at instantiation time (i.e. when the rule is inserted onto the eligibility queue) which extracts the variable bindings for the instantiation and stores them. No righthand side actions are actually executed during this phase and the cost is reasonable. If any rule-specific control functions (specified as meta-information) need to be executed, they take place during the pre-evaluation phase. The rule is then placed on the eligibility queue (formerly the conflict set) where it is immediately scheduled by the control process and placed in the execution queues according to its rating and queue priority. Currently, the only way a rule is pruned during the scheduling process is if there is a conflict with an already executing rule, however, it would be trivial to execute a heuristic pruning function at this time.

The third opportunity for heuristic control occurs immediately prior to a rule's execution. Once a rule is placed on the execution queue, it is theoretically executed immediately. However, in most applications, the number of rules to be executed will temporarily exceed the number of processors available to execute them, and rules will remain on the execution queues for potentially extended periods of time. During this time, the state of the system can change, causing a rule instantiation on the queue to become redundant. Control functions can be attached to rules determine whether the rule is still valid given the current state of the system. The control functions must be extremely fast and simple if they are not to decrease performance, so they are typically designed to simply check a value being asserted by the rule against a global variable. The control function also resets the value of this global variable if appropriate. (It is a problem in OPS and production systems in general that working memory cannot be easily accessed from procedural code, and data must sometimes be stored redundantly in order to be accessible from both rule-based and imperative code.)

When a rule demon removes an instantiation from the queue, it checks to see whether the rule has an attached control function. If so, the demon executes the control function in the context of the rule instance's righthand side. If the function returns a non-nil, the rule is executed, otherwise it is killed. Killing a rule does not necessarily mean that no action is taken – there may be some clean up operations associated with the state represented by that rule. The programmer has the option of attaching "kill actions" to the rule which are executed if, and only if, the rule is pruned by a control rule.

1.5.1 Dynamic Control of Rule-Firing Policies

Simply assigning static priorities to rule *types* and control functions to rules which do not change during the course of a computation is insufficiently flexible to provide truly sophisticated control. It is possible, albeit awkward, to dynamically modify the priorities and control functions associated with rules during execution. For example, special control *meta-rules* can be devised which, when they perceive a particular state occuring in the system, can execute, modifying the priority as was done in the BB1 blackboard architecture[Hayes-Roth, 1985].

Such control rules should, of course, be given the highest execution priorites. Even so, it is likely that queue and execution latencies would render such control rules less responsive to the state of the system than is desirable. A modification to UMPOPS is being contemplated which would allow true control rules to be distinguished. Because the principal activity of such meta-control rules would be to send messages to the scheduler (which is very inexpensive as compared to modifying working memory), control latency could be minimized by sending the control messages during the pre-execution phase (i.e. *immediately* after the control rule becomes enabled) and before the rule enters the eligibility set. Because the new control state of the system would likely have to be mirrored in working memory to prevent iterative firings of the meta-control rules, the meta-control rule would still have to be executed in a conventional fashion. An alternative scheme is to simply bypass the scheduling mechanism and place control meta-rules on the highest priority execution queue.

Another issue in the dynamic control of rule-firing policies is the apportionment of rule demons to the various execution queues. The default assignment is that the highest priority rules get executed first, however this is an *unfair* scheduling policy and it is possible that rules on lower priorities queues could be "starved" if large numbers of high priority rules were to arrive. UMPOPS allows the user to specify a queue examination protocol when invoking rule demons; the demons will examine the rule queues in the given order. If more than one protocol is given, rule demons the protocols will be divided evenly among rule demons. It is possible, for example, to assign a single rule demon to monitor a single queue or a range of queues, and to specify the order in which they are visited. The programmer can also specify whether the queue protocol is fair, that is, whether after executing a rule, a demon should begin at the highest priority queue or whether it should visit all queues in its protocol before restarting its traversal of the execution queues.

1.6 Interactions between Consistency Maintenance and Heuristic Control

The heuristic control mechanisms can interact with the working memory locking scheme described previously in potentially pathological ways. The first problem occurs when a rule which has acquired the necessary working memory locks is then pruned by a heuristic control mechanism. There will always be an interval between lock acquisition and the pruning. During this time, it is possible that another rule which is capable of satisfying the heuristic will become eligible to fire. Because the first rule has acquired the necessary working memory locks, this competing rule will be prevented from executing and will be removed from the eligibility set by the lock manager. Because neither rule is ever executed, the result of this sequence of events is that the appropriate action never takes place. Simply reversing the order in which lock acquisition and heuristic control takes place will not solve this problem, and performing both operations simultaneously would require delaying lock acquisition until execution time and performing lock management within a critical region. One solution (not yet implemented) is to modify the lock management routines so that rules which are locked out due to competing write operations are not eliminated from the eligibility set until the competing rule has successfully begun execution.

A similar problem arises when an asynchronous rule-firing policy is employed. Because rules are scheduled on a first-come-first-served basis, standard conflict resolution techniques in which all eligible rules are ordered and the "best" rule is selected cannot be applied. It is possible, therefore, that a rule will be selected to be fired and acquire all working memory locks only to have a heuristically superior rule arrive in the eligibility queue. If execution queue latencies are short, then the second rule will simply be disabled by the first rule as it changes working memory; if the rules are designed correctly, the change to working memory will re-stimulate a similar instantiation and the superior result will eventually be reasserted into working memory at the cost of some delay. If execution queue latencies are long, the rule which asserts an inferior answer may remain in the execution queue for a long time. This will block the assertion of the superior result and reduce the efficiency of the heuristic pruning mechanisms by allowing more inferior solution paths to be explored during the extended period in which the better solution is blocked. A solution (unimplemented) to this problem is to allow heuristic override of locks. That is, one could record the identity of rule instances which have acquired locks to working memory elements. If a rule attempts to acquire a lock on that element and finds that it is possessed by another rule, then the following algorithm could be performed.

1. Check to see if the blocking rule is currently executing. If so, fail.
2. If the blocking rule is still in the execution queue, mark it as temporarily non-executable.
3. Compare the blocked and blocking rule using a situation specific conflict resolution function.
4. If the blocked rule is superior, mark the blocking rule as 'killed', release its locks, and schedule the new rule. If the blocking rule is superior, mark it as executable and remove the blocked rule from the eligibility queue.

1.7 Multiple Worlds

One of the biggest advantages of parallelism in complex system is the ability to explore multiple alternatives simultaneously. If the search is partitioned appropriately, then problems of rules interacting no longer apply. So we can trade the cost of detecting syntactic rule interactions against the expense of creating partitioned (and possibly redundant) states. Copying states can be done informally in situations in which the entire problem-solving state can be represented as one or two working memory elements; in these case, the partitioned state can be created by copying the elements in question and annotating them with a unique tag (see the Travelling Salesperson example in Chapter 3 for an example of this technique). For larger, more complex states, possibly consisting of linked data structures, the problem of accessing and copying states becomes more problematic. Not only is it difficult to explicitly reference each appropriate working memory element in the lefthand side of the rule and explicitly copy it on the right, but the use of explicit tags to denote state places an undue strain on the pattern matcher and can lead to matching inefficiencies.

There's a simple approach which is to copy each new state into a logically separate version of the Rete net. That is, we can imagine each node of the Rete net being sliced into an infinite number of dimensions. Each dimension represents a new state, and as each new state is created, the working memory elements associated with that state are placed in the appropriate world, or dimension (see figure 1.3). Thus, creation of a new state consists very simply of the transformation $WME_i -> WME_i + 1$ where the arrow denotes a copying/transformation operator. The advantage of this approach is that working memory elements in one state can only be compared with working memory elements in the same state and therefore identical elements can appear in multiple states, thus no re-naming or tag generation is necessary. The principal disadvantage, and it is potentially a large one, is that the copying operation is likely to be very expensive (it effectively violates the *temporal redundancy* requirement of the OPS5 Rete net). The response to this objection is two–fold; first, given sufficient resources to implement action parallelism, copying of working memory elements can be done concurrently and the creation of new states can be reduced to O(n) where n is the depth of the network. The second response is that, with a sufficiently clever copying algorithm, one state can be copied directly into another from one slice of each node of the Rete net to the next, thus maintaining the partial match state contained in the Rete net. This copying operation is potentially suitable for large scale SIMD parallelism and could potentially be carried out in O(1) time.

A second disadvantage is that of space usage – the copying scheme inevitably will result in redundant copies of working memory elements which could be inefficient to maintain and garbage collect. We can somewhat reduce this problem by maintaining a *base* space; that is, a space in which all facts not actually modified during the course of the computation are stored. The ultimate solution to the problem of space (and to a certain extent, that of copying overhead) is to employ a scheme in which successor states simply inherit the working memory elements from predecessor states via pointers augmented with truth maintenance-style IN/OUT lists.

An experimental version of UMPOPS has been developed with a partitioned Rete net and operators for performing parallel search in multiple worlds. This multiple worlds implementation is described more fully in [Neiman, 1992].

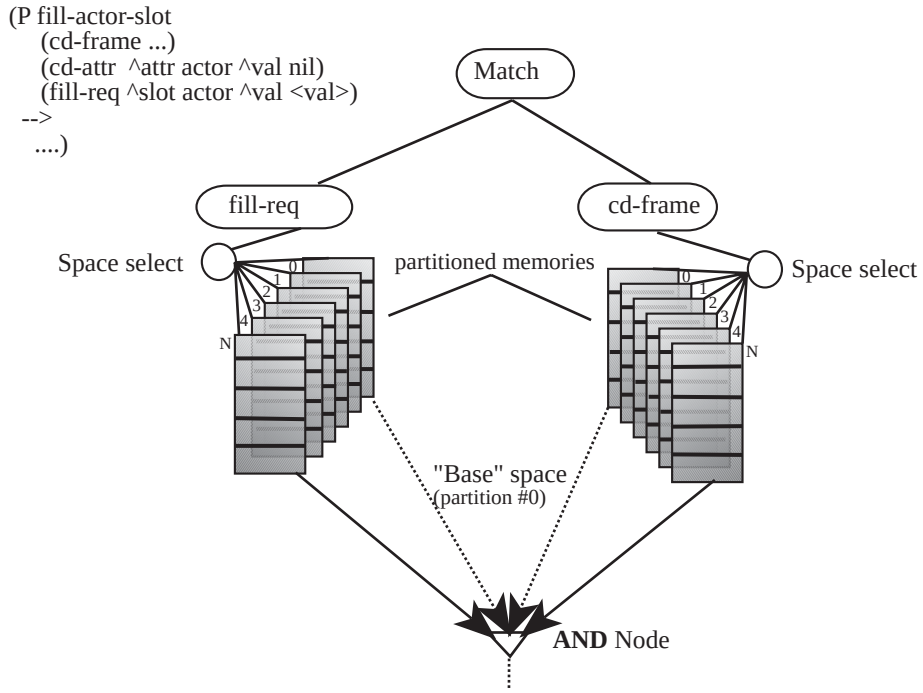

Figure 1.3: By partitioning the memories of the Rete net, a multiple world implementation suitable for parallel search can be transparently achieved. To minimize copying, a "base" space or partition can be defined which contains knowledge guaranteed to remain stable over the course of the search.

1.8 Summary

UMass Parallel OPS5 has been modified to support parallelism at the rule, action, and match levels. Language features have been added which allow correct programs to be designed without the overhead of complete checking for rule interactions. "Hooks" for control functions have been added to the language to increase its expressive ability and to compensate in part for the lack of explicit conflict resolution while firing rules asynchronously. The following chapter discusses how to invoke the system and explains the syntax and usage of the new language features.

Chapter 2

User's Manual

This chapter discusses how to invoke the parallel OPS5 system. It should be noted that much of the information is specific to the current version of Top Level Common Lisp[1] and UMass Parallel OPS5 and may change in later releases. The section on invocation can be skipped by the casual reader. Information on the language which is not specific to parallel OPS5 can be obtained from the *OPS5 User's Manual* [Forgy, 1981].

2.1 Invocation:

The system is contained in the file **POPSHASH.lisp**; the compiled version is in the file **POPSHASH.zoom**[2]. The file should be loaded into a TopCL image. It is recommended that the TopCL image be invoked with the `-gcpages 2500` option, and, if using a Lisp machine front-end, the `-netdebug` *remote-host* option. Once the file is loaded, the control stack limit should be set using `(setf sys::*control-stack-limit* 15000)`.

Startup files: A standard start file looks like this:

```
;Standard start file for Toru-Waltz.


(in-package 'user)
(load "hash.zoom")
(load "popshash.zoom" :print nil)
(load "lock-time.zoom")
(load "make-unique.zoom")
(load "scheduler.zoom")
(load "pops-set-fns.zoom")
(load "action-par-fns-mmq.zoom")
(load "float-string.zoom")
(load "graph.zoom")
```

[1] Top Level Common Lisp and TopCL are trademarks of Top Level, Inc.

[2] POPSHASH is the modification of the experimental parallel OPS system with hashed memories.

```
Where:
   hash is the file containing the hash table functions
   popshash is the parallel OPS5 code.
   scheduler is the current main loop and scheduling package.
   lock-time contains utilities for timing lock functions.
   make-unique is for the unique locking mechanism.
   pops-set-fns is the code for set-oriented productions.
   action-par-fns is the code for implementing action parallelism.
   graph is a utility for showing the activities of the rule demons
         graphically.


(format t "~%Parallel OPS5 loaded. ~%")
(setf sys::*control-stack-limit* 15000)
(i-g-v)
;;YOU HAVE TO EXPLICTLY DEFINE THE NUMBER OF OPS PRIORITY QUEUES BEFORE
;;RUNNING.
;;Even if running serially, because this also initializes the conflict
;;set queue.
(init-ops-queues 4)
;;THIS IS HOW YOU DEFINE THE NUMBER OF QUEUE DEMONS WHEN RUNNING IN
;PARALLEL.
(init-rule-demons 12)

(load "/user1/dis/dann/bench/toru-waltz/dwaltz.ops")

;;IT'S A GOOD IDEA TO DEFINE A RESET FUNCTION TO GET THINGS BACK TO
;;GROUND ZERO...
(defun reset()
   (oremove *)
  (setf *killed* nil)
   (clear-unique-trees)
   (reset-queues)
   (init-time)
   (reset-control-variables)
   (setf *cycle-count* 0)
   (setf *action-count* 0)
 )



;;;FLAGS TO ENABLE VARIOUS SELF-EXPLANATORY THINGS
;;;NOTE: Setting action and node parallelism to true don't automatically
;;;initiate parallel activities anymore, but allow them to be toggled.

(setf user::*production-parallelism* t)
(format t "Setting *production-parallelism* flag to ~a~%" *production-parallelism*)
(setf user::*action-parallelism* t)
(format t "Setting *action-parallelism* flag to ~a~%" *action-parallelism*)
(setf user::*node-parallelism* t)
(format t "Setting *node-parallelism* flag to ~a~%" *node-parallelism*)
(setf *control-enabled* t)
(format t "Setting *control-enabled* flag to ~a~%" *control-enabled*)
```

The `init-ops-queues` command takes one required argument – the number of queues reserved for rule instantiations – and two optional arguments, the number of queues reserved for action and match parallelism, respectively. It is recommended that at least two queues be allocated for match-level operations as the magnitude of match-level node activations is sufficent to generate some contention for the queues between the queue demons. The `init-ops-queues` command must precede the `init-rule-demons` command.

The `init-rule-demons` command takes one required argument, the number of rule(/action/match) demons to be activated. An optional argument can be provided to specify the order in which the demons should examine the queues. This argument has the form of a list of lists, each list having the form (*fair-flag visit-list*). A visit list is simply a list of integers in the range between 0 and $1 - N$ where N is the number of queues assigned to rules. The "fair-flag" indicates whether the traversal will be fair, that is, whether the queue demon will examine all of the rule queues in its list before starting over, or if it will begin at the start of its list after each rule execution. The traversal lists are apportioned out evenly to the rule demons as they are created; a rudimentary check is performed to ensure that each rule queue is visited at least once.

Rule demons may be killed using the `kill-rule-demons` function.

2.2 Rule Firing Policies

High-level control over rule-firing algorithms in UMPOPS is primitive; the global policies are determined by the setting of global variables. Rules may be fired either synchronously or asynchronously by setting the **async** flag. If the **production-parallelism** flag is set to nil, rule firing takes place serially. If the variable `*cr-scheme*` is set to `'rating` when running serially, rules are fired according to priority functions, otherwise they are fired according the conventional MEA conflict resolution scheme.

2.2.1 Control Tasks

It is likely that the programmer will occasionally wish to perform synchronous conflict resolution within the context of certain *tasks* or groups of rules, while allowing other activities to take place asynchronously. (An example of an application which requires such a rule-firing architecture is the Alexsys system developed at Columbia University [Stolfo *et al.*, 1990, Stolfo *et al.*, 1991]; a discussion of experiments with this system will appear in future reports.) A task-based rule-firing architecture is required when the computation can be divided into multiple asynchronous tasks, each of which:

- Has a specific preferred order in which operators should be tried.
- Applying one operator invalidates all others in that task.
- Performing search by applying operators in parallel would be prohibitively expensive due to copying or computational costs.

2.2.2 Defining Tasks and Task Quiescence

A task can be defined informally as a named subproblem within the scope of a larger computation. Rules executing in different tasks can fire asynchronously; however they are not guaranteed to access discrete resources, so locking of working memory elements accessed within tasks is necessary. This possible interaction between tasks distinguishes UMPOPS's tasks from Miranker and Kuo's notion of a set of independent clusters firing asynchronously [Miranker *et al.*, 1989, Kuo *et al.*, 1991]. A task is a *control* mechanism which defines the context in which a computation's conflict resolution routines (if any) and rule-firing policy are defined and in which a local quiescence may be determined. Multiple rules from within a task may execute concurrently; this is dependent only on the conflict resolution routine assigned to that task; if the routine returns multiple instantiations, they all may fire. Rule-firing within a task may be asynchronous (in which case no conflict resolution routine is defined) or synchronous (in which case, a situation specific conflict resolution routine may be specified, or a default may be used).

Control activities in tasks requiring synchronous conflict resolution can only occur once the task has become *quiescent*. The quiescence of a task can be defined as a state in which all current computation corresponding to that task has been completed. Because of the data-directed, pattern-matching nature of rule-based systems, determining the quiescence of tasks (or even the scope of tasks) is non-trivial. Certainly if all working memory changes in the entire system have become quiescent, then we can say that a particular task is quiescent and that all relevant operators corresponding to that task have been determined. However, in a system in which tasks are executing asynchronously with respect to each other, it is unreasonable to expect system-wide quiescence to occur. Thus, we have the problem of determining whether a working memory change which is currently occuring is part of the current task. In general, this can only be determined by waiting to see if that element matches against a rule which also matches against an element previously determined to be in the task.

For example, consider the pseudo-OPS code below:

```
(P
  (task ^name find-a-wombat)
  (wombat ^name Keith)
-->
  ...)
```

If the working memory element `(wombat ^name Keith)` was being asserted, then the task defined by the element `(task ^name find-a-wombat)` could not be said to be quiescent. Dynamic rule conflict detection cannot be used to detect the quiescence of tasks, not only because of the overhead, but because new rules may continously be created (or about to be created); thus once again, only global synchronization would allow run-time consistency checking to accurately determine whether a task was quiescent.

We must assume that for the large part, working memory is quiescent and that all active working memory changes can be annotated as directly affecting particular tasks. This is

done by starting with a quiescent working memory. Goal elements are created in the context of a task; a goal element is simply any element which is specifically annotated as belonging to a task and which stimulates further rule firings. From that point on, any rule stimulated by the goal element is considered to be executing in the context of that task, and any working memory changes stimulated by that rule are in the context of the task. Quiescence is achieved when all working element changes being performed within the context of a task have terminated.

Because all rules execute in the context of the task which created their stimulated elements, communication or sharing of data is difficult to arrange. (The task mechanism is still experimental and not all of the necessary mechanisms have been developed.) Currently, all initial working memory elements are created in the context of an initial task. All tasks are allowed to access these elements. If any of these elements are modified by a control task, the resulting element remains in the context of the initial task. This allows tasks to communicate with the initial task, but means that quiescence of tasks is only partial; quiescence of the initial database pool cannot be guaranteed. The question of determining partial quiescence remains an interesting research issue...

2.2.3 The Task Implementation

Incorporating tasks into the UMPOPS rule-firing architecture was straightforward. Rule instances and working memory elements have to be annotated to record their parent tasks and the rule firing routines have to modified to annotate them correctly.

Each task has its own conflict resolution routine which is to be applied only to rules instantiations within the context of that task, thus, each task must be assigned its own conflict/eligibility set. A new eligibility set structure was devised for UMPOPS; the eligibility set is now an array of sets, with the index of each set being uniquely assigned to a task. Various initialization routines were modified to ensure that the new data structures are initialized and reset properly. The conflict set reporting routines were modified so that they correctly interpret and print out the contents of the various conflict sets.

Task Syntax

The task syntax is similar to that used in UMPOPS for specifying action-level parallelism.

Defining tasks: Tasks must be defined before they can be created. A task definition consists of a task name, a rule firing policy (asynchronous or synchronous) and, if synchronous, a conflict resolution routine specified as a function call taking an eligibility set.

```
(deftask <task-name>
         :conflict-resolution-routine <CR-function-name>
         :type {synchronous | asynchronous}
         )
```

Initiating tasks: A task must be explicitly invoked with one exception; all initial rule firings take place within the scope of an initial default task. (The user may define this default task to be asynchronous or synchronous). The syntax is:

```
(with-new-task(<task-name> body))
```

All RHS actions contained within the body are executed within the context of the task. Otherwise, all RHS actions within a rule are executed within the context of the task which stimulated that rule.

Modifiying or Terminating Tasks: When phases of a computation change, the nature of the task may have to change as well. The `(redefine-task <task-type>)` operator allows rules to change the rule-firing policy and conflict resolution routine of a task in mid-computation. The `(kill-task)` function allows a rule to terminate the task context in which it is executing. This allows the resources used by the task, specifically the eligibility set, to be reclaimed and assigned to other tasks.

Summary: Tasks

The control task is a flexible construct which allows a parallel rule-firing system to pursue multiple independent activities, each of which possesses its own conflict resolution routine and appropriate rule-firing policy. The propagation of task contexts through the use of working memory elements allows at least a partial quiescence to be defined and determined for each task. If tasks can also be guaranteed to be fully data-independent in a specific application, complete local quiescence of a task can be determined.

2.3 Functions for Timing and Benchmarking

UMPOPS is instrumented to provide a great deal of information about the execution characteristics of parallel programs. The following functions can be used to retrieve this timing information:

- `Analyze-tasks`: The `analyze-tasks` function returns a list of the utilization times for all activated rule demons, divided into time spent executing rule, action, and match level activities. Maximum, minimum, average and total times are given. (The 'tasks' referred to are not the control tasks discussed in the previous section, but the tasks assigned to processors.)
- `Analyze-instances`: The `analyze-instances` function prints out the execution statistics for each type of rule executed, including number of executions, average, minimum, and maximum execution times, and number of times the rule was locked out. Information about residency on the eligibility set and queues and time spent performing heuristic ratings is also given.
- `Print-time`: For the real statistics aficionado, the function `print-time` will print out queue and execution statistics for every individual rule executed during a program.

- `Anomalies`: Given an argument representing a time limit, returns all rules exceeding that execution time.

Graphing functions are available to show processor utilizations. All graphing functions should be preceeded by the function call `(init-graph)`. The function `(do-graph start end)` prints out processor utilizations for the indicated time period. Each type of parallelism is indicated by a "*" (rule parallelism), an "A" (action parallelism), or an "M" (match parallelism). For each time period, the average number of rules in the eligibility set and all execution queues is given. The graphing data is obtained by sampling recorded execution times at a given level of granularity. The default resolution is 50 samples/sec. This resolution is approximately that of a rule execution, however to adequately examine activities at a finer level of granularity, the sampling interval can be reduced by resizing the graph. The function `(resize-graph sample/sec time-period)` will resize the graph to record data at the given resolution for the given number of seconds. For example, `(resize-graph 2000 2)` will display activities at a resolution of every .0005, fine enough to display even match-level activity accurately. The function `(write-graph file-name start end)` will write the graph data to a file with the columns delimited by tabs so that the graph can be reproduced using more sophisticated graphics.

2.4 LHS Meta-level Notation

The lefthand syntax of the rules in UMass parallel OPS5 has been modified to allow the specification of "meta-level" information. The notation has the syntax `(meta (meta-type value) (meta-type value) ...)`. Although the `meta` construct is present in the lefthand side, it does not generate patterns or change the match in any way; it is simply used to send messages to the rule compilation routines.

Originally, the purpose of the `meta` notation was to allow the specification of control information, however, in actual use, the `meta` notation has been used as a catch-all for any language modification which would otherwise require a modification to the original OPS5 syntax. The current usages of the meta notation are summarized below.

2.4.1 Annotating mode-changing productions:

Rule-based programs are usually organized in phases. Each production contains a reference to particular working memory element of a class such as *mode* or *stage*. The production is only enabled when the mode is set to a particular value. In order to change the mode, special rules such as the following are used:

```
(p go-to-next-phase
         (stage ^is current-phase)
        -->
         (modify 1  ^is next-phase))
```

In a serial OPS5 system, the standard conflict resolution strategy is used to ensure that the mode-changing production only fires after all other eligible productions have done so.

In a system which fires all eligible productions in parallel, the mode-changing production may execute prematurely. To prevent this, mode-changing productions should be explicitly annotated in the following way:

```
(p go-to-next-phase
         (meta
            (rtype mode-changer))
         (stage ^is current-phase)
        -->
         (modify 1  ^is next-phase))
```

'Mode-changing' rules annotated in this manner are handled specially by the scheduler, they are prevented from firing until all other rules in the conflict set have fired and all working memory changes have been processed.

2.4.2 Other uses of the `meta` notation

The `meta` notation has been used as a general purpose mechanism to specify information about the rules which would otherwise require changes to the existing OPS5 rule syntax. These usages are summarized below:

- **Priority:** A number between zero and N-1 where N is the number of priority queues. All rules of this type will be placed within that queue.
- **Priority-queue:** If non-nil, then rule instantiations of this type are placed on a priority queue.
- **Priority-fn:** The function which is executed during the pre-eval phase in order to determine the rating of rules placed in a priority queue.
- **Lock-not-required:** If it is certain that a rule will never interact with other rules (e.g., it works in its own space), then locking is not required. Setting lock-not-required to be non-nil informs the controller that locks need not be acquired for this rule.
- **Control-fn:** A control function which, when executed in the environment of the righthand side, will determine whether or not the rule should fire. Variables bound by the rule can be accessed using the $varbind function (i.e. ($varbind '<foo>) returns the current binding of <foo>). Usually the control function compares some combination of instantiation variables with a global variable describing the current state of the solution.
- **Control-generator:** A function which allows a control value to be associated with a keyword at instantiation time which is then stored in the rule instantiation for later reference by control functions. The user specifies a form, i.e. ((gen-control-data indicator exp)), and the cons-cell `(indicator . exp)` is placed on an association list associated with the rule instance. It can then be accessed by various control functions using a standard `assoc` call, i.e. `(assoc '*tsp-distance* (rule-instance-control-data instance))`.

- **Rhs-kill-actions:** A list of righthand side actions to be taken if the rule is killed by a control rule. These are usually "clean up" actions which delete the current state so as to reduce the overall size of working memory.

Examples of the use of the `meta` construct can be seen in the programming examples in section 3 and the appendices.

2.5 New Righthand Side Functions in UMPOPS

This section describes the additions to the OPS5 righthand side instruction set and syntax.

2.5.1 Action and Match-level Parallelism

In the previous version of UMass parallel OPS5, action and match level parallelism were enabled by global flags. Using this technique, it was not possible to selectively employ these levels of parallelism in specific rules. Because action and match parallelism are not appropriate to all situations and may cause saturation of processing capability, it proved necessary to develop RHS language constructs to allow action or match parallelism to be specified at the level of individual working memory changes. To avoid having to maintain separate versions of programs with and without action and match level parallelism, the global switches `*node-parallelism*` and `*action-parallelism*` are retained. If these flags are set to `nil`, parallelism is disabled and all actions and match activities will take place serially.

A disadvantage of the scheme of using explicit constructs to implement match parallelism is that it is not possible to simply activate match parallelism to measure the speedup achievable on existing OPS5 programs; the programs must be modified to explicitly invoke parallelism in their righthand sides.

Action Parallelism:

To invoke action parallelism, there are two constructs, `in-parallel` and `in-parallel-sync`. Each of these constructs takes one or more RHS actions as an argument. Each working memory change carried out within the scope of these commands is executed concurrently with all other working memory changes. If the `in-parallel` construct is used, the flow of control continues on to the actions following the construct, if any, as soon as the working memory changes are initiated. It is frequently necessary to ensure that all working memory changes have completed before a rule terminates, for example, when performing an initialization routine. In such cases, the `in-parallel-sync` construct is used; this construct initiates all the RHS actions included within its body, then waits until they have all been completed before any further actions are taken.

Match-Level Parallelism:

Match-level parallelism does not normally yield great speedups because of the small granularity of the match operations, the relatively high overhead of invoking parallel operations

at that level of granularity, and the small number of rules affected by the average working memory change [Gupta, 1987]. There are, however, certain cases in which a significant improvement can be achieved. The most common of these is the mode-changing production. When a working memory element which triggers a new phase of the computation is added, unusually large amounts of matching activity occur and many rules are triggered. Under these circumstances, match-level parallelism can greatly reduce the rule execution time.

To invoke match-level parallelism, new righthand side actions are provided. These are `make-match-parallel`, `modify-match-parallel`, and `remove-match-parallel`. The syntax of these actions is identical to their serial counterparts, however, the matching of the working memory changes triggered by these commands will take place in parallel. The commands do not terminate until all the matching processes have been completed.

2.5.2 Make-unique

UMPOPS provides working memory locks to enforce consistency in working memory during concurrent activities. But the working memory locking scheme is not adequate to ensure correct rule firings for rules which contain negated elements on their lefthand sides as it is not possible to acquire a lock on an element that does not yet exist. However, it is possible, through the `make-unique` function, to require that a rule "ask permission" before creating an element that it (or another rule) references through a negative condition element.

The `make-unique` function (actually a pseudo-function) is used to create a working memory element with certain values once and only once. This mechanism, very useful when performing initializations, allows the user to specify a working memory element of a specific class with given key values. Before creation, a check is performed to see if such an element previously exists. If so, the rule is not allowed to execute, otherwise the rule is allowed to create that element, and all other instantiations are prevented from doing so. This is called acquiring a "unique lock". An extended example of the use of the `make-unique` mechanism is given in section 3.3.3.

The element which is to be created must be *declared* to be unique using the `unique-attribute` command. This command, which must precede any rule definitions, usually immediately follows the `literalize` command defining the working memory element. For example, to define a unique solution element for each of a number of tasks, one could use the following syntax:

```
(literalize task-solution task value)
(unique-attribute task-solution task)
```

After this declaration, only one element with the class 'task-solution' and a given value of the task field can be created by a call to `make-unique`; thus each task will have a unique solution element and clashing cannot occur. Uniqueness is only guaranteed if the element is created using the `make-unique` call; UMPOPS doesn't prevent the programmer from later changing the key fields or values of a unique element. The syntax of the `make-unique` call (and, in fact, the function itself) is identical to that of the OPS5 `make`. The elements required to be created uniquely are annotated at compile time and the `make-unique` call

is retained as syntactic sugar to highlight the elements that are intended to be created as unique elements.

Once obtained, unique locks are never released; this allows the programmer to implement "one-shot" rules which fire once and once only. The function `clear-unique-trees` must be executed between runs of a program in order to release all existing unique locks. (The underlying mechanism behind the unique locks is a discrimination tree upon whose leaf nodes the locks are hung.)

The `make-unique` function differs from the more general region locks in that it allows the user to specify a particular create operation to be singled out for special attention. Instead of having to check all new working memory elements against all currently defined regions, only elements declared to be unique are examined; this reduces the overhead of the locking mechanism significantly.

2.5.3 Set Functions

Recent work on incorporating rule-based systems into database systems has indicated that increased efficiency and ease of programming can be obtained through the use of *set*-based rules rather than *instance*-based rules [Gordin and Pasik, 1991, Widom and Finkelstein, 1990, Sellis *et al.*, 1987]. In the instance-based implementation of a rule-based system, one and only one rule instantiation is created for each set of working memory elements which match a lefthand side. That is, if the lefthand side of a rule has N positive condition elements, then the resulting instantiation will refer to N working memory elements which match those conditions. A set-based rule modifies this scheme by allowing the programmer to specify that certain groups of condition elements on the lefthand side should be considered as sets. Therefore, the resulting instantiation can be considered as a set of all the instantiations which would have normally been matched by an instance-based system. By creating righthand side functions which operate on these functions, many algorithms can be represented with greater efficiency and a reduction in rule-firings.

The principal influence of set operations on synchronous parallel rule firing is the reduction of rule firings devoted to fundamentally serial algorithms. For example, counting and marking algorithms can be performed by a single set-oriented production. This simplifies programming and eliminates many control structures which depend on conflict resolution to succeed. Because the overhead of rule instantiation and invocation is reduced for set rules, the waiting time for any rule which must synchronize with the counting or marking task is reduced. However, individual set-oriented rules, because they must operate on greater amounts of data, may take longer to execute than instance-based rules.

Synchronization Groups

Implementing set-oriented rules in an asynchronous rule-firing system is somewhat more difficult. The problem is that set construction proceeds incrementally as one or more working memory elements which stimulate the set rule in question propagate through the network. An asynchronous rule firing scheme may attempt to execute a set-oriented rule before the set has been completely constructed. This could result in the execution of the set rule mul-

tiple times with different data sets, causing inconsistencies within the database. Therefore, it is necessary to determine when the working memory changes which affect a particular instantiation of a set rule are complete and to fire the instantiation only at this time. The problem of associating working memory changes with a particular set-oriented rule instantiation is similar to that of associating working memory elements with particular tasks or of associating rule instances with a particular conflict set.

For now, this problem is solved by using a signaling mechanism. It is assumed that each set operation has some trigger; that is, that the rules are in some sense goal-oriented. Before the triggering element (or elements) is (are) added to working memory, the beginning of a *task* is signalled. After the working memory modifications, the end of the task is signalled. If any set rule is triggered during the period during which the task is active, it is placed on that task's completion list. When the task completes, the rule instantiation is placed in the execution queue as soon as all the working memory changes invoked within the task become quiescent. Note that the idea of task synchronization does some damage to the pure notion of data-directed programming embodied by rule-based systems. In order to create a task group, it is necessary to know in advance when a working memory being created is likely to trigger a set-based rule.

The following righthand actions are used to implement *synchronization groups*:

Generate-sync-group: Returns a pointer to a synchronization group. If invoked within a righthand side, all working memory elements subsequently created by this rule will be considered as members of that synchronization group.

End-group: Terminates a synchronization group. Any rules enabled by the working memory elements within the group will become enabled as soon as all elements have completed matching.

Signal-quiescence-to-group: Used by the working memory match routines to signal to a group that a working memory element has become quiescent. This simply decrements a counter within a critical region and then, if the counter becomes zero and the group has been terminated, allows enabled rules to fire.

Signal-wme-active-to-group: Adds a new working memory element to the synchronization group.

Set-group-completion-demon: Allows a function to be attached to a group to be executed when the group terminates and becomes quiescent.

The synchronization group mechanism is sufficiently flexible to be used for purposes other than synchronizing set-rules, for example, it is used for avoiding race conditions when performing modify actions in parallel.

Syntax of the set notation

The addition of set-oriented rules to OPS5 required changes to both the left- and right-hand rule syntax. Set-oriented rules must have one or more *set patterns* in their left-hand side. A set pattern is a condition element surrounded by square brackets, e.g. `[(block ^color <x>)]`. Any such condition element matches *all* elements which satisfy the pattern. If more than one set pattern is present in the LHS, then one instance of a set production matching the cross product of all these elements will be produced. One can think of a set rule as simply compressing all the rule firings which would occur in standard OPS5 into a single rule firing. The `map-set` function allows the programmer to map RHS operations across the set of rule instantiations.

```
(map-set
    (rhs-action)
    (rhs-action)
       :     :
    (rhs-action))
```

The righthand side actions which are encased in the `map-set` function are mapped across the set of instances. Any variables or condition element variables are bound to the appropriate values in turn. Any righthand side actions which should only be executed once can either precede or follow the `map-set` construct. The set notation can also be used to count occurrences of working memory elements; when a set-oriented rule is executed, the variable `<set-count>` is bound to the number of instances contained in the set.

2.5.4 Map-vector

OPS5 is oriented towards using rules as the basic unit of control This is not particularly efficient as each rule firing consumes an unavoidable amount of overhead in terms of matching, rule instantiation, conflict resolution (if any), variable binding, and so on... Thus, using rules as the basic unit of iterative operations is a bad idea from the standpoint of minimizing the number of rule firings which must be performed serially. The `map-vector` command is an example of the kind of syntactic mechanism which can be incorporated into a rule-based language to allow iterative operations to be performed within the scope of a single rule firing. Much more sophisticated structures than vectors have been incorporated into rule-based languages since OPS5 was written and the `map-vector` command should be considered significant not for its expressive power (which is simply making up for a language deficiency) but for the reduction in sequential rule firings that it allows.

The `map-vector` command is used to map RHS operations over elements in a vector. A vector is a field of a working memory element corresponding to a list of values; it can be thought of as a one-dimensional array of arbitrary length. The vector is not a particularly flexible mechanism and the mechanisms for manipulating them are particularly crude. Iterating over a vector typically involves maintaining a working memory element with a counter and a vector, extracting a value from the vector using a `substr` command, then

deleting the element and reasserting a modified version with an incremented counter. The `map-vector` command has the syntax

```
(map-vector {<ce-variable> | ce-index} vector-name
           {<field-bindings> | nil}
           body)
```

The ce-variable or ce-index is simply a pointer to the element containing the vector. The vector-name is the name of the field in which the vector is stored. The <field-bindings> is a list of the form (<variable-name> key <variable-name> key) where *key* is one of prev | item | rest | index | length | vector-less-item. Body, of course, is the set of RHS actions to be executed in the context of the `map-vector`. The <field-bindings> need some explanation. During the execution of the `map-vector`, each element of the vector is considered in turn. Local variables are bound to the list of elements previously seen(prev), the list of elements yet to be seen(rest), the current element(item), the index of the current element(index), the length of the vector(index), or the entire vector except for the current item (vector-less-item). Because `map-vectors` can potentially be nested, the programmer is given the ability to bind these values to appropriate variable names. The ability to refer to the previous and subsequent items in the vector allows permutations of vector elements to be produced. A typical use of `map-vector` from a travelling salesperson example is show below:

```
(p start-city
  {<start> (start ^start-city <sc> ^length <length> )   }
           (initialized ^value t)
   -->
  (oremove 2)
  (map-vector <start> city-list (item <city> vector-less-item <vli>)
       (bind <tag>)
     (in-parallel
       (make connect-goal ^tag <tag> ^city1 <sc> ^city2 <city>
                          ^length (compute <length> - 1)
                          ^city-list  <vli>)
       (make so-far ^tag <tag> ^distance 0 ^cities-seen <sc>)))
   )
```

Chapter 3

Writing Parallel OPS5 Programs: Structure and Performance

This chapter discusses the problem of actually programming a parallel OPS5 program. The first section discusses the general nature of parallel OPS5 programs and the restrictions that the implementation places on concurrently executing rules in order to ensure program correctness. The remainder of the chapter discusses some of the actual programming issues involved in parallel rule firing. For purposes of illustration, two programs, Toru-Waltz and Travelling Salesperson, are analyzed in terms of their potential for parallelism. Language constructs which are particularly pathological when employed in parallel programs are highlighted and methods of avoiding them are discussed. The use of RHS functions and programming methods specific to UMPOPS are demonstrated. The parallel behavior of these benchmarks is analyzed in terms of both speedup and processor utilization and the features of the programs which contribute or detract from their performance are examined.

3.1 Programming Productions in Parallel

A conventional production system usually has a very definite structure consisting of phases of processing. Each rule belongs to a single phase and is relevant only during that phase. Only a single production is selected and executed in each cycle. Transition between phases is controlled by *mode* working memory elements which are added or deleted by *mode-changing* productions. Order of execution is controlled by a rigid conflict resolution strategy which is frequently used to impose control upon the computation.

This traditional program structure does not support a great deal of parallelism. Because a working memory change is likely to affect only a small number of productions within the current phase, the benefits of node (matching) parallelism is limited. Because only a single production is selected at a given time, there is no opportunity for production parallelism. Even the scope of action parallelism is limited, because there is frequently an implicit assumption that the working memory changes in the righthand side take place in a specific order.

Writing an OPS5 program which takes advantage of parallel rule firings can be done in

several ways. The first and easiest is to find a domain with obvious parallel decompositions so that each subtask can be assigned to one or more rule firings. An example of this is the circuit simulation benchmark[1] in which each device in the circuit is simulated by a separate rule firing. Applications that display such a large amount of parallelism may be few and far between.

Much of the performance gain in programs using rule parallelism will likely result from shifts in the fundamental program writing paradigm. As an example, consider the current meaning of conflict resolution. Rules are expected to conflict, and if more than one is applicable to a given situation, the "best" is chosen. In a parallel system, there is no reason why all productions applicable in a given situation should not be executed, assuming that the number of rules in the conflict set does not grow exponentially. This is equivalent to performing an exploration of a search space in parallel. Because the elimination of conflict resolution reduces the opportunity for heuristic evaluation, the parallel activation of rules gives rise to a number of control issues, The programmer must be able to represent the relative utility of rules, control the number of rules being activated, and be able to terminate incorrect or irrelevant sequences of rule firings. These control issues are the focus of our on-going research in parallel production systems.

3.2 Restrictions on Parallel Rule Firings

When rules are executed concurrently, the potential exists for interactions between the rules which can potentially leave working memory in an inconsistent state, or which can produce results which could not be achieved by any serial execution order of the productions [Ishida and Stolfo, 1985, Schmolze, 1989, Schmolze, 1991]. UMPOPS provides mechanisms for detecting and preventing interactions due to positive interactions, however, this is only sufficiently powerful to allow correct programs to be written, it does not guarantee correctness. So the programmer is required to design rule-based programs so that interactions do not occur. The basic assumptions underlying correct parallel programs are listed below:

- Only one production may modify a given working memory element during an execution cycle (enforced by locking).
- A production may not reference or modify a working memory element which is being (or has already been) modified by another co-executing production. Because the *instantiations* of a rule contain their own copies of relevant working memory, there are circumstances where it is valid (or at least harmless) to refer to a working memory element which is currently being modified.
- A rule which contains negative condition elements should not be executed if any rule is currently executing which would add a working memory element which would disable the rule. (*Not* enforced by locking.)
- *All* working memory operations which affect a particular production must be completed before that production is scheduled and executed.

[1] Supplied with the release as an example program.

The last restriction is to avoid firing transient instantiations of a production when employing an asynchronous rule-firing policy. If the entry of a rule instance into the eligibility set is not *monotonic*, the rule should not be fired asynchronously. For example, given the following production and changes to working memory, a transient production instantiation appears in the eligibility set; because it will be immediately disabled, it should not be executed.

```
(p example-prod
  (A)
 -(B  ^field2 wombat)
  -->
   ...)

(remove B ^field1 wallaby ^field2 wombat)  ---> example-prod into eligibility set.
(make B  ^field1 koala ^field2 wombat)    <---  example-prod out of eligibility set.
```

The changes in the above example are exactly those which would take place by the execution of the OPS5 statement `(modify <B> ^field1 baz)` in the righthand side of some rule.

3.3 Analyzing Two Benchmarks for Rule Parallelism

This section discusses the construction of two parallel rule-based programs in terms of their potential for various levels of parallelism: node, action, rule, and for their potential for asynchronous rule execution. The first benchmark was written by Toru Ishida and modified by the author; it will be referred to as Toru-Waltz[2]. The second program is an implementation of the travelling salesperson problem which illustrates the issues underlying the implementation of heuristic control in a parallel asynchronous rule-based program[3] .

3.3.1 Analyzing the Toru-Waltz Benchmark for Rule Parallelism

It turns out that the Toru-Waltz program is reasonably amenable to parallel rule execution for a fundamental reason – during the course of the program, conflict resolution is never[4] used for the purpose of distinguishing between two valid rules. In most cases, if a rule appears in the conflict set, it is either executable, superfluous, or transient. So the principal problems in adapting this program to parallelism is removing extraneous rules from the conflict set and firing the eligible rules at the earliest possible time without accidentally executing a transient instantiation.

The Toru-Waltz benchmark is well-suited for parallel rule execution. It is divided into a number of stages, each of which supports a considerable degree of concurrent rule firing.

[2]Schmolze has referred to this benchmark as 'Neiman-Waltz', but modesty forbids...

[3]The text of the benchmarks is given in the appendix.

[4]Well...hardly ever.

- Initialize: Creates a database of the legal junction labels.
- Make-data: Loads the scene to be analyzed into working memory.
- Enumerate-Possible-Candidates: Lists all the labellings for all the junctions.
- Reduce-Candidates: Eliminates all illegal line labellings.

Both the `initialize` and `make-data` phases of the computations simply consist of adding data to working memory. Rule parallelism can be employed by executing both rules concurrently. To further reduce the initialization overhead, action parallelism can be used to assert the initial working memory elements concurrently. In Toru-Waltz, initialization time was reduced by approximately a factor of ten by combining rule and action parallelism.

The `enumerate-possible-candidates` phase of the computation is ideal for asynchronous rule parallelism. Each instantiation is monotonically enabled by the creation of a junction in the `make-data` phase of the computation. Each instantiation corresponds to a unique junction and labelling, so rules never conflict. The righthand side of the `make-data` and `enumerate` rules perform no `modify` commands, so no transient instantiations appear in the conflict set. Although the `enumerate` rules contain a negated condition element, its sole purpose is to render the rule self-disabling.

A brief digression is in order here. In the discussion of the rationale underlying the selection of the locking mechanism, it has previously been asserted that attempts to automatically ensure serializable behavior require both a compile-time and run-time component. Then why should we feel that it is possible to *design* a parallel rule-set which executes correctly when design, of course, precedes run-time? The answer is given above: because the designer is granted a certain knowledge of the nature of the data which is input to the program, it is possible to make deductions about the uniqueness of each rule instantiation. For example, in Toru-Waltz, we know that each junction is assigned a unique label, and therefore there will only be one rule instantiation for each junction/possible-junction-label combination. Thus, while an automated interaction detection mechanism must rely on run-time analysis of rules with instantiated variables to conclude that each instance of the `enumerate-possible-line-label` rule can be run in parallel, the designer, by means of a certain omnipotence, can conclude the rules will not interact. This, of course, raises the potential of including syntactic methods of conveying this information to compile-time analysis methods, but given the philosophy of designing for parallel activity, the principal virtue of such a mechanism would be to verify the correctness of the programmer's design.

Because the enumeration rules are added to the eligibility set monotonically (that is, once added, a rule instance will never be removed from the conflict set by another rule firing), it is possible to combine the enumeration and initialization phases so that the `enumerate` rules can fire asynchronously as soon as they are enabled. Because the mode-changing mechanism continuously monitors the rule demons for quiescence and the eligibility set for contents, there is no possibility that the system will move onto the next phase before the initialization and enumeration phases have terminated.

The `reduce-candidates` phase consists of rules which detect junctions whose labels are not consistent with any possible labelling of adjacent vertices; these junctions are then

deleted. Deleting elements which represent junctions is the only action which takes place during the reduce-candidates phase of processing. It takes a certain amount of inspection to conclude that the rules in this phase can actually execute concurrently. This is because it is actually possible for rules to mutually disable each other by deleting existing possible-line-label elements. By examination, it can be seen that this interaction is actually harmless; the effect of executing both rule instantiations concurrently is to execute a redundant `remove` operation which is actually semantically valid in OPS5. However, because the labelling-candidate element is referenced via a positive condition element, the working memory locking mechanism of UMPOPS automatically detects the interacting rule instances and prevents one of them from firing.

3.3.2 Mode Changes

One serializing feature in Toru-Waltz (and most other rule-based programs) is the use of the modal or gating type of working memory element. An example of this usage is the *stage* working memory element in the Toru-Waltz benchmark. Modes are typically used to distinguish between the major stages of a computation. In cases where there is no significant parallelism between the stages, the use of mode elements serve a useful purpose in denoting an explicit partitioning among rules. Specific semantics can be assigned to each mode change; for example, in the Toru-Waltz program, the `reduce-candidates` mode declares that no new junctions will be created, therefore the relaxation phase can begin. If stages of the computation can overlap or be pipelined, the use of mode elements can cause unnecessary serialization of the computation; in these cases, the rules in the overlapping stages should be placed in the same partition so that they may fire asynchronously.

The use of modal working memory elements can seriously slow down a computation because otherwise eligible productions can not enter the conflict set until the mode of the element is changed. Typically, many productions are enabled by a modification to a modal element and creation of instantiations is relatively expensive, so the overhead of a mode-changing rule can be very high. What is more, because the order of matching within the Rete net is determined by the order of the condition elements within the rule, the traditional location of the gating condition element as the first element in the rule prevents partial matching from occurring between other elements within the rule. This causes a significant amount of matching to take place once the gating element finally arrives (see Figure 3.3.2). To give an idea of the magnitude of the problem posed by mode-changing rules, the single rule `go-to-reduce-candidates` can consume anywhere from 33% to 50% of the run-time of Toru-Waltz (out of a total of 370 rule firings) depending on whether match-level parallelism and asynchronous rule-firing are enabled.

The delay due to the gating effect can be minimized by placing the gating element as the final positive condition element in the rule. This positioning allows more partial matching to occur before the gate element is created. This technique only works if the gating element is not used to pass parameters to the rule, that is, no field in the gating element may be unified with any field in any other condition element of the rule. The problem with this re-ordering of terms is that placing the gating condition as the final element may cause the rule to partially match in situations in which the rule is not applicable. The overhead

caused by this unnecessary matching may exceed the advantage gained in minimizing rule activation time by changing the placement of the gating element. The exact nature of the trade-off can only be determined by examining each case separately.

Parallelism can be used to minimize the delays caused by gating in two ways. Because a gating element typically affects many production instantiations, node parallelism, as mentioned previously, can be effective in minimizing the time consumed in the matching process. In Toru-Waltz, the time consumed by the `go-to-reduce-candidates` mode-changing rule is reduced by a factor of five by using match-level parallelism. If asynchronous production execution is allowed, then productions enabled by the addition of the modal element can be executed as soon as they enter the conflict set, thus maintaining processor utilization and avoiding a bottleneck when processes have to be assigned to each instantiation in the conflict set.

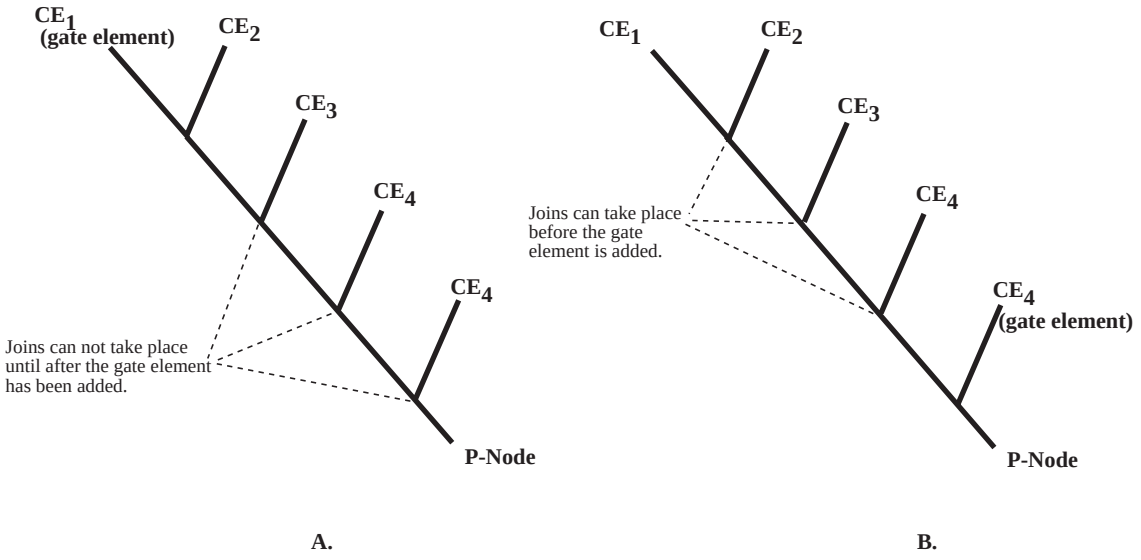

Figure 3.1: The location of the gating element affects the amount of partial matching which can take place in the match process.

3.3.3 The Travelling Salesperson Benchmark

The Toru-Waltz program was simple to parallelize because it presented opportunities for *data* parallelism in which individual rule instances could fire on unique data items. No control mechanisms were required because one instance fired for each potential junction labelling. The travelling salesperson benchmark (TSP) is a more complex example because the search process has to be managed so that there is no interaction between working memory elements in different search states, and rules must be heuristically pruned in order to reduce execution time to a reasonable number of rule firings. Finally, solutions from different paths must be compared and the best solution selected.

Search spaces are managed in TSP by assigning all elements in the same state a unique identifier. In the example shown below, a new state is created by creating a new tag using the `ngenatom` command. (NGENATOM generates a unique integer rather than a symbol in order to avoid the overhead of interning a variable which is quite high in a parallel system). The tag is used to annotate the working memory elements which comprise a state in the search space, i.e. the `so-far` and `connect-goal` element. The `so-far` element records the distance travelled and the cities seen, while the `connect-goal` element records the next city to be visited. Because each combination of elements in a state are unique and there is only one operator in TSP, there will be only one rule instance firing per state. This allows the use of the `(meta (lock-not-required t))` notation; because of the partitioning, no rules can conflict during the propagation phase and the overhead of locking working memory elements is not necessary. As will be seen shortly, this is not the case when asserting solutions.

Note the use of the `map-vector` command which allows iteration across vectors to be performed in the righthand side, thus avoiding having to use multiple rule firings to achieve the iteration. Because the nature of the search process in Travelling Salesperson is that few rules are initially active, the `propagate-cities` rules were divided into two types. The initial `start-cities` rule and rule instances matching states with more than five cities left to visit have action parallelism invoked in their righthand sides so that matching need not be completed before the next state can be constructed and visited. As more states in the search space are created and more rules become more active, action parallelism is not longer used. This is because with fewer successor nodes, the righthand sides do not take as long to execute and, with more active states, all available processors are already usefully employed in performing search.

```
(p propagate-city-5
   (meta (priority 1) (control-fn compare-with-solution)
         (priority-queue t)
         (lock-not-required t)
         (priority-fn propagate-city-priority-fn)
         (rhs-kill-actions ((oremove <sofar>)
                            (oremove <cg>)))
         )
   {<sofar> (so-far ^tag <tag> ^distance <d-so-far> ^home-city <home> ) }
   {<cg> (connect-goal ^tag <tag> ;^est-cost {<> nil <e-cost>}
                       ^city1 <city1> ^city2 <city2>
                       ^length { >= 5 <l> } ) }
   (distance ^city1 <city1> ^city2 <city2> ^distance <d>)

-->
   (bind <cities-seen> (litval cities-seen))
   (oremove <sofar>)
      (oremove <cg>)
   (map-vector <cg> city-list ( item <new-city> vector-less-item <vli>)
      (bind <newtag> (ngenatom))
```

```
  (in-parallel
   (make connect-goal ^tag <newtag> ^city1 <city2> ^city2 <new-city>
        ^length (compute <l> - 1)
        ^city-list  <vli> )
   (make so-far ^tag <newtag> ^distance (compute <d-so-far> +  <d>)
                ^home-city <home>
          ^cities-seen
                (substr <sofar> <cities-seen> inf) ;previously visited
                   <city2> ) ;and the new city
)))
```

The meta-information in the above rule states that its basic priority is 1 (and all instances of this rule will be placed in queue number 1), that this is a priority queue, the function that computes the priority is `propagate-city-priority-fn`, the heuristic control function associated with this rule is `compare-with-solution`, and if the rule is pruned by the heuristic control function, it should remove the working memory elements indicated by the condition elements <sofar> and <cg>. Note that both the priority function and the heuristic control function are encoded in lisp, but are executed within the context of the rule's righthand side and can therefore access not only global Lisp variables, but also can reference any OPS variable bound in the rule's lefthand side by using the OPS5 $varbind function.

Heuristic Control in TSP

The TSP problem is NP-hard, and if all possible solutions were examined, the search space would grow unmanageably for even small values of N. There are a number of well-known admissable heuristics for ordering the traversal of the search space in TSP; the one chosen for this example is the minimum spanning tree[Pearl, 1984]. Rules are placed in the execution priority queue according to the value returned by the MST heuristic. A record is kept of the best (lowest) solution developed so far, and if the value of the heuristic exceeds this value for any search space, the corresponding rule instance is not executed, instead, the rule's *kill-actions* are executed to remove the corresponding working memory elements and reduce the size of memory.

The heuristic functions are implemented procedurally in Lisp and are specified using the LHS `meta` notation. An early version of this program attempted to compute the MST heuristic using rule firings, however, this resulted in more rule firings to compute the heuristic than to perform the search. For the same reason, rules are pruned using procedural functions in the rule demons; at the level of granularity of search operations in TSP (one rule per state), meta-rule implementations of control methods are not efficient.

Asynchronous Rule-Firing in TSP

An asynchronous rule-firing policy is used in the travelling salesperson program. Because rules are scheduled and executed as soon as they enter the eligibility queue, there is never a time when the system is quiescent and it is impossible to select the 'best' of all possible

solution paths as indicated by the heuristic evaluation function. It is reasonable to ask whether this causes any degradation in the quality of the solution process as measured by the number of nodes expanded in the search space. In fact, the number of nodes expanded by the TSP program is approximately the same whether the program is executed serially or asynchronously in parallel. There are a number of reasons why this is the case. First, solutions tend to be placed in the execution queues faster than they can be handled by the rule demons. Because the `propagation` rules are placed on a priority queue, the lower quality rules tend not to be executed until after a solution has been found; they can then be pruned without being executed. The second and most important reason why asynchronous rule firing is admissable is that the heuristics used for TSP are only reasonably precise. Each invalid solution path must be developed to a certain depth before the estimate of the quality of the developing solution becomes good enough that the path can be pruned. So, in even a strictly best-first algorithm for solving TSP using the minimum spanning tree heuristic must develop a minimum number of nodes in order to ensure that the current solution is indeed the minimum distance. Executing rules asynchronously just means that a certain amount of this work takes place before the solution is found. An analysis of the parallel nature of the travelling salesperson problem can be found in [Kumar *et al.*, 1988].

Merging Solutions

The Travelling Salesperson problem was developed primarily to illustrate two points about parallel rule-firing; the elimination of the need for rule interference detection by partitioning the problem into independent states (this idea is developed further in [Neiman, 1992]) and the idiom for merging results from parallel search processes. Eventually, each parallel search path which has not been pruned terminates and posts a possible solution. Only one solution is acceptable and when competing rules post solutions there is a chance for conflict; either the posting of multiple solutions or the posting of an inferior solution.

Consider the first case: in order to create a `solution` element, one needs a rule of the form:

```
(p init-solution
   (meta (priority 0))
   {<new> (solution-goal ^distance <dist> ^tag <tag> ) }
         -(solution-goal ^distance < <dist>)
         -(solution)
  -->
 (bind  <cities-seen> (litval cities-seen))
  (make-unique solution ^tag <tag> ^distance <dist>
        ^cities-seen (substr <new> <cities-seen> inf)))

    (If there is a goal to create a solution,
      and there is no goal to create a better solution
      and there is no current solution
     Then
      create a solution element.)
```

If rules were executed instantaneously, then there would be no difficulties with this rule. However, there is an unavoidable delay between the instantiation of a rule and its firing. This leads to the following possible scenario. An instantiation of the `init-solution` rule arrives with a goal to create a solution of say, 10,000. It is scheduled to fire; once on the execution queue, the rule cannot be disabled, even if a better solution goal arrives. Suppose then, a better solution (produced by some parallel search process) does arrive while the first instantiation is still on the execution queue. It is also scheduled and placed on the execution queue. The end result of this scenario is that two solution elements are posted in working memory and a possibly erroneous result is produced. How can this be prevented? The locking scheme cannot prevent this situation because the element to be locked, the `solution` element, does not exist until after the execution of the initialization rule. Instead, we provide a mechanism for creating a *unique* working memory element. A unique element is defined as a class of working memory element with one or more optional key fields. Only one element with a given combination of working memory class and key field values is allowed to exist. In the above scenario, if the `solution` class is declared to be unique, the second rule instantiation will not be allowed to fire, because an instance of the `solution` element is already being created. Thus, the rule instance will be disabled and the assertion of the `solution` element will trigger an instantiation of the `new-and-improved` rule, causing the correct solution to eventually be asserted.

```
(p new-and-improved
 (meta (priority 0))
 {<new> (solution-goal ^distance <dist> ^tag <tag> ) }
       -(solution-goal ^distance < <dist>)
 {<old> (solution ^distance > <dist>)  }
-->
 (bind  <cities-seen> (litval cities-seen))
 (make solution ^tag <tag> ^distance <dist>
       ^cities-seen (substr <new> <cities-seen> inf))
 (remove <old>)
 )
```

A similar possibility for interactions occurs with the `new-and-improved` rule; two solutions may compete to modify the existing solution. Once again, multiple copies of the `solution` element could potentially be created. In this case, however, an existing element is being replaced and the rule instantiation replacing it must first acquire a write lock on that element. Thus, only one instance of the `new-and-improved` rule is ever scheduled to fire at a given time.

There are a number of points of interest in the solution merging rules: First, note that the correctness of the solution merging is guaranteed. If a superior solution is ever locked out, the assertion of the new `solution` element will re-trigger the instantiation; thus, the data-directed nature of the rule-based system serves to automatically correct temporary errors. Although the correct solution will always eventually be asserted, it is important that the solution-merging rules perform a check to ensure that the solution being asserted is from the best known solution goal. Otherwise, if many solution-goals were present, each time the solution was modified, it would take many cycles of rule firings before it was

ensured that the correct solution was achieved; and many rules would be locked out during each of these cycles.

Finally, note the order in which the actions take place in the `new-and-improved` rule. The new solution is asserted and then the previous solution is removed from memory. This is to avoid the creation of transient instantiations of the `init-solution` rule. Although the unique-lock mechanism would prevent the `init-solution` rule instantiations from firing, avoiding the overhead of creating the instantiations is a good idea.

3.3.4 Queue Latencies in TSP

TSP was chosen as a benchmark because even small problems generate large enough search spaces to provide opportunities for rule-level parallelism and the necessity for heuristic pruning. Each node in the search space (except for the leaf and penultimate nodes) generates multiple successor states and thus the rate of rule generation will exceed the execution rate for any reasonable number of processors. The scheduled but unexecuted rules (which represent the *open* list) remain on the execution queues until processors become available to execute them. This queue latency time can greatly exceed the actual time of rule execution. It's interesting to briefly examine the effects of queue latency on the performance of the benchmark.

Because the `solution` element is used for performing heuristic pruning, it is necessary to assert new solutions as rapidly as possible. If there were only one execution queue, new solutions would remain on the execution queue for extended periods of time. Propagation and solution rules would compete for resources and inferior search paths would be traversed during the interval between the scheduling of a solution rule and its execution. For this reason, rules which assert solutions are given a higher priority than propagation rules, ensuring that the information used for heuristic pruning is as current as possible.

Once an initial solution has been found (not necessarily the final solution), heuristic pruning can take place. At this point, it would appear possible that the node which would generate the final solution could potentially languish in the execution queue for a long time while lesser solutions were developed, and this could adversely affect the performance of the benchmark. In fact, the combination of the use of priority queues and a reasonable heuristic cause the solution to be developed rather expeditiously. Because the search problem then has to develop many additional nodes in order to eliminate the possibility of a better solution, it turns out that queue latency is not a major problem in a heuristic program that requires a "best" solution. In fact, the contents of the execution queues can be divided up into rules representing nodes that must be "opened" and nodes that will eventually be pruned; it is only if a significant number of the latter are executed while waiting for the former that queue (and scheduling) latency becomes a problem.

3.4 Performance Analysis of Toru-Waltz and TSP

The rationale behind parallelizing a rule-based system is to increase the performance of the system. Ideally, the nature of the speedup should be linear or near-linear to the number of

available processors. In practice, there are a number of effects which limit the performance of UMPOPS and they are discussed in this section.

3.4.1 Methodology of Benchmarking Programs

As has been discussed earlier, UMPOPS has been instrumented to record timing information during execution. The most visible and useful measurement is simply the time required to execute a particular benchmark program. Because of the time-sharing nature of the Sequent's operating system and the user's inability to prevent processors from being redirected to perform system tasks, there can be a significant variation between runs of the same benchmark. The loss of processors during a parallel run can be especially significant when the swapped process was in the midst of a critical region; in this case, the delay is multiplied by the number of processors attempting to enter that region. The variability of run times is most noticeable during benchmarks which require a number of processors near the limit of the machine; because these runs also tend to be the shortest, the variation due to system scheduling can range from 10%–20%. For these reasons, when benchmarking, the simple expedient of choosing the best time over a series of runs was adopted. Thus, the following benchmarks should be viewed much as you would view the EPA stickers on a new car – your mileage may vary.

3.4.2 Performance Measurements: Toru-Waltz

The speedup due to all levels of parallelism in Toru-Waltz appears to be roughly a factor of seven or eight, independent of the number of processors available beyond that number (see Figure 3.4.2). Because both action and match parallelism are employed in Toru-Waltz to reduce the overhead due to initialization and mode-changing, the overall speedup is not expected to be particularly linear. As a general rule, each successive level of granularity of parallelism, rule, action, and match, generates less of an overall speedup due to the proportionate overhead required in generating the parallel action. Not only is the overhead of smaller granularity operations greater relative to the cost of the operation, but, because there are typically many more small granularity operations, the absolute cost of the overhead becomes significant. The speedup for action parallelism in a single rule is approximately 8-fold in UMPOPS. Match-level parallelism yields a speedup factor of five in the single mode-changing rule in which it is employed[5].

For an insight into the performance of Toru-Waltz, we can examine the processor utilization graph for this benchmark (see Figure 3.4.2.).

During the initialization phase, Toru-Waltz makes use of rule and action parallelism. Two rules fire simultaneously to add data to the system and each rule employs action parallelism to reduce run time. Because all actions must be completed before moving to the next phase, two processors must be reserved to perform synchronization and the maximum level of action parallelism is 12 processors. During the mode-change from the `enumerate` phase to the `reduce candidates` phase, only a single rule can fire and match parallelism is

[5]There is no reason to think that this speedup is an absolute limit, but optimization of match-level parallelism is not a high-priority in this research project.

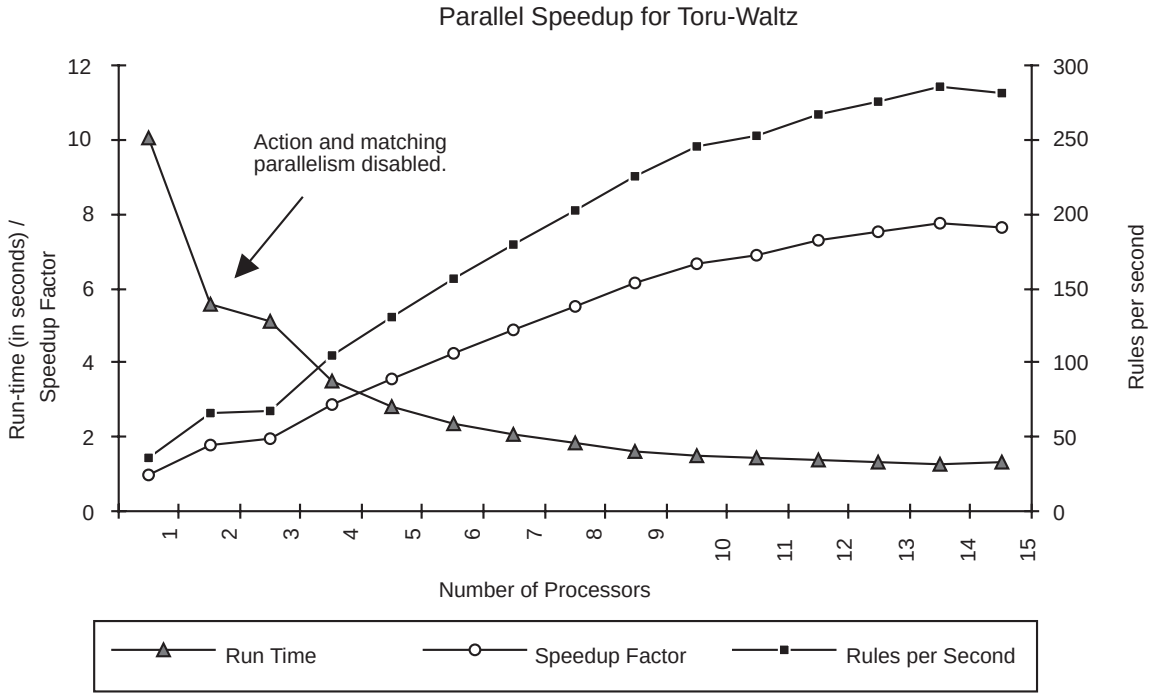

Figure 3.2: The parallel speedup graph for Toru-Waltz. The left axis shows both run-time in seconds and the speedup factor. The right axis shows the average rule firing speed per second for each run.

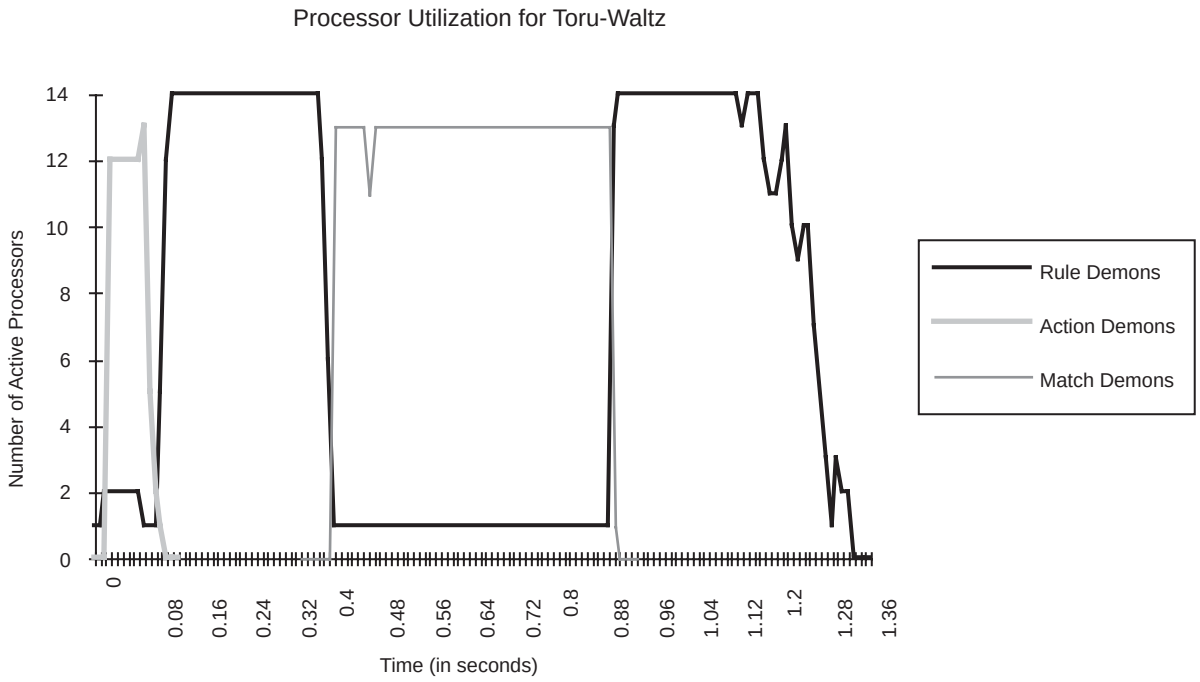

Figure 3.3: The processor utilization graph for Toru-Waltz with 14 "demon" processors.

employed to reduce the serial bottleneck. Finally, as the algorithm approaches completion, the amount of work to be performed (and the demand for processors) falls off significantly. This is due to the nature of the constraint propagation algorithm; each junction is connected to at most three other junctions and thus, each rule firing can initiate at most three rule firings. Because the Toru-Waltz benchmark is fairly small, the significance of these three limiting phases is relatively large, and the potential speedup is limited.

The principal bottleneck in Toru-Waltz is the overhead due to the `go-to-reduce` mode-changing production. If this rule is included, the rule execution rate for Toru-Waltz with 14 processors is 280 firings/sec while if this rule is disregarded, the average rule execution speed is 380 firings per second and exceeds 500 rules/sec in the reduce phase.

3.4.3 Performance Measurements: TSP

The processor utilization for the Travelling Salesperson benchmark is shown in figure 3.4.3; as can be seen in this graph, the level of rule parallelism is very high in this benchmark after the initialization phase has been completed. Theoretically, the speedup due to rule parallelism in the Travelling Salesperson Problem should be essentially linear and, as can be seen in figures 3.4.3 and 3.4.3, this is approximately the case. The results shown in this graph are the best results of a series of benchmarking runs. Speedup factors are measured with the initialization rules included and with them deleted from the total runtime. The

best observed speedup is approximately 12 times using 15 processors[6]. In general, the observed speedup departs from the linear due to the time-sharing nature of the Sequent's scheduler and for reasons discussed in the following section.

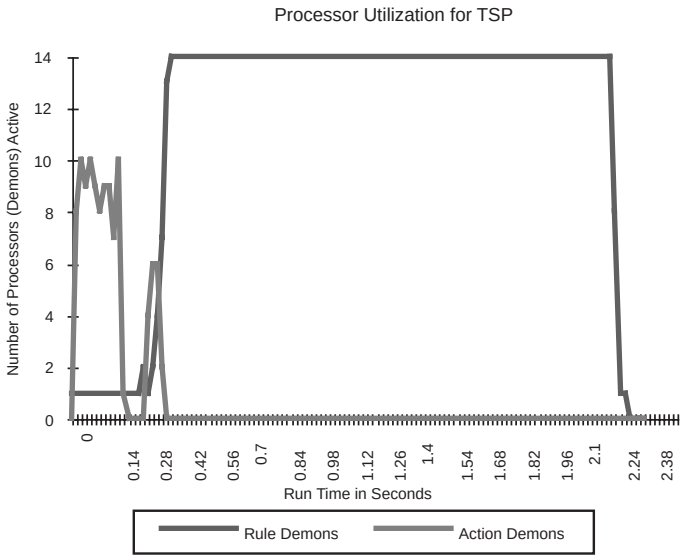

Figure 3.4: The processor utilization for the TSP benchmark.

Contention and Cache Failures

In our experiments with TSP, we have seen a steady decrease in run time (increase in performance) as more processors are applied to the problem, however, the increase departs from the linear as the number of processors increase. There are a number of possible explanations for this decrease including the character of the Sequent's scheduler, contention for resources within the Rete net, and cache faults within the shared memory architecture.

Although the contention of the various queue demons for the rule, match, and action queues would appear to be a potential bottleneck, measurements of processor utilization indicate that processors spend very little idle time when performing rule parallelism.

Measurements of both rule execution times and the time that processors spend attempting to gain access to critical regions indicate that contention for resources within the Rete net and for the eligibility set might account for as much as 5% of the decrease from linear performance. In TSP, average rule execution times tend to increase by only a small amount, no more than 3/1000 of a second as the number of processors utilized increases, reflecting a slight contention for resources. This increase is roughly 5% of the average rule execution time. Contention for resources can rise considerably if the number of processors being employed approaches the number available on the machine available processors are employed and other processes require cycles. Under these load conditions, it becomes probable that

[6]In UMPOPS, one processor is always reserved for the scheduler and therefore, the maximum number of rule demons is 15 on a 16 processor machine.

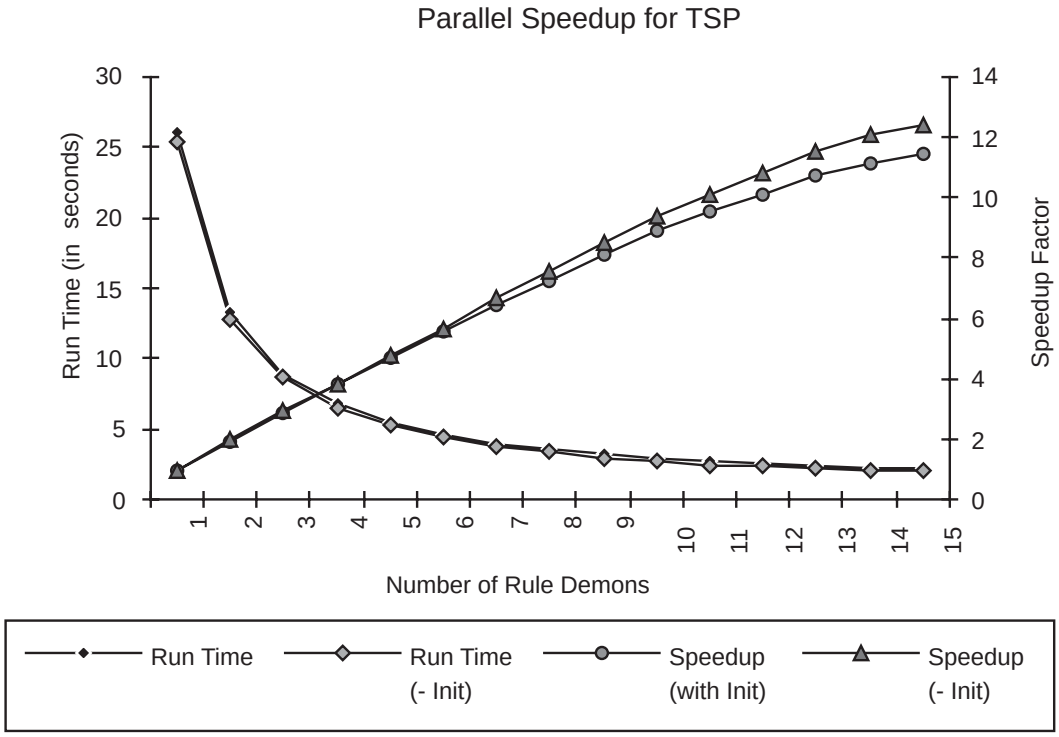

Figure 3.5: The parallel speedup for the TSP benchmark. One set of lines shows the decrease in run-time as the number of processors increases; the other shows the ratio of parallel run-time to serial run-time.

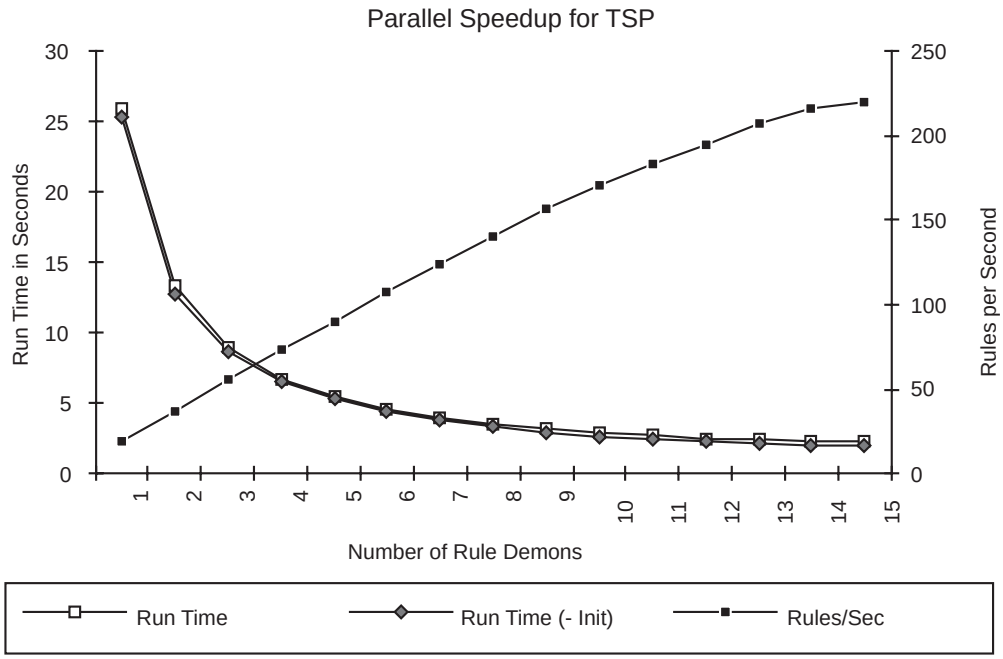

Figure 3.6: The parallel speedup for the TSP benchmark showing run-time and the number of rules firing per second.

a processor will be swapped out while it has acquired control over a critical region; the swapping delay is then multiplied by the number of processes waiting to get access to that critical region.

There is one anomaly associated with the average rule execution time metric; it jumps significantly when moving from serial to parallel execution. The explanation for this is apparently the loss of cache consistency as multiple processors begin accessing the same areas of shared memory, particularly within the Rete net memory nodes. Unfortunately, the Sequent operating system does not provide any tools for measuring this effect directly and this explanation must remain a hypothesis. It is to be expected that the number of cache faults would increase with the number of executing processors, and the slight increase in average rule execution times may also be due partly to bus and cache contention.

The final explanation for the decrease in linearity is simply that as the run-time of the benchmark decreases, small effects due to the time-sharing scheduler, paging and other artifacts of the operating system, and even the time required to access and store timing data become more significant.

3.5 Summary – Programming Parallel OPS5

This section has examined two programs written to take advantage of parallel rule execution. These programs illustrate two situations in which parallelism is effective: data parallelism, in which operations can be applied to many different data objects in parallel, and parallel

search, in which many search paths can be explored concurrently. Some of the issues underlying parallel rule-based programming have been discussed, including effective use of lock mechanisms, avoiding transient instantiations, merging solutions, and making effective use of action- and match-level parallelism during initialization and mode-chaning rules. In both programs, an asynchronous rule-firing control policy was used and, in the travelling salesperson problem, it was shown how heuristic control mechanisms could be incorporated into a rule-based program without resorting to synchronizing conflict resolution schemes. The performance characteristics of each program were analyzed and it was shown how multiple levels of parallelism could be used to optimize performance. The limited number of processors available on the Sequent prevents the determination of the full speedup due to parallelism for these benchmarks. However, the fact that each benchmark is able to fully utilize all available processors for much of their execution indicates that many additional processors could gainfully be employed.

Chapter 4

Implementation

This chapter discusses the data structures and algorithms which are used to implement the parallel OPS5 matching process and the changes that were necessary to allow parallel activity. During the re-implementation process, it was discovered that there are several assumptions concerning the order in which activities take place within the matcher which are no longer valid in a parallel system – the potential errors and their solutions are also described in this section.

Many of the details of the implementation were inspired by Gupta's study of the issues involved in parallelizing the Rete net [Gupta, 1987]. Because this work is undoubtedly familiar to the interested reader, this report concentrates primarily on the implementation details which are unique to parallel OPS5, particularly the synchronization of two-input nodes.

4.1 The Rete Net

Because the following discussion hinges on an understanding of the internals of the OPS5 pattern matching process, a short overview is given of the processing which takes place within the Rete net, the principle data structure in OPS5.

In production systems, most of the processing time is spent determining which rules are eligible to fire. In OPS5, this process consists of matching the lefthand sides of productions against working memory. When a set of working memory elements is found such that there is a working memory element for every non-negated condition element in the lefthand side and there exist no elements which match negated condition elements, the rule is eligible to fire. As a principal bottleneck in rule firing, this matching process should be as fast as possible.

The matching process in OPS5 takes place using a data structure called the *Rete net*. The Rete net is an efficient implementation of a pattern matcher based on the following observations:

- Working memory changes only incrementally from cycle to cycle.

- Many productions in a rule base are frequently structurally similar and may share one or more terms.

The first observation implies that it should be possible to store partial matches and only match against those working memory elements which change, rather than implementing the naive approach of comparing each production against all of working memory after each set of working memory changes. Sharing of tests between productions reduces the total number of comparisons that must take place.

Rete Net Overview: The matching process works by passing tokens consisting of one or more working memory elements through the net, performing tests on them at each node. The 'top' of the Rete net is composed of *alpha* nodes which consist of simple tests on the class of the working memory element and specific fields. This part of the network possesses no memory and resembles a conventional discrimination net; tokens are passed to suceeding nodes in the network only if the tests at the current node succeed. Alpha tests are not very time-consuming and parallelizing their execution does not lead to large improvements in performance.

Beta tests are responsible for unifying variable values between two condition elements (inter-element tests). Each of the beta nodes has two inputs and two memories, one associated with each input. As a token arrives at a beta node, it is stored in memory and tested against the *opposite* memory to see if one or more consistent bindings can be achieved. If so, a new token is constructed from the incoming token and the stored token. This new token is then propagated through the beta node's out list (a list of successor nodes). The memories associated with the beta nodes store partial matches, making it unnecessary to repeat the entire computationally expensive unification process after each working memory modification. The cost of executing a beta node is proportional to the size of the memory against which the incoming token is tested. The two primary beta nodes are the AND and NOT nodes. Beta nodes present numerous opportunities for parallelism; for example, multiple beta nodes can be executed in parallel, or, if the architecture supports sufficiently fine-grained processing, an incoming token can be compared to each corresponding token in memory simultaneously. Beta nodes also present a number of obstacles to implementing parallelism. First, they contain memory nodes which must remain consistent despite possible parallel accesses. Secondly, each beta node refers to at least two tokens which can change asynchronously during the match process. Finally, new data may arrive during a match episode; synchronization constructs are needed to ensure that the new data does not stimulate spurious matches or none at all.

At the bottom of the Rete net is a series of *production* nodes; when a token arrives at one of these nodes, the production corresponding to the node is placed in the conflict set, instantiated with variable bindings from the incoming token. The production node has no memory, thus only one production firing ever results from a given combination of working memory elements.

AND Nodes: The operation of an AND node is illustrated in Figure 4.1. In part A of the figure, a token is shown arriving at the memory node of the AND. The token (which

represents a partial match) is inserted into the memory of the AND node and then processed (part B). (In a serial system, it does not matter whether the node is placed in memory before or after processing, although this is not the case when node parallelism is allowed.) Part C of the figure shows the processing of the AND node. The incoming token is compared to each token in the *opposite* memory according to the list of tests contained within the node. A typical test might compare the value of the third slot of the second element of the incoming token to the fifth slot of the first element of the memory token.

Pairs of tokens which satisfy the tests are concatenated into a new token and passed to the succeeding nodes in the network. Because an AND node is basically symmetrical, this description covers the case of tokens arriving from both the left and right sides. In the case of a *negated* token (that is, a token resulting from a `remove` working memory command), the AND node functions in the same way except that the token is removed from the memory node and, if the token satisfies the tests, a new *negated* token is passed to succeeding nodes so that partial matches will be deleted from memory nodes lower down in the network. If the succeeding node is a production node, then the negated token is used to remove the corresponding instantiation from the conflict set[1].

NOT Nodes: A NOT node is used to implement negated clauses in a production lefthand side. The NOT nodes are structurally similar to AND nodes, but the processing is quite different. A NOT node must ensure that for a given negated clause, there is no working memory element that matches that clause in such a way that there are consistent variable bindings with the working memory elements matching the preceding LHS clauses. Like the AND node, the NOT node has two memories. One memory is devoted to working memory elements which potentially match the negated clause. The other memory contains a list of tokens corresponding to the non-negated condition elements of the LHS and, associated with each token, a count of the number of matches which occur in the opposite memory. The processing of tokens arriving at a NOT node differs according to whether the token arrives from the left or righthand side.

A token arriving from the left (the choice of sides is arbitrary) represents a list of elements which match the lefthand side condition elements of the production; this token will be propagated through the net only if no token is present in the opposite memory which satisfies the tests of the NOT node. The arriving token is placed in the lefthand memory of the NOT node and assigned a counter value of zero (Figure 4.1, parts A and B). For each element in the right memory, the test is performed; if successful, the counter is incremented. If the count is zero after the entire righthand memory has been examined, the token is propagated.

A token arriving from the right (Figure 4.1, parts A and B) is placed in the righthand memory. Then, for every token in the lefthand memory, if the tests are satisfied, then the corresponding counter is incremented. If the counter was formerly 0, then the new token has disabled the production; in this case, the lefthand token is negated and propagated to

[1] A negated token should not be confused with a negated condition element. A negated token is simply a token tagged for removal while a negated condition element specifies a working memory element which must not exist if a rule containing it is to fire.

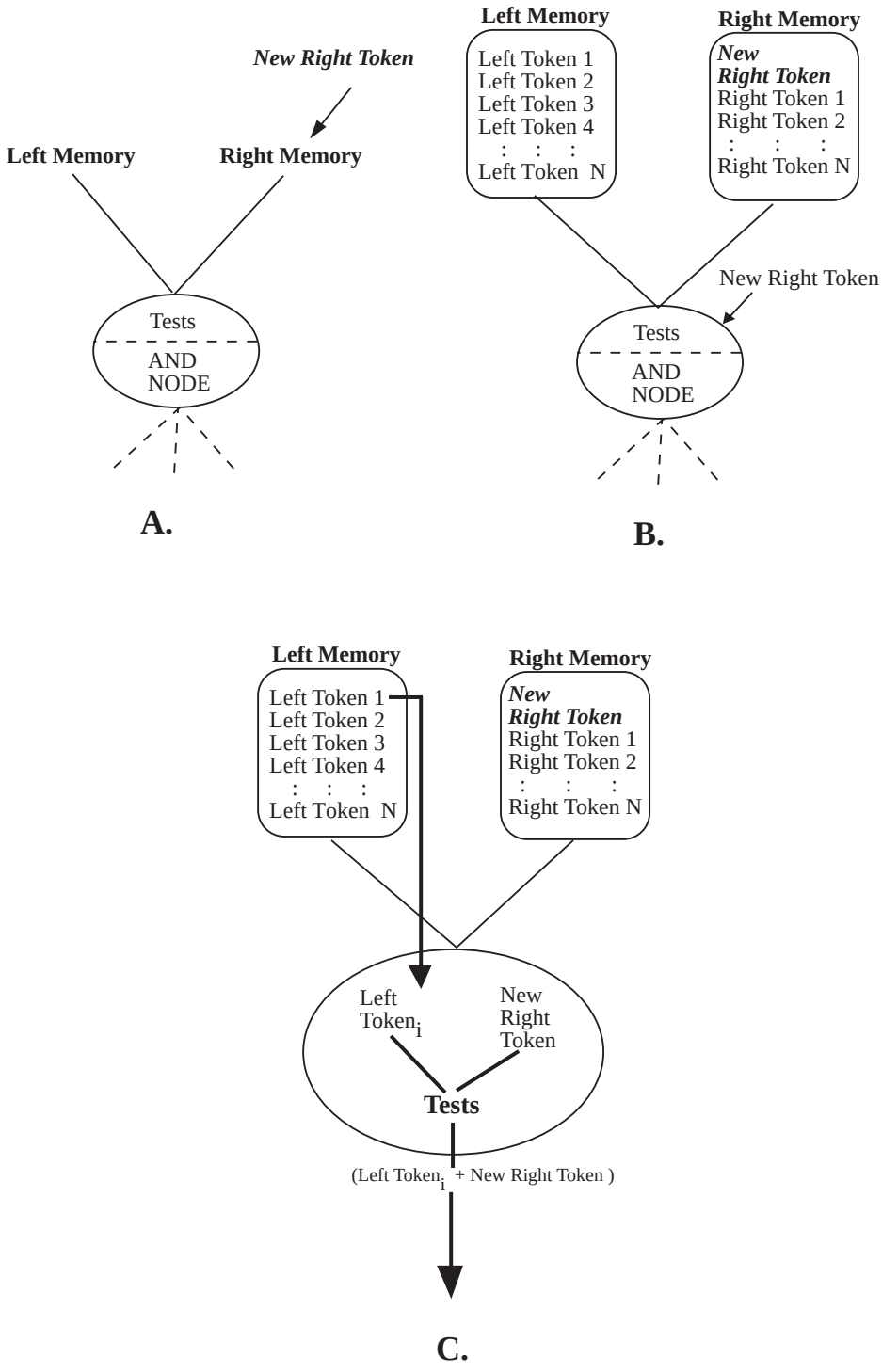

Figure 4.1: A token arrives at an AND node.

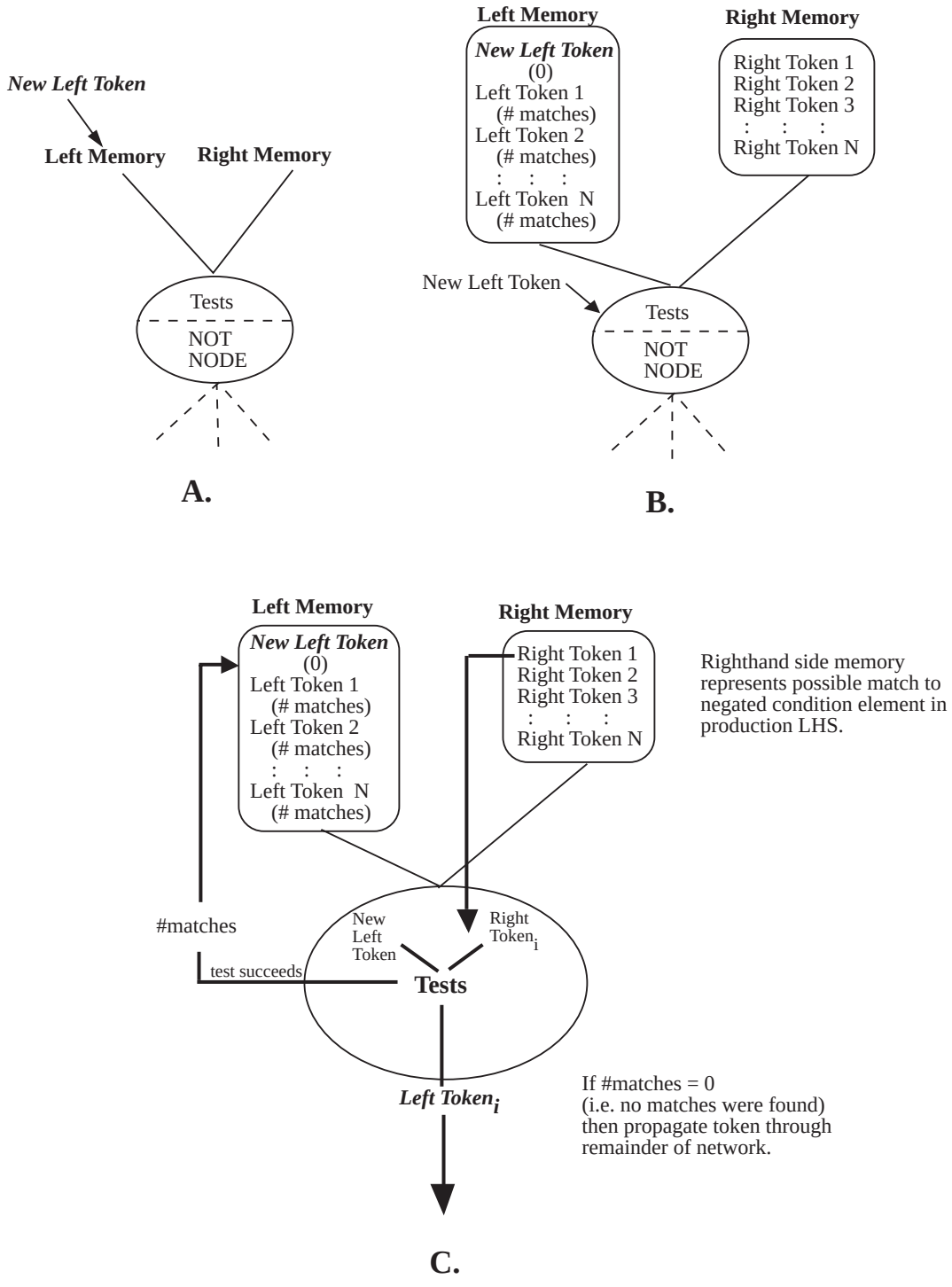

Figure 4.2: A token arrives at the lefthand input of a NOT node.

remove it from memory nodes further down the net. For negated (deleted) tokens arriving at the NOT node, the process is the same except that the token is removed from the memory and the counters are decremented. If any counter becomes 0, then the lefthand token is propagated.

4.2 Implementing Match-level Parallelism

Any degree of parallelism in a system which uses a Rete net pattern matcher implies the presence of (or at least the capability for) node and intra-node parallelism. Technically, the node level of parallelism allows more than one test node in the network to be active at the same time while intra-node parallelism allows multiple activations of a single node. The first attribute increases the speed of a single match episode because multiple paths of the network can be traversed in parallel. The second attribute allows multiple match episodes to take place at once; a situation which arises when multiple actions in a RHS are executed concurrently, or when the righthand sides of multiple productions are executed concurrently.

Match-level parallelism occurs when node parallelism is used to increase the matching speed of a single working memory element change. Implementing match parallelism is relatively simple. Each node possesses an *out-list*; that is, a list of the nodes which succeed it in the network. In a serial system, this out-list is traversed in a depth-first fashion. To parallelize the net traversal, each item in the out-list is traversed in parallel. This approach to match parallelism spawns one new process for each node in the net traversed by a given token. Depending on the structure of the Rete net, the branching factor, the amount of computation performed at each node, and the overhead of invoking parallel processes, this might involve more overhead than is gained by the parallelism. Variations on the scheme involve only invoking node parallelism when the out-list is large, not invoking node parallelism for the simple alpha nodes, only creating parallel processes at the first level of beta nodes, or creating less than N processes for an N-element out-list, each process then traversing part of the out-list in a depth-first fashion.

When node parallelism is employed, there is a chance that multiple production nodes may be simultaneously active, causing multiple instantiations to be entered into the conflict set at the same time. For this reason, the conflict set (or, if the implementation doesn't require conflict resolution, the list of productions waiting to be executed) must be considered a critical resource, and the add and delete functions must take place within a critical region.

Intra-node parallelism, in which multiple tokens can be processed by multiple activations of the same node at the same time, is required for action- or rule-level parallelism. The major difficulty is maintaining the consistency of the associated memory (for beta nodes) during simultaneous accesses. If two tokens are added to memory at the same time, then the memory list could end in an inconsistent state. To avoid this problem, each memory node is assigned a unique lock which allows token insertion and deletion to be performed within a critical region. For AND nodes, the memories do not have to be locked during the actual token processing as the synchronization mechanism described in the following section ensures that the state of the network remains consistent.

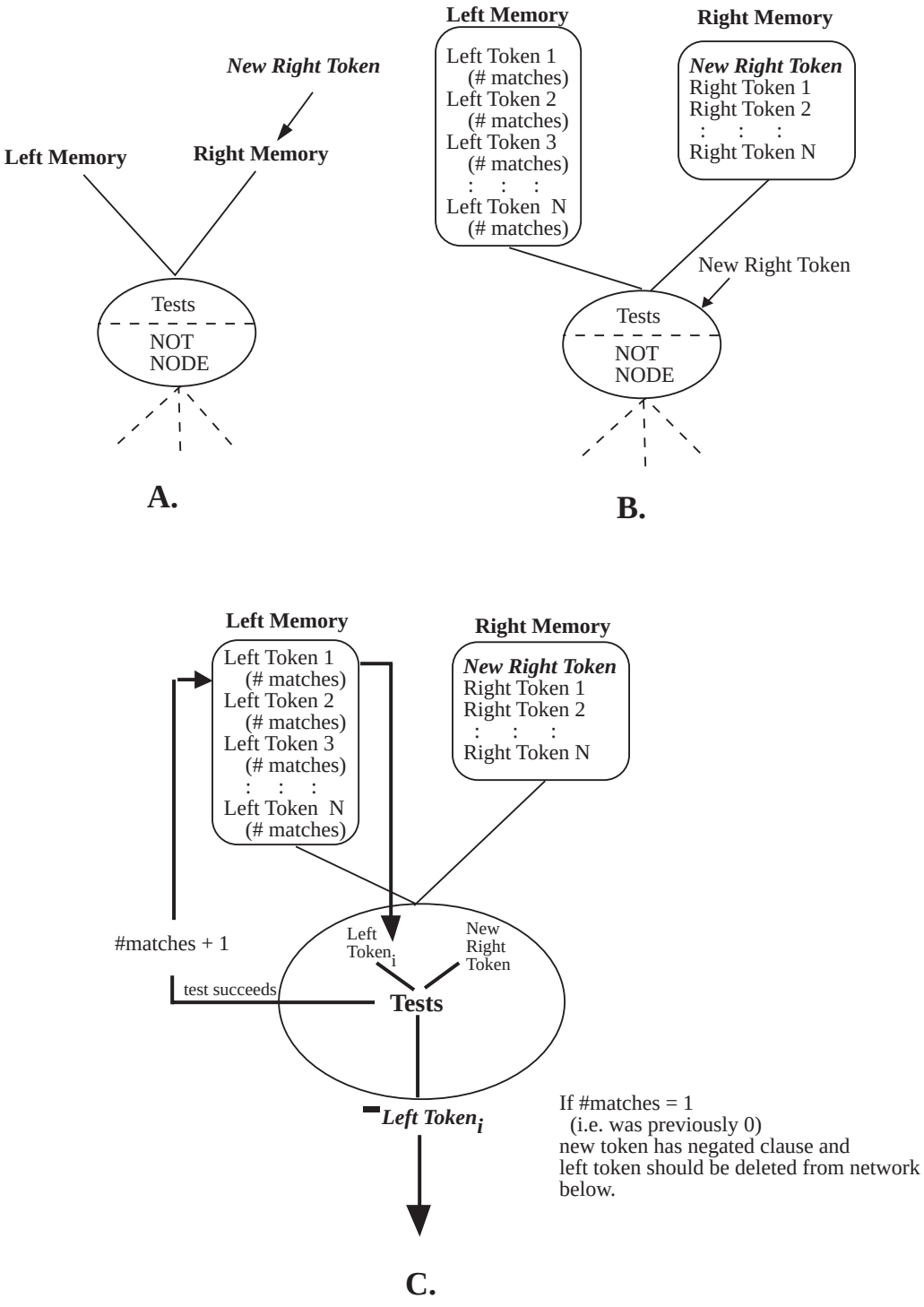

Figure 4.3: A token arrives at the righthand input of a NOT node.

4.3 Synchronization of 2-input Nodes

The Rete net makes the implicit assumption that only one token is processed by a two input node at one time. When the system supports action or production parallelism, this assumption is no longer true; multiple tokens might arrive at a two-input node at any time, and at either input [Forgy, 1979]. It is inevitable that eventually a token will arrive at either the left or right input while a matching token is still being processed on the opposite side. This can cause serious synchronization problems.

There are two possible failure modes, depending on when the token is added to the node's memory. Figure 4.3 depicts the case in which the implementation adds tokens to the memory *before* passing the token to the AND node. When tokens arrive simultaneously, it is possible that the left token will match against the right token, and the right token will match against the left token. This will result in two identical tokens being propagated through the network. The inevitable result is that the conflict set will eventually contain multiple identical instantiations, multiple copies of tokens will proliferate in memory, and the state of the network will be corrupted.

If the tokens are added to the node's memory *after* the matching process takes place, then it is possible that neither token will match. The righthand matching process will examine the lefthand memory and not find a matching token and the lefthand process will examine the righthand memory and not find a matching token. Then both tokens will be added to memory on their respective sides. This would result in a situation in which the node's memories contain two tokens satisfying all tests but which have not been passed further down the net.

One possible solution to this problem, which was adopted by Gupta, is to *lock* one side of a node when a token arrives from the opposite side so that the problem of simultaneous arrival never occurs. This is unduly restrictive, however, for the occurrence of a simultaneous arrival of matching tokens is very rare. Non-matching tokens arriving at the opposite inputs can be processed without difficulty. So a locking approach will unduly reduce throughput. Instead, UMPOPS adopts the following solution:

In UMPOPS, tokens are added to memory before they are passed to the AND node, so the case in which two identical tokens are propagated must be detected. The solution taken to this synchronization problem was to add a *completion flag* and a *match list* field to each token being passed through the network. As each token enters a two-input node, the flag is set to false. It is not set to true until all tests have been completed on that token and it has been added to memory; obviously, if a node's match flag is set, then its matching process is complete and it is not going to generate any more matches unless a matching token arrives on the opposite side.

When a test in the two-input node succeeds, the matching token is checked to see if its completion flag is set. If the flag is set, then the token is propagated as usual. If not, then that token is currently being processed and a case of simultaneous activation exists. The matching token is stored on the incoming token's match list. After the incoming token has been compared against the entire opposite memory, both it and its match list are passed to a synchronization process. The synchronization process iterates over the match list examining the completion flags of each item. If the flag becomes set, then the two

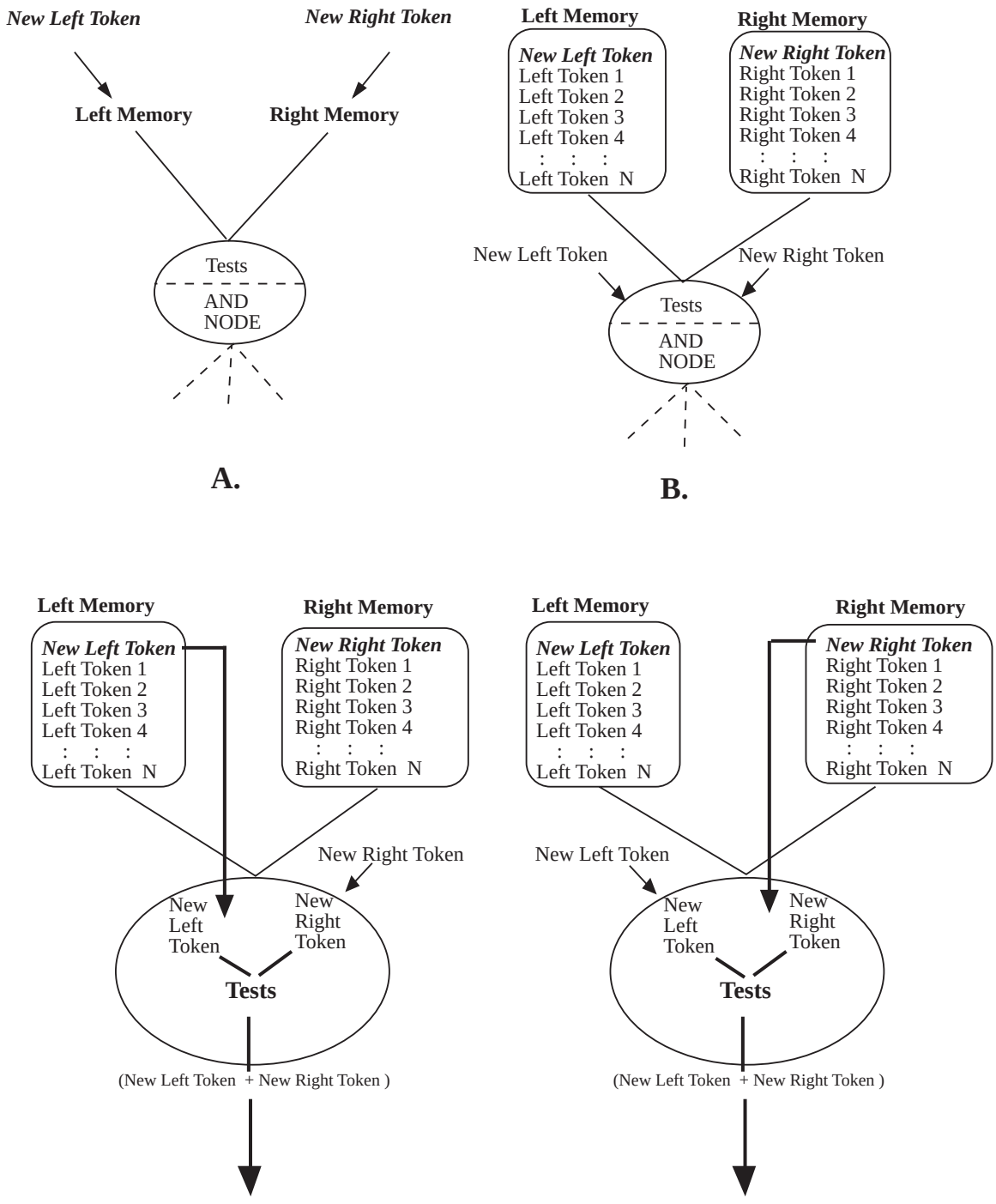

Figure 4.4: When matching tokens arrive at an AND node simultaneously.

tokens are concatenated and propagated. If the matched token's flag is not set, then its match list is examined to see if it contains the current token. If so, then the two nodes are mutually matching and a synchronization error exists. In this case, the result of one match is suppressed by removing it from the token's match list (The choice of which side the token is removed on is arbitrary). Once the matching token is removed, the match list for the incoming token becomes empty and its completion flag is set. This allows the opposite synchronization process to propagate the concatenated token further down the net.

The overhead for this synchronization check is not high because it is rare for two tokens to arrive at a node simultaneously, therefore the usual overhead is the creation of the token data structure, and the checking and setting of the completion flags.

Figure 4.3 demonstrates the synchronization process.

To prove that this mechanism correctly solves the simultaneous token problem, consider the following cases.

Case A: The left token(T_L) arrives and completes matching before the right token T_R arrives. This is the same as the serial case. The left token does not find a righthand match and is not propagated, but sets its completion flag. The righthand token then arrives, successfully matches against the lefthand token, and, because the completion flag for T_L is set, the result is propagated.

Case B: T_L does not complete matching before T_R arrives, but, because T_R is concatenated to the front of the memory list, T_L does not match against T_R. In this case, the completion flag for T_L is not yet set, so T_L is placed on T_R's match list. The synchronization mechanism ensures that T_R cannot complete until the match list is empty. The token T_L, however, has an empty match list and completes, setting its completion flag. T_R can then propagate the result of concatenating T_L and T_R.

Case C: T_L and T_R arrive at the two input node simultaneously. Both are entered in to memory, and each successfully matches against the other. Left uncorrected, two identical tokens (T_L + T_R) would be propagated through the network. This is the pathological case which the synchronization mechanism was designed to avoid. The completion flag can not be set on either token because in order to do so, the opposing token would have to have its flag set. Therefore, the matching token is placed on each incoming token's match list, that is, T_R stores T_L and T_L stores T_R on its list. Each token is then passed to the synchronization routine. The synchronization routine observes that the token T_R has T_L on its match list which in turn has T_R on its match list. It arbitrarily deletes T_R from T_L's match list. T_L then has a null match list and the match process terminates, setting the completion flag for T_L. Once the flag is set, the synchronization process monitoring T_R can then propagate T_L+T_R and remove T_L from T_R's match list, allowing its match process to terminate. Only one copy of the outgoing token is propagated.

Because of the symmetry of the two-input AND node, the tokens T_L and T_R can be reversed in the above discussion. There are no other cases. It remains only to consider the case of deadlock. Is it possible for a token to never set its completion flag, thus resulting in

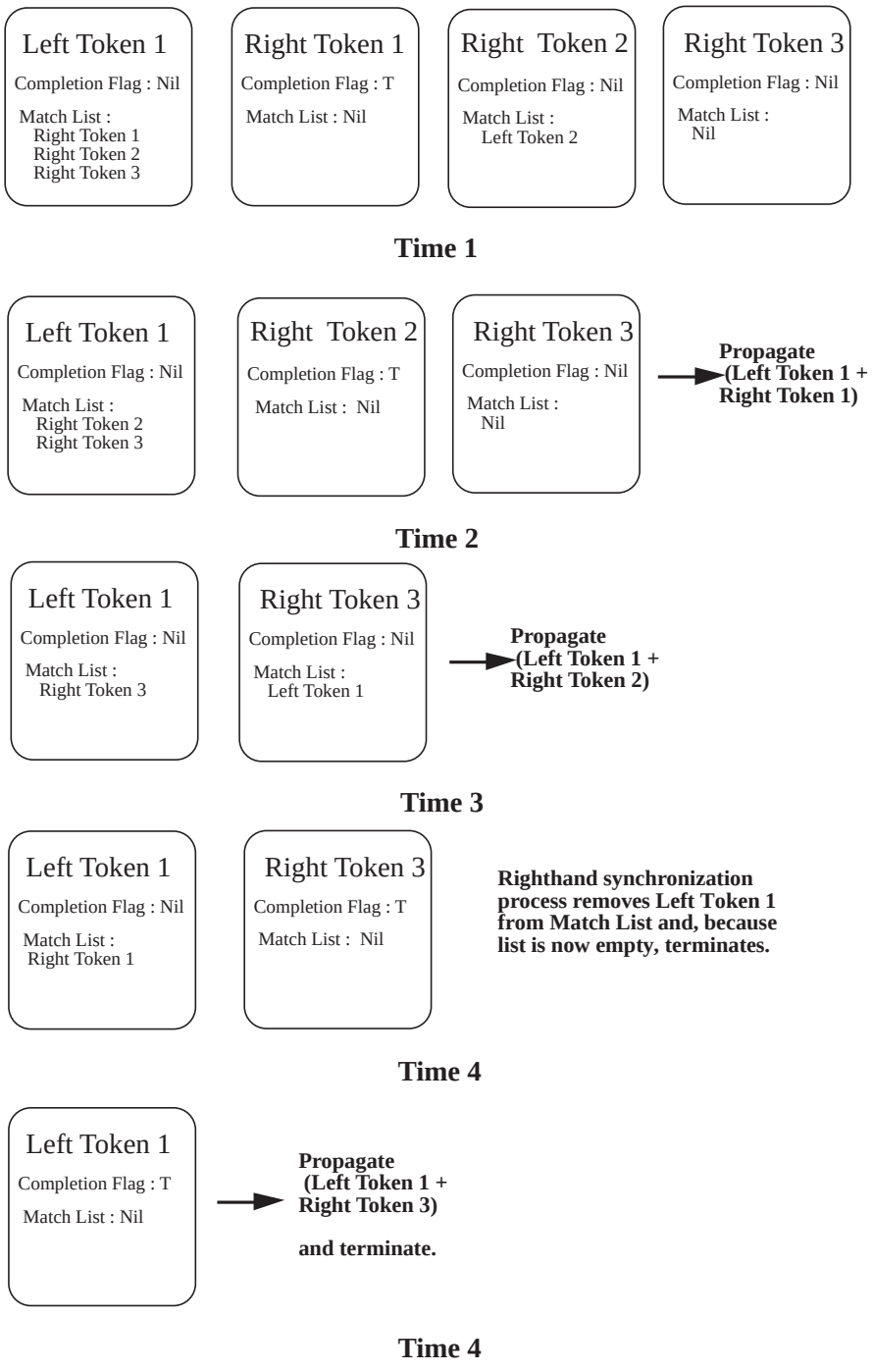

Figure 4.5: The synchronization process for an AND node.

a synchronization process which never terminates? The answer, briefly, is no, for the only way for a token to never complete is for it to match a token whose completion flag is never set. But the only way for this to happen is for the two tokens to be mutally matching, and this deadlock is arbitrarily broken by the synchronization routine.

The synchronization problem may also appear in NOT nodes, however this was solved by another mechanism. Because the NOT node modifies its memory nodes during processing (by incrementing counters), it proved easier to simply lock both memories while the node was being executed. Therefore, the simultaneous synchronization problem does not arise in this implementation. However, locking the memory nodes dramatically reduces throughput, and a less restrictive algorithm should eventually be developed.

Synchronization and Sharing of Memory Nodes

The approach taken to synchronization effectively prevents the sharing of memory nodes in UMPOPS. If a memory node has an *out-list* of more than one beta node, then a token's synchronization flag might be set in any of these nodes. This could cause synchronization of any of the other sibling nodes to take place improperly. Because the proliferation of memory nodes is inefficent in terms of space usage and increases the number of critical regions that must be acquired during processing, this restriction should be removed. Methods such as arrays of synchronization flags in which the arity of the array is the same as the arity of the *out-list* should be applicable, but have not yet been implemented.

4.4 Race Conditions

There is one additional hazard due to intra-node parallelism which must be guarded against. Consider the case in which a token T_1 enters a two input node and matches with a token contained on the opposite side T_2. A new token consisting of the two tokens concatenated together, $(T_1 + T_2)$ is propagated through the tree. Now suppose that a remove working memory element episode takes place, which causes T_2 to be removed from the node memory. This causes the token $-(T_1 + T_2)$ to be propagated, where the minus sign represents a flag specifying deletion. For any number of reasons, it is possible that this negated token could arrive at a beta node or production node before the original token. If this happens, the deletion will fail, and the positive token will remain in memory despite the fact that one of its supporting working memory elements has vanished (figure 4.4). Currently, race conditions are avoided by using the task synchronization mechanisms provided by UMPOPS; no action stimulated by an embedded add or delete operation in a `modify` command is allowed to execute until the opposing match operation has completed.

4.4.1 Avoiding critical regions

When running in parallel, it is necessary to reduce the time that processes spend in critical regions as this tends to serialize performance. The most notable critical regions in UMPOPS are the eligibility (conflict) set and the node memories. Conflict for the node memories is greatly reduced by hashing. Inserting instantiations into the eligibility set is inexpensive,

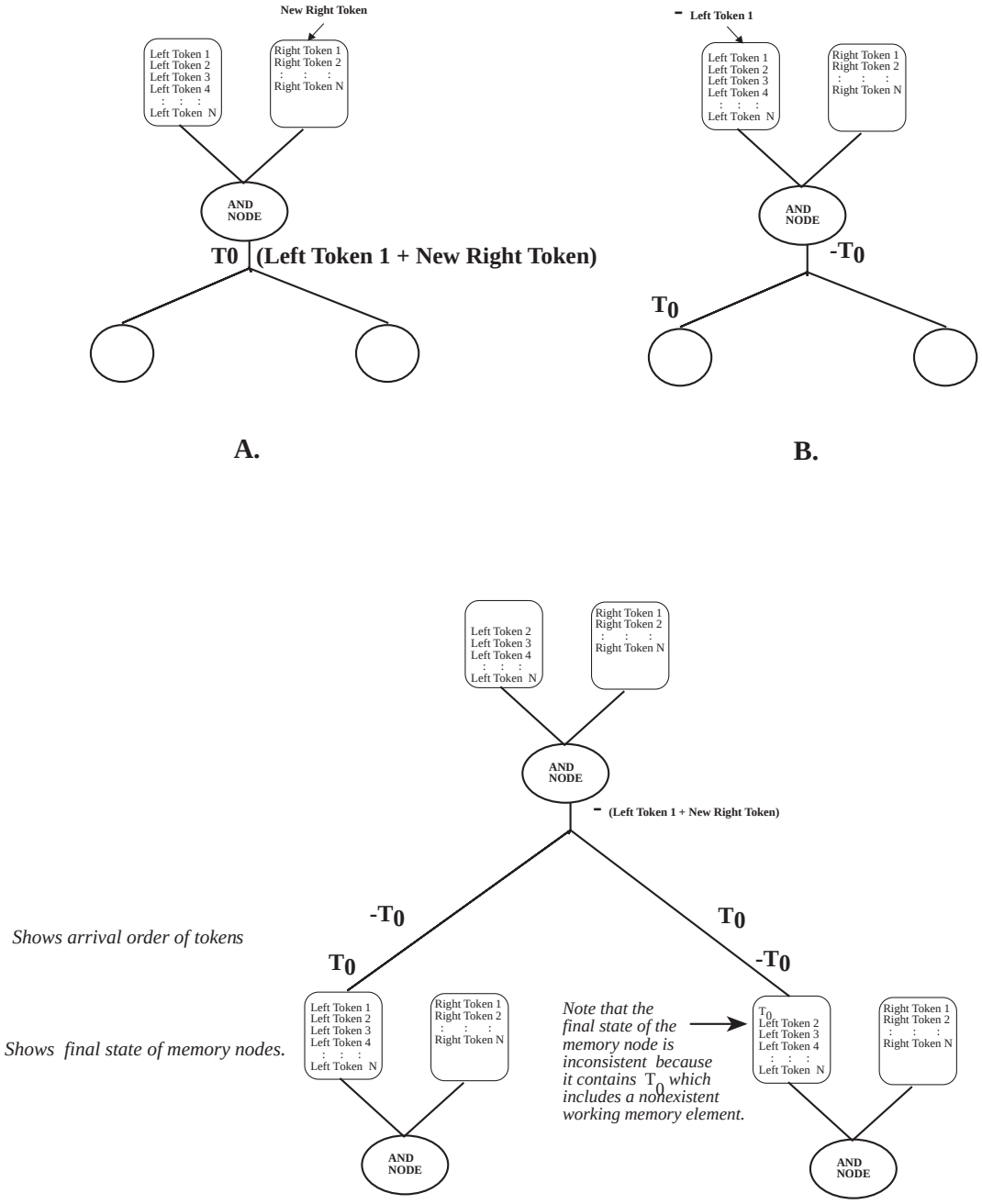

Figure 4.6: Race conditions due to intra-node parallelism.

however, deletion is costly because the traversal of the list and the actual deletion must take place within a lock. To avoid this delay, instantiations which are deleted from the eligibility set are simply marked as "killed", but are not actually removed from the list. As the scheduling process removes instantiations from the list, it checks to see if the instantiations have been killed, if so, they are simply discarded. A variation of this scheme could be used to reduce the overhead of deleting tokens from node memories, however this would require a garbage collection process to periodically examine the node memories and remove unneeded tokens.

4.5 Implementing Action-level Parallelism

Action-level parallelism occurs when multiple working memory changes stimulated by a single rule take place concurrently. From a matching standpoint, once the working memory changes are asserted, they are handled no differently than changes invoked by separate rule instances. The challenge of implementing action parallelism comes from the implementation of OPS5 which defines or compiles the righthand side into monolithic code in which the individual actions are not accessible to be invoked concurrently. In order to access the individual actions, the `in-parallel` and `in-parallel-sync` constructs were devised. These constructs, actually compile-time macros, examine their arguments (which consist of make, modify, or remove functions) and modify them so that they spawn off individual match processes.

In effect, each working memory operation is expanded into a function which pushes a change operation and arguments onto an *action* queue where it is then executed by a rule/action demon. If the `*action-parallelism*` flag is not set, the action-queue-push operation executes the operations instead of pushing them onto the action queue.

The initial naive implementation of action-level parallelism allowed the righthand side of the rule to compute the token to be added to (deleted from) memory. This token was then placed on the action queue with a flag indicating whether it was an add or delete operation. It turned out, however, that the process of constructing the token to be matched against working memory is actually fairly expensive relative to the matching process. Thus, the righthand side of the rule was only able to push about four actions onto the queue before the first had finished executing, and the benefit due to action parallelism was thus limited to four-fold. This rather contradicts the widely quoted statistic that matching consumes at least 90% of the work of executing a rule – one reason for this may be that action parallelism is most useful during initialization routines in which there is not a lot of matching performed against the elements being asserted.

To increase the potential speedup due to action parallelism, the righthand side was modified so that instead of creating the token to be matched, it simply placed the appropriate make, modify or delete function, its arguments, and the necessary environment variables onto the action queue. The action demons now were responsible for both constructing the tokens and matching them against working memory. Although this increased the time that the individual action demons were active, the rule instances were able to push actions onto the queue much more quickly and the speedup due to action parallelism increased to at

least eight-fold. The (obvious) object lesson to be learned from this is that when spawning off processes sequentially (especially at a low level of granularity), the launch time must be minimized. Therefore, all initialization and processing work which can be passed on to the parallel processes should be.

Chapter 5

Conclusion

This manual has described the features and syntax of the UMass Parallel OPS5. These features include the ability to invoke parallelism at the rule-,action-, or match-levels, specify heuristic pruning and ordering funcitons, and (partially) enforce correctness using a working memory-based locking scheme. Some of the issues involved in designing programming parallel rule-based systems were addressed and two sample programs were studied. Finally, low-level implementation details of parallel matching and synchronization were discussed.

Bibliography

[Forgy, 1979] C. L. Forgy. *On the Efficient Implementation of Production Systems.* PhD thesis, Carnegie-Mellon University, 1979.

[Forgy, 1981] C. L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, CMU Computer Science Department, July 1981.

[Gordin and Pasik, 1991] Douglas N. Gordin and Alexander J. Pasik. Set-oriented constructs: From rete rule bases to database systems. In *Proceedings 10th ACM Symposium on PODS*, pages 60–67, 1991.

[Gupta *et al.*, 1988] A. Gupta, M. Tambe, D. Kalp, C. Forgy, and A. Newell. Parallel implementation of OPS5 on the encore multiprocessor: Results and analysis. *International Journal of Parallel Programming*, 17(2), 1988.

[Gupta, 1987] Anoop Gupta. *Parallelism in Production Systems.* Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[Hayes-Roth, 1985] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26:251–321, 1985.

[Ishida and Stolfo, 1985] T. Ishida and S. Stolfo. Towards the parallel execution of rules in production system programs. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages 568–575, 1985.

[Ishida, 1990] Toru Ishida. Methods and effectiveness of parallel rule firing. In *6th IEEE Conference on Artificial Intelligence Applications*, March 1990.

[Kalp *et al.*, 1988] Dirk Kalp, Milind Tambe, Anoop Gupta, Charles Forgy, Allen Newell, Anurag Acharya, Brian Milnes, and Kathy Swedlow. Parallel OPS5 user's manual. Technical Report CMU-CS-88-187, CMU Computer Science Department, November 1988.

[Kumar *et al.*, 1988] Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 122–127, 1988.

[Kuo *et al.*, 1991] Chin-Ming Kuo, Daniel Miranker, and James C. Browne. On the performance of the CREL system. *Journal of Parallel and Distributed Computing*, 13(4):424–441, December 1991.

[Miranker *et al.*, 1989] Daniel Miranker, Chin-Ming Kuo, and James C. Browne. Parallelizing transformations for a concurrent rule execution language. Technical Report TR-89-30, Department of Computer Science, University of Texas at Austin, October 1989.

[Neiman, 1991] Daniel Neiman. Control issues in parallel rule-firing production systems. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 310–316, 1991.

[Neiman, 1992] Daniel E. Neiman. A multiple worlds implementation for parallel rule-firing production systems. In *preparation*, January 1992.

[Pearl, 1984] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison–Wesley, Reading, Massachusetts, 1984.

[Schmolze and Neiman, 1992] James G. Schmolze and Daniel E. Neiman. Comparison of three algorithms for ensuring serializability in parallel production systems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-92)*, July 1992.

[Schmolze, 1989] James G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. Technical Report 89-5, Department of Computer Science, Tufts University, October 1989.

[Schmolze, 1991] James G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. *Journal of Parallel and Distributed Computing*, 13(4), December 1991.

[Sellis *et al.*, 1987] Timos Sellis, Chih-Chen Lin, and Louiqa Raschid. Implementing large production systems in a dbms environment: Concepts and algorithms. Technical Report CS-TR-1960, Dept. of Computer Science, University of Maryland at College Park, 1987.

[Stolfo *et al.*, 1990] Salvatore J. Stolfo, Leland Woodbury, Jason Glazier, and Philip Chan. The ALEXSYS mortage pool allocation expert system: A case study of speeding up rule-based programs. In *AI and Business Workshop, AAAI-90*, 1990.

[Stolfo *et al.*, 1991] Salvatore J. Stolfo, Ouri Wolfson, Philip K. Chan, Hasanat M. Dewan, Leland Woodbury, Jason S. Glazier, and David A Ohsie. PARULEL: Parallel rule processing using meta-rules for redaction. *Journal of Parallel and Distributed Computing*, 13(4):366–382, December 1991.

[Widom and Finkelstein, 1990] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *ACM-SIGMOD International Conference on the Management of Data*, pages 259–270, 1990.

Appendix A

Enhancements to OPS5

A.1 Efficiency considerations in Lisp-based OPS5

UMPOPS is implemented in Top Level Common Lisp[1] a concurrent Lisp supporting lightweight threads which was originally developed at the University of Massachusetts and has since been commercialized. The principal reason for this choice of implementation language is that the public domain code was already available in Lisp. Although there are now public domain C-based OPS5s which offer higher performance, this increased speed is largely due to extremely optimized compilation of the rules into a Rete net. The compiled nature of this code makes it extremely difficult to modify the pattern matching algorithms in order to add levels of parallelism or synchronization mechanisms; in most cases, the rule-firing algorithms are equally inflexible. Lisp is still the language of choice for experimental vehicles such as UMPOPS which are frequently modified or augmented with new language features or control mechanisms. During the development of UMPOPS, a number of enhancements were made to the lisp-based OPS5's pattern matcher and data structures. These enhancements have improved the performance of UMPOPS by a factor of 5 to 10 times, despite the overhead due to the extra data structures, locking, and general bookkeeping required to implement rule parallelism. Because considerable experimental work is still performed using Lisp-based versions of OPS5, the magnitude of this performance increase is sufficient to make it worth briefly describing the modifications to the OPS5 code which are not directly related to parallelism.

A.1.1 Representation of working memory elements and tokens

OPS5 represents working memory elements as flat lists, the first element of which is the class of the element, and all following items are field values. Extraneous information such as the timetag of the element is stored on an association list, indexed by the class of the element and the element itself. This representation scheme proved inadequate for parallel implementations of OPS5. The association list corresponding to the class of each element had to be locked everytime a new element was created or deleted which caused undesirable

[1] Top Level Common Lisp and TopCl are trademarks of Top Level, Inc.

contention for the critical region. This contention was increased by the necessity for storing additional information about each working memory element (such as the wme locking information and benchmarking information such as match and assertion times). A new data structure, the `wme-info` structure was created to store all information about the working memory element. To avoid having to access a global structure to access the `wme-info` structure, it was stored as the last element of each working memory element. Because this requires cdr'ing to the end of the element every time the element is accessed, this is not as efficient as, say, implementing each working memory element as a structure; however this was far simpler than rewriting all the OPS5 code which references working memory elements. Although the `wme-info` structure was developed simply to avoid contention for critical resources when asserting working memory elements in parallel, it turned out that a 10% increase in performance was achieved even when running serially.

During pattern matching using the Rete net, *tokens* are passed through the network. In OPS5, these tokens are simply lists of working memory elements. In UMPOPS, the necessity of storing synchronization information in each token required that tokens be represented as structures. Although more expensive than the conventional OPS5 representation for tokens, this representation allowed other efficiency measures to be implemented, notably the storage of tests values for pattern matching and hash keys for memory node access.

A.1.2 Reduction of GELM calls

When pattern matching, OPS5 performs binary tests on working memory tokens. These tests consist of tests applied to the values of specific fields of the tokens. The fields are accessed using indices compiled into the tests. OPS5 makes frequent calls to the `gelm` (get element) function which takes an index into a token, selects the appropriate working memory element and field and extracts the value specified by the index. The `gelm` function is easily the most frequently called function in OPS5. However, it is clear that many of the `gelm` calls are redundant; at each node in the Rete net, a call to `gelm` for a specific token will always return the same value. It's more efficient, therefore, to simply call `gelm` once when the token arrives at each node memory and to store a list of the values required for the node tests. Because UMPOPS requires a structured representation for tokens instead of the simple list used by OPS5, this modification was simple and resulted in a 10-20% decrease in match times and a considerable reduction in the calls to the `gelm` function.

For reasons which remain mysterious but are probably related to the small amount of memory available to previous implementations of Lisp, OPS5 originally stored the gelm indices as integers in which the working memory index(WM) of the token and the field number(F) were encoded as $WM * 10000 + F$. Decoding this involved a call to floating point routines to perform division and rounding operations. The performance of OPS5 was sped up considerably by replacing this representation with a simple dotted pair (WM . F).

A.1.3 Hashed Memories

The original OPS5 memories were simply unstructured lists. In UMPOPS, the memories of each node are represented as hash tables whenever possible. (Hashing is only possible when

the node contains an equality test.) The hashing scheme is similar to that used in the CMU parallel-matching OPS5 [Gupta *et al.*, 1988], however, that implementation maintains a single global hash table rather than a hash table at each node. Once the appropriate hashed memory has been retrieved for an incoming token, the corresponding equality test has already been performed, so that test is not repeated during evaluation of the beta tests.

The hashtable functions provided by Common Lisp could not be used in implementing hashed memory nodes because of the requirement that memory nodes be accessed in parallel. Because the CL hashtable is expandable, it is necessary to acquire a lock on the entire hashtable whenever the table is referenced, whether or not it is actually being modified. This is unacceptable when executing in parallel as a reference to any bucket of the hashtable prevents any further access to that table and thus serializes access to the Rete node. This problem was solved by developing a special hashtable in which each bucket is assigned a lock. The hashtable is of fixed size, so each bucket contains a list of entries. The bucket is only locked when changes are made to its list of entries. In addition, each entry (which represents a hashed slice of a memory node) also possesses a lock which must be acquired when tokens are inserted into that sub-bucket. Although more complex to implement, this scheme greatly reduces contention for memory locks.

The addition of hashed memories to UMPOPS improved performance by approximately a factor of two.

A.1.4 Compilation of Righthand Sides

In a Lisp-based OPS5, righthand side actions are represented as macros. The principal reason for this is to avoid having to 'quote' arguments passed to the actions. In order to execute the righthand side actions, an `eval` operation is performed on each action. This involves a macro expansion at runtime for each RHS action; the macro expansion is quite inefficient and can cause serious variation in RHS execution times. Although the development of a full RHS compiler was beyond the scope of our research, we developed a preprocessing mechanism which expanded all embedded macros in the RHS actions and then passed the body of the RHS to a function compiler. The compiled RHS function is stored in the `PNODE` structure attached to each rule type. Instead of `eval'ing` the RHS, it is now invoked using the more efficient `apply` operator.

Appendix B

The Toru-Waltz Benchmark

```
;
;  INITIAL OPS5 VERSION OF WALTZ'S ALGORITHM         by Toru Ishida
;

;  Modified by Dan Neiman
;              COINS Dept.
;              University of Massachusetts

;  11/16/90: Added possible-line-label element.  One element is added for each possible
;  labelling of each end of each line.   This allows easy testing for consistent
;  line labelling without proliferation of rules.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                     ;
;       Data & Knowledge Structure for Waltz's Algorithm                              ;
;                                                                                     ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (literalize possible-junction-label junction-type line-1 line-2 line-3)

 (literalize junction junction-type junction-ID line-ID-1 line-ID-2 line-ID-3)

 (literalize labelling-candidate junction-ID line-1 line-2 line-3 l-c-ID)

 (literalize possible-line-label line candidate junction label)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Knowledge of Possible Junction Labeling   ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;(literalize possible-junction-label junction-type line-1 line-2 line-3)

;                           \   /
; Junction type : L        1 \ / 2
;                             V
```

```
 (p initialize
    (meta  (no-lock-required t))
    (stage initialize) --> ;(remove 1) (make stage make-data)
 (in-parallel-sync
 (make possible-junction-label ^junction-type L
                               ^line-1 out ^line-2 in ^line-3 nil)

 (make possible-junction-label ^junction-type L
                               ^line-1 in  ^line-2 out ^line-3 nil)

 (make possible-junction-label ^junction-type L
                               ^line-1 +   ^line-2 out ^line-3 nil)

 (make possible-junction-label ^junction-type L
                               ^line-1 in  ^line-2 +   ^line-3 nil)

 (make possible-junction-label ^junction-type L
                               ^line-1 -   ^line-2 in  ^line-3 nil)

 (make possible-junction-label ^junction-type L
                               ^line-1 out ^line-2 -   ^line-3 nil)

;                             1 \ / 3
; Junction type: FORK           V
;                              2 l

 (make possible-junction-label ^junction-type FORK
                               ^line-1 +   ^line-2 +   ^line-3 +  )

 (make possible-junction-label ^junction-type FORK
                               ^line-1 -   ^line-2 -   ^line-3 -  )

 (make possible-junction-label ^junction-type FORK
                               ^line-1 in  ^line-2 -   ^line-3 out)

 (make possible-junction-label ^junction-type FORK
                               ^line-1 -   ^line-2 out ^line-3 in )

 (make possible-junction-label ^junction-type FORK
                               ^line-1 out ^line-2 in  ^line-3 -  )

;                            1 _____ 3
; Junction type: T              l
;                               l2

 (make possible-junction-label ^junction-type T
                               ^line-1 out ^line-2 +   ^line-3 in)

 (make possible-junction-label ^junction-type T
                               ^line-1 out ^line-2 -   ^line-3 in)

 (make possible-junction-label ^junction-type T
                               ^line-1 out ^line-2 in  ^line-3 in)
```

```
 (make possible-junction-label ^junction-type T
                               ^line-1 out ^line-2 out ^line-3 in)

;                                /1\
; Junction type: ARROW          1 / 1 \ 3
;                                /  12 \

 (make possible-junction-label ^junction-type ARROW
                               ^line-1 in ^line-2 +  ^line-3 out)

 (make possible-junction-label ^junction-type ARROW
                               ^line-1 -  ^line-2 +  ^line-3 -  )

 (make possible-junction-label ^junction-type ARROW
                               ^line-1 +  ^line-2 -  ^line-3 +  )))


;;;;;;;;;;;;;;;;;;;;;;;;;
; Scene to be Analyzed   ;
;;;;;;;;;;;;;;;;;;;;;;;;;

;(literalize junction junction-type junction-ID line-ID-1 line-ID-2 line-ID-3)

;
;                            ------------>
;
;
;                              A               B
;                             / \             / \
;                          1 /   \ 2        3/   \ 4
;                           /  C  \         /  D  \
;                          / 5/1\6 \       /  /1\  \
;                       E /10/+1 \  \     /  / 1 \  \
;                        1\+/  1  \  \   / 7/ 81  \9 \
;                        1 1 F 1-  \  \ /  /   1   \  \
;                        1 112 141 +\  G  /+   1-   \  \
;                       111 1+  1     \   /     1     \  \
;                        1 1   1     L\ /    M 1     +\  \
;                        1 114/K\      1      / \      \39\ O
;                     H  1 1-/   \15   116 17/   \18   N\+/1
;      ^              /-\1/  P  \-   1+  -/  Q  \-     1 1       1
;      1             / 13J  /1\  \   1   /  /1\  \   191 120     1
;      1         21/      / 1 \  \  1  /  / 1 \  \   +1 1       1
;      1           /    22/  1  \  \ 1 /  /  1  \  \   1 1       1
;      1       R/      /   1 24\  \1/  /   1 27\  \  1 1       V
;              1\30   /+ 231   +\  T  /+ 261   +\  \ 1 1
;              1 \+  /    -1     \   /25  -1     \ W\1 1
;              1  \S/      1    U \ /      1    V \ /+ 1
;            291   1       1       1       1       128 1
;              1 311      /X\    321+     /Y\     +1   1
;              1   1+    /   \     1     /   \   331   1
;              1   1    /     \    1    /     \    1   1
;            Z \   1   /40     \   1   /       \   1   / DD
;               \  1  /       35\  1  /36     37\  1  /38
```

```
;              34\ 1 /           \ 1 /           \ 1 /
;                 \1/             \1/             \1/
;                  AA              BB              CC
;
;                              <---------


 (p make-data
    (meta  (no-lock-required t))
    (stage initialize)
   -->

   ;(remove 1)
   ;(make stage enumerate-possible-candidates)
 (in-parallel-sync
 (make junction ^junction-type  L ^junction-ID  A
               ^line-ID-1       2  ^line-ID-2    1   ^line-ID-3    NIL)

 (make junction ^junction-type  L ^junction-ID  B
               ^line-ID-1 4 ^line-ID-2    3   ^line-ID-3    NIL)

 (make junction ^junction-type  ARROW ^junction-ID  C
               ^line-ID-1 5 ^line-ID-2    41  ^line-ID-3    6)

 (make junction ^junction-type  ARROW   ^junction-ID  D
               ^line-ID-1 7 ^line-ID-2    8   ^line-ID-3    9)

 (make junction ^junction-type  ARROW ^junction-ID  E
               ^line-ID-1 11 ^line-ID-2    10  ^line-ID-3  1)

 (make junction ^junction-type  FORK ^junction-ID  F
               ^line-ID-1 10 ^line-ID-2    12  ^line-ID-3    5)

 (make junction ^junction-type  L ^junction-ID  G
               ^line-ID-1 2 ^line-ID-2    3   ^line-ID-3    NIL)

 (make junction ^junction-type  FORK    ^junction-ID  H
               ^line-ID-1 11 ^line-ID-2    21  ^line-ID-3    13)

 (make junction ^junction-type  ARROW ^junction-ID  J
               ^line-ID-1 14 ^line-ID-2    12  ^line-ID-3    13)

 (make junction ^junction-type  FORK ^junction-ID  K
               ^line-ID-1 41 ^line-ID-2    14  ^line-ID-3    15)

 (make junction ^junction-type  FORK ^junction-ID  L
               ^line-ID-1 6 ^line-ID-2    16  ^line-ID-3    7)

 (make junction ^junction-type FORK ^junction-ID  M
               ^line-ID-1 8 ^line-ID-2    17  ^line-ID-3    18)

 (make junction ^junction-type  FORK ^junction-ID  N
               ^line-ID-1 9 ^line-ID-2    19  ^line-ID-3    39)
```

```
 (make junction ^junction-type  ARROW ^junction-ID  O
              ^line-ID-1 4 ^line-ID-2    39  ^line-ID-3    20)

 (make junction ^junction-type  ARROW ^junction-ID  P
              ^line-ID-1       22 ^line-ID-2    23  ^line-ID-3    24)

 (make junction ^junction-type  ARROW ^junction-ID  Q
              ^line-ID-1 25 ^line-ID-2    26  ^line-ID-3    27)

 (make junction ^junction-type  ARROW ^junction-ID  R
              ^line-ID-1 29 ^line-ID-2    30  ^line-ID-3    21)

 (make junction ^junction-type  FORK ^junction-ID  S
              ^line-ID-1 30 ^line-ID-2    31  ^line-ID-3    22)

 (make junction ^junction-type  ARROW   ^junction-ID  T
              ^line-ID-1 17 ^line-ID-2    16  ^line-ID-3    15)

 (make junction ^junction-type  FORK ^junction-ID  U
              ^line-ID-1 24 ^line-ID-2    32  ^line-ID-3    25)

 (make junction ^junction-type  FORK ^junction-ID  V
              ^line-ID-1 27 ^line-ID-2    33  ^line-ID-3    28)

 (make junction ^junction-type  ARROW ^junction-ID  W
              ^line-ID-1 19 ^line-ID-2    18  ^line-ID-3    28)

 (make junction ^junction-type  FORK ^junction-ID  X
              ^line-ID-1 23 ^line-ID-2    40  ^line-ID-3    35)

 (make junction ^junction-type  FORK ^junction-ID  Y
              ^line-ID-1 26 ^line-ID-2    36  ^line-ID-3    37)

 (make junction ^junction-type  L ^junction-ID  Z
              ^line-ID-1 29 ^line-ID-2    34  ^line-ID-3    NIL)

 (make junction ^junction-type  ARROW ^junction-ID  AA
              ^line-ID-1 40 ^line-ID-2    31  ^line-ID-3    34)

 (make junction ^junction-type  ARROW ^junction-ID  BB
              ^line-ID-1 36 ^line-ID-2    32  ^line-ID-3    35)

 (make junction ^junction-type  ARROW ^junction-ID  CC
              ^line-ID-1 38 ^line-ID-2    33  ^line-ID-3    37)

 (make junction ^junction-type  L ^junction-ID  DD
              ^line-ID-1 38 ^line-ID-2    20  ^line-ID-3    NIL)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Temporal Labelling Candidates ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;(literalize labelling-candidate junction-ID line-1 line-2 line-3)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
;           Production Rules for Waltz's Algorithm                             ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;
; Start ;
;;;;;;;;;

 (p start-Waltz
   (meta  (no-lock-required t))
(start)
-->
(remove 1)
(make stage initialize))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Enumerate Possible Candidates ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (p enumerate-possible-candidates
   (meta  (no-lock-required t))
(stage initialize)
(junction ^junction-type <j-type> ^junction-ID <j-ID>
                  ^line-ID-1 <l1> ^line-ID-2 <l2> ^line-ID-3 <l3>)
(possible-junction-label ^junction-type <j-type>
                        ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
-(labelling-candidate ^junction-ID <j-ID>
                        ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
-->
        (bind <l-c-ID> (ngenatom))
(make labelling-candidate ^junction-ID <j-ID>  ^l-c-ID  <l-c-ID>
                        ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
        (make possible-line-label ^line <l1> ^label <line-1> ^candidate <l-c-ID>
              ^junction <j-ID>)
        (make possible-line-label ^line <l2> ^label <line-2> ^candidate <l-c-ID>
              ^junction <j-ID>)
        (make possible-line-label ^line <l3> ^label <line-3> ^candidate <l-c-ID>
              ^junction <j-ID>))


 (p go-to-reduce-candidates
        (meta (rtype mode-changer)
              (no-lock-required t))
(stage initialize)
-->
(remove-match-parallel 1)
(make-match-parallel stage reduce-candidates))
```

```
;;;;;;;;;;;;;;;;;;;;
; Reduce Candidates ;
;;;;;;;;;;;;;;;;;;;;

;If a line is labelled '+' on one end, than it must be labelled '+' on the
;other end.
 (P  consistent-plus
(stage reduce-candidates)
 {<line>(possible-line-label ^line <line> ^junction <junction> ^label + ^candidate <c>) }
 {<l-c> (labelling-candidate ^l-c-ID <c>) } ;find candidate which label belongs to.
       -(possible-line-label ^line <line> ^junction <> <junction> ^label +)
-->
        (remove <line>)
(remove <l-c>))


;If a line is labelled '-' on one end, than it must be labelled '-' on the
;other end.
 (P  consistent-minus
(stage reduce-candidates)
 {<line>(possible-line-label ^line <line> ^junction <junction> ^label - ^candidate <c>) }
 {<l-c> (labelling-candidate ^l-c-ID <c>) } ;find candidate which label belongs to.
       -(possible-line-label ^line <line> ^junction <> <junction> ^label -)
-->
        (remove <line>)
(remove <l-c>))

;If a line is labelled 'in' on one end, than it must be labelled 'out' on the
;other end.
 (P  consistent-in-out
 (stage reduce-candidates)
 {<line> (possible-line-label ^line <line> ^junction <junction> ^label in ^candidate <c>) }
 {<l-c>  (labelling-candidate ^l-c-ID <c>) } ;find candidate which label belongs to.
        -(possible-line-label ^line <line> ^junction <> <junction> ^label out)
-->
        (remove <line>)
(remove <l-c>))


;If a line is labelled 'out' on one end, than it must be labelled 'in' on the
;other end.
 (P  consistent-out-in
 (stage reduce-candidates)
 {<line> (possible-line-label ^line <line> ^junction <junction> ^label out ^candidate <c>) }
 {<l-c>  (labelling-candidate ^l-c-ID <c>) } ;find candidate which label belongs to.
        -(possible-line-label ^line <line> ^junction <> <junction> ^label in)
-->
        (remove <line>)
(remove <l-c>))

;When a labelling-candidate is deleted, we want to also delete all possible line
;labels associated with that labelling-candidate.
```

```
 (P eliminate-line-labels
           (stage reduce-candidates)
   {<old>  (possible-line-label ^candidate <c>) }
          -(labelling-candidate ^l-c-ID <c>)
           -->
           (remove <old>))

;Commented out the remove because this part of the code is serial.
;We're interested in benchmarking parallel rules...
 (p go-to-print-out
        (meta (rtype mode-changer))
(stage reduce-candidates)
-->
;(remove 1)
(make stage print-out))

;;;;;;;;;;;;;
; Print Out ;
;;;;;;;;;;;;;

 (p print-out
        (stage print-out)
-->
(remove 1)
(halt))
```

Appendix C

The Travelling Salesperson Problem

```
:TSP
;Travelling salesperson problem modified to incorporate the minimum
;spanning tree heuristic.
(literalize go )
(literalize home-city name)
(literalize start tag start-city  length city-list)
(vector-attribute city-list)
(literalize connect-goal tag est-cost city1 city2 length city-list)
(literalize so-far tag distance cities-seen)
(vector-attribute cities-seen)
(literalize solution distance tag cities-seen)
;Note the use of the UNIQUE-ATTRIBUTE for the solution element.
(unique-attribute solution)
(literalize solution-goal distance tag cities-seen)
(literalize distance city1 city2 distance)

;Working memory element declarations for minimum spanning tree calculation
(literalize mst-city tag name flag)
(literalize goal-compute-mst cost-so-far seed-city tag unseen-cities)
(vector-attribute unseen-cities)
(literalize mst-data cost-so-far tag)

(vector-attribute city-list)


(literalize initialized value)
;8/27/91 Modified start-city to use new map-vector rhs-macro.  This will
;cut down on instantiations of start-city and create new nodes to "open"
;in quick succession.
(p start-city
  (meta (priority 0))
  {<start> (start ^start-city <sc> ^length <length> )   }
           (initialized ^value t)
   -->
```

```
  (oremove 2)
  (map-vector <start> city-list (item <city> vector-less-item <vli>)
       (bind <tag>  (ngenatom))
     (in-parallel
       (make connect-goal ^tag <tag> ^city1 <sc> ^city2 <city>
                          ^length (compute <length> - 1)
                          ^city-list  <vli>)
       (make so-far ^tag <tag> ^distance 0 ^cities-seen <sc>)))
   )

(p finish-trip
   (meta (priority 0) (control-fn compare-new-solution-with-solution)
         (control-generator ((gen-control-data *tsp-distance* (ari (<d-so-far> + <d> + <d2>))))))
   {<sofar> (so-far ^tag <tag> ^distance <d-so-far> ^cities-seen <start-city> ) }
   {<cg> (connect-goal ^tag <tag> ^city1 <city1> ^city2 <city2> ^length 0 ) }
   (distance ^city1 <city1> ^city2 <city2> ^distance <d>)
   (distance ^city1 <city2> ^city2 <start-city> ^distance <d2> )
  -->
   (bind <cities-seen> (litval cities-seen))
   (make solution-goal  ^distance (compute <d-so-far> + <d> + <d2> )
           ^cities-seen (substr <sofar> <cities-seen> inf) <city2> <start-city>))

;Use action parallelism to reduce the run time of rules which do a
;lot of processing in their righthand sides.
(p propagate-city-5
   (meta (priority 1) (control-fn compare-with-solution)
         (priority-queue t)
         (lock-not-required t)
         (priority-fn propagate-city-priority-fn)
         )
   {<sofar> (so-far ^tag <tag> ^distance <d-so-far> ) }
   {<cg> (connect-goal ^tag <tag> ;^est-cost {<> nil <e-cost>}
                       ^city1 <city1> ^city2 <city2> ^length { = 5 <l> } ) }
   (home-city ^name <home>)
   (distance ^city1 <city1> ^city2 <city2> ^distance <d>)
-->
   (bind <cities-seen> (litval cities-seen))
   (map-vector <cg> city-list ( item <new-city> vector-less-item <vli>)
      (bind <newtag> (ngenatom))
     (in-parallel
      (make connect-goal ^tag <newtag> ^city1 <city2> ^city2 <new-city>
           ^length (compute <l> - 1)
           ^city-list  <vli> )
      (make so-far ^tag <newtag> ^distance (compute <d-so-far> +  <d>)
             ^cities-seen (substr <sofar> <cities-seen> inf) ;previously visited
                      <city2> ) ;and the new city
   )))

;No action parallelism in righthand sides as by the time these rules are
;invoked, full rule parallelism will be in use.
(p propagate-city-lt-5
   (meta (priority 1) (control-fn compare-with-solution)
         (priority-queue t)
         (lock-not-required t)
```

```
        (priority-fn propagate-city-priority-fn) ;for now, a variable, later a function...
        ;make control generator an externally compiled function...
        ;(control-generator ((gen-control-data *tsp-distance* (ari (<e-cost> + 0))))))
        )
   {<sofar> (so-far ^tag <tag> ^distance <d-so-far> ) }
   {<cg> (connect-goal ^tag <tag> ;^est-cost {<> nil <e-cost>}
                       ^city1 <city1> ^city2 <city2> ^length { > 0 < 5 <l> } ) }
   (home-city ^name <home>)
   (distance ^city1 <city1> ^city2 <city2> ^distance <d>)
-->
   (bind <cities-seen> (litval cities-seen))
   (map-vector <cg> city-list ( item <new-city> vector-less-item <vli>)
      (bind <newtag> (ngenatom))
      (make connect-goal ^tag <newtag> ^city1 <city2> ^city2 <new-city>
           ^length (compute <l> - 1)
           ^city-list  <vli> )
      (make so-far ^tag <newtag> ^distance (compute <d-so-far> +  <d>)
             ^cities-seen (substr <sofar> <cities-seen> inf) ;previously visited
                      <city2> ) ;and the new city
   ))



(p init-distance-table
   (start)
  [(distance ^city1 <c1> ^city2 <c2> ^distance <d>)]
  -->
   (map-set
     (add-to-distance-table ($varbind '<c1>) ($varbind '<c2>)
                            ($varbind '<d>)))
   (make initialized ^value t))


 (p start
   (go)
 -->
 (in-parallel-sync
  (make start ^start-city NY ^length 6 ^city-list SEATTLE HTFD SF  CHI PHOENIX BOSTON)
   (make home-city ^name NY)
   (make distance ^city1 NY  ^city2 SF ^distance 3000)
   (make distance ^city1 NY  ^city2 HTFD ^distance 80 )
   (make distance ^city1 NY  ^city2 SEATTLE  ^distance 3500)
   (make distance ^city1 NY  ^city2 CHI ^distance 1500)
   (make distance ^city1 NY  ^city2  PHOENIX ^distance 2300)
   (make distance ^city1 NY  ^city2  BOSTON  ^distance 190)

   (make distance ^city1 SF  ^city2 HTFD ^distance 3040)
   (make distance ^city1 SF  ^city2 SEATTLE ^distance 450)
   (make distance ^city1 SF  ^city2 CHI ^distance 2000)
   (make distance ^city1 SF  ^city2 NY ^distance 3000)
   (make distance ^city1 SF  ^city2  PHOENIX ^distance 1850)
   (make distance ^city1 SF  ^city2  BOSTON  ^distance 2900)
```

```
   (make distance ^city1 SEATTLE ^city2  SF ^distance 450)
   (make distance ^city1 SEATTLE ^city2  HTFD ^distance 3600)
   (make distance ^city1 SEATTLE ^city2  NY ^distance 3500)
   (make distance ^city1 SEATTLE ^city2  CHI ^distance 2200)
   (make distance ^city1 SEATTLE ^city2  PHOENIX ^distance 2300)
   (make distance ^city1 SEATTLE  ^city2  BOSTON  ^distance 3400)

   (make distance ^city1 CHI  ^city2  NY ^distance 1500)
   (make distance ^city1 CHI  ^city2  SF ^distance 2000)
   (make distance ^city1 CHI  ^city2  HTFD ^distance 1450)
   (make distance ^city1 CHI  ^city2  SEATTLE  ^distance 2200)
   (make distance ^city1 CHI  ^city2  PHOENIX ^distance 1400)
   (make distance ^city1 CHI  ^city2  BOSTON  ^distance 1610)


   (make distance ^city1 HTFD  ^city2  NY ^distance 80)
   (make distance ^city1 HTFD  ^city2  SF ^distance 3040)
   (make distance ^city1 HTFD  ^city2  SEATTLE ^distance 3600)
   (make distance ^city1 HTFD  ^city2  CHI ^distance 1450)
   (make distance ^city1 HTFD  ^city2  PHOENIX ^distance 2350)
   (make distance ^city1 HTFD  ^city2  BOSTON  ^distance 110)


   (make distance ^city1 PHOENIX  ^city2  NY ^distance 2750)
   (make distance ^city1 PHOENIX  ^city2  HTFD ^distance 2700)
   (make distance ^city1 PHOENIX  ^city2  SF ^distance 1000)
   (make distance ^city1 PHOENIX  ^city2  SEATTLE ^distance 2300)
   (make distance ^city1 PHOENIX  ^city2  CHI ^distance 1800)
   (make distance ^city1 PHOENIX  ^city2  BOSTON  ^distance 2600)


   (make distance ^city1 BOSTON ^city2 NY ^distance 190)
   (make distance ^city1 BOSTON ^city2 SF ^distance 2900)
   (make distance ^city1 BOSTON ^city2 SEATTLE ^distance 3400)
   (make distance ^city1 BOSTON ^city2 CHI ^distance 1610)
   (make distance ^city1 BOSTON ^city2 HTFD ^distance 110)
   (make distance ^city1 BOSTON ^city2 PHOENIX ^distance 2350)

  )
   (init-distance-table 5)
  )

;If better solution found, propagate it.
;Note:  Without locking mechanism, this rule has two error modes,
;depending on when remove is performed.  If the remove comes
;second, then there will be two solutions for a brief period.
;Another rule might fire on the old solution value, and create
;a superfluous solution (or, in some systems, might overwrite the
;new, better solution).  If the remove comes first, then there
;will be a brief period in which no solution exists, in which
;case an init production referencing -(solution) might fire.

(p init-solution
   (meta (priority 0))
```

```
   {<new> (solution-goal ^distance <dist> ^tag <tag> ) }
         -(solution-goal ^distance < <dist>)
         -(solution)
  -->
 (bind  <cities-seen> (litval cities-seen))
  (make-unique solution ^tag <tag> ^distance <dist>
        ^cities-seen (substr <new> <cities-seen> inf)))

;Now here's an application for functionally accurate programming.
;Note that this rule may fire, even though, while it's firing,
;a solution-goal whose distance is < <dist> may appear, in
;effect disabling this rule.   But, because solution is locked,
;competing rule won't fire until new solution is postulated, so
;no harm is done, and correct solution eventually becomes asserted.

(p new-and-improved
  (meta (priority 0))
  {<new> (solution-goal ^distance <dist> ^tag <tag> ) }
        -(solution-goal ^distance < <dist>)
  {<old> (solution ^distance > <dist> ^tag <oldtag>)  }
 -->
  (bind  <cities-seen> (litval cities-seen))
  (make solution ^tag <tag> ^distance <dist>
        ^cities-seen (substr <new> <cities-seen> inf))
  (remove <old>)
  )
```

```
;SALES-CONTROL.LISP
;These are the lisp functions used to implement heuristic control
;in the travelling salesperson program.


(defvar *solution-so-far* nil "The solution generated so far")
(defvar *control-variables* '(*solution-so-far*))

;A pruning function which determines if a new solution is better than
;the current solution.  If so, records the new value and returns t.

(defun  compare-new-solution-with-solution(instance)
  (declare (optimize (speed 3) (safety 0) (space 0)))
  "Compare-with-solution(control-data): Control-data is an a-list derived from
   the rule-instance-control-data slot.   If the control function returns a
   non-nil value, the rule should be executed, otherwise it should be pruned."
  (let ((new-solution (cdr (assoc '*tsp-distance* (rule-instance-control-data instance)))))
     (cond ((not *solution-so-far*)
            (setf *solution-so-far* new-solution))
           ((< new-solution *solution-so-far*)
            (setf *solution-so-far* new-solution))
           (t
            nil ;indicating bad rule -- don't execute
            ))))

;This function is used to compare a developing solution with a complete solution.
;If the developing solution ever exceeds the current best, then return nil.
;If no current best solution, return t.
(defun compare-with-solution(instance)
  (declare (optimize (speed 3) (safety 0) (space 0)))
  "Compare-with-solution(control-data): Control-data is an a-list derived from
   the rule-instance-control-data slot.   If the control function returns a
   non-nil value, the rule should be executed, otherwise it should be pruned."
  (if *solution-so-far*
       (< (rule-instance-rating instance) ;distance travelled < best-solution
           *solution-so-far*)
      t))


;Priority computation for propagate city

(defun propagate-city-priority-fn()
  (+ ($varbind '<d-so-far>)
     (compute-mst (list ($varbind '<city2>))
                  (cons ($varbind '<home>) (substr-to-list '<cg> 'city-list 'inf)))))

;Minimum Spanning Tree Computation:


(defvar *distance-table* nil)
(defun init-distance-table(n-cities)
     (setf *distance-table*
           (new-dhash-table n-cities)))
```

```
(defun add-to-distance-table(city1 city2 distance)
    (let ((tmp (get-dhash city1 *distance-table*)))
      (set-dhash city1
                 (push (cons city2 distance)
                       tmp)
                 *distance-table*)))

(defun do-mst-fn(city-list)
    (compute-mst (list (car city-list))
                 (cdr city-list)))

(defun compute-mst(seen-cities not-seen-cities)
    (let ((min-so-far 0)
          (min-city nil)
          (dist 0))
      (while not-seen-cities
        (setf min-city (car not-seen-cities))
        (setf min-so-far (apply #'min
                                (mapcar #'(lambda(city1)
                                            (city-distance city1 min-city))
                                        seen-cities)))
        (mapc #'(lambda(seen)
                  (mapc #'(lambda(not-seen)
                            (setf dist (city-distance seen not-seen))
                            (cond ((< dist min-so-far)
                                   (setf min-so-far dist)
                                   (setf min-city not-seen))))
                        (cdr not-seen-cities)))
              seen-cities)
        (push min-city seen-cities)
        (setf not-seen-cities (delete min-city not-seen-cities)))
      min-so-far))

(defun city-distance(city1 city2)
     (cdr (assoc city2
                 (get-dhash city1 *distance-table*)
                 :test #'eq)))
```