Parallel OPS5
User's Manual and Technical Report

Daniel E. Neiman

COINS Technical Report 91-1

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003
CSNET: DANN@CS.UMASS.EDU

Abstract

This report describes the use of the experimental Parallel OPS5 developed at the University of Massachusetts and discusses a number of the implementation issues. The report also contains a brief tutorial on programming parallel rule-based systems. This tutorial covers the topics of debugging techniques and program structure, and concludes by analyzing two versions of an OPS5 benchmark in terms of their potential for parallel execution.

[0]This work was supported in part by the Office of Naval Research under a University Research Initiative grant, ONR N00014-86-K-0764, and NSF-CER contract DCR-8500332

Contents

Chapter 1

Introduction and Language Overview

OPS5 is a well-known language for implementing production systems. This report describes the first release of a version of OPS5 which has been modified to support a number of levels of parallel activity, primarily *rule* parallelism which allows rules to be fired concurrently, and *matching* parallelism which allows the pattern matching to be performed in parallel. The first section of this paper 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 use of the current version of parallel OPS5. The current version of OPS5 provides only the basic capability for executing parallel rules and does not provide language constructs or utilities for enforcing correctness or serializability in computations. A version which will incorporate more language constructs for supporting program development is planned, and a number of these language constructs are discussed. In the meantime, the third section of the paper discusses restrictions on parallel rule firing, briefly explains how to debug a parallel production system, and concludes with tutorial on programming rule-based systems in parallel in the form of an extended analysis of the potential parallelism in two versions of an OPS5 program. The final section is devoted to discussing the issues involved in implementing a parallel production system based on the Rete network.

1.1 Language Overview

With a few exceptions, the parallel OPS5 is the same as that described in the OPS5 Technical Report[Forgy81]. 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

```
(block ^color red ^size medium ^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.

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[Gupta87].

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 parallelism at the *node* and *production* levels and will eventually support *action* 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.

Node Parallelism: When node 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).

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 parallel OPS5 and may change in later releases. Information on the language which is not specific to parallel OPS5 can be obtained from the *OPS5 User's Manual*[Forgy81].

Invocation: The system is contained in the file **POPSEXP5.lisp**; the compiled version is in the file **POPSEXP5.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)`. The user should then define the number of needles using the `sys::needles` command and set the correct flags to establish the level of parallelism and the conflict resolution paradigms. These flags are described below: Parallel OPS5 has exactly the same syntax as the standard OPS5, although the RHS commands `write` and `remove` have been changed to `owrite` and `oremove` to avoid conflicts with Common Lisp commands. The commands for backing up production runs don't make much sense in a parallel system and have not been tested, so they probably don't work even when running the system serially.

Flags and Variables: Parallel OPS5 currently supports production and node parallelism. They are controlled by the global variables `*production-parallelism*` and `*node-parallelism*`. Setting the variables to `t` enables that level of parallelism.

When benchmarking a parallel system, it's usually a good idea to suppress output so that the timings will be accurate. The flag `*ptrace*` controls the verbosity of the OPS5; setting `*ptrace*` to nil will turn off rule tracing.

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:

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

[2]POPSEXP5 is the fifth major modification of the experimental parallel OPS system.

```
 (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. In later versions of the parallel OPS5, a meta-language facility will allow control to be expressed in a less round-about fashion. For now, mode-changing productions should be explicitly annotated in the following way:

```
(setf (get 'go-to-next-phase 'mode-changer) t)
```

This will prevent the mode-changing production from being executed until all rules in the conflict set have fired and all working memory changes have been processed.

Rule Firing Algorithms As described in the later sections, there are three algorithms provided for firing productions concurrently.

- Fire N : Default. Fires the first N productions in the conflict set in parallel.
- Fire N with conflict-set emptying: Fires at least N productions. If more than N items are in the conflict set, all are executed. Load the file `async.zoom`.
- Asynchronous production firing: Fires rules as they enter the conflict set. Load the file `acssync.zoom`.

Chapter 3

Writing Parallel OPS5 Programs

This section discusses the somewhat thorny problem of actually programming a parallel OPS5 program. The first section discusses the general nature of parallel OPS5 programs. The restrictions that the implementation places on concurrently executing rules in order to ensure program correctness are then, outlined, and some hints on debugging parallel programs are given. The final section of this chapter discusses some of the actual programming issues involved in parallel rule firing. Because there is so little practical experience in actually writing such programs, I have chosen to analyze two programs 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. In some cases, parallel OPS5 does not yet possess mechanisms which will allow a particular idiom to be programmed; in these cases I will discuss possible language mechanisms to be incorporated in a future version of parallel OPS5.

3.1 Programming Productions in Parallel

Writing parallel production systems requires a change of paradigm. 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 this kind of programming reduces the opportunity for conflict resolution, the parallel activation of rules gives rise to a number of control issues, The programmer has no way of representing the relative utility of rules, controlling the number of rules being activated, and no way to terminate incorrect or irrelevant sequences of rule firings. These control issues are the focus of my current 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-Stolfo85, Schmolze89]. The current implementation of parallel OPS5 does not contain any built-in mechanisms for detecting and avoiding rule interactions; instead certain assumptions about the interactions of co-executing productions are made and it is the responsibility of the programmer to ensure that these assumptions hold. Briefly, these assumptions are as follows:

- Only one production may modify a given working memory element during an execution cycle.

- A production may not refer to 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 working memory, there are circumstances where this restriction does not hold).

- *All* working memory operations which affect a particular production must be completed before the production is instantiated. This is to avoid transient instantiations of a production. For example, given the following production and changes to working memory, a transient production instantiation appears in the conflict set.

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

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

```
          -->
           ...)

      (remove B ^field1 foo ^field2 wombat)      --->    example-prod into conflict set.
      (make B  ^field1 baz ^field2 wombat)       <---    example-prod out of conflict set.
```

Note that 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.

There are several modes of production parallelism, for example, in one, all the productions in the conflict set are executed concurrently after all rules in the previous execution cycle have finished executing, in another, each instantiation is executed as soon as it become eligible, and in a third, only a subset of productions are executed.

A user (who is sufficiently familiar with Lisp programming) can implement conflict resolution routines which partition the conflict set according to the above rules and which will allow arbitrary subsets of the rules in the conflict set to be executed concurrently.

3.3 Debugging Parallel OPS5 Programs

Even serial OPS5 is not the most intuitive programming language, and constructing a correct parallel OPS5 program can be quite a challenge. Debugging a parallel OPS5 program is difficult, because the actual error may occur long before it manifests itself in some obvious way. The usual error occurs because of some unexpected interaction between concurrently added working memory elements and it can be quite difficult to determine when a working memory element was added or deleted and by which production. Unfortunately, the conventional trace mechanisms involve printing out information which does not adequately represent the temporal relationships between activities. Printing out debugging information can also upset the timing between actions so as to cause the anomalous behavior to not occur.

The following section describes the tools and techniques which currently exist for debugging a parallel OPS5[2].

TopCL is equipped with a window interface which allows an output stream to be assigned to each thread. By enabling the production trace mechanisms, some idea of the order of firing and processor utilization can be obtained. The interface to TopCL allows tasks to be paused and their current stack traces to be examined. It is possible to get a pretty good model of processor utilization by successively pausing and restarting processes.

Parallel OPS5 does not yet provide mechanisms for gathering such useful data as when instantiations enter the conflict set, when working memory elements are created, which

[2]Such as they are. It should be noted that the tracing and debugging of a parallel production system is still a research issue.

productions created/deleted which elements, and so on. This information is vital for determining the effectiveness of various control schemes. Currently, the user must modify the OPS5 code to collect the necessary data, but the next release of parallel OPS5 will provide mechanisms for metering. The most useful locations for inserting timing code is at the `match` function (for gathering information about the length of time required to match a particular working memory element), and in the production execution loop in the function `main` (which is useful for gaining information about start time and execution time of individual rules). The file `time-acssync.lisp` contains examples.

3.4 Analyzing the Waltz Benchmark for Rule Parallelism

In this section, I discuss two simple versions of the Waltz line labelling algorithm in terms of their potential for various levels of parallelism: node, action, rule, and asynchronous rule execution. The origin of the first benchmark is unknown and is simply called Waltz; the second was written by Toru Ishida and will be referred to as Toru-Waltz[3].

It turns out that the Waltz and Toru-Waltz programs are 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 principle problems in adapting these programs to parallelism are removing the extraneous rules from the conflict set and firing the eligible rules at the earliest possible time without accidentally executing a transient instantiation.

The first program I will examine is the Waltz benchmark. This program is very serial in nature and contains a number of standard OPS5 idioms which limit or eliminate the potential for parallelism (which is why it was chosen for analysis). The Toru-Waltz program proves to be much easier to parallelize, but still requires modification to allow rules to be executed in parallel.

3.4.1 Waltz Benchmark Program Structure

Like most OPS5 programs, the Waltz benchmark is partitioned into logical sections. These are

- Start: A single production which sets the initial mode.
- Duplicate: An operation which iterates over the list of lines and transforms them into edges, one edge for each endpoint of the line segment.
- Detect Junctions: Each junction is detected and labelled as either a three-way junction or a two-way junction (an "L").
- Find-initial-boundary: Locates the initial point to be processed and determines its type.

[3]The text of the benchmarks is given in the appendix.
[4]Well...hardly ever.

- Secondary-boundary-junction: Locates the second boundary point and determines its type.
- Labelling: Each unlabelled edge is labelled using the constraints of adjacent points.

Initialization:

The initialization phase reads in a number of line segments. By using *action* parallelism, this phase can be speeded up by a factor proportional to the number of available processors and the number of working memory elements which must be asserted. But note that due to the gating effect of the `stage` working memory element, very little matching actually takes place until the stage element is modified – this limits the potential speedup due to action parallelism.

Duplication:

This phase is implemented using a typical OPS5 loop. An initial mode flag is set indicating that the phase is duplication. A production, `reverse-edges` executes, replacing each 'line' element with two edges. A final production modifies the mode when no more line elements exist. When the conflict set is examined, it can be seen that there is one instance of the duplication rule for each line element in memory. Because there are no potential interactions, all the instantiations can be executed simultaneously. This is an example of object or data parallelism in which an operation can be applied to each element in memory indepedently. The expected speedup is proportional to the number of instantiations in the conflict set modulo the number of processors available.

Junction Detection

As the duplication rules fire, they create the data necessary to enable the junction detection rules. We would like to discover how much overlap exists between the two phases and whether rule firings from one phase can occur in parallel with rule firings from the other.

There are two types of junction detection rules, one for 3-way junctions and the other for 2-way junctions. As can be seen in Figure 3.1, the rule `make-3-junction` contains no negated elements and cannot fire until all the data is present. There is no difficulty with executing these rules as soon as they are enabled by the creation of the appropriate edges. The rule `make-2-junction` is more problematical. Because it contains a negated condition element, it may become enabled by the addition of *edge* working memory elements, only to be disabled by the addition of a *third* element. Furthermore, transient instantiations of the `make-L` rule can be created by the execution of the `make-3` rule as it modifies edge elements. Therefore, asynchronous production execution is contraindicated in this instance.

Combinatorial Growth in the Conflict Set: Examination of the conflict set reveals another problem: each set of points eligible for a 3-way junction generates six entries in the conflict set, and each set of points eligible for a two-way junction generates two! This is because the rules do not specify a specific ordering among the elements, so the number of instances grows combinatorially. Even in a serial system this represents a fair bit of overhead as the extra instances are generated, entered into and deleted from the conflict set, however it is a major disaster in a parallel production system which attempts to execute all the instances in parallel. This would cause redundant rules to be executed which would not only corrupt the database but would also occupy many processors inefficiently.

At present, the only way to avoid the problem of combinatorial growth in the conflict set is to specify explicit distinctions between the working memory elements. For example, in the `make-3-junction` rule, we can eliminate the problem of combinatorial growth by specifying an ordering of the edges according to the values of the second edge, e.g.

```
(edge ^p1 <base-point> ^p2 <p1> ^joined false)
(edge ^p1 <base-point> ^p2 {<p2> > <p1>} ^joined false)
(edge ^p1 <base-point> ^p2 {<p3> > <p2>} ^joined false)
```

Because this code specifies a strict ordering of the lines, only one instantiation of the rule can result. Pragmatically, this is an adequate solution, however a more careful analysis reveals that this approach can still cause excessive growth in the node memories, as there are several cases which satisfy the first inequality (<p2> > <p3> , <p1> > <p3> , and <p1> > <p2>). Only the last will actually result in an instantiation. Because the collect-all-working-memory-elements-with-a-given-value idiom is so common, a language construct to implement the match in a non-combinatorial fashion would be very useful and is under consideration for inclusion in the next language release.

Again, the whole point here is to insure that only instantiations that should be executed enter the conflict set.

Labelling Junctions – The Relaxation Process

The labelling process proceeds by relaxation, that is, the program propagates junction labels by using already labelled adjacent edges to constrain the types of labels that can be assigned to a junction. The relaxation process is well suited for data-driven programming in that rules only become enabled when there is sufficient information to unambiguously label a junction. If properly formulated, the constraint propagation is monotonic – possible labellings are only eliminated, never created.

Because there are arbitrarily many junctions in a diagram, the potential for rule parallelism is high. The potential for node parallelism is more difficult to estimate. The number of productions that each rule execution is likely to stimulate is equal to the maximum number of connected junctions (2) times the maximum number of incorrect possible labellings of each connected junction (between 2 and 5). So the *maximum* benefit of node parallelism would be a factor of 10, and the average would be much less.

Because the labelling algorithm, as implemented in the Waltz benchmark, is not suitable for production level parallelism and is implemented in an unclear fashion, I will leave the further discussion of relaxation until the Toru-Waltz program is discussed. I will note only that the continued use of the modify command in the righthand side of the Waltz program labelling algorithms prohibits the use of asynchronous rule execution during constraint propagation, which would seriously limit performance, if executed in parallel.

```
;If three edges meet at a point and none of them have already been joined in
;a junction, then make the corresponding type of junction and label the
;edges joined.  This production calls make-3-junction to determine
;what type of junction it is based on the angles inscribed by the
;intersecting edges
(p make-3-junction
(stage ^value detect-junctions)
(edge ^p1 <base-point> ^p2 <p1> ^joined false)
(edge ^p1 <base-point> ^p2 {<p2> <> <p1>} ^joined false)
(edge ^p1 <base-point> ^p2 {<p3> <> <p1> <> <p2>} ^joined false)
-->
(make junction
      ^type (make-3-junction <base-point> <p1> <p2> <p3>)
              ^base-point <base-point>)
(modify 2 ^joined true)
(modify 3 ^joined true)
(modify 4 ^joined true))

;If two, and only two, edges meet that have not already been joined, then
;the junction is an "L"
(p make-L
(stage ^value detect-junctions)
(edge ^p1 <base-point> ^p2 <p2> ^joined false)
(edge ^p1 <base-point> ^p2 {<p3> <> <p2>} ^joined false)
- (edge ^p1 <base-point> ^p2 {<> <p2> <> <p3>})
-->
(make junction
^type L
^base-point <base-point>
^p1 <p2>
^p2 <p3>)
(modify 2 ^joined true)
(modify 3 ^joined true))
```

Figure 3.1: The Detect Junction Rules.

3.4.2 Mode Changes

One serializing feature in the Waltz system (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 Waltz benchmarks. 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 Waltz program, the `detect-junctions` mode declares that no new edges will be created, therefore any rule referencing edges can now fire. 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.

The use of modal working memory elements can slow down a computation because otherwise eligible productions can not enter the conflict set until the mode of the element is changed. 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 "burst" effect in which a significant amount of matching must take place once the gating element finally arrives (see Figure 3.4.2).

The delay due to the burst 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 must be unified with any field in any other condition element of the rule.

Parallelism can be used to minimize the delays caused by gating in a number of ways. Because a gating element typically affects many production instantiations, node parallelism can be effective in minimizing the time consumed in the matching process. 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 second burst effect when processes have to be assigned to each instantiation in the conflict set.

3.5 Analyzing the Toru-Waltz Benchmark for Rule Parallelism

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.

- 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.

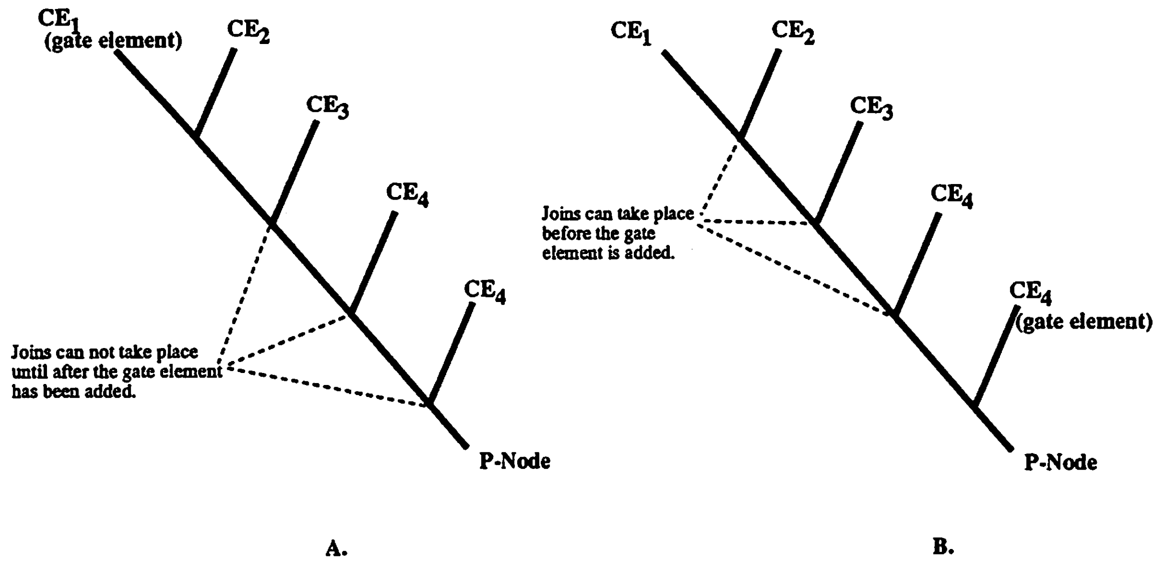

CE1
(gate element)
CE2
CE3
CE4
CE4
Joins can not take place until after the gate element has been added.
P-Node
A.
CE1
CE2
CE3
CE4
Joins can take place before the gate element is added.
CE4
(gate element)
P-Node
B.

Figure 3.2: The location of the gating element affects the amount of partial matching which can take place in the match process.

- 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. Implemented as single productions, they would be somewhat facilitated by the addition of action parallelism. As discussed previously, however, the real bottleneck in these phases is the propagation of the mode-changing working memory element.

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. When asynchronous rule parallelism is allowed, the `enumerate` phase actually overlaps the `make-data` phase.

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. This phase is also monotonic, because there are no rules which reference junctions in a negative condition element; rules are never enabled when junctions are deleted. The entire constraint propagation phase can be carried out in parallel. It should be noted, however, that it *is* possible for two rules to attempt to delete a working memory element representing a junction at the same time. Because this is semantically correct, there is no need for synchronization. The overhead of synchronizing the superfluous rule would be greater than the cost of executing the production as a no-op.

3.5.1 Control in Toru-Waltz

The Toru-Waltz program has one feature which makes it difficult to run in parallel; it uses the standard conflict resolution techniques in order to change modes. With this technique, a special production is written which appears in the conflict set with such a low priority that it does not execute until no other instantiation is in the conflict set, then the production fires, changing the current mode. This construct simply does not work in a parallel rule-firing system, as the mode-changing production will be executed concurrently with all other rules, causing the mode to change prematurely.

At the present time, parallel OPS5 has no language mechanisms to support mode-changing. As described earlier, the current solution is to explicitly tag the mode-changing productions; they are then placed in a separate conflict set which is not checked until all rules in the main conflict set are exhausted. This particular kludge will be replaced by a general purpose meta-rule facility in the next version of the language.

3.6 Summary – Programming Parallel OPS5

This section has examined a number of common OPS5 idioms in terms of their suitability for parallel execution. In general, the following generalizations hold true.

- A computation which makes monotonic changes to working memory is usually safe for rule-level parallelism, provided the rules match discrete objects.
- Initialization phases support action parallelism.
- Rules which contain `modify` commands in their righthand sides typically are not compatible with asynchronous rule execution.
- Conflict resolution cannot be used as a control mechanism if rule-level parallelism is employed.
- Mode-changing productions should be supported by either node-level parallelism or by asynchronous rule execution in order to maximize processor utilization.

Chapter 4

Implementation

In this section, I discuss 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[Gupta87]. Because this work is undoubtedly familiar to the interested reader, I concentrate 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, I will give a short overview 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 principle 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 fields of a condition element (intra-element tests) or 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 main 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.

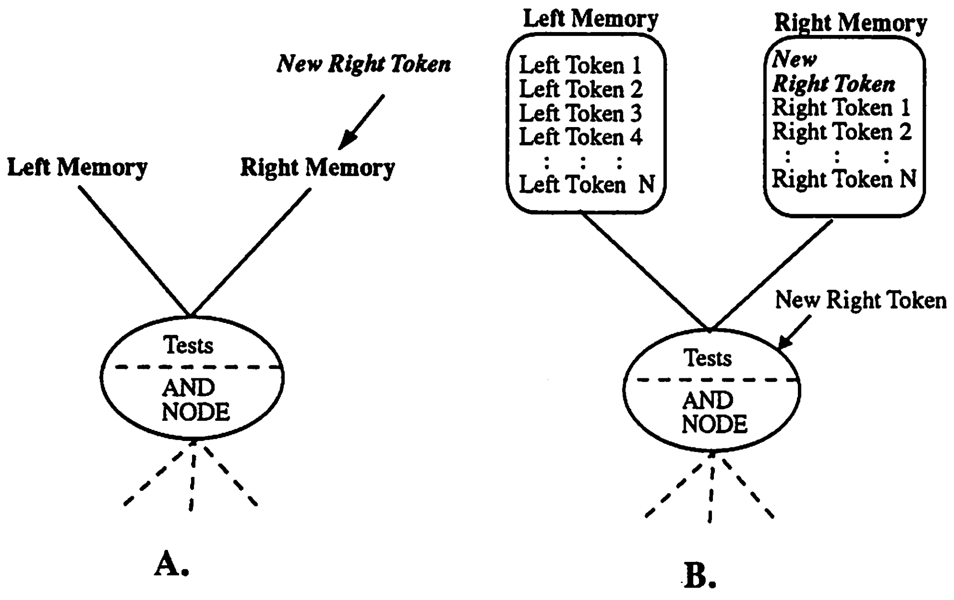
Pairs of tokens which satisfy the tests are concatenated into a new token and passed to the succeeding nodes in the network. Because an AND node is basically symmetrical, this description covers the case of tokens arriving from both the left and right sides. In the case of a *negated* token (that is, 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 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.

[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.



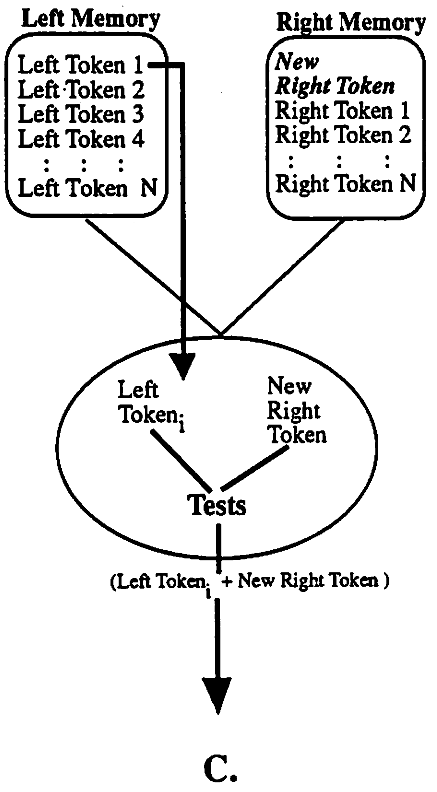


Figure 4.1: A token arrives at an AND node.

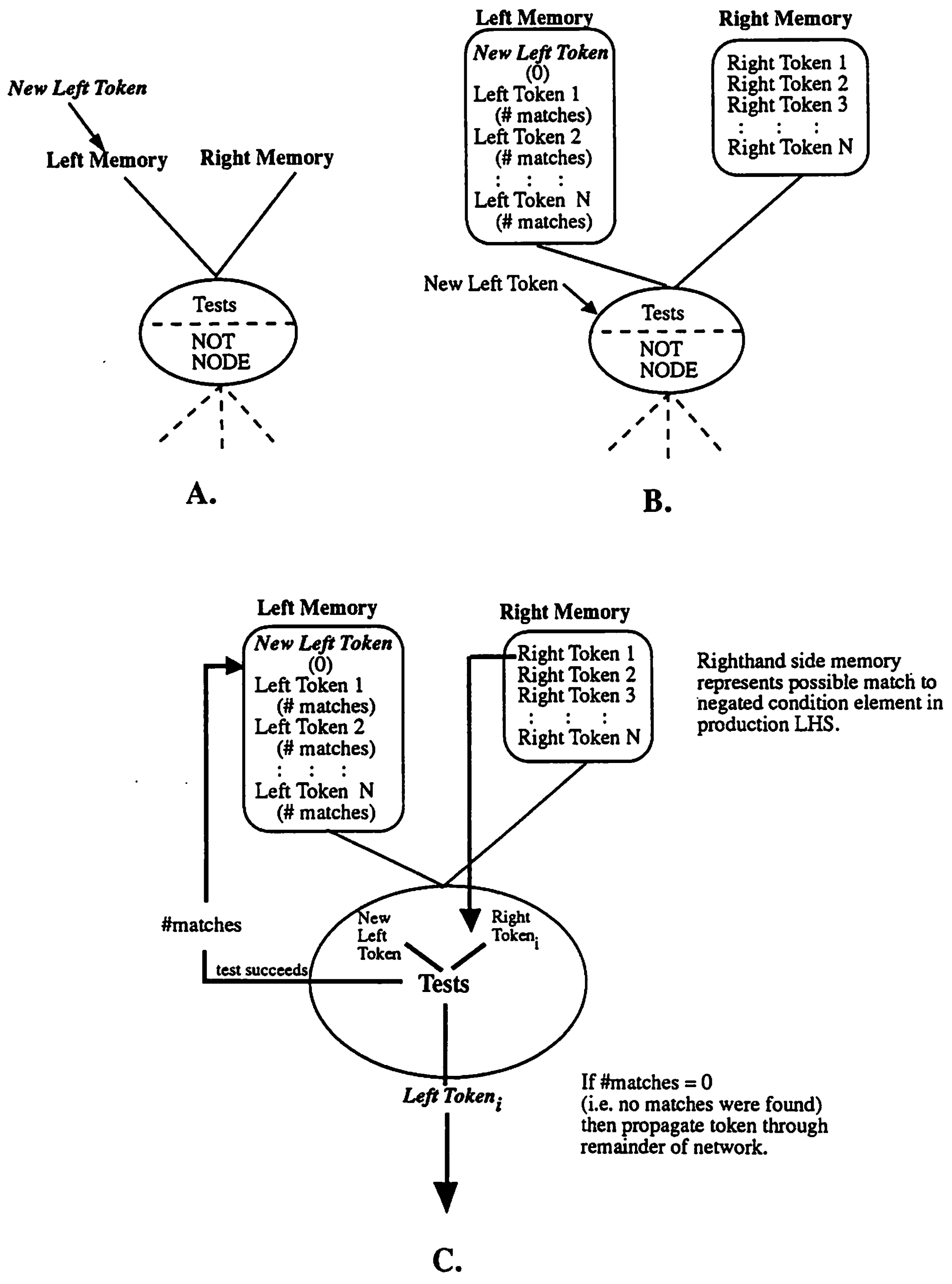


Figure 4.2: A token arrives at the lefthand input of a NOT node.

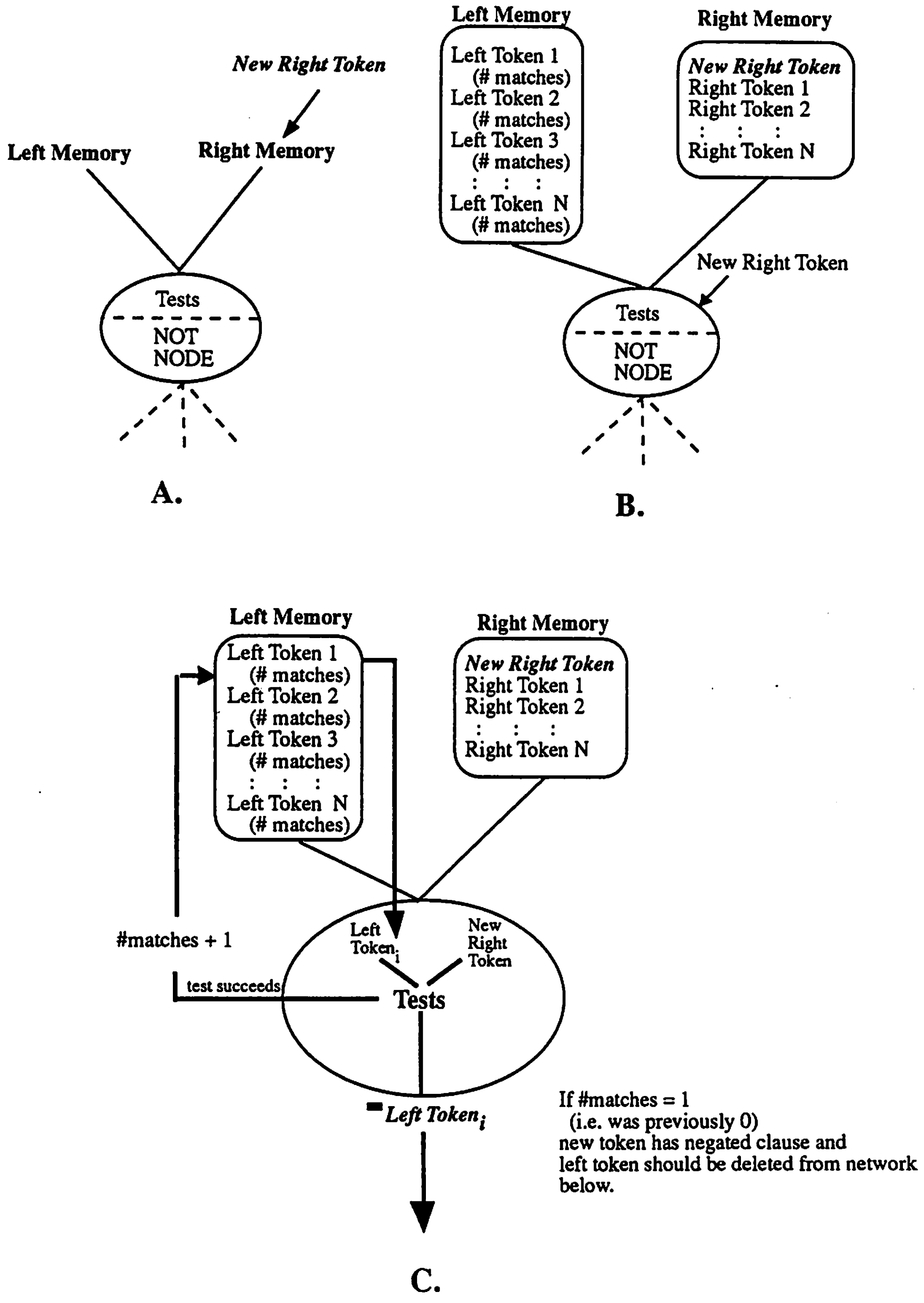


Figure 4.3: A token arrives at the righthand input of a NOT node.

4.2 Implementing Node-level Parallelism

Any degree of parallelism in a system which uses a Rete net pattern matcher implies the presence of (or capability for) node and intra-node parallelism. 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.

Implementing node 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 node 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 traverses 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 more complicated to implement. 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. So 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 below ensures that the state of the network remains consistent.

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. It is inevitable that eventually a token will arrive at either the left or right input while a token is still being matched 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 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. But, the problem only arises when two *matching* tokens arrive simultaneously. A non-matching token arriving at the opposite input can be processed without difficulty. So a locking approach reduces throughput.

In my implementation of parallel OPS5, the 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 I took 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 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 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 total 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.

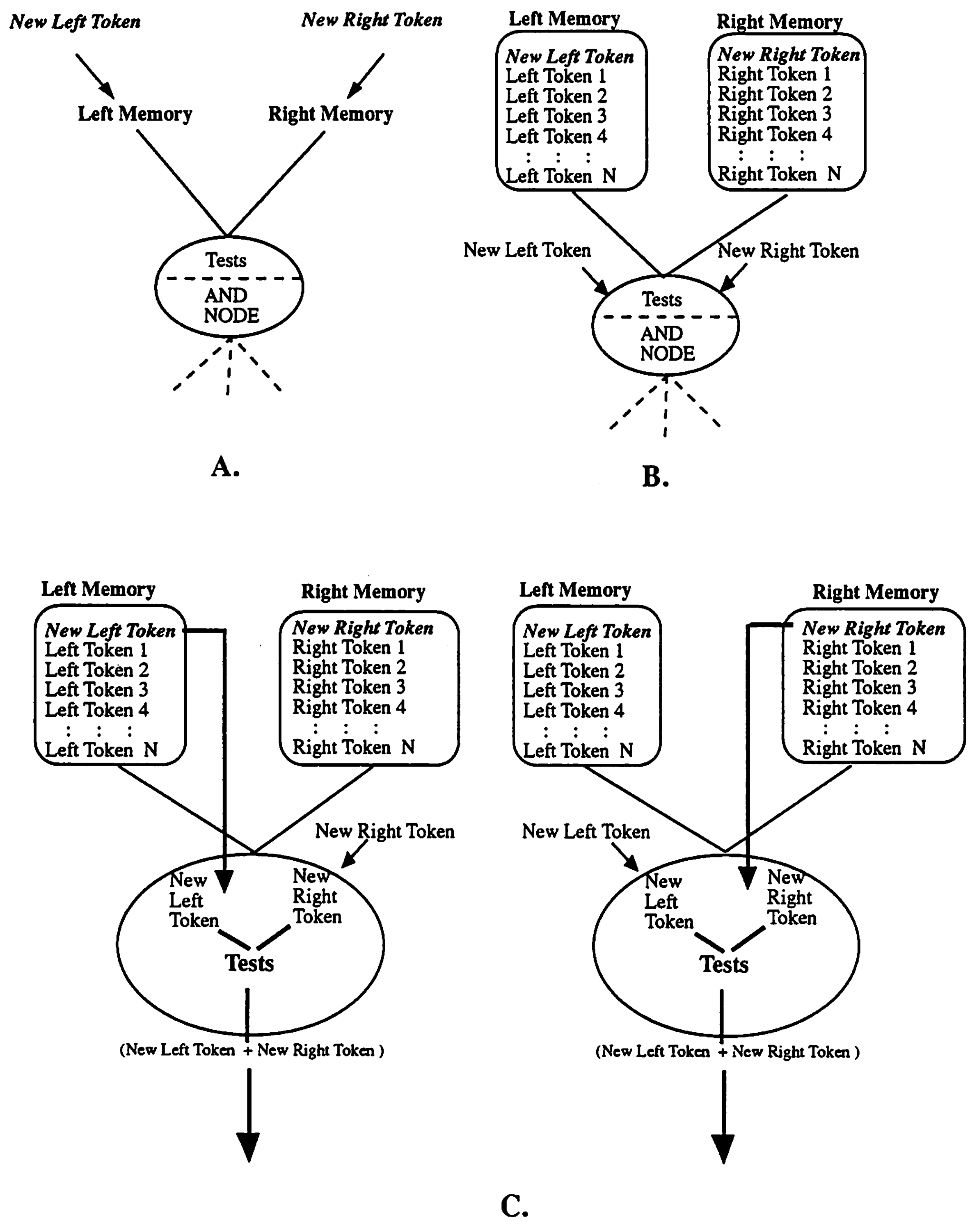


Figure 4.4: When matching tokens arrive at an AND node simultaneously.

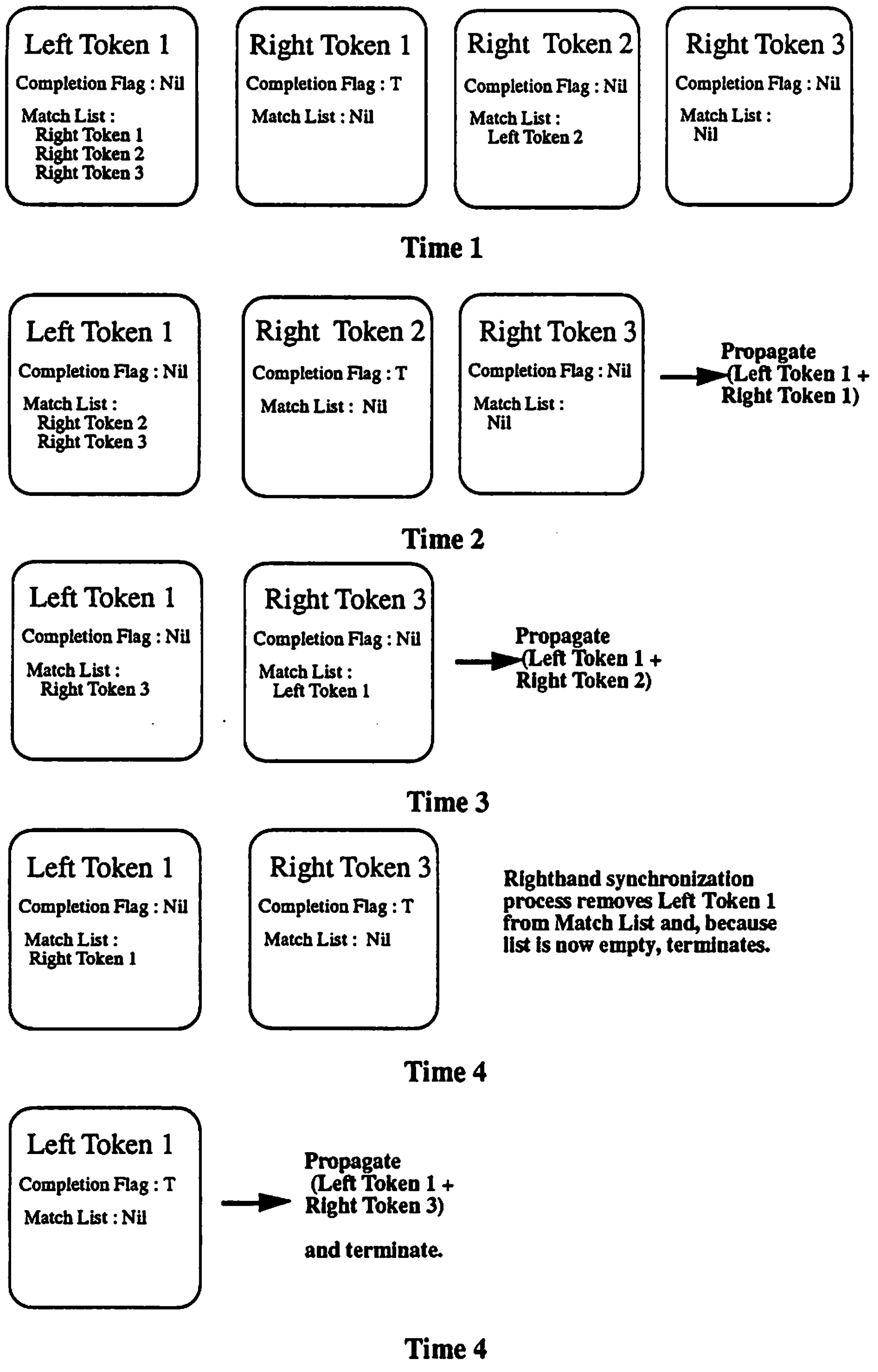


Figure 4.5: The synchronization process for an AND node.

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 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, 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 will eventually be developed.

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). If the guidelines for writing systems using production-level parallelism are followed strictly, this problem will not manifest itself in the current system, however, it will inevitably arise in any implementation which supports action parallelism. Several possible solutions are being investigated, including storing the negative token until its complement arrives and tagging individual working memory elements as deleted so that "sanity checks" can be performed on node memories.

4.5 Production Parallelism

In parallel OPS5, production parallelism is defined as executing the righthand sides of multiple productions in parallel[2] Assuming that node parallelism has already been implemented, production parallelism is relatively easy to implement. The righthand sides of the productions have to be executable in parallel. This requires the removal of all reliance on global variables for binding variables and condition element variables. The code for each righthand side is then executed by a separate processor which sequentially executes each righthand command.

Conflict Resolution Schemes In general, when production parallelism is allowed, traditional conflict resolution schemes are not appropriate. In general, I have assumed that if a production is present in the conflict set, then it should be executed, however the programmer is free to write whatever conflict resolution scheme is appropriate to the application. I present three approaches which I have tried.

- First N: In this approach, when the user specifies `(run N)`, the first N productions in the conflict set are selected. The system selects the first N items in the conflict set, or all the items if the number is less than N. A process is assigned to execute each of the N righthand sides. If there are more items in the conflict set than processors, the remainder is placed on a queue (automatically by the thread mechanism). The system then waits for all the righthand sides to execute, using the *group* synchronization mechanism. The process repeats until N productions have been executed.

[2]There is an alternative usage of production parallelism as implying that the match process for each production is carried out in parallel.

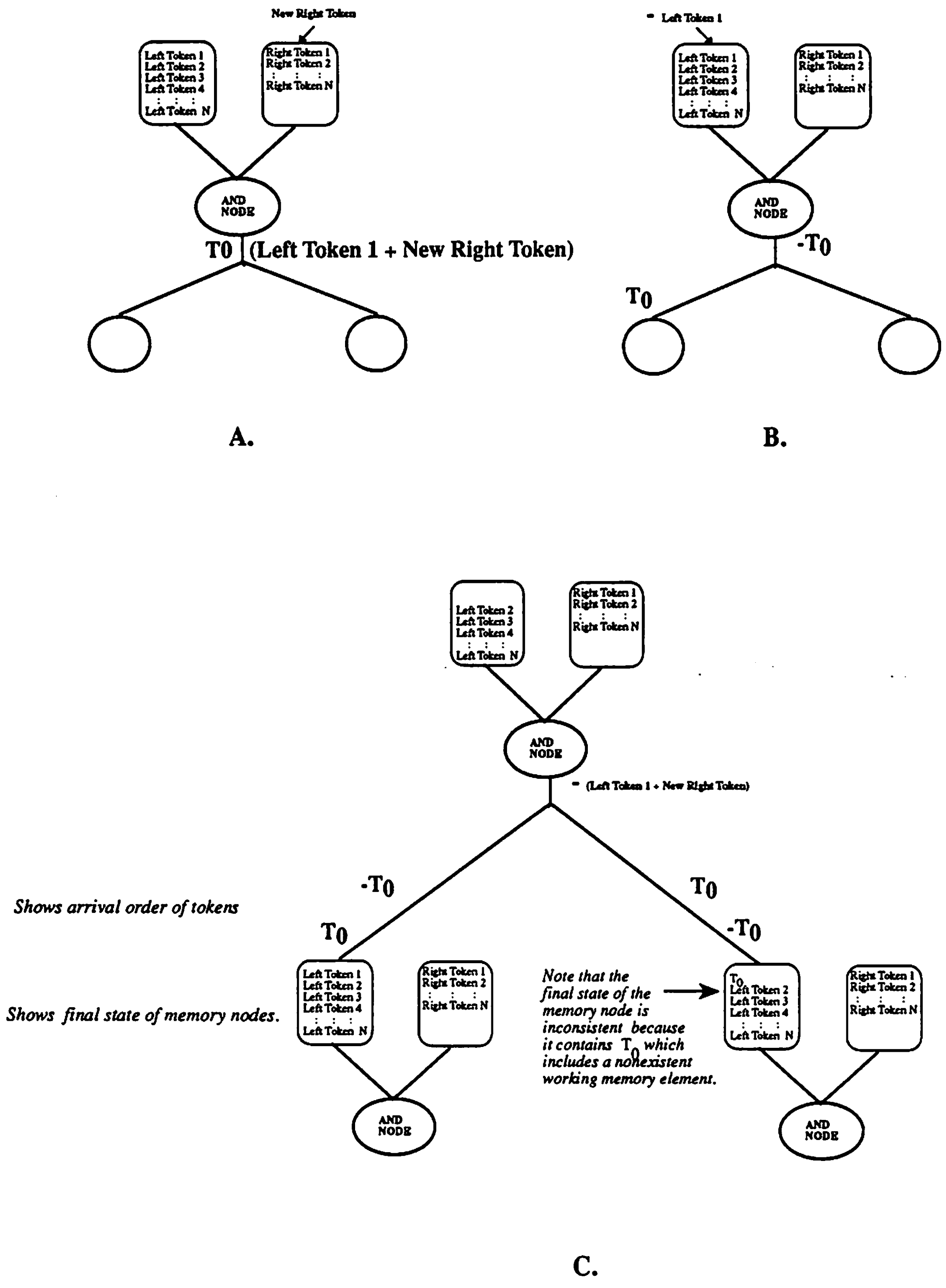

New Right Token
Left Token 1
Left Token 2
Left Token 3
Left Token 4
Left Token N
Right Token 1
Right Token 2
Right Token N
AND NODE
T0 (Left Token 1 + New Right Token)
A.
- Left Token 1
-T0
T0
B.
- (Left Token 1 + New Right Token)
Shows arrival order of tokens
Shows final state of memory nodes.
Note that the final state of the memory node is inconsistent because it contains T0 which includes a nonexistent working memory element.
C.

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

- First N with Conflict Set Emptying: A problem arises with the First N approach to conflict resolution. A production may be guaranteed to run safely in parallel with all the productions in the conflict set, but may conflict with a production instantiated by one of those productions. If the user specifies Run N, where $N < |ConflictSet|$, a situation could result in which the instantiations in the conflict set result from different phases of the computation (where phase is defined as a set of productions which produce instantiations which can be safely executed concurrently). For an example of this, see Figure 4.5. In part A, we see the First N algorithm applied. In parts B and C, we see two possible results of incremental execution of the conflict set. In part B, the program is designed so that no additional items get added to the conflict set until the entire phase has been completed. In part C, no such assurance is made, and new instantiations are added to the conflict set. If it were simply a matter of the new instantiations being executed before the original contents of the conflict set, a First-Come-First-Served algorithm would rectify the situation. However, no dynamic methods are taken to ensure that the contents of the conflict set can be safely executed in parallel, so instantiations I_{11} and I_{12} may conflict with instantiations I_4 and I_5. To ensure that this situation does not occur, a control loop is provided which ensures that the entire contents of the conflict set are executed during each execution cycle whenever the user specifies a value for N less than the size of the conflict set.

  This conflict resolution scheme is most appropriate for iterative computations which can be thought of as consisting of activities taking place at discrete time intervals, i.e. time 1, time 2, ... At any time, the contents of the conflict set consist of all actions applicable at time N.

- Asynchronous Production Execution: Both of the previous methods of production invocation involve a synchronization delay caused by waiting for all co-executing productions to terminate. A third control loop is available which executes an instantiation as soon as it enters the conflict set. If a processor is not available, instantiations are queued for execution. This approach provides the highest rate of execution, and highest degree of processor utilization, but also provides the highest probability that two conflicting productions will co-execute.

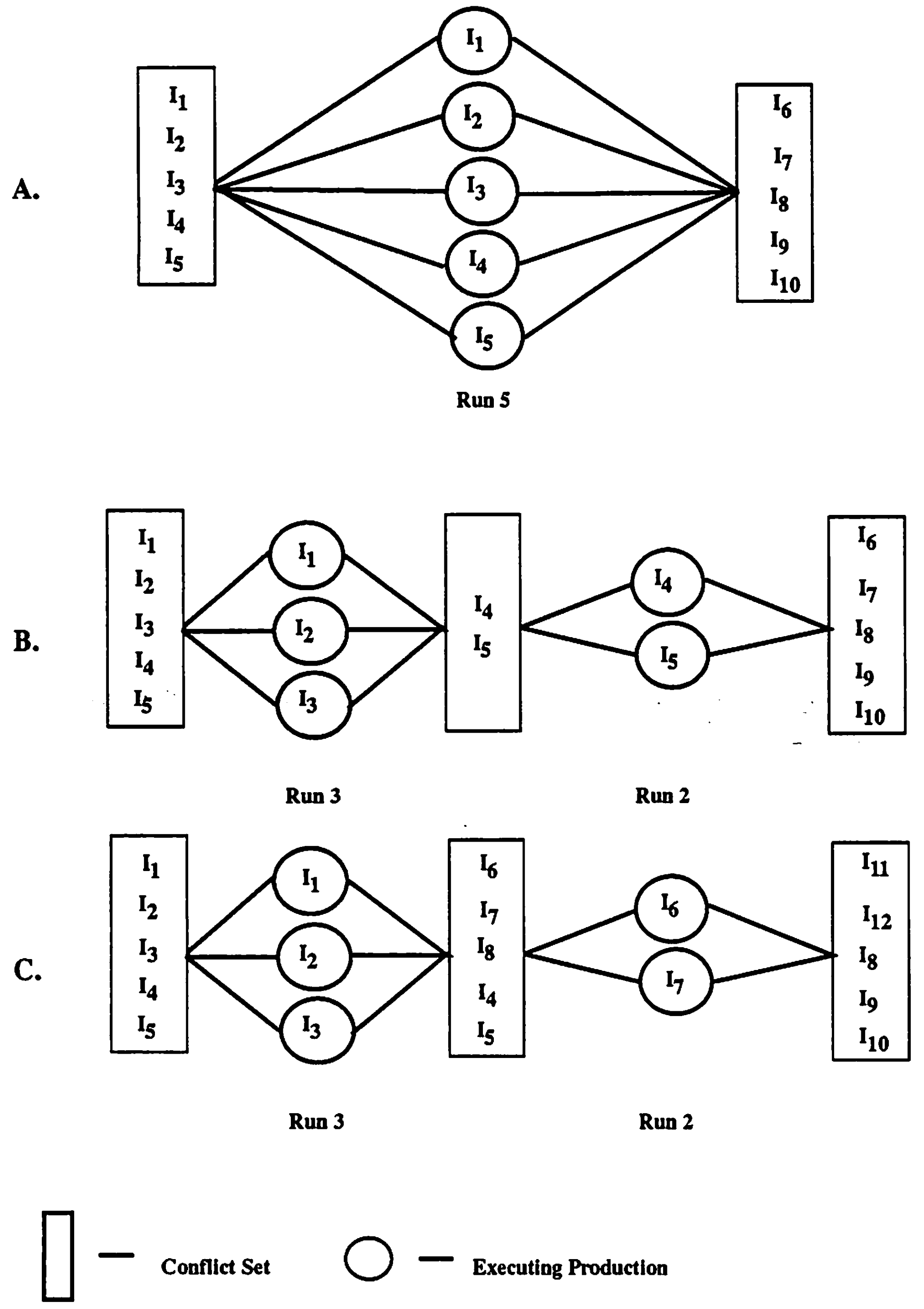

A.
I_1 I_2 I_3 I_4 I_5
I_1 I_2 I_3 I_4 I_5
I_6 I_7 I_8 I_9 I_{10}
Run 5
B.
I_1 I_2 I_3 I_4 I_5
I_1 I_2 I_3
I_4 I_5
I_4 I_5
I_6 I_7 I_8 I_9 I_{10}
Run 3
Run 2
C.
I_1 I_2 I_3 I_4 I_5
I_1 I_2 I_3
I_6 I_7 I_8 I_4 I_5
I_6 I_7
I_{11} I_{12} I_8 I_9 I_{10}
Run 3
Run 2
— Conflict Set
— Executing Production

Figure 4.7: The state of the conflict set depends on policy by which production instantiations are selected and executed.

Chapter 5

Conclusion

This document has described the operation and implementation of the parallel OPS5 developed at the University of Massachusetts. Parallel programming of rule-based programs is, as yet, not a well-defined science. Many of the programming techniques and language features discussed herein are tentative or experimental. A second release of the language is under development which is intended to provide more flexible control mechanisms, diagnostic and metering mechanisms, and robust language features for expressing parallel constructs.

Bibliography

[Forgy81] Forgy, C.L., "OPS5 User's Manual". Technical Report CMU-CS-84-135, Dept. of Computer Science, Carnegie-Mellon University, July 1981.

[Gupta87] Gupta, Anoop, *Parallelism in Production Systems*, Morgan Kaufman Publishers, Inc., Los Altos, CA, 1987.

[Ishida-Stolfo85] Ishida, T. and Stolfo, S., "Towards the Parallel Execution of Rules in Production System Programs", *Proceedings of the IEEE International Conference on Parallel Processing*, pp. 568-575, 1985.

[Schmolze89] Schmolze, James G.,"Guaranteeing Serializable Results in Synchronous Parallel Production Systems", Technical Report 89-5, Department of Computer Science, Tufts University, October, 1989.

Appendix A

The Waltz Benchmark

```
;This is the Waltz benchmark.  Its origin is unknown.
;Given a set of lines, it locates and labels junctions,
;then assigns labels to lines according to the Waltz filtering
;algorithm.

;Our WM elements.  Lines have the label line followed by the 2 points
;defining the line.  Edges are like lines accept that they can be labeled,
;permanently labelled and plotted.  Junctions are defined by 4 points.  The
;basepoint is where the 3 (2) lines intersect.  The points p1, p2, p3 are the
;other endpoints of the lines at this junction

(literalize stage value)
(literalize line p1 p2)
(literalize edge p1 p2 joined label plotted)
(literalize junction p1 p2 p3 base-point type)

(external make-3-junction)

;The Waltz Algorithm using OPS5 Production System Interpreter
;This is our production memory

;Our starting production.  It checks to see if the start flag is in WM,
;and if it is, it deletes it, and clears the screen
(p begin
(stage ^value start)
-->
(write clr)
(modify 1 ^value duplicate))

;If the duplicate flag is set, and there is still a line in WM, delete the line
;and add two edges. One edge runs from p1 to p2 and the other runs from p2 to
;p1.  We then plot the edge.
(p reverse-edges
(stage ^value duplicate)
(line ^p1 <p1> ^p2 <p2>)
-->
(write draw <p1> <p2>)
```

```
(make edge ^p1 <p1> ^p2 <p2> ^joined false)
        (make edge ^p1 <p2> ^p2 <p1> ^joined false)
(remove 2))

;If the duplicating flag is set, and there are no more lines, then remove the
;duplicating flag and set the make junctions flag.
(p done-reversing
(stage ^value duplicate)
- (line)
-->
(modify 1 ^value detect-junctions))


;If three edges meet at a point and none of them have already been joined in
;a junction, then make the corresponding type of junction and label the
;edges joined.  This production calls make-3-junction to determine
;what type of junction it is based on the angles inscribed by the
;intersecting edges
(p make-3-junction
(stage ^value detect-junctions)
(edge ^p1 <base-point> ^p2 <p1> ^joined false)
(edge ^p1 <base-point> ^p2 {<p2> <> <p1>} ^joined false)
(edge ^p1 <base-point> ^p2 {<p3> <> <p1> <> <p2>} ^joined false)
-->
(make junction
      ^type (make-3-junction <base-point> <p1> <p2> <p3>)
              ^base-point <base-point>)
(modify 2 ^joined true)
(modify 3 ^joined true)
(modify 4 ^joined true))

;If two, and only two, edges meet that have not already been joined, then
;the junction is an "L"
(p make-L
(stage ^value detect-junctions)
(edge ^p1 <base-point> ^p2 <p2> ^joined false)
(edge ^p1 <base-point> ^p2 {<p3> <> <p2>} ^joined false)
- (edge ^p1 <base-point> ^p2 {<> <p2> <> <p3>})
-->
(make junction
^type L
^base-point <base-point>
^p1 <p2>
^p2 <p3>)
(modify 2 ^joined true)
(modify 3 ^joined true))


;If the detect junctions flag is set, and there are no more un-joined edges,
;set the find-initial-boundary flag
(p done-detecting
(stage ^value detect-junctions)
- (edge ^joined false)
-->
```

```
(modify 1 ^value find-initial-boundary))

;If the initial boundary junction is an L, then we know it's labelling
(p initial-boundary-junction-L
(stage ^value find-initial-boundary)
        (junction ^type L ^base-point <base-point> ^p1 <p1> ^p2 <p2>)
        - (junction ^base-point > <base-point>)
(edge ^p1 <base-point> ^p2 <p1>)
(edge ^p1 <base-point> ^p2 <p2>)
-->
        (modify 3 ^label B)
(modify 4 ^label B)
(modify 1 ^value find-second-boundary))

;Ditto for an arrow
(p initial-boundary-junction-arrow
(stage ^value find-initial-boundary)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
- (junction ^base-point > <bp>)
(edge ^p1 <bp> ^p2 <p1>)
(edge ^p1 <bp> ^p2 <p2>)
(edge ^p1 <bp> ^p2 <p3>)
-->
(modify 3 ^label B)
(modify 4 ^label +)
(modify 5 ^label B)
(modify 1 ^value find-second-boundary))

;If we have already found the first boundary point, then find the second
;boundary point, and label it.

(p second-boundary-junction-L
(stage ^value find-second-boundary)
        (junction ^type L ^base-point <base-point> ^p1 <p1> ^p2 <p2>)
        - (junction ^base-point < <base-point>)
(edge ^p1 <base-point> ^p2 <p1>)
(edge ^p1 <base-point> ^p2 <p2>)
-->
        (modify 3 ^label B)
(modify 4 ^label B)
(modify 1 ^value labeling))

(p second-boundary-junction-arrow
(stage ^value find-second-boundary)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
- (junction ^base-point < <bp>)
(edge ^p1 <bp> ^p2 <p1>)
(edge ^p1 <bp> ^p2 <p2>)
(edge ^p1 <bp> ^p2 <p3>)
-->
(modify 3 ^label B)
(modify 4 ^label +)
(modify 5 ^label B)
(modify 1 ^value labeling))
```

```
;If we have an edge whose label we already know definitely, then
;label the corresponding edge in the other direction
(p match-edge
(stage ^value labeling)
(edge ^p1 <p1> ^p2 <p2> ^label {<label> << + - B >>})
(edge ^p1 <p2> ^p2 <p1> ^label nil)
-->
(modify 2 ^plotted t)
(modify 3 ^label <label> ^plotted t)
(write plot <label> <p1> <p2>))

;The following productions propogate the possible labellings of the edges
;based on the labellings of edges incident on adjacent junctions.  Since
;from the initial boundary productions, we have determined the labellings of
;of atleast two junctions, this propogation will label all of the junctions
;with the possible labellings.  The search space is pruned due to filtering,
;i.e. - only label a junction in the ways physically possible based on the
;labellings of adjacent junctions.


(p label-L
(stage ^value labeling)
(junction ^type L ^base-point <p1>)
(edge ^p1 <p1> ^p2 <p2> ^label << + - >>)
(edge ^p1 <p1> ^p2 <> <p2> ^label nil)
-->
(modify 4 ^label B))


(p label-tee-A
(stage ^value labeling)
(junction ^type tee ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p1> ^label nil)
(edge ^p1 <bp> ^p2 <p3>)
-->
(modify 3 ^label B)
(modify 4 ^label B))


(p label-tee-B
(stage ^value labeling)
(junction ^type tee ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p1>)
(edge ^p1 <bp> ^p2 <p3> ^label nil)
-->
(modify 3 ^label B)
(modify 4 ^label B))


(p label-fork-1
(stage ^value labeling)
(junction ^type fork ^base-point <bp>)
```

```
(edge ^p1 <bp> ^p2 <p1> ^label +)
(edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label nil)
(edge ^p1 <bp> ^p2 {<> <p2> <> <p1>})
-->
(modify 4 ^label +)
(modify 5 ^label +))


(p label-fork-2
(stage ^value labeling)
(junction ^type fork ^base-point <bp>)
(edge ^p1 <bp> ^p2 <p1> ^label B)
(edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label -)
(edge ^p1 <bp> ^p2 {<> <p2> <> <p1>} ^label nil)
-->
(modify 5 ^label B))


(p label-fork-3
(stage ^value labeling)
(junction ^type fork ^base-point <bp>)
(edge ^p1 <bp> ^p2 <p1> ^label B)
(edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label B)
(edge ^p1 <bp> ^p2 {<> <p2> <> <p1>} ^label nil)
-->
(modify 5 ^label -))


(p label-fork-4
(stage ^value labeling)
(junction ^type fork ^base-point <bp>)
(edge ^p1 <bp> ^p2 <p1> ^label -)
(edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label -)
(edge ^p1 <bp> ^p2 {<> <p2> <> <p1>} ^label nil)
-->
(modify 5 ^label -))



(p label-arrow-1A
(stage ^value labeling)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p1> ^label {<label> << B - >>})
(edge ^p1 <bp> ^p2 <p2> ^label nil)
(edge ^p1 <bp> ^p2 <p3>)
-->
(modify 4 ^label +)
(modify 5 ^label <label>))


(p label-arrow-1B
(stage ^value labeling)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p1> ^label {<label> << B - >>})
```

```
(edge ^p1 <bp> ^p2 <p3> ^label nil)
(edge ^p1 <bp> ^p2 <p2>)
-->
(modify 4 ^label +)
(modify 5 ^label <label>))


(p label-arrow-2A
(stage ^value labeling)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p3> ^label {<label> << B - >>})
(edge ^p1 <bp> ^p2 <p2> ^label nil)
(edge ^p1 <bp> ^p2 <p1>)
-->
(modify 4 ^label +)
(modify 5 ^label <label>))

(p label-arrow-2B
(stage ^value labeling)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p3> ^label {<label> << B - >>})
(edge ^p1 <bp> ^p2 <p1> ^label nil)
(edge ^p1 <bp> ^p2 <p2>)
-->
(modify 4 ^label +)
(modify 5 ^label <label>))


(p label-arrow-3A
(stage ^value labeling)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p1> ^label +)
(edge ^p1 <bp> ^p2 <p2> ^label nil)
(edge ^p1 <bp> ^p2 <p3>)
-->
(modify 4 ^label -)
(modify 5 ^label +))

(p label-arrow-3B
(stage ^value labeling)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p1> ^label +)
(edge ^p1 <bp> ^p2 <p3> ^label nil)
(edge ^p1 <bp> ^p2 <p2>)
-->
(modify 4 ^label -)
(modify 5 ^label +))


(p label-arrow-4A
(stage ^value labeling)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p3> ^label +)
(edge ^p1 <bp> ^p2 <p2> ^label nil)
```

```
(edge ^p1 <bp> ^p2 <p1>)
-->
(modify 4 ^label -)
(modify 5 ^label +))

(p label-arrow-4B
(stage ^value labeling)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p3> ^label +)
(edge ^p1 <bp> ^p2 <p1> ^label nil)
(edge ^p1 <bp> ^p2 <p2>)
-->
(modify 4 ^label -)
(modify 5 ^label +))


(p label-arrow-5A
(stage ^value labeling)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p2> ^label -)
(edge ^p1 <bp> ^p2 <p3> ^label nil)
(edge ^p1 <bp> ^p2 <p1>)
-->
(modify 4 ^label +)
(modify 5 ^label +))


(p label-arrow-5B
(stage ^value labeling)
(junction ^type arrow ^base-point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
(edge ^p1 <bp> ^p2 <p2> ^label -)
(edge ^p1 <bp> ^p2 <p1> ^label nil)
(edge ^p1 <bp> ^p2 <p3>)
-->
(modify 4 ^label +)
(modify 5 ^label +))


;The conflict resolution mechanism will only execute a production if no
;productions that are more complicated are satisfied.  This production is
;simple, so all of the above dictionary productions will fire before this
;change of state production
(p done-labeling
(stage ^value labeling)
-->
(modify 1 ^value plot-remaining-edges))

;At this point, some labellings may have not been plotted, so plot them
(p plot-remaining
(stage ^value plot-remaining-edges)
(edge ^plotted nil ^label {<label> <> nil} ^p1 <p1> ^p2 <p2>)
-->
(write plot <label> <p1> <p2>)
(modify 2 ^plotted t))
```

```
;If we have been unable to label an edge, assume that it is a boundary.
;This is a total Kludge, but if we assume only valid drawings
;will be given for labeling, this assumption generally is true!

(p plot-boundaries
(stage ^value plot-remaining-edges)
(edge ^plotted nil ^label nil ^p1 <p1> ^p2 <p2>)
-->
(write plot B <p1> <p2>)
(modify 2 ^plotted t))

;If there is no more work to do, then we are done and flag it.
(p done-plotting
(stage ^value plot-remaining-edges)
- (edge ^plotted nil)
-->
(modify 1 ^value done))

;Prompt the user as to where he can see a trace of the OPS5
;execution
(p done
(stage ^value done)
-->
    (write "see trace.waltz for description of execution- hit CR to end")
(halt))
```

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 (stage initialize) --> (remove 1) (make stage make-data)

(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 1

(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              1
;                               12

(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 (stage make-data)
  -->

  (remove 1)
  (make stage enumerate-possible-candidates)

(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
(start)
-->
(remove 1)
(make stage initialize))

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

 (p enumerate-possible-candidates
(stage enumerate-possible-candidates)
   (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>)
(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>)
        )


;A meta-property is attached to the mode-changing rule so that
;it will not be executed in parallel with any other rule.

(setf (get 'go-to-reduce-candidates 'mode-changer) t)

 (p go-to-reduce-candidates
(stage enumerate-possible-candidates)
-->
(remove 1)
(make 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>))

(setf (get 'go-to-print-out 'mode-changer) t)

 (p go-to-print-out
(stage reduce-candidates)
-->
(remove 1)
(make stage print-out))

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

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