Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-89-47

1989-12-01

A Methodology for Developing Correct Rule-Based Programs for Parallel Implementation

Rosanne Fulcomer Gamble

Production systems, also called rule-based systems, are very useful in automating certain human expert tasks, but the current technology exhibits many problems. We believe that parallelism is difficult to exploit in production system programs for two reasons. First, the original serial programs are designed with a priori knowledge of an explicit global control mechanism which must be simulated for correct execution in parallel. The second reason for the difficulty is that no formal language exists in which to express these programs and no verification techniques are utilized to prove properties which guarantee correct execution in parallel. With these two... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
Gamble, Rosanne Fulcomer, "A Methodology for Developing Correct Rule-Based Programs for Parallel Implementation" Report Number: WUCS-89-47 (1989). *All Computer Science and Engineering Research.* https://openscholarship.wustl.edu/cse_research/757

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/757

A Methodology for Developing Correct Rule-Based Programs for Parallel Implementation

Rosanne Fulcomer Gamble

Complete Abstract:

Production systems, also called rule-based systems, are very useful in automating certain human expert tasks, but the current technology exhibits many problems. We believe that parallelism is difficult to exploit in production system programs for two reasons. First, the original serial programs are designed with a priori knowledge of an explicit global control mechanism which must be simulated for correct execution in parallel. The second reason for the difficulty is that no formal language exists in which to express these programs and no verification techniques are utilized to prove properties which guarantee correct execution in parallel. With these two obstacles removed, a correct rule-based program can be designed to exploit increased parallelism when mapped to a parallel production system model for execution. This research will concentrate on the development of parallel production system programs. The objective is to define a theoretical foundation to describe parallel production systems for implementation in parallel architectures. The Swarm language will be used as the vehicle for encoding the programs. Swarm's associated proof theory will be used, and possibly extended, to show correctness of properties necessary to guarantee correct parallel execution of the rule-based programs. Thus, the overall contribution of this research will be a methodology for defining, developing, and encoding correct parallel production system programs.

A METHODOLOGY FOR DEVELOPING CORRECT RULE-BASED PROGRAMS FOR PARALLEL IMPLEMENTATION

Rosanne Fulcomer Gamble

WUCS-89-47

Center for Intelligent Computer Systems
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

A proposal presented to the Department of Computer Science of Washington University in preparation for doctoral work.

Prepared under the direction of William E. Ball.

A Methodology for Developing Correct Rule-based Programs for Parallel Implementation

Rosanne Fulcomer Gamble

WUCS-89-47

Abstract

Production systems, also called rule-based systems, are very useful in automating certain human expert tasks, but the current technology exhibits many problems. We believe that parallelism is difficult to exploit in production system programs for two reasons. First, the original serial programs are designed with a priori knowledge of an explicit global control mechanism which must be simulated for correct execution in parallel. The second reason for the difficulty is that no formal language exists in which to express these programs and no verification techniques are utilized to prove properties which guarantee correct execution in parallel. With these two obstacles removed, a correct rule-based program can be designed to exploit increased parallelism when mapped to a parallel production system model for execution.

This research will concentrate on the development of parallel production system programs. The objective is to define a theoretical foundation to describe parallel production system programs for implementation in parallel architectures. The Swarm language will be used as the vehicle for encoding the programs. Swarm's associated proof theory will be used, and possibly extended, to show correctness of properties necessary to guarantee correct parallel execution of the rule-based programs. Thus, the overall contribution of this research will be a methodology for defining, developing, and encoding correct parallel production system programs.

1 Introduction

Production systems, also called rule-based systems, are very useful in automating certain human expert tasks, but the current technology exhibits many problems. The sequential systems are far too slow to be useful for any complex problem where resources such as time are limited. There is no formal specification language in which to write these programs easily. Also, no verification techniques are utilized to guarantee specific correctness criteria, such as program specifications and constraints.

To solve the speed problem, attempts at making production systems execute in parallel have been made. Significant speed up has not yet been achieved. Gupta[11] concludes the reasons for limited parallelism are that there are only a small number of affected productions per working memory change, there are large variations in processing times for the productions, and only a small number of working memory elements change per cycle. Since production systems spend as much as 90% of their execution time in the matching and processing of working memory elements, attempts have been made to increase the speed of this portion of the system execution [11, 21, 1, 17, 3]. If successful speed up of this area can be achieved, then the rest of the system should be executed in parallel as well. Research in this area has resulted in models [12, 19] which are overly synchronous because they use a global control mechanism to guarantee their parallel programs give results that can be produced by the serial systems. Because synchronization causes unnecessary bottlenecks in the parallel execution of a rule-based program, work to achieve more asynchronous systems is ongoing [14, 15, 20]. Both the synchronous and asynchronous systems lose global control, and once the system is executing in parallel, problems such as serializability, termination detection, and task ordering arise.

We believe that parallelism is difficult to exploit in production system programs for two reasons. First, the original serial programs are designed with a priori knowledge of an explicit global control mechanism which must be simulated for correct execution in parallel. Parallelism may be better exploited if the rule-based programs for parallel architectures are designed directly from the specifications. The second reason for the difficulty is that no formal language exists in which to express these programs. Therefore no verification techniques are utilized to prove properties which guarantee correct parallel execution. Designing a rule-based program for parallel execution demands such formal methods, due to the complexity of the execution, to ensure correct behavior. With these two obstacles removed, a correct parallel rule-based program can be designed to overcome the problems that arise when the whole system is executed in parallel, along with lessening the limitations stated above.

This research will concentrate on the development of parallel production system programs. The objective is to define a theoretical foundation to describe parallel production system programs for implementation in parallel architectures. The Swarm language [18] will be used as the vehicle for encoding the programs. Swarm's associated proof theory[6, 5] will be used, and possibly extended, to show correctness of properties necessary to guarantee correct parallel execution of the rule-based programs. Contributions made by this research will be the following developments: 1) methods for eliminating conflict resolution strategies from serial rule-based programs, 2) methods for formally encoding and proving rule-based programs, and 3) methods for enhancing the programs to exploit parallelism while maintaining correctness criteria. Thus, the overall contribution of this research will a methodology for defining, developing, and encoding correct parallel production system programs.

The paper is organized in the following manner. The next section details the problems associated with executing a production system program in parallel and eliminating global control for this type of execution. Section 3 presents general background on related systems which have taken different approaches to solving one or more of the problems presented in Section 2. Included in Section 3 is a brief description of a sequential production system and the use of global control. Section 4 contains a description of the Swarm language, its

associated proof theory, and the motivation for using Swarm as a tool for conducting the research. Section 5 presents the research proposal and contributions.

2 General Problems

Most production systems rely on some form of global control which determines when a rule can fire. To execute a production system in parallel and asynchronously, global control must be eliminated, simulated or encoded locally. In this section, we list several problems that must be addressed when developing parallel production system programs. These problems stem from the attempt to execute a rule-based program in parallel and from the elimination of global control.

Rule division among processors. In order to achieve parallel computation, the rules must be divided among the processors so that the processors can execute asynchronously. Ways to divide the rules include partitioning or copying certain rules for all or a limited number of processors. Distributing working memory does not pose a problem provided that a processor has access to all data needed.

Termination detection. Processors operating asynchronously with only working memory as their medium may not have a way to determine when a solution is reached or when no solution can be found.

Solution correctness. In sequential systems, a global control mechanism, such as conflict resolution, is responsible for directing the search for a solution. If the search space is too complex, more control, such as adding additional task ordering, is utilized to prune it. Using this control, the solutions reached are acceptable. Guaranteeing a solution path will converge is difficult without global control. It is also difficult to prove the solution reached is among the set of solutions deemed satisfactory.

Non-determinism. Once the order of rule execution cannot be controlled globally, the same solution will not be guaranteed to occur on two independent executions of the same program.

Serializable solution. Rules may interfere with each other by changing data upon which other rules initially relied. The interleaving of actions of rules fired in parallel may produce solutions that cannot be obtained in a serial execution of the rules.

Task-order dependence. Some tasks depend on the execution or non-execution of other tasks. With no global scheduler, such dependence must be encoded within the rules or working memory.

Multi-rule execution ordering. This problem is intertwined with serializability and task-order dependence. When multiple rules are fired asynchronously from different sources, restrictions must be placed on the order in which changes are made to working memory. Also, the order in which rules match working memory may be restricted. These restrictions are necessary for achieving a correct solution.

Consistency of working memory. When a rule-based program executes in parallel, it is possible for a rule to view a working memory element in the middle of another rule changing it. Avoidance of such inconsistency can be done by determining which actions are atomic.

3 Background

This section begins with a description of the general sequential production system and the important role played by conflict resolution. Descriptions of synchronous parallel production systems are given next. Also, the approaches these systems use to solve the problems of serializability and rule distribution are discussed. A previously developed asynchronous model is then presented along with an asynchronous model still being developed. Both the synchronous and asynchronous systems avoid the control problem by using only those rule-based programs that make no assumptions regarding conflict resolution. We discuss the general blackboard architecture, and its relation to this research, last.

3.1 Production Systems

A simple production system consists of production memory (PM), which is defined by a set of rules, and working memory (WM), which is a database of assertions. A production rule has the following form:

LHS $\longrightarrow$ **RHS**

where

LHS is made up of a conjunction of condition elements (CEs)
RHS is made up of action elements (AEs)

Every CE represents a pattern or template matched against working memory elements (WMEs). WMEs are data objects with class and attribute-value pairs. A positive CE is a CE satisfied when there exists a matching WME. A negative CE is a CE that succeeds when no matching WME can be found. Pattern variables are consistently bound throughout the positive CEs. Free variables may exist in negative CEs. Every AE is made up of working memory elements (WMEs) and specifies modification to WM. Positive AEs add to WM and negative AEs delete from WM.

Execution of a serial production system consists of repeatedly executing a **match-select-act** (Figure 1) cycle until a halt statement (a type of AE) is reached or the match phase returns the empty set. The match phase compares the LHS of all rules to working memory. A match for every CE in a rule constitutes an instantiation of that rule. A rule may have one or more instantiations. All instantiations from all rules are gathered at the end of the match phase to form a set. This non-empty set of instantiations, called the *conflict set*, is passed through the select phase. During this phase, a conflict resolution strategy determines a single instantiation or some subset of the conflict set which is passed to the act phase. The act phase performs the actions of RHS of the instantiations passed. The results of this phase may be modifications to WM, and/or calling subroutines, and/or modifications to PM. Once all actions are performed the cycle begins again with the match phase.

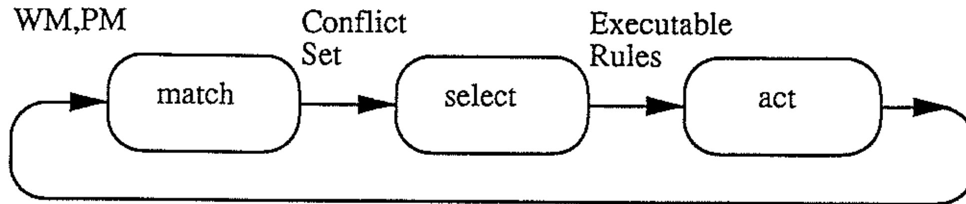

Figure 1: Simple Production System

3.1.1 Use of Conflict Resolution

Conflict resolution is one form of global control. Production systems use conflict resolution strategies to control the search for a solution, and to achieve an efficient execution of a rule-based program. Most production systems have a conflict resolution strategy built in. Thus, programs encoded within a system make implicit use of its strategy. In general a conflict resolution strategy is made up of a combination of conflict resolution rules. The strategies may come in different forms to accommodate any particular rule-based system design. McDermott and Forgy[13] detail thirteen of these rules, grouping them into five categories: 1) **production order**–based on a relationship defined among the rules, 2) **special case**–based on subsumption and redundancy of rules, 3) **recency**–based on the time of assertion for WMEs, 4) **distinctiveness**–based on history of rules fired, 5) **arbitrary decision**–causes only one rule to be chosen randomly for execution. Some programmers use variations on these thirteen rules or detail another type of conflict resolution rule for a particular type of rule-based program.

McDermott and Forgy[13] believe the function of conflict resolution is to provide a mechanism that can preserve *sensitivity* and *stability* in a production system, without sacrificing autonomy. Sensitivity refers to the degree of responsiveness in the system. Stability refers to the degree to which the system maintains continuity in its behavior. They state the useful characteristics in all production systems as follows.

1. Both sensitivity and stability should not be ignored.
2. The system must be sensitive to its own processing.
3. The systems should sort relevant information from non-relevant information when conflict occurs in working memory.
4. The system should make a decision about execution when

multiple rules can be fired.
5. All actions taken should be observably deterministic.

For Fickas, et al [9], the important characteristics of a production system with respect to conflict resolution are:

1. *Flexibility:* the system must be flexible enough to permit different kinds of strategies to be matched to different domains.
2. *Representation:* control knowledge should be represented separate from domain knowledge.
3. *Dynamic Control:* if a system deduces control knowledge while executing, it should make use of this knowledge.

ORBS [9, 8, 10] is a production system which allows the user to choose the desired conflict resolution strategy and create additional rules on a program by program basis. The motivation for this allowance is strategy flexibility. The conflict resolution strategy should be flexible for the system to respond in a reasonable fashion to frequent, sometimes competing and unexpected, demands from its environment. Such responsiveness comes from limiting the number of productions fired on each cycle. Without conflict resolution, the easiest way to enforce this limitation is to make the rules applicable in mutually exclusive situations, requiring that productions be given knowledge of each other's domains of applicability. This knowledge will limit the system's flexibility because new productions cannot easily be added.

Though ORBS displays these characteristics, the programmer is burdened with the responsibility of determining how to match the strategies to the programs, when a change in strategy for a program is necessary, and how to perform that change.

3.2 Parallel Production Systems

This section describes the synchronous parallel production system models from Ishida and Stolfo [12] and Schmolze [19], and the asynchronous parallel production system models from Schmolze [20] and Miranker [14, 15]. A brief description of each model is given, followed by the approaches each system takes to solve the problems of serializability and rule distribution.

3.2.1 Ishida and Stolfo's Model

Ishida and Stolfo[12] propose a parallel firing mechanism for production system execution. The aim of this mechanism is to reduce the total number of sequential production rule cycles by executing multiple matching rules simultaneously on multiple processor systems.

The two major problems addressed are the *synchronization problem* (or **serialization** problem), and the *decomposition problem* (or **distribution of rules** problem). The synchronization problem occurs when production rules are written without considering rule interference. The definitions for this interference are defined under the **serializability**

heading below. To achieve a solution, it is necessary to determine all the rules in production memory that interfere with the rule in question. These rules are then *synchronized* with the given rule, i.e., they are not to executed simultaneously with that rule. The decomposition problem occurs when a partition or distribution of a given production system must be found for multiple rules to be fired as often as possible.

In the model developed by Ishida and Stolfo, the production rules are distributed between a control processor (CP) and all processing elements (PEs). Only production system programs written without any assumptions of particular selection algorithms for conflict resolution are used, though the system does not have to be a commutative production system[1]. If the result of a parallel execution of two rules cannot be achieved in a sequential execution of those rules, then interference occurs and the two rules must be synchronized. A data dependency graph, explained in more detail below, is used to determine interference. The execution cycle, pictured in Figure 2, is described as follows:

1. **match:** Each PE executes this phase simultaneously and reports its instantiations to the CP.
2. **select:** The CP chooses at most one of the reported rules from the match phase for each PE, so that interference between PEs does not occur.
3. **act:** Each PE executes this phase simultaneously. If WM is distributed among the PEs, changes are reported to the CP and broadcasted to the PEs. Rules are only fired by the CP if no PE can execute a rule.

Synchronization occurs at each step. A processor cannot continue in the cycle until all processors finish each step.

Serializability

A data dependency graph of production rules is used for the analysis of interference between rules. Within this graph, a P-node represents a production rule and a W-node represents a class of WMEs. A directed edge from a P-node to a W-node means the RHS of the P-node modifies the W-node. If an element of the W-node is added, then the W-node is *+changed*. A *−changed* W-node means an element of the W-node is deleted. The directed edge is labeled with a + or − to indicate the action. A directed edge from the W-node to the P-node, means the LHS of the P-node refers to the W-node. If the class is positively referenced the W-node is called *+referenced*, and similarly for negative references. The corresponding edge will be labeled with a + or −.

Analysis of interference of two distinct rule firings is based on the data dependency graph. Given rules A and B, in order to prevent a non-serializable execution of A and B, synchronization must occur between them if there exists a W-node which satisfies at least one of the following:

[1] A commutative production system is one in which, given a rule set R of some program P, then $\forall$A,B where A,B $\in$ R and A,B appear in the conflict set together, the same result occurs when A executes before B, as it does when B executes before A.

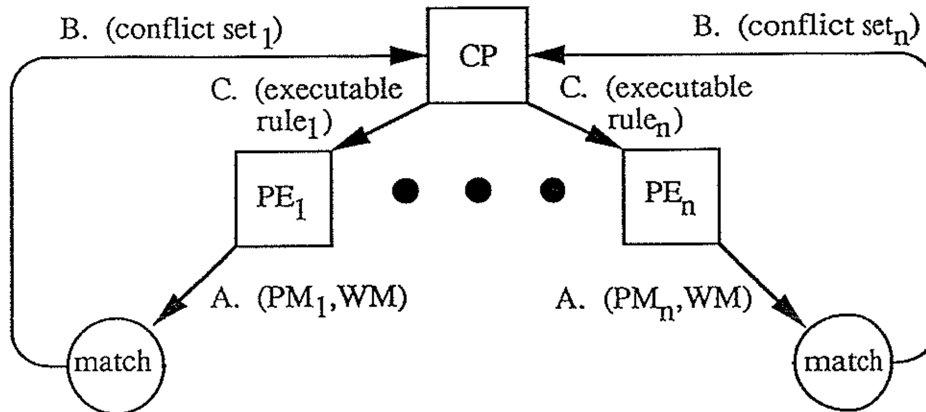

Figure 2: Ishida and Stolfo's Synchronous Parallel Production System

1. $+changed$ ($-changed$) by A and $-referenced$ ($+referenced$) by B.
2. $+changed$ ($-changed$) by B and $-referenced$ ($+referenced$) by A.
3. $+changed$ ($-changed$) by A and $-changed$ ($+changed$) by B.

From this analysis, a *synchronization set* is produced for each rule. This set contains all rules which must be synchronized with the rule, or all rules that interfere with the rule. Because the analysis is static more synchronization is reported than is necessary in execution.

Rule Distribution

Operations, such as I/O, are allocated to the CP, and the remaining system is decomposed among the PEs. The goal is to achieve a decomposition in which multiple rules may be concurrently fired as often as possible, reducing the number of production cycles. The two major difficulties in devising such an algorithm are that the number of possible decompositions is too large to take an exhaustive approach, and the optimal decomposition may change as the usage of the production system changes.

The decomposition is based on the *parallel executability* of two rules. Parallel executability refers to the number of cycles which can be reduced by allocating the two rules to different processors. Rules are allocated to the processors in order of their parallel executability, with the highest two rules allocated first. Parallel executability is determined by sample execution traces of the program. The *hierarchical decomposition* algorithm first produces a hierarchical data structure called a *rule tree*, and then creates partitions for the rule set from the rule tree.

For evaluating the effectiveness of the parallel execution of production systems, Ishida and Stolfo developed a simulation environment consisting of an *Analyzer*, a *Simulator* and a *Decomposer* subsystem. OPS5 production systems are input to the Analyzer which produces the synchronization sets from the data dependency graph constructed. The Simulator simulates any partition of the rule set of the production system and creates the sample execution traces. The synchronization sets produced by the Analyzer are referenced by the Simulator. The Decomposer examines the execution traces of the Simulator and measures the parallel executability between each pair of rules, then partitions the rule set using the rule tree produced.

3.2.2 Schmolze's Synchronous Model

This model[19] was developed to provide a solution to the serializability problem which improves upon the one given by Ishida and Stolfo above. It is targeted for a set of processors with shared memory. The execution cycle, pictured in Figure 3, is defined as follows:

1. **match:** All processors perform a parallel match, producing a set of instantiations.
2. **select:** All processors jointly select the instantiations to be executed and place them on a shared instantiation queue.
3. **post:** Processors examine each instantiation on the queue, form the actions for the instantiation and post the actions to a shared action queue.
4. **act:** Processors perform all posted actions.

Synchronization is performed at each step. The shared queues only provide sequential access to each instantiation and action. There is no restriction as to the number of actions a processor can execute, i.e., there is no longer a restriction of one instantiation per processor.

Serializability

Two causes of non-serializability are identified as *LHS disabling* and *RHS clashing.* LHS disabling occurs when the simultaneous execution of two rules causes one of the rules to become irrelevant and no longer match. The select step above will be responsible for disallowing the co-execution of any set of rules with a cycle of disabling relations. RHS clashing occurs when the actions of two simultaneously execution rules overlap, such that one rule adds a working element that the other deletes. If two instantiations clash and one rule can disable the other, then non-serializable results may be obtained. The co-execution of such rules can be prohibited in either the select step or by imposing a partial order on the execution of actions in the act step. The paper gives three complex methods for determining what rules can be co-executed. Using these methods, more rules are executed simultaneously than using the method given by Ishida and Stolfo[12].

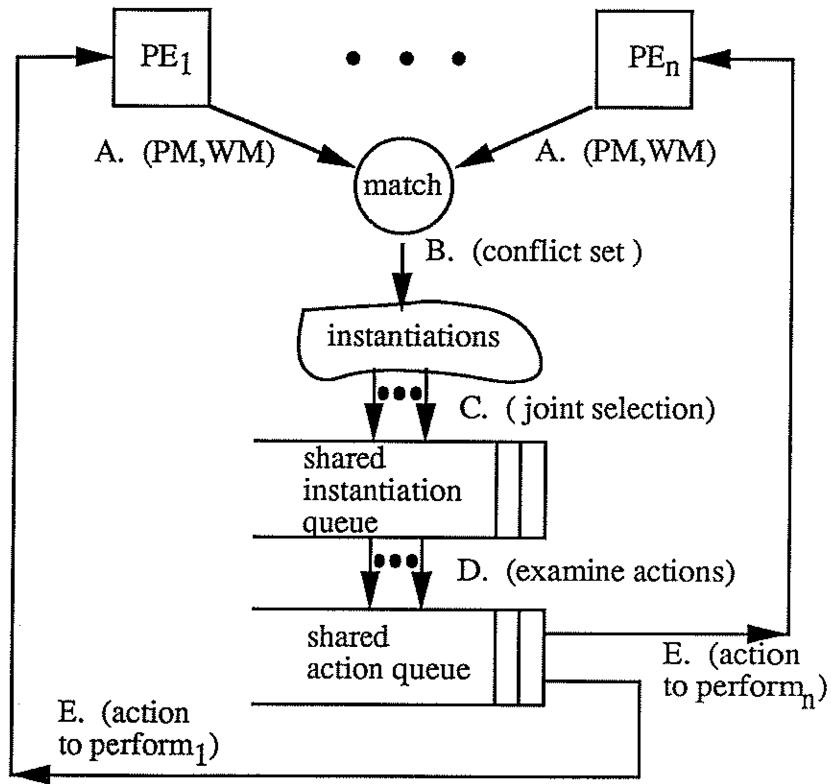

Figure 3: Schmolze's Synchronous Parallel Production System

3.2.3 Schmolze's Asynchronous Model

Schmolze[20] describes PARS (Parallel Asynchronous Rule System), a system in which each processor in a multiple processor system is a separate production system with local production and working memories. This model is different than the previous two presented because there is almost no synchronization performed, thus eliminating the time wasted in synchronous systems where some processors are forced to wait for other processors to finish the current step. Their goal is to improve upon the performance of the Ishida and Stolfo model previously described through asynchronous computation.

The set of rules are partitioned over the processors. Working memory elements may be duplicated among the processors, so the every processor has all the necessary classes of elements that the rules need to match. Processors also have knowledge of where duplicate working memory elements are located and where interfering rules are located on other processors.

Each processor executes the following execution loop at its own pace, asynchronously. Figure 3 depicts the cycle of one processor.

1. **match:** Performed using local production and working memories.
2. **select:** Also performed locally. Instantiations of temporarily disabled rules are ignored. Only arbitrary selection is used. The processor returns to the match phase if the conflict set is empty. It does not terminate.
3. **disable:** Requests to disable interfering instantiations are sent out. The processor blocks until all acknowledgements are received.
4. **act:** Actions of the firing instantiations are sent to appropriate processors, and the processor waits for receipt of all acknowledgements. Actions in local working memory are taken immediately.
5. **enable:** Request are sent to enable previously disabled rules and the processor returns to the match step.

Possible synchronization occurs through the communication protocols in the disable and act step. Processors not involved in communicating messages may continue to execute asynchronously. Also, after a processor completes a step, it does not have to wait for the other processors to complete that step before continuing to the next step in the cycle.

Processors accept messages from other processors before entering the match phase and while waiting for message acknowledgements in the disable and act phases. A problem with message passing may occur in the disable phase if a processor receives actions invalidating the chosen instantiation while waiting for its acknowledgements. If the instantiation is invalid, after all acknowledgements are received, the processor skips to the enable phase.

As seen from the execution cycle, termination does not occur as in the earlier models because a processor may receive work even though its conflict set from a match phase is empty. Thus, in PARS, termination is controlled by a designated processor that determines, through message passing, when all processors are idle and broadcasts termination

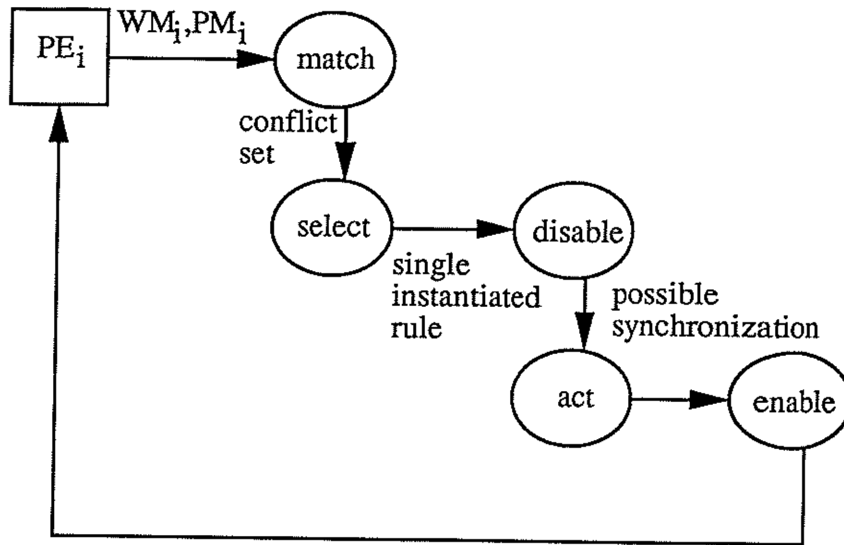

Figure 4: Execution of a single processor in PARS

commands.

Because working memory elements are duplicated, a processor may receive a change while it is matching, causing working memory inconsistencies. A remedy for this problem is a double acknowledgement protocol used when sending actions. The original receipt of actions from another processor indicates that certain WMEs will be added or removed. But the receiver prevents the use of this information until it is sure that all appropriate processors have received it, to prevent inconsistency.

The reasoning behind the development of an asynchronous model, is that by reducing the amount of synchronization, there is more potential for parallelism. The estimated results achieved by PARS substantiate this claim. Through simulation of their algorithms, Schmolze and Goel improve upon Ishida and Stolfo by having each rule synchronize with less than half (on average) the number of rules that Ishida and Stolfo require and by co-executing about twice as many instantiations as that of Ishida and Stolfo.

Serializability

To guarantee serializability, concern needs to be placed only on sets of co-executing instantiations. In this model, instantiations are said to co-execute if and only if their *period of execution* overlaps. This period for an instantiation is the time of commitment to execute an instantiation (made in the disable step) to the later of the following two times: 1) the time that instantiation's local actions are taken or 2) the time that all acknowledgement have been received from the act step for that instantiation.

To analyze the interference between instantiations, a directed graph *IDO(I)* is used, where I is the set of instantiations. This graph represents *instantiation disabling order*, such that each $i \in I$ is a node in *IDO(I)*. If i and j are in I and j disables i, there is an

arc from i to j, making i execute before j. The co-execution of a set of instantiations, I, in PARS, is serializable if $IDO(I)$ is acyclic and no two distinct instantiations in I clash.

A guarantee is made such that a set of instantiations that co-execute meet the above conditions of serializability by: 1) performing an off-line analysis of the rules, 2) determining pairs of rules whose instantiations should not co-execute, and 3) enforcing rule synchronization from the determination in 2).

Additional steps are taken which reduce the work of enforcing synchronizations between processors and preventing deadlock. If a pair of rules are known to disable each other only one rule of a pair is required, instead of both, to initiate disabling. Also, by ensuring that the graph for this requirement is undirected and acyclic, deadlock is prevented.

Rule Distribution.

To partition the rules among the processors, several approaches are combined including the parallel executability[12] measure. The final result is a distribution that attempts to maximize rule co-execution, which increases the number of rules that execute simultaneously, and minimizes communication costs, which reduces the length of each processor's basic cycle time.

3.2.4 Miranker, Kuo, and Browne's Model

The implementation of a recent parallel production system with a production rule language called CREL [14, 15] is underway. This system will execute asynchronously and in parallel on a shared memory multi-processor system. CREL (Concurrent Rule Execution Language) has a syntax identical to OPS5, and a semantics derived from a combination of the theories of serializability from databases and the UNITY language[4]. CREL programs that run correctly in a sequential environment are guaranteed to run correctly in a parallel environment. A guarantee must be made by the program designer that the result of the parallel execution of the system is serializable and that all serial executions reach a correct fixed point. Though the CREL authors hoped to have a rule execute as soon as it is able to fire, the semantic constraints due to negation disallow this approach. They believe one problem of executing a production system in parallel is that only a small amount of parallelism can be exploited when dynamic methods of determining which tasks can be performed simultaneously are utilized. Their approach is to develop static ways for this determination, which are applied at compile time.

Serializability

Rules that interfere with each other are termed *mutually exclusive*. Compiler algorithms determine independent sets of rules, called *clusters*, such that a rule in one cluster is not mutually exclusive with any rule in any other cluster. By definition, clusters can operate asynchronously with respect to each other. The rules within each cluster are mutually exclusive and form a synchronization set, according to Ishida and Stolfo's definition, since a rule in one cluster cannot interfere with any rule in any other cluster. Using the

dependency graph originated by Ishida and Stolfo as described above, they found that direct application of the mutual exclusion analysis did not produce a substantial number of clusters. Therefore, they developed a set of optimizing transformations which break unnecessary mutual exclusion dependencies, increasing the potential parallelism. For example, static analysis may predict unnecessary dependencies that, due to the state of current working memory, do not exist. Such transformations include the identification of the control variable. This control variable breaks dependencies in the graph between rules which match different control variable values. The utilization of additional constants in the condition element patterns to create finer divisions of working memory within the dependency graph to reduce its connectivity.

To correctly execute multiple rule firings, they compute the mutual exclusion sets from the dependency graph, followed by repeating the execution cycle (match-select-act) where selection from the conflict set of multiple rules from the same mutual exclusion set should not form a cycle with conflicting interferences. They prove that a correct CREL program is guaranteed to reach a correct terminal state under the execution scheme described above.

Rule Distribution

Clusters can be distributed among processors since no global synchronization is necessary for rules across clusters. But, among rules interfering with each other, synchronization is still required to insure correctness of the parallel execution. Thus if interfering rules are present on different processors, these processors must synchronize. They say that these processors can synchronize through message passing. Local synchronization of the rules must occur at each processor.

3.3 Blackboard Architectures

The blackboard architecture is an informal extension of a production system. By informal we mean, in the abstract blackboard system, the production system operation has been extended to include additional features such as separate inferencing systems operating in parallel on a shared working memory. Actual implementations of blackboards differ significantly. Some of these differences are the result of attempts to solve the problems of inferring and performing related activities in parallel. Thus, the results achieved by blackboard architectures may provide insight and information for solving some of the problems posed by parallel production systems.

In the general blackboard architecture, knowledge is segmented into modules called knowledge sources (KSs). Each KS has its own set of rules (or procedures) and a local inference engine which can be different from the other KSs. Communication between modules is limited to the reading and writing of a shared memory data structure called a blackboard. The blackboard contains the global state data. The KSs produce changes to the blackboard which lead incrementally to a solution to the problem. Data on the blackboard are objects organized hierarchically into levels of analysis. The KSs transform data from one level to the next, until a solution is found. As the blackboard changes, different KSs are automatically activated to work on it. Included in the blackboard model are control modules that monitor the changes on the blackboard and determine which KSs can use the blackboard. These control modules can be in the form of a KS or can be located separately on the blackboard.

Each KS can be thought of as a separate production system sharing working memory with other KSs. Some KSs are dependent on others, causing some type of scheduling to occur between KSs so that there is no memory contention and no interference. The problems that arise in the implementation of a blackboard system are very similar to those presented in section 2. Thus the solutions given by researchers of different blackboard systems will contribute to our research.

The development of the CAGE[16] blackboard system, a concurrent problem-solving system, caused questions to arise that are similar to those our research is attempting to answer.

1. Are problem-solving systems that rely heavily on centralized control doomed to fail in a concurrent environment?
2. Can control be distributed? If so, to what extent?
3. If more knowledge results in less search, can a similar tradeoff be made between knowledge and control?
4. In concurrent systems where control, especially global control, is a serializable process, can knowledge be brought to bear to alleviate the need for control.

The types of parallelism to be expressed in CAGE are 1) knowledge parallelism, in which the knowledge sources and rules within each KS can run concurrently; 2) pipelining parallelism, the transfer of information from one level of the blackboard to another, allowing multiple KSs to operate at different levels; and 3) data parallelism, in which the blackboard can be partitioned into solution components for concurrent operation. The following assumptions are utilized: a) all of the agents can see all of the blackboard all of the time and what they see represents the current state of the solution; b) any agent can write its conclusions on the blackboard at any time, without getting in anyone else's way; and c) the act of an agent writing on the blackboard will not confuse any of the other agents as they work. In CAGE, centralized control mechanisms are provided for selecting as many events as possible to be processed in parallel and causing many KSs to run. The application programmer assumes much of the burden for developing this centralized control, ensuring the correctness of the control, and ensuring it is the minimal control necessary.

Global control is used to determine the *focus of attention* (FOA), defined to be the next KS to use the blackboard, the next object to be processed, or some combination of an object to be processed by a certain KS. In Crysalis[22], the goal is to address the FOA problem. An efficient sequence of rules must be chosen in response to a specific problem. If some rules in the conflict set of a KS are not fired when they match, some solutions may be lost. Global knowledge specifies the order of the tasks to be performed, and the choices among competing alternatives. Thus, the control is very explicit and restrictive to ensure proper execution of the system. A solution to the FOA problem in Crysalis uses control rules to directly state which KS to run next, instead of picking a set of KSs as possibly useful and then ruling most of them out during global conflict resolution.

In Crysalis [22], a *hierarchical production system* was designed to incorporate explicit strategies of control within the blackboard system. In a hierarchical production system, control proceeds through many levels of strategy heuristics until one specific action at

the problem-solving or bottom-most level is selected. During execution, inferences can be made from any level to any other level. Each level is a complete production system containing rules that examine the current state and choose actions at the next lower level. The highest level consists of a single control KS that examines the current state of the panels to select a KS or sequence of KSs to invoke at the next level. The process repeats with the execution of rules in a KS at one level causing the KSs at the next level to be invoked until the bottom level is encountered where object rules actually construct a solution. When a lower level KS terminates, control returns to the KS at the previous level which invoked the terminating KS.

A transactional blackboard [7] aims to maintain the integrity of the shared data which is accessed asynchronously. A transaction manager is associated with the database. Any reference to the blackboard must be part of a transaction. A locking protocol is used with commit-abort actions. Data consistency is maintained using time-stamps.

In the transactional blackboard, two mechanisms are provided for safe access to blackboard data. First, knowledge sources can communicate by accessing shared data in separate transactions. Also, several KSs can participate in a common transaction if they need to see a common consistent view of shared data. If the system state matches the general preconditions of a KS, that agent takes the specified actions. These actions include requesting that the controller schedule a KS activity by placing an entry on the controller goal queue, performing some operations on the blackboard, and/or sending a message to another KS.

A transaction manager associated with the blackboard receives requests from agents, executes on their behalf and packages the responses. Two and three-phase commit protocols are used. Atomicity ensures that at the end of a transaction, all write actions associated with the transaction have taken place or all the data referenced by the transaction is restored to the state that existed when the transaction began.

POLIGON[16] is a demon-driven blackboard system, in which global control, as it is used in the previously described systems, is eliminated. In the POLIGON system, KSs are attached to data elements in the blackboard. Changes to the data elements result in the activation of all KSs attached. The goal for developing POLIGON is the elimination of all global synchronization, so that a KS is immediately executed as relevant data are changed. Within this framework, there is no need from any separation of the KSs for the blackboard. The programmer must specify the classes of blackboard nodes in which a particular piece of knowledge is interested. Overall communication delay and memory contention is eliminated because the interpreter does not have to examine the entire knowledge base to decide on a subset of rules, i.e., there is no global conflict resolution. Since there are no global variables, POLIGON provides a mechanism for defining sharable, mutable data, and with minimal bottlenecks.

In order to solve the problems caused by eliminating control, POLIGON uses control at various levels within the system structure. For example, control may be placed at each node where actions of rules must be executed serially. Other types of explicit and restrictive node control are also used such that control is global with respect to the data, not the rules.

4 Parallel Languages

Because we are interested in developing parallel rule-based programs, we need a language in which to encode these programs. Concentration will be on program development, thus we need a language that is independent of hardware. Also, the language must have mechanisms to express concurrent operations. In this section, the languages of Linda[2] and UNITY[4] are briefly discussed, followed by a description of the Swarm language and why we have chosen it for this research.

4.1 Linda

The goal of the Linda programming language[2] is to make it largely unnecessary to think about the coupling between parallel processes. Linda's uncoupled processes never deal with each other directly. A parallel program in Linda is a spatially and temporally unordered bag of processes. When Linda operators are injected into a host language, the host language turns into a parallel programming language. In the Linda framework we can get parallelism by replicating as well as partitioning tasks to multiple processes.

Linda's unit of memory is the logical tuple, or ordered set of values. Elements in Linda memory have no addresses. They are accessed by logical name, where a tuple's name is any selection of its values. Linda's shared memory is referred to as the tuple space. A process with data to communicate adds it to the tuple space and a process that needs to receive data seeks it in the tuple space. To be changed, tuples must be physically removed, updated and reinserted. This makes it possible for many processes to share access to a Linda memory simultaneously. Messages in Linda are never exchanged between two processes directly.

There are three basic operations on tuples: out(), in(), and read(). The operation out(t) causes tuple t to be added to the tuple space, and the executing process continues immediately. The operation in(s) causes some tuple t, that matches template s, to be withdrawn from the tuple space The process that executes an in(s) suspends until a match is found. Matches are chosen arbitrarily. The operation read(s) is the same as in(s) except the tuple remains in the tuple space when a read(s) occurs.

Dynamic scheduling of tasks is important in Linda, since tasks may require varying amounts of time to complete. Thus, new tasks may be developed dynamically as old ones are processed, using a distributed data structure called a task bag. Workers repeatedly draw their next assignment from the task bag, carry out the specified assignment, and drop any new tasks generated by the process back into the task bag. The program completes when the bag is empty. If it were necessary to process tasks in a particular order a task queue would be built instead of a task bag.

Linda has no proof theory and cannot stand alone. It must be used as a tool within a host language.

4.2 UNITY

For Chandy and Misra[4], the first concern of program development is to design a solution to the problem. The second concern, separate from the first, is to implement the

solution in a given language on a given hardware. Their goal for developing UNITY is to build a computation model and proof system which both is simple and has the expressive power necessary to permit the refinement of specifications and programs to suit target architectures. Central issues for this development were: nondeterminism, absence of control flow, proof systems that support program development by stepwise refinement of specifications and the decoupling of correctness from complexity. They wanted to incorporate synchrony and asynchrony in a unified theory of parallel programming based on state-transition systems.

A UNITY program consists of declaration of variables, a specification of their initial values and a set of multiple-assignment statements. There is no sequential flow of control. An assignment statement is chosen nondeterministically. The choice is fair in that every statement is executed infinitely often. Execution continues indefinitely, though each assignment statement is guaranteed to terminate.

The proof system allows one to develop and study a proof in its own right. Chandy and Misra state that correctness of a program is independent of the target architecture and the manner in which the program is executed; by contrast, the efficiency of a program execution depends on the architecture and manner of execution.

For example, UNITY and OPS5 are different systems of programming. OPS5 was developed to facilitate knowledge representation and inference, while UNITY was developed for writing programs without concern for specific languages. In UNITY, the set of statements is static, and therefore UNITY cannot accommodate self-modifying programs. Since there is no control flow in a UNITY program, it cannot simulate the global selection strategy. Also in UNITY, variables are accessed by name, not content as in Linda.

Throughout the following description of Swarm, more comparisons with UNITY will be made.

4.3 Swarm

Swarm is a *shared dataspace* language [18]. The term shared dataspace refers to the general class of programming languages in which the principal means of communication among the concurrent components of programs is a common, content-addressable data structure called a **dataspace**. Linda was first to use this concept as its tuple space. The state of a Swarm program consists of a dataspace, i.e., a set of transaction statements and data tuples. Tuples in Swarm are similar to tuples in Linda. Executing a Swarm transaction is similar to a process taking a task from the Linda task bag.

Underlying the Swarm language is a state-transition model similar to that of UNITY[4], but recast into the shared dataspace framework. In the model, the state of a computation is represented by the contents of the dataspace, a set of content-addressable entities. The model partitions the dataspace into three subsets: the *tuple space*, a finite set of data *tuples*; the *transaction space*, a finite set of *transactions*; the *synchronic group*, discussed later. An element of the dataspace is a pairing of a type name with a sequence of values.

Swarm is very similar to UNITY. For this research, Swarm is the language of choice because the constructs that appear in Swarm, such as transactions and tuples, more closely relate to rule-based programs than the constructs of assignment statements and named

variables in UNITY.

A program in Swarm begins execution with a specified initial dataspace. On each execution step, a transaction is chosen nondeterministically from the transaction space and executed atomically. This selection is fair in the sense that every transaction in the transaction space at any point in the computation will eventually be chosen. An executing transaction examines the dataspace and then, depending upon the results of the examination, can delete tuples (but not transactions) from the dataspace and insert new tuples and transactions into the dataspace. Unless a transaction explicitly reinserts itself into the dataspace, it is deleted as a by-product of its execution. Program execution continues until there are no transactions in the dataspace.

Within the program description there exists sections in which to declare all (and only those) tuples and transactions that are (and can be) used in the program. Each tuple type declaration defines a tuple schema whose instances can be examined, inserted, and deleted by the program. The transaction types section declares the types of transactions that can exist in the transaction space. Each transaction type declaration defines a set of transaction instances that can be executed, inserted, or examined by a program.

The body of a transaction instance consists of a sequence of subtransactions connected by the || operator. Each subtransaction definition consists of a variable list, a query, and an action. A subtransaction's action specifies sets of tuples to insert and delete and transactions to insert. A subtransaction is executed in three phases: query evaluation, tuple deletions, and tuple and transaction insertions. Evaluation of the query seeks to find values for the subtransaction variables that make the query predicate **true** with respect to the dataspace. If the query evaluation succeeds, then the dataspace deletions and insertions specified by the subtransaction's action are performed using the values bound by the query. If the query fails, then the action is not executed. The subtransactions of a transaction are executed simultaneously: all queries are evaluated first, then the indicated tuples and the transaction itself are deleted, and finally the indicated tuples and transactions are inserted. By default, both the tuple and transaction spaces are empty. The initialization section establishes the dataspace contents that exist at the beginning of a computation.

The Swarm logic is defined in terms of the same logical relations as UNITY: (**unless**, **ensures**, and **leads-to**). Some of the concepts of these relations have been reformulated. A proof rule for transaction statements is defined which replaces UNITY's rule for multiple-assignment statements, the **ensures** relation is redefined to accommodate the creation and deletion of transaction statements, and UNITY's use of fixed-point predicates has been replaced with other methods for determining program termination. Because of this carefully constructed logic, most of the theorems developed for UNITY can be directly adapted to the Swarm logic.

With the close relation between the Swarm logic and UNITY logic, anything that we can do in Swarm can be done in UNITY, since we will restrict our programs to those that are not self-modifying. However the task in UNITY would be far more difficult. Thus, by using Swarm, we have the ease of using a language closer to a rule-based language, the concept of designing a program without regard to a specific hardware, and the use of a proof theory for developing correct programs.

The programming logic constructed in Swarm also utilizes a synchronic group, a unique

feature. Synchronic groups are dynamically constructed groups of transactions which are executed synchronously as if they were a single atomic transaction. The synchronic groups should not be confused with the synchronization sets discussed earlier. In section 3.2, a synchronization set is produced for each rule which contains all rules which must be synchronized with the rule because of possible interference with that rule. All the transactions in a synchronic group are executed simultaneously, and their actions are synchronized, i.e., all deletions before all actions. The synchrony relation defines synchronic groups of transactions. This relation can be stated initially. It can also be added or deleted dynamically as transactions execute.

4.3.1 The Swarm Proof Theory

In this section, we will briefly present the Swarm proof theory[18].

The proof system for Swarm is similar in style to that of UNITY. The UNITY logic seeks to derive the proof from the program text by relying upon proof of program-wide properties, e.g., global invariants and progress properties. With the Swarm logic, these ideas are extended into the shared dataspace framework. Swarm's underlying computational model is similar to that of UNITY, but has a few key differences. A UNITY program consists of a static set of deterministic multiple-assignment statements acting upon a shared-variable state space. The statements in the set are executed repeatedly in a nondeterministic, but fair, order. Swarm is based on less familiar programming language primitives—nondeterministic transaction statements which act upon a dataspace of anonymous tuples—and extends the UNITY-like model to a dynamically varying set of statements. The programming logics are identical, with the exception of the multiple assignment statement, the **ensures** relation, and the fixed point predicates as stated above.

Notational conventions are also similar to UNITY. They use Hoare-style assertions of the form $\{p\}\ t\ \{q\}$ where p and q are predicates and t is a transaction instance. UNITY assignment statements are deterministic; execution of a statement from a given state will always result in the same next state. This determinism, plus the use of named variables, enables UNITY's assignment proof rule to be stated in terms of the syntactic substitution of the source expression for the target variable name in the postcondition predicate. In contrast, Swarm transaction statements are nondeterministic; execution of a statement from a given dataspace may result in any one of potentially many next states. This arises from the nature of the transaction's queries. A query may have many possible solutions with respect to a given dataspace. The execution mechanism chooses any one of these solutions nondeterministically—fairness in this choice is *not* assumed. Since the state of a Swarm computation is represented by a set of tuples rather than a mapping of values to variables, finding a useful syntactic rule is difficult.

As in UNITY's logic, the basic safety properties of a program are defined in terms of **unless** relations. Defining **TRS** to be the set of all possible transactions, the Swarm definition mirrors the UNITY definition:

$$p \textbf{ unless } q \equiv [\forall t :\ t \in \textbf{TRS} : \{p \wedge \neg q\}\ t\ \{p \vee q\}]$$

The **unless** relation is defined informally as follows. If p is *true* at some point in the

computation and q is not, then, after the next step, p remains *true* or q becomes *true*. In other words, either $p \wedge \neg q$ continues to hold indefinitely or q holds eventually and p continues to hold at least until q holds.

Stable and **invariant** properties are fundamental notions of the proof theory. Both can be defined easily.

$$\textbf{stable } p \equiv p \textbf{ unless false}$$
$$\textbf{invariant } p \equiv (INIT \Rightarrow p) \wedge (\textbf{ stable } p\ )$$

INIT is a predicate which characterizes the valid initial states of the program. A stable predicate remains *true* once it becomes *true*, although it may never become *true*. Invariants are stable predicates which are *true* initially.

We use the **ensures** relation to state the most basic progress (liveness) properties of programs. UNITY programs consist of a static set of statements. In contrast, Swarm programs consist of a dynamically varying set of transactions. The dynamism of the Swarm transaction space requires a reformulation of the **ensures** relation. Without considering the synchrony relation, the **ensures** relation is defined informally as follows. If p is *true* at some point in the computation, then (1) p will remain *true* as long as q is *false*; and (2) if q is *false*, there is at least one transaction in the transaction space which can, when executed, establish q as *true*. The second part of this definition guarantees q will eventually become *true*.

Let $[t]$ mean that the transaction t is present in the transaction space. Formally **ensures** is defined as:

$$\begin{aligned} & p \textbf{ ensures } q \equiv \\ & \qquad (p \textbf{ unless } q) \wedge \\ & \qquad (\exists t : t \in \textbf{TRS} : (p \wedge \neg q \Rightarrow [t]) \wedge \\ & \qquad\qquad\qquad\qquad \{p \wedge \neg q\}\ t\ \{q\}\ ) \end{aligned}$$

The only way a transaction is removed from the dataspace is as a by-product of its execution; the fairness assumption guarantees that a transaction in the transaction space will eventually be executed. Thus, the Swarm definition of **ensures** is a generalization of UNITY's definition.

The **leads-to** property, denoted by the symbol $\longmapsto$, is commonly used in Swarm program proofs and is identical to the definition in UNITY. Informally, $p \longmapsto q$ means once p becomes *true*, q will eventually become *true*. However, p is not guaranteed to remain *true* until q becomes *true*.

UNITY makes extensive use of the fixed-point predicate *FP* which can be derived syntactically from the program text. Since *FP* predicates cannot be defined syntactically in Swarm, verifications of Swarm programs must formulate program postconditions differently—often in terms of other **stable** properties. However, unlike UNITY programs, Swarm programs can *terminate*; a termination predicate *TERM* can be defined as follows:

$$TERM \equiv (\forall t : t \in \textbf{TRS} : \neg[t])$$

Other than the cases pointed out above (i.e., transaction rule, **ensures**, and *FP*), the

Swarm logic is identical to UNITY's logic. The theorems (not involving FP) developed in Chapter 3 of UNITY can be proven for Swarm as well.

4.3.2 From Swarm to Rule-based Programs

As stated earlier, we will be using the Swarm language to describe the production system programs. The Swarm proof theory will be used to show correctness of program behavior. We can make a correlation of a Swarm program to a rule-based program. The shared dataspace is divided into three subsets: the tuple space, the transaction space, and the synchrony relation.

The working memory of a production system is equivalent to the tuple space in Swarm, as is a working memory element to a tuple. However, production memory has no direct equivalent, because a rule in production memory may be a transaction or it may be a subtransaction. The distinction depends on the independence of the production rule to other production rules and the way that the transaction space is organized. There is no equivalent of the synchrony relation in production systems.

Termination of a serial production system occurs when the conflict set produced by the match phase is empty. Termination in Swarm occurs when the transaction space is empty. Thus, it will be necessary to carefully define when a transaction reasserts itself and when it does not, in order for the rule-based program to correctly terminate.

The Swarm execution cycle, in the context of production systems is as follows:

i. Non-deterministically choose a transaction from the production memory.
ii. Match subtransactions simultaneously against the working memory.
iii. Execute simultaneously all subtransactions that evaluate to true and change working memory, with all deletions preceding all additions .

4.3.3 Motivation for Using Swarm

It is necessary for us to use some language for defining and developing our parallel rule-based programs. The Swarm language and model provides us with some inherent qualities that we believe will be beneficial toward achieving our goals. The major benefit of using this model is that we can use its proof theory. It is possible that a formal model with a different rule execution cycle will yield better results, and this will be determined. The main disadvantage is that the execution cycle is unlike any production system language currently available. But, this different rule execution cycle does not make it necessary to match all rules to determine which ones have instantiations, since only one rule is chosen at a time to match and execute.

In Swarm, there is no general structure for a global control module or global scheduler as in production systems or blackboard systems. The type of global control mechanism

present in Swarm is a scheduler which filters out all undesirable execution sequences. Thus, proofs are performed only on those execution sequences that have the behaviors desired. For example, given a set of transactions, $T = \{t_1, t_2, t_3, t_4\}$, we want only to reason (prove properties) about those executions where t_2 executes before every execution of t_4. Then the filter would accept the finite execution sequence, $t_1, t_2, t_4, t_3, t_2, t_2, t_4$, but not $t_1, t_2, t_3, t_4, t_4, t_2, t_1$. We will only use this scheduler when no other alternatives exist.

Even using this filtering mechanism, no global conflict set will be produced. Any rule that is chosen and has subtransactions that are matched successfully, will be executed. There is no control mechanism to decide which transaction is chosen. This lack of control will force an analysis of the importance of global control in parallel rule-based programs. Therefore, by using a formal language such as Swarm, finding specific solutions to the control problems and rule ordering problems will produce insights into general solutions.

Other properties in Swarm also make it a desirable model. Asynchronous computation can be simulated and synchronous computation can be performed dynamically through the synchrony relation. Transactions or rules are considered atomic. Use of the proof system will alleviate the problems of non-determinism, solution correctness, and termination detection within a rule-based program.

5 Research Proposal

This section begins with an overview of the proposed research. The tasks in bold face are discussed separately in detail. The overall contribution of this research will be methods for defining, developing, and encoding correct parallel production system programs. Concentration will be placed on transforming current serial programs. The problems presented in section 2 will be addressed within the tasks. The transformations and derivations that are described will be performed manually.

1. **Eliminating global control.** This task will occupy the largest portion of the research. We will define a class of conflict resolution rules or strategies relied on by different serial rule-based programs. The role that these strategies play in the serial programs will be analyzed and their worth in parallel rule-based programs will be determined. Those strategies that are necessary to keep will be emulated by methods which we will develop.

2. **Formal description and proof.** We will show that specifications and constraints of rule-based programs can be formally described. Also, we will show how a rule-based programs can be incrementally developed and proven to maintain correct behavior. This task will be done by encoding programs into Swarm. Once these programs are encoded in Swarm, we will utilize the proof theory of Swarm to show correctness of the program according to initial specifications and constraints.

3. **Enhancing rule-based programs.** A series of derivations of the programs in Swarm will be defined to change the program into one which exhibits parallelism while maintaining correctness. These derivations will be in the form of rules and/or heuristics. Issues concerning serializability and synchronization will be addressed under this heading.

4. **Mapping programs to architectures.** By analyzing the enhanced programs, we will discuss ways to project them onto parallel architectures. Target architectures will be defined in which the issues of serializability and proper rule distribution and ordering will be addressed.

5. **Evaluation of research.** The methodologies for transforming the serial rule-based programs to rule-based programs without conflict resolution will be evaluated. Swarm will be evaluated to determine its usefulness in the context of parallel production systems. The benefits and disadvantages of Swarm will be presented as well as suggestions for extensions and changes. The programs developed in Swarm will be evaluated on the basis of 1) the degree to which the problems presented in section 2 are overcome, and 2) the degree of parallelism achieved.

5.1 Eliminating Global Control

Researchers in parallel production systems need rule-based programs which do not rely on any conflict resolution strategy. They utilize programs that never had this reliance or they transform their example programs into the types they need on a program by program basis. These programs may be more easily expressed in parallel if they do not initially rely on conflict resolution. Such programs may also exhibit more parallelism. No methodologies for general transformation of these programs exist.

We will begin by gathering a set of representative rule-based programs. These programs

will be those used by parallel production system researchers and will include some toy problems. Most of these programs will rely on a conflict resolution strategy, a combination of different conflict resolution rules. Programs will be chosen so that the set contains some current benchmark programs for parallel production systems, programs which rely on different conflict resolution rules, programs which are known to be very sequential or very parallel, and programs which have not been studied. For the purpose of clarity, we call this set of programs INITIAL.

An analysis of the role that conflict resolution plays in the programs will be performed. The importance this role has will be determined along with the changes that must be made to the program if certain conflict resolution rules are eliminated. For those rules that must remain to achieve desired results, methods for which the rules can be emulated will be developed.

Because Swarm has no notion of a conflict set or global conflict resolution, it will be very useful as a vehicle for this analysis. We want to encode the rule-based program in Swarm so that it executed as it did in the simple production system. Thus we can determine the importance of certain conflict resolution rules by examining sample executions of the program in Swarm. Since Swarm rules are independent of each other, control must be placed within the rule's interaction with the dataspace.

The objective is to transform from INITIAL a set of programs that can be encoded in Swarm. Then, by using Swarm, we can generalize the methods, by which the transformation was performed, to determine when to eliminate conflict resolution rules, and how to emulate, or encode those necessary conflict resolution strategies. Part of this analysis will be done by using the global scheduling filter mechanism in Swarm. By formally describing the conflict resolution rules we do not wish to eliminate and looking at the behavior of executions that are satisfactory, we hope to gain insight into efficient methods for encoding those conflict resolution rules. This mechanism may also be used for those conflict resolution rules which are necessary but cannot be encoded into Swarm.

The set of programs, that are transformed for execution in Swarm will be called SW. We recognize that Swarm offers mechanisms to make the some transformations easier. For example, the entire dataspace can be examined and also a transaction may be defined to add a different transaction to the transaction space depending on the state of computation. If mechanisms such at these are used to create programs with independent rules and no reliance on global conflict resolution, they cannot be utilized directly in a production system. Therefore, another objective is to develop general methods for transforming the programs in INITIAL to ones which can execute on a production system model. We will use the information and insight gained by creating SW, to transform INITIAL into another set of programs for the general production system model.

This new set of programs, call it PR will contain programs that can be executed on a production system model and do not rely on any form of global conflict resolution. PR may be a subset of SW, if some transformations from INITIAL to SW rely specifically on characteristics of Swarm, such as the capability to utilize the global scheduling filter. PR may be a superset of SW, if there are ways to transform programs from INITIAL that cannot be done in Swarm, and all programs in SW can be transformed into PR. There may be no relation between PR and SW because of the differences between the general production system model and Swarm.

Overall, an analysis will be performed to determine the conflict resolution strategies which can be eliminated, ignored, simulated and/or encoded within production rules and working memory for use in parallel production systems. Methodologies of simulating and encoding these strategies specifically in Swarm will be developed. General methods of simulating and encoding these strategies in a program targeted for a general production system model will be developed. The differences between the general methods and the Swarm methods will be given.

5.2 Formal Description and Proof

Some of the problems presented in section 2 may arise in parallel (and serial) production systems as the result of informal problem description and lack of capability to guarantee certain results. Currently a desirable solution for a rule-based program is guaranteed to be achieved through multiple executions of the program. The use of conflict resolution controls how solution paths are traversed, in order to guarantee such paths converge. Neither multiple executions nor explicit control strategies prove program correctness. There does not exist a commonly accepted formal system for which a rule-based program can be expressed and proven correct. It is possible that a new way to express and prove rule-based programs, as in Swarm, may lead to a change in a new production system model.

Concentration will be placed on the set SW defined in section 5.1. The set SW is the set of rule-based programs that can be encoded in Swarm which were transformed from INITIAL. The set INITIAL is a set of example rule-based programs which rely on various conflict resolution strategies. In order to prove correctness of the programs, certain specifications and constraints for each program must be formally encoded in Swarm and serve as invariants. The set of acceptable solutions to the rule-based programs must also be encoded formally into the terminating conditions for Swarm. Many of the specifications for expert systems are vague, and extremely difficult to state formally. Therefore, we will attempt to prove correctness of the behavior of the rule-based programs. It is possible that during the proof process, rewriting of the program may be necessary, due to oversights in the encoding or oversights in the transformation. Rewriting may also result in easier proofs. Any insight gained from the formal description of the program and proving program correctness can be reflected back onto the initial transformation methods for both SW and PR.

Relationships between the methods of proving correctness and characteristics of the program, including the initial conflict resolution strategy used, will be reported. If such methods can be generalized, we will detail heuristics for proving a certain type of rule-based program to be correct. This generalization applies also to encoding the specifications and constraints, and for proving the equivalence between the solution results from the program in INITIAL and the solution results from the program in SW', the set of transformed programs in Swarm that have been proven correct.

The results of this part of the research will be formal descriptions of a set of rule-based programs which can be proven correct. Such descriptions and proofs have not yet been performed in Swarm, though their benefits have been discussed[5]. Through the use of the Swarm proof theory, we can show equivalence of the globally controlled programs in INITIAL and their counterparts in SW'. Lastly, providing general methods for expressing rule-based programs in Swarm and for proving correctness according to

common characteristics and common encodings will be a contribution.

5.3 Enhancing Programs

In Section 3, methods were presented to solve certain problems that arise in attempts to execute a production system program in parallel. Swarm provides a means to express concurrent computations by allowing: 1) all satisfied subtransactions of a transaction to execute simultaneously, and 2) the synchrony relation to dynamically gather related transactions for simultaneous execution. Once a program is encoded in Swarm, it can possibly be further enhanced to exploit more parallelism. Using the proof of the original program, the enhanced program can maintain its original correctness criteria, plus any additional correctness criteria that becomes necessary as parallelism is increased. This task is of importance in giving insight into which type of computation, (synchronous, asynchronous, or combination) allows more rules to execute simultaneously in Swarm and in parallel production system programs.

Using those programs in SW', we will perform a series of transformation on each Swarm program. These transformations will yield a final program in Swarm, in which: 1) a large number of rules that match are executed simultaneously, and 2) serializability is guaranteed. Some techniques used for transformation will be taken from other sources which detail program derivation, such as UNITY[4]. These transformations will be reflected onto the set of programs in PR, the general production systems programs which do not rely on conflict resolution but exhibit desired behavior during execution. This reflection will enhance these programs to exploit parallelism when executed on a parallel production system model. Again some transformations may be specific to Swarm, but attempts to generalize will be made.

The transformations in Swarm will include synchronizing transactions which do not interfere and requiring additional control to be encoded in transactions which may interfere. Interference definitions will be those defined in section 3. The transformation rules will define ways to control multiple rule executions and guarantee serializability. In essence, the evaluation of the transaction and tuple space will be static, and will rely on previous research results on guaranteeing serializability, and controlling task ordering. New correctness criteria will be added to the programs due to the increase in parallelism. This criteria will be generalized as before across programs with similar characteristics.

From the accomplishments and restrictions defined at this point in the research, we can determine if general methods and/or heuristics exist for directly encoding a rule-based program in Swarm which exhibits the parallelism achieved by enhancement. If so, possible derivation rules will be discussed. Two approaches can be taken to perform this task. One approach will be to examine the initial rule-based programs. These programs will have the initial rules and solution paths, along with the conflict resolution and control strategies. From the structure of the program, and the previous methods necessary to transform it to a parallel program, we will determine if there are general strategies to encode the serial program directly into a parallel program. The second approach is to derive the program by examining the initial specifications and constraints. In this approach, there will be no concern over what conflict resolution strategies have been utilized in the past. The programmer will begin by determining where parallelism can occur in the program, and encode the program directly in Swarm, proving its correctness during development. It is

possible that the derivation rules will apply to a smaller subset of program than SW.

During this task, we will evaluate INITIAL as a set of example programs. This evaluation will determine what types of if INITIAL is a good representative set of programs which can be used as benchmarks for parallel production system, and what other types of rule-based programs would be beneficial to this research specifically.

There are several by-products of this task. It will provide a means for analysis of the benefits and disadvantages of the Swarm model in the context of parallel production systems. We will be able to utilize previous research on serializability, rule division, task ordering, etc., to enhance the programs, in conjunction with the proof theory of Swarm. Also, developing rule-based programs that exhibit a large amount of simultaneous computation in Swarm has not yet been performed.

In general, there are three main goals associated with this task. The first is to determine general characteristics of rule-based programs which exploit parallelism, and characteristics that hinder parallelism. The second goal is to lay the foundation for a design methodology such that if the rules are encoded in some way within a serial production system model, transforming the rules to a parallel production system model can be performed easily and correctly. By repeatedly examining the programs and improving them, we hope to achieve this goal and contribute such a design methodology. Lastly, we will discuss methods and/or heuristics for directly encoding rule-based programs from specifications and constraints, so that these programs exploit any inherent parallelism.

5.4 Mapping Programs to Architectures

Because Swarm is a theoretical language, it is necessary to speculate how these parallel rule-based programs will be mapped onto parallel architectures. We believe that well-designed programs should be easily mapped. But since the programs will be designed without regard to efficiency of execution on a particular architecture, some added constraints may be necessary.

Before entering this task, we have transformed a set of rule-based programs, INITIAL, to a new set, SW'', which are those programs encoded in Swarm, proven correct and enhanced to exploit parallelism. Within this task we will discuss how these programs can be mapped onto target architectures. Some general assumptions applied to the execution of Swarm on multiprocessors, will help us to speculate on the possible ways to implement the programs on parallel architectures running Swarm.

One constraint of the mapping is that the correctness criteria established and proven for the Swarm program, must be maintained. Otherwise, correct behavior cannot be guaranteed during actual execution of the program on an architecture. For example the mapping must define ways to partition production memory to control multiple rule executions and guarantee serializability. Also, the rule distribution among processors should not inhibit any rule that can execute from doing so.

Results from this task will be 1) an understanding of how the parallel rule-based programs which we have developed can be implemented in current architectures, and 2) a discussion on when partitioning of the transaction space is necessary versus replication of the transaction space on every processor.

5.5 Evaluation of Finished Research

There are three major areas of the this research that will be under evaluation. First, the methodologies for transforming the serial rule-based programs to rule-based programs without conflict resolution will be evaluated. The metrics that will be used are 1) the total number of conflict resolution strategies and/or rules eliminated, 2) which strategies can be generalized from Swarm to the simple production system, and 3) determine the properties of the program and/or conflict resolution strategy that restrict or ease generalization.

As stated earlier, it is possible that a new way to express and prove rule-based programs, as in Swarm, may lead to a change in the way production systems are modeled. Since Swarm is a new model for this area, evaluation of the model for formally describing and proving such programs is necessary. We will evaluate its performance in this context. Swarm may be wholly beneficial to the development of parallel production system programs. It may also be that the Swarm model, in the manner in which it was used for this research, was not capable of yielding a satisfactory solution. In our evaluation, we will determine if this paradigm has aided us in the development of such programs and that the solutions found within this paradigm do not exhibit more serious drawbacks and restrictions the solutions presented in other models.

Finally, the programs developed in Swarm will be evaluated on the basis of 1) the degree to which the problems presented in section 2 are overcome, and 2) the degree of parallelism achieved. These evaluations are consistent with those performed by other researchers in this area. Because there will be no centralized control and we will be developing parallel rule-based programs, it is necessary determine the extent to which the problems given in section 2 have been solved by our efforts and by previous research efforts. Most likely there will be trade offs between reaching a good solution for one problem and a fair solution for another. If certain problems cannot be solved within the model, we will explore reasons for this failure. The degree of parallelism will be measured against the parallelism expressed by the similar programs in other models. Measures are the number of rule firings necessary to achieve a solution and the number of rules that can be fired in parallel against that number from other parallel production system models.

References

[1] A. Acharya. Message passing computers as production system engines. In *IJCAI-89 Workshop on Parallel Algorithms*, August 1989. Unpublished proceedings.

[2] S. Ahuja, N. Carriero, and D. Gelertner. Linda and friends. *Computer*, 19(8):26–34, August 1986.

[3] F. Barachini, H. Mistelberger, and E. Bahr. The art of parallel pattern matching. In *IJCAI-89 Workshop on Parallel Algorithms*, August 1989. Unpublished proceedings.

[4] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, Reading, MA, 1988.

[5] H.C. Cunningham and G.-C. Roman. Toward formal verification of rule-based systems: A shared dataspace perspective. Technical Report WUCS-89-28, Washington University, June 1989.

[6] H.C. Cunningham and G.-C. Roman. A UNITY-style programming logic for a shared dataspace language. Technical Report WUCS-89-5, Washington University, March 1989. To appear in IEEE Transaction on Distributed and Parallel Computing.

[7] J.R. Ensor and J.D. Gabbe. Transactional blackboards. In Alan H. Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[8] S. Fickas and D. Novick. Control knowledge in expert systems: Relaxing restrictive assumptions. Technical Report CIS-TR 85-04, University of Oregon, February 1985.

[9] S. Fickas, D. Novick, and R. Reesor. Building control strategies in a rule-based system. Technical Report CIS-TR 85-04, University of Oregon, January 1985.

[10] S. Fickas, D. Novick, and R. Reesor. An environment for building control rule-based systems: An overview. Technical Report CIS-TR 85-10, University of Oregon, 1985.

[11] A. Gupta. *Parallelism in Production Systems.* Pitman Publishing, London, England, 1987.

[12] T. Ishida and S.J. Stolfo. Towards the parallel execution of rules in production system programs. In *Proceedings of the IEEE International Conference on Parallel Processing*, Washington, D.C., 1985. IEEE Computer Society Press.

[13] J. McDermott and C. Forgy. Production system conflict resolution strategies. In D.A Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems.* Academic Press, New York, NY, 1978.

[14] D.P. Miranker, C.M. Kuo, and J.C. Browne. Compiling parallelism among rules. In *IJCAI-89 Workshop on Parallel Algorithms*, August 1989. Unpublished proceedings.

[15] D.P. Miranker, C.M. Kuo, and J.C. Browne. Parallel transformation for a concurrent rule execution language. Technical Report TR-89-30, University of Texas at Austin, October 1989.

[16] H.P. Nii, N. Aiello, and J. Rice. Frameworks for concurrent problem solving: A report on CAGE and POLIGON. In Robert Engelmore and Tony Morgan, editors, *Blackboard Systems*. Addison-Wesley Publishing Company, Wokingham, England, 1988.

[17] M. Perlin. Match box: Fine-grained parallelism at the match level. In *IJCAI-89 Workshop on Parallel Algorithms*. Unpublished proceedings, August 1989. Also tech report CMU-CS-89-163.

[18] G.-C. Roman and H.C. Cunningham. A shared dataspace model of concurrency-language and programming implications. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 270–9, Los Alamitos, CA, 1989. IEEE Computer Society Press.

[19] J.G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. Technical Report 89-5, Tufts University, Medford MA, October 1989.

[20] J.G. Schmolze and S. Goel. A parallel asynchronous distributed production system. In *Submitted to ICPP-90*, 1990. Technical Report in Preparation.

[21] M. Tambe, D. Kalp, A. Gupt, C. Forgy, B. Milnes, and A. Newell. Soar/PSM-E: Investigating match parallelism in a learning production system. In *ACM/SIGPLAN PPEALS*, pages 146–160, Sept. 1988.

[22] A. Terry. Using explicit strategic knowledge to control expert systems. In Robert Engelmore and Tony Morgan, editors, *Blackboard Systems*. Addison-Wesley Publishing Company, Wokingham, England, 1988.