An Implementation of Multiple Worlds for Parallel Rule-Firing Production Systems[1]

Daniel E. Neiman
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
Phone: (413)545-3444
Email: DANN@CS.UMASS.EDU

Abstract

One of the principal advantages of parallelizing a rule-based system, or more generally, any A.I. system, is the ability to pursue alternate search paths concurrently. Conventional memory representations for production systems cannot easily or efficiently support parallel search because of the essentially flat structure of working memory and the combinatorics of pursuing pattern matching in a large memory space. A further obstacle to the effective exploitation of parallelism is the problem of maintaining the internal consistency of each search space while performing parallel activities in other spaces. This paper presents an approach to parallel search for rule-based systems which involves maintaining multiple separate worlds, each representing a search space. Constructs for creating, manipulating, and merging separate spaces are discussed. We describe how the addition of a language mechanism for specifying multiple worlds simplifies the design of parallel search algorithms, increases the efficiency of the pattern matcher, provides a framework for implementing advanced control mechanisms such as task-based or hierarchical problem solvers, and reduces or eliminates the overhead of runtime consistency checking.

[1]This work was partly supported by the Office of Naval Research under a University Research Initiative grant, number N00014-86-K-0764, NSF contract CDA-8922572, and DARPA contract N00014-89-J-1877.

1 Introduction

In recent years, the parallel firing of rules has been explored as a method of increasing the performance of rule-based systems [9, 10, 11, 12, 16, 17, 20, 22, 24, 27]. Much of this work has concentrated on automatically identifying sets of concurrently executable rules based on a syntactic analysis of rule sets. Our own research in this area has indicated that the automatic extraction of parallelism and run-time detection of concurrently executable rules limits the performance of the rule-firing system to a single order of magnitude speedup [23, 19]. As a result, our research has focused on explicitly reformulating problems to exploit the underlying parallelism within the domain and designing the rule-based algorithms accordingly. To this end, a number of enhancements have been made to our basic rule-firing system which enable the design of parallel and correct programs. Many of these enhancements take the form of language mechanisms, or idioms, that make it easier to express parallel algorithms. One of these mechanisms, a set of constructs for supporting the creation of multiple worlds, is described in this paper. We describe how the addition of a semantics for multiple worlds simplifies the design of parallel search algorithms, increases the efficiency of the pattern matcher, provides a framework for implementing advanced control mechanisms such as task-based or hierarchical problem solvers, solves the problem of determing local quiescence of tasks, and reduces or eliminates the overhead of runtime consistency checking.

1.1 Motivation

In research reported in [17, 18], we demonstrated that conventional conflict resolution techniques would cause synchronization delays in parallel rule execution due to unequal rule matching times and proposed an asynchronous rule-firing scheme in which rules are executed as soon as they become enabled. One method of eliminating conflict resolution is to interpret competing rule instances as alternative operators within a search space and to exploit rule-level parallelism by concurrently exploring alternative solution paths. In many cases, parallel exploration of alternatives can result in a superior solution by allowing a greater number of possible solutions to be examined rather than selecting a single alternative using a heuristic conflict resolution scheme. However, neither the structure of working memory, nor the semantics of production systems is designed to support search, and current techniques for implementing search are cumbersome to program and potentially inefficient.

The process of searching a state space involves applying one or more operators to a current state in order to generate new states. In a rule-based system, each state is composed of a set of facts or *working memory elements* and the operators are the rules themselves. In order to preserve consistency while performing search, the system must be able to distinguish between working memory elements that are contained in separate states. However, most production systems record their facts in a flat structure that does not provide facilities for manipulating or maintaining separate search spaces. For this reason, search can only be performed in a limited number of ways in a conventional rule-based system. One can employ backtracking, creating and deleting states and evaluating each alternative in turn until an acceptable solution is achieved. Such an approach minimizes the memory requirements of search since fewer states need to be represented at any given time, however there is no appreciable

rule-level parallelism in this approach.

A more common idiom for performing search in production systems is to *annotate* each working memory element with a specific tag indicating which state of the search space it belongs to. Only working memory elements with the same annotation are allowed to match. The SOAR system, for example uses this technique to link all related objects within a sub-task [21]. This approach is suitable for cases in which there are few states and each state is represented by only a small set of working memory elements, however, there are a number of disadvantages. First, annotation increases the complexity of both the left and righthand sides of a production. Working memory elements must be modified to contain fields whose sole purpose is to represent control data (tags), resulting in more obscure rules. When copying large states, the corresponding lefthand side of the production must be very large in order to refer to each element of the previous state. If the number of elements within a state is not a constant, multiple almost-identical versions of rules with varying lefthand sides must be written. In order to create a new state, each new working memory element must be specifically asserted in the righthand side of a production.

The most significant problem with annotation is that it reduces the efficiency of the pattern matcher. If memory is represented as a flat list (as in the original OPS5 memories), then elements from each state must be matched against elements in all other states to see if they possess the appropriate tag. Thus, the matching time and the space required to store partial matches increases combinatorially in proportion to the number of states. If memory is hashed, as in more recent implementations of production systems (such as CParaOPS5 [7]), the most obvious choice for a hash key is the state tag, but this reduces the effectiveness of the hashing implementation to that of a linear memory representation: within a state there is no internal hashing and all matching is done on the contents of the bucket accessed by the state tag.

The flat representation of working memory also has implications for parallel implementations of production systems. Because memory nodes are shared by elements in all states of the search space, as the number of processors concurrently performing pattern matching increases so does the contention for the memory nodes, and processors spend an increasing amount of time waiting for access to critical regions.

2 Partitioned Memory for Parallel Search

The method we have chosen to enable search algorithms to be expressed more easily in production systems is to employ a multiple worlds representation implemented by partitioning working memory. The concept of partitioned spaces is not a new one in A.I. problem solvers. The CONNIVER system introduced the notion of *contexts* to represent hypothetical worlds [25]. YAPS [1], an object-oriented version of OPS5, supported multiple databases, as well as a global database. The knowledge-based systems KEE [3] and ART [26], also incorporate methods for reasoning about hypothetical worlds. Many of these systems use an assumption-based truth maintenance approach [2] although there is reason to believe that the overhead of maintaining beliefs would limit pattern-matching performance and require synchronization between parallel processes. An approach to managing multiple worlds

for parallel production systems which is similar in concept to assumption-based truth maintenance was investigated by Matsuzawa [13]. In this work, a method was proposed for a parallel system that produces all possible results in an expert system by firing all rules simultaneously. This method involves creating a separate tag for each working memory element created. This tag describes the inference history of the rule creating the working memory element. Rules are matched against all working memory elements, regardless of the worlds in which they exist and then a separate *merge* step takes place to ensure that all elements enabling a rule instance are consistent. While this method works in principle, in practice it quickly suffers from a combinatorial growth in terms of length of rule inference history, and in terms of the increasing size of working memory (because all rules are matched within a flat database). Other drawbacks include the inability to access all consistent elements in a state (all elements valid in a particular world), and the inability to execute more than a single rule in the context of any given world.

In our research, we sought a solution to the multiple world representation problem that was conceptually simple, easy to implement, reasonably efficient, and that would exploit, at least to some degree, the fact that many facts in the database are either static or change only at specific times. The approach that we have implemented is to create separate memory partitions such that each state in the search space is represented in its own world, and all working memory elements representing that state are assigned to that partition. A special partition, called the *base space* (or *base partition*), is defined for elements that are guaranteed not to change during the course of problem solving; these elements represent the immutable "world facts" of the system in question. All working memory elements within the same state are placed within the same partition and are matched only against elements within that state or elements in the base partition.

The motivation for the base partition is our observation that working memory elements typically are employed in a dual role: first, as state variables describing the state of the computation and second, as facts describing the environment in which problem solving is taking place. It would prove inefficient to copy all of working memory into each new state in situations in which a large number of elements remain static during the course of problem solving. For example, when solving a configuration problem, one would not expect the database of known components to change during the course of problem solving. Therefore, we provide the facility to specify a "base state" of facts that do not change during the course of problem solving. The relationship between the base space and all other spaces is depicted in Figure 1.[1]

2.1 Semantics of World Spaces

We define the following semantics for rules operating in a multiple world environment:

- The lefthand side of rules can match only elements within a single space, or within the base space. A rule which matches any element not in the base space is said to be executing in the space of that element.

[1] For similar reasons, the mechanism in KEE for maintaining possible worlds [3] allowed the specification of "background facts" which were not subject to assumption-based truth maintenance.

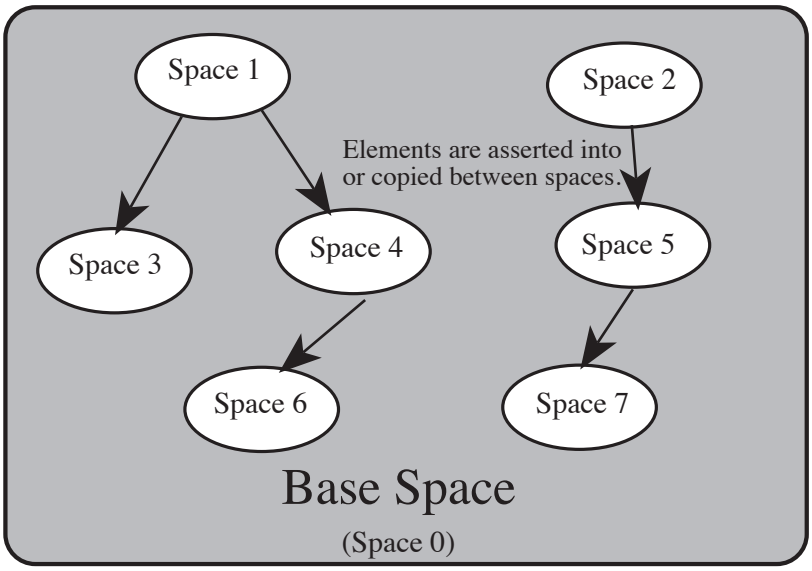

Figure 1: All spaces in the universe of multiple worlds have access to facts represented within the base space.

- Unless explicitly specified, working memory elements added by a rule are asserted into the space in which the rule is executing.
- Changes made to working memory elements in one space (excepting the base space) are guaranteed not to affect any other space in memory, and by implication, any rule executing in any other space.

2.2 Language Constructs for Manipulating Spaces

In order to create and manage spaces, our parallel rule-firing system has been augmented with the following set of commands that are expected to be executed within the righthand sides of rules.

- Generate-new-space: Generate a unique identifier describing a partition.
- Kill-space: Delete all working memory elements in a given state, remove any rules enabled by these elements from the execution queues, and reclaim the space.
- Assert-into-space: Identical to the working memory creation command, but allows the partition to be specified.
- Modify-into-space: Modify an existing working memory element and place the result in the given space.
- Copy-space-to-space: Copy the entire contents of a space into a new space.
- Copy-and-move-to-space: Copy the space in which the rule is executing to a new space and execute the remainder of the righthand side in the context of the new space.
- Space-quiescent-p: Are any changes being made in the indicated space?

- Lock-space: Forbid any changes to be made to the memory space.
- Current-space: Return the identifier of the current space.

In addition to the above commands, our production system includes a set-oriented semantics similar to that described by Gordin and Pasik [6] – the use of this construct allows the programmer to match selected *sets* of working memory elements on the lefthand side of the rule and copy them into a new state. The addition of many new righthand side operators to the language represents a tradeoff in programming versus representational complexity. By representing the manipulation of states explicitly in the imperative code of the righthand side of the rule, the pattern-matching lefthand side is considerably simplified.

3 Advantages of Partitioned Worlds

The use of partitioned spaces, although simple in concept, has surprisingly far-reaching implications for parallel rule-firing systems; the problems of control, working memory consistency enforcement, locking, and program design are all simplified and the match process itself becomes more efficient.

3.1 Reduction of Memory Sizes

One of the principal advantages of partitioning memory into multiple worlds is the attendant reduction of memory sizes during the pattern match. Instead of examining large numbers of elements belonging to multiple states, the pattern matcher need only examine the set of elements corresponding to the current state. Because pattern-matching within the Rete net frequently leads to combinatorial growth in the number of partial matches, restricting the number of elements in each space will result in faster matching. We expect, based on our experiences in problem solving in production systems, that the number of elements in each problem space will be relatively few and the small amount of memory required to represent each state will compensate somewhat for the additional overhead of maintaining many states at once in a parallel system.

3.2 Working Memory Consistency and Multiple Worlds

When multiple rules execute concurrently, there is the chance that the changes that they make to working memory will interact, resulting in a working memory state that could not be achieved by any serial order of rule firing; results of this nature are said to be *non-serializable*. The problem of maintaining working memory consistency during the course of parallel rule-firing has been extensively studied [9, 10, 22, 23]. One of the motivating factors for the development of the multiple world paradigm for parallel production systems is the virtual elimination of overhead for consistency maintenance. Because rules in separate spaces do not share state-specific data, analysis of interactions between rules executing in separate spaces is not necessary. The complete independence of partitions means that rule firings in different spaces can be scheduled and executed asynchronously with respect to each other, increasing the potential processor utilization. The asynchronous execution of rules in multiple worlds is similar to the control architecture proposed by Miranker, Kuo and Browne [14, 11], however, the

parallel partitions are generated explicitly by the task-based semantics of the problem solving process rather than extracted via an analysis mechanism.

It is still possible for rules operating *within* the same partition to interact. However, the number of rules executing within each space can reasonably be expected to be small; because the cost of interaction detection is dependent on the number of rules concurrently executing [19, 23] the cost of ensuring serializability will be greatly reduced.

3.2.1 Communication between Spaces

In general, the separation between spaces is held to be absolute, that is, there is no interaction between rules executing in separate partitions. However there are situations in which it is desirable to support communication between spaces, for example, when asserting a solution or returning a result from a subtask to a superior task. The command set of the production system is augmented with an *assert-into-space* command that allows rules to place results in the working memory of another space. When this occurs, the assumptions of space independence are violated and locking of working memory elements, detection of interactions, or very careful design are required to ensure correct and serializable behavior of the system when running in parallel. Because the communication between spaces occurs only at well-defined points in the computation and only for specific purposes, it is possible to design inexpensive constructs for ensuring correctness, i.e. the idiom for merging multiple solutions described in [18].

3.3 Multiple Worlds and Control of Parallel Rule-firing

In [17], we raised a number of issues relevant to the sophisticated control of production systems, namely:

- How can a task- or goal-oriented control scheme be imposed on the essentially data-driven paradigm of rule-based programming?
- How can we insure that all related rules have become quiescent before beginning a control process?
- How can rules be dynamically scheduled to ensure high utilization of each processor?

We can show how the use of *spaces* makes each of these problems manageable.

3.3.1 Task-based Control

The principal difficulty in implementing task-based control in a parallel rule-based system is identifying when a particular rule is applicable to a specific task. Because multiple rules can execute simultaneously and assert working memory elements at unpredictable times, it is not possible to attribute the activation of a rule instance to any specific working memory element or rule firing, nor can it be determined when all rules applicable to a specific goal have been discovered. Multiple worlds can be used as a framework to construct a task-based architecture. Specific goal elements representing individual tasks

can be asserted into new spaces; a single task group then consists of all rules executing within that space. A task can be said to have completed when all applicable rules have been executed within that space. Communication of results from subtasks to parent tasks can be performed by the *assert-into-space* operator. Subtasks that prove to be redundant can be pruned by the use of the *kill-space* operator.

3.3.2 Determining Quiescence

We can increase the utilization of processing resources and eliminate synchronization delays by allowing rules (or at a more coarse-grained level, tasks) to execute asynchronously in parallel. However, asynchronous execution in a data-driven paradigm introduces a new problem, that of determining *quiescence*, where quiescence is defined as the termination of pattern-matching activities and rule instantiation with respect to a subset of working memory elements and rules. The determination of quiescence is especially important when the nature of the problem being solved requires a decision to be made between alternative operators or rules. When solving many subproblems asynchronously in parallel, waiting for complete global quiescence is not practical; this would impose an unnecessary synchronization relationship between unrelated computations. Instead, we wish to ensure that quiescence has been achieved relative to a single task. However, because rule-based systems are data-driven, it is not easy, based purely on syntactic information, to determine the scope of a working memory change. We can solve this problem by partitioning working memory according to a semantic task decomposition. If we limit the effects of all rule instances in a task to a single space, we can define local quiescence in the following way:

- No working memory element is currently being asserted, modified, or removed within the task space.
- No rule that asserts elements into the task space has been scheduled for execution.
- No modifications are made or pending for the base space.

Control can then be performed on locally quiescent tasks while computation proceeds in other spaces and delays due to global synchronization can be avoided.

3.3.3 Process Management in Multiple Worlds

There is a time/space trade-off associated with multiple worlds. In exchange for processing worlds concurrently, many spaces must be represented concurrently, with each space potentially containing redundant information. We can minimize the space overhead by observing that most states are transient: they are created, processing occurs, and successor states are generated. States are never referenced after generating successors, therefore we can reclaim spaces as processing is completed. In general, the number of spaces actually in existence need never greatly exceed the number of processors available to generate concurrent tasks. To ensure this, we need to modify our rule-scheduling mechanisms to ensure that all rules referencing a currently instantiated state are executed expeditiously so that the state may generate its successors and be reclaimed.

4 Implementation

A multiple world implementation has been developed for UMass Parallel OPS5 (UMPOPS) [18]. The system runs on a 16 processor Sequent shared-memory multiprocessor and is implemented in Top Level Common Lisp[2]. Although the following discussion is restricted to a Rete net implementation, the concept of partitioned memory spaces is also applicable to non-Rete algorithms such as TREAT [15].

4.1 Partitioned memory in the Rete net

The Rete net is a data structure that has been widely used in the implementation of production systems [4, 7]. Much of its efficiency results from taking advantage of *temporal redundancy* by storing partial matches. Because the database is expected to change only slowly, this allows rapid matching as new data arrives. The Rete net can be considered as a discrimination net augmented with memory at the individual nodes. Working memory elements that are propagated through the net are formed into lists of *tokens* corresponding to partial matches against a rule's lefthand side. A token of length N represents the working memory elements corresponding to the first N positive condition elements in the rule's righthand side; all variable bindings within a token are guaranteed to be consistent. These tokens are stored in memory nodes within the network. The Rete-net has two types of *beta* nodes which require that state be preserved. Simplifying somewhat, **AND** nodes concatenate tokens together and ensure consistent bindings of variables. **NOT** nodes serve as gates and pass tokens only if they have no consistent variable binding with any working memory element matching a negated condition element. Each *beta* node has two inputs, one representing an incoming working memory element and one representing an incoming token consisting of a list of previously matched elements. Because each element arriving at one input of the node must be compared with each element arriving at the other input, all tokens passing through a beta node are stored in either a lefthand or righthand memory (see Figure 2). A description of the functioning of the Rete net can be found in [5].

The modifications to the Rete net are fairly straightforward. Instead of being represented as a flat list or a single hashed table, each memory node with the Rete net is defined as an array, or as an array of hash tables (where the space allocated to the array may grow as the number of states increase). As tokens are propagated through the network, they are placed in the appropriate memory space. Because each node may not contain elements from all spaces, nodes may be assigned tables to map from space indices to array indices in order to conserve space. In the case of **AND** nodes, processing proceeds very much as in the single-space Rete net, but with the following distinction: tokens entering the node are matched *only* against items in the same partition in the corresponding memory *or* tokens in the base partition of the node memory. The modified representation of the Rete node is depicted in Figure 3.

NOT nodes, which represent negated condition elements are somewhat more complex.[3] Briefly, the reason for this is that the **NOT** stores a count of the currently matching elements for each negated token in one of its memories. Because it is possible that the negated token might exist in the base

[2]Top Level Common Lisp is a trademark of Top Level, Inc.

[3]It is our experience that *any* aspect of the implementation of rule-based programming involving negation becomes considerably more complex.

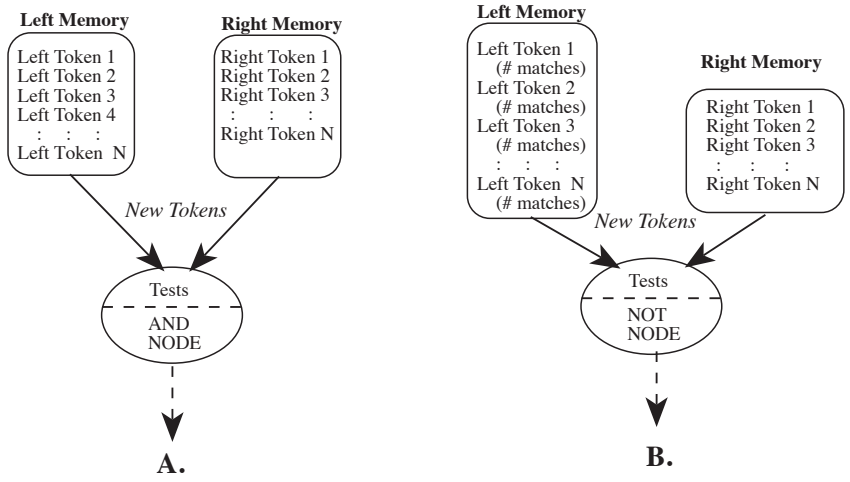

Figure 2: The basic memory structures for the **AND**(A) and **NOT**(B) nodes of the Rete net.

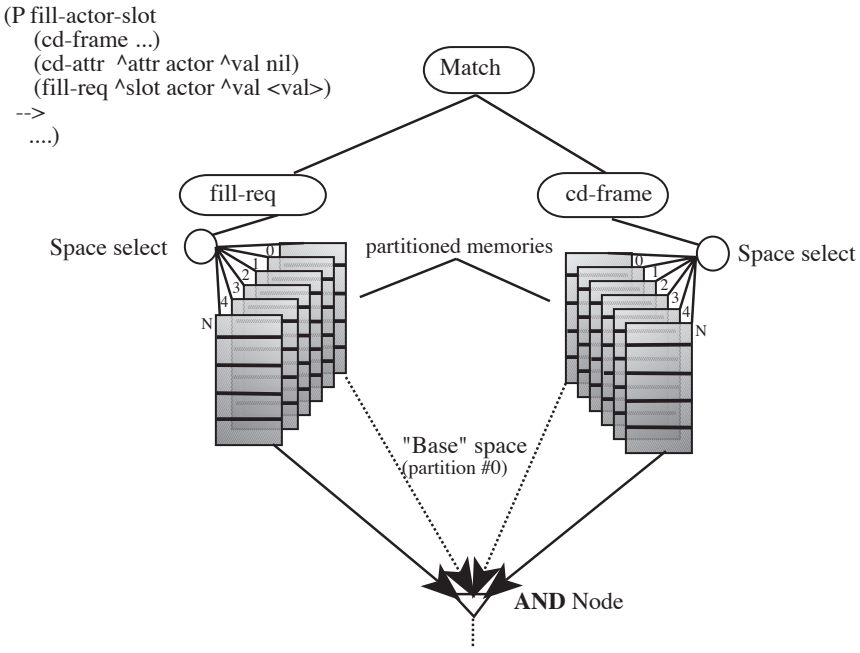

Figure 3: An **AND** node of the Rete net in a system modified to support partitioned memory spaces.

partition, it is not possible to simply direct a pointer to the base memory since any modification to the base partition would affect the match process for all partitions. For this reason, the contents of the base partition of a **NOT** node must be copied to each new space at initialization time.

4.1.1 Performance Considerations

Because of the large variations possible in state sizes and number of states, it is difficult to make any specific claims about the efficiency of our multiple worlds implementation. We have tested the system using a version of the Travelling Salesperson problem and have seen approximately a 20% decrease in execution time when compared with an implementation that explicitly annotates working memory elements with discrete state tags. The relative increase in performance is independent of number of processors assigned to the benchmark and can be largely attributed to the elimination of the overhead of insuring consistent bindings for the state variables and the reduced memory size during pattern matching. In these experiments, it proved critical to the performance of the system that the size of individual node memories be initially set to close to the required size because dynamically increasing the number of states represented at runtime was extremely expensive.

The creation of a state, or the copying of one state into another, is the largest bottleneck and potential source of overhead in a multiple world representation. One method of reducing the cost of creating new states is to insert working memory elements into the new state in parallel using *action* level parallelism [8]. In UMPOPS, we have observed an approximately 8-fold decrease in execution time for rules which assert multiple elements in parallel.

5 Conclusion

This paper has described a scheme for implementing multiple worlds in a parallel production system and has discussed a number of the implications on the design and efficiency of programs implemented using this mechanism. We have found that the use of partitioned memory spaces greatly simplifies a number of issues involved in the successful design of parallel rule-firing programs including the representation of states for parallel search, avoidance of rule interactions, reduction of pattern matching times, ease of design, and development of sophisticated control idioms. The principal drawback to the scheme that has been presented here is the possible redundancy due to the complete copying of state from world to world. We are currently implementing a multiple world scheme based on state inheritance which will be benchmarked against the existing implementation.

References

[1] Elizabeth Allen. YAPS: A production rule system meets objects. In *Proceedings of the Second National Conference on Artificial Intelligence*, pages 5–7, 1983.

[2] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[3] Robert E. Filman. Reasoning with worlds and truth maintenance in a knowledge-based programming environment. *Communications of the ACM*, 31(4):382–401, April 1988.

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

[5] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

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

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

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

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

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

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

[12] Steve Kuo and Dan Moldovan. Implementation of multiple rule firing production systems on hypercube. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 305–309, 1991.

[13] Kazumitsu Matsuzawa. A parallel execution method of production systems with multiple worlds. In *First International Conference on Tools for AI*, pages 339–344, 1989.

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

[15] Daniel P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[16] D. I. Moldovan. Rubic: a multiprocessor for rule-based systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(4):699–706, July/August 1989.

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

[18] Daniel Neiman. Parallel OPS5 user's manual and technical report. COINS Technical Report 92–28 (Supersedes TR 91-1), Computer and Information Sciences Dept., University of Massachusetts, April 1992.

[19] Daniel E. Neiman. *Design and Control of Parallel Rule-Firing Production Systems*. PhD thesis, University of Massachusetts, September 1992.

[20] A. Pasik and S. Stolfo. Improving production system performance on parallel architectures by creating constrained copies of rules. Technical report, Computer Science Dept., Columbia University, 1987.

[21] Daniel J. Scales. Efficient matching algorithms for the SOAR/OPS5 production system. Technical Report KSL 86–47, Knowledge Systems Laboratory, Computer Science Dept., Stanford University, 1986.

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

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

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

[25] Gerald Jay Sussman and Drew Vincent McDermott. Why conniving is better than planning. Artificial Intelligence Memo 255A, A.I. Laboratory, Massachusetts Institute of Technology, April 1972.

[26] Chuck Williams. Art – the advanced reasoning tool, conceptual overview. INFERENCE Corporation, 1984.

[27] Ouri Wolfson and Aya Ozeri. A new paradigm for parallel and distributed rule-processing. In *Proceedings of the ACM-SIGMOD 1990 International Conference on Management of Data*, pages 133–142, May 1990.