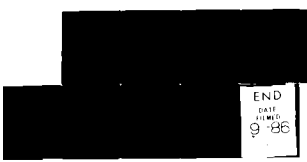
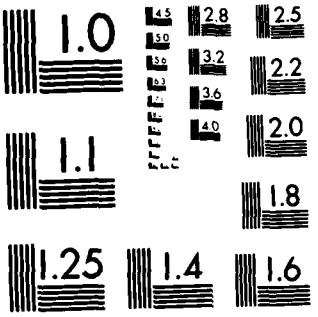
AD-A170 967 A REWRITE RULE MACHINE ARCHITECTURAL OPTIONS AND TESTBED FACILITIES FOR THE REWRITE RULE MACHINE(U) SRI INTERNATIONAL MENLO PARK CA S LEINWAND ET AL. JUL 86 1/1

UNCLASSIFIED N00014-85-C-0417 F/G 9/2 NL



END
DATE FILMED
9-86



SRI International

A REWRITE RULE MACHINE

Architectural Options and Testbed Facilities for the Rewrite Rule Machine

Final Report
July 1986

By: Sany Leinwand, Senior Research Engineer
Joseph Goguen, Program Manager

SRI Project ECU 1243

Prepared for:

Office of Naval Research
Information Sciences Division
800 N. Quincy St.
Arlington, VA 22217-5000

Attn: Dr. Charles Holland , Code 1133

Contract No. N00014-85-C-0417

SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
(415) 326-6200
TWX: 9100-373-2046
Telex: 334486

Architectural Options and Testbed Facilities for the Rewrite Rule Machine*

Sany Leinwand and Joseph Goguen
SRI International, Computer Science Lab

~~June 1~~, 1986
July

Abstract

The Rewrite Rule Machine (RRM) project at SRI International unites advanced architectural concepts with advanced software concepts. The unification is achieved through a novel model of computation, called *concurrent tree rewriting*, which supports both ultra high level programming and extreme concurrency of execution.

This report presents some options for RRM implementation. We expect to use custom VLSI design to place many small processors on a single chip. With a suitable high-level architecture, this will allow unprecedented concurrency, with many thousands of small processors cooperating on a reduction task. Our performance estimate for a reasonable prototype is one million MIPS.

The proposed RRM architecture is described at four different levels. The individual tokens that constitute trees are stored in *cells*, while *nodes* consist of many cells with a common controller. *Clusters* include many nodes sharing a common name space for tokens, and several clusters together constitute the *network* level. Scalability and fault tolerance are also discussed, as is a *testbed* to ease the validation of architectural concepts.

1 Introduction

This report summarizes recent progress of the Rewrite Rule Machine (RRM) project at SRI International. The RRM project blends advances in software and hardware to achieve a novel computer architecture featuring:

- *ultra high level programability*, and
- *highly concurrent* program execution.

We describe the architecture on four levels. The most detailed level is a cell holding a single token from a tree, and the next level is a processor node, consisting of many cells. The largest is a conglomerate of clusters, where each cluster is a network of nodes. The final section of this report presents plans for an architectural evaluation testbed.

*Supported by Office of Naval Research Contract N00014-85-C-0417.

1.1 The Concurrent Tree Rewriting Computation Model

In contrast to many concurrent computer architecture projects, the RRM project starts with a clearly defined model of computation, *concurrent tree rewriting*. Simulated concurrent tree rewriting of typical programs written in the ultra high level functional language OBJ2 [1] reveals enormous potential concurrency [2], and recent research shows how to extend this model of computation from functional programming to logic programming [3] and object-oriented programming [4].

Basically, tree rewriting consists of applying rewrite rules to tree-structured data. Tree rewriting is inherently concurrent in sofar as rewrite rules can be applied irrespective of order[1]. The data being processed consists of tokens organized into a tree. A *rewrite rule* consists of two *templates*,

1. the left-hand side and
2. the right-hand side.

The left-hand side of the rule defines a pattern to be matched against subtrees of the data tree. When the match is successful, the rule's right-hand side is used to modify the matching subtree. Rule *variables* collect individual tokens and subtrees from the matched data for redistribution according to the right-hand side.

Rules may also have a *condition*, which is another expression that must evaluate to true, using the variables determined by the match, before the replacement is actually performed.

Non-sequentiality is inherent in this computational model: any rule that matches a subtree of the data tree can modify it according to the rule's right-hand side. However, correct implementation requires that no part of the data tree be rewriten simultaneously by more than one rule.

1.2 Formal Computational Models and Architecture

The RRM project is organized to exploit continuing interaction between hardware, software and theoretical studies. As a symbiosis of hardware and software concerns, the *concurrent-* and the *partitioned concurrent-* tree rewriting models studied in [5] can guide the solution of machine organization questions. On the other hand, architectural problems will stimulate the development of new formal models. Further phases of RRM research will involve even closer cooperation of hardware and software research to achieve efficient execution control units and compilers.

1.3 Architectural Levels

We have identified four levels of architectural concern, each with specific problems:

[1] The semantics of most computations guarantees that the order of rule application does not matter. However, sometimes one wishes to control the order of application to improve efficiency, and other times it is demanded by the semantics of the computation - for example, with communication protocols. See [5] for a much more detailed discussion.

1. The *cell level* is concerned with implementing *rewrite cells*, which store the individual tokens of which trees are composed.

2. The *node level* is concerned with the organization of single RRM processors, composed of many cells plus a common controller.

3. The *cluster level* considers interconnected RRM nodes having a common name space for tokens[2].

4. The *network level* considers interconnections of several clusters, each with its own name space for tokens.

These levels will be clarified and developed in the following sections, as some of the architectural options under consideration for each level are described.

2 Cell Architecture

A rewrite task may be composed of many thousands, or even hundreds of thousands, of tokens organized into a tree structure. As discussed above, this tree is constantly matched against a set of rewrite rules. Whenever a rule template (left-hand side) matches, the tree is locally changed (rewriten) according to the rule's right-hand side.

The basic RRM component is the *cell*. Each cell can store one token of the tree being processed, and can also perform simple operations on tokens. The interaction between cells during pattern matching and subtree replacement is of major interest. Note that matching and replacement together consistitute tree rewriting.

2.1 Token Representation

We expect to partition problems into sufficiently local computations so that only a few bits (say 8 to 10) are needed to represent any token inside a cluster. As described in the section on cluster architecture, local token values are translated into globally unique names when data is exchanged between clusters.

2.2 Operations on Tokens

Operations performed on a stored token include:

- Match the token for equality with an externally supplied pattern.
- Cooperate with immediate neighbor cells to determine the success or failure of a pattern match.
- Replace the token with a new value when the pattern match is successful.

Since cell size is at a premium, we are investigating ways to do these operations with a minimum of logic gates. A more detailed view of cell operations is given in Section 3.1.

[2] i.e., a mapping of tokens into cell values.

2.3 Numerical Operations

Although numerical operations could be implemented from "basic principles" by using only the successor (i.e., "add one") operation on integers, this would be much too expensive. Alternatives under consideration are:

- Use of stand-alone *numerical coprocessors*. Since the RRM is intended for "symbolic" computation (such as software development and artificial intelligence applications) rather than for "number crunching," this does not seem unreasonable. But for some problems, these numerical tasks could slow down tree rewriting.
- Implement the traditional ALU functions within each cell. This alternative would not be overly costly, since hardware for token equality is already needed in each cell. The compiler could support more complex arithmetic by calling these built-in ALU functions. Because token size will probably limit direct arithmetic computation to byte size numbers, longer and more complex arithmetic will have to be implemented from simple byte operations. The cell should also keep track of inter-byte carries and borrows. Special shift functions supporting multiplication could also be included in the basic ALU set.

2.4 Cell Control

Although we are far from a final cell design, it is clearly desirable to pack as many cells as possible on a single custom VLSI chip. This implies we should minimize the logic that each cell must have. Currently, we favor having all cells in a chip share a common controller, which will broadcast matching and replacement commands. Thus, cells will operate in lock-step (SIMD) mode.

3 Node Architecture

An RRM node consists of many cells and their controller; in effect, it is a block of *active memory*. Fast inter-cell communication is of paramount importance at this level, and this implies that each node should be implemented on a single VLSI chip.

3.1 Node Functionality

A matched pattern is a subtree of tokens. All possible matches of a given pattern are performed concurrently, for all possible subtree positions. Therefore each cell must:

1. check that it contains one of the tokens present in the pattern,
2. initiate a dialog with cells that are subordinate to it in the subtree structure to insure that they have also matched successfully,
3. resolve any match conflicts (e.g., one cell matching two different positions in the same pattern) and
4. notify the subtree root of the outcome.

Following a successful pattern match, the tree will be rewritten according to the rule's right-hand side. In simple cases, the current token is just changed to a new value, but in most cases, tree structure must also be changed; this may place severe demands on the cell interconnection structure.

3.2 Physical Connection Structure

It seems clear that the cells inside a node should be physically arranged in a regular pattern so as not to waste silicon area. First, note that we need only map binary trees to the physical level; in fact, the RRM compiler can always reduce more complex tree structure to binary tree tructure, by introducing several auxiliary binary operations to represent a single operation having more than two arguments at the logical level.

We are still debating what underlying physical structure would best implement the logical tree structure. We will use extensive simulation to explore the following major alternatives:

- Physical structure mapping puts logically related tokens into strictly adjacent array locations.
- Virtual structure mapping places tree tokens at arbitrary array positions and provides non-local communication channels for tree connections.
- Hybrid structure mapping embeds a logical tree into some physical structure having richer connectivity.

These options are discussed in more detail in the subsections below.

3.2.1 Physical Structure Mapping

In a physical structure mapping, directly connected tree tokens must be placed in adjacent array locations. Two of many possible mapping schemes are

- the H-tree and
- *the X-tree.*

Mapping trees onto the physical interconnection pattern provides very fast intercell communication during pattern matching. On the other hand, structure changes (resulting from tree rearrangements) are very costly. Just moving a subtree to a different location requires a very high bandwidth[3]. We are investigating the possibility of using regular arrays with additional connections to support tree reshaping.

3.2.2 Virtual Structure Mapping

In a virtual structure mapping the physical placement of tokens is immaterial, but connected tokens must use an available communication channel to exchange data. Possible implementation solutions are:

[3]Tree rearrangement potentially occurs after each match, so the two operations should take about the same time.

- Intervening cells route token messages to the correct destination. Since the array is reasonably small, the routing overhead is not insufferable. However the space overhead (logic gates) required to provide this function in each cell may be unjustifiable.
- A *switching network* within the node allows direct connection between any two cells. A switching networks may however be too space consumming for a node implemented on a single VLSI chip.
- Some form of *bus* could also be used, but this would sharply reduce the concurrency, because of communication conflicts.

Each of these schemes can easily accomodate tree reshaping. However, during pattern matching the non-local communication channels introduce delays which translate into performance penalties. Also, *routing blockage*[4] could reduce the degree of concurrency.

3.2.3 Hybrid Structure Mapping

Hybrid solutions are also possible. Here a tree logical structure is embedded into a physical structure which has richer physical interconnections than trees, in order to facilitate reshaping. One example of such a physical structure is the hexagonal tesselation of the plane. One disadvantage of this approach is that typical embeddings may have many empty cells. Another is that publicly available VLSI design systems do not support the non-rectilinear layouts that would be most advantageous.

3.3 Rule Execution

The node controller broadcasts commands to all cells inside its chip. We expect rules to be already compiled into sequences of simple microoperations. Due to limited storage capabilities, only active rules will be kept inside a node. This raises issues similar to those associated with "working sets" for von Neumann virtual memory architectures.

Another difficulty is the need to avoid decisions based on cell status. As much as possible, the control sequence should avoid global branches that depend on local data. This is because, if many cells reply, the controller becomes a bottleneck in processing the test results. Much more work is needed in both defining the architecture at this level and in devising efficient compiler techniques.

3.4 Loading Trees

Assuming a node is a single VLSI chip, its connections with the outside world are of very limited bandwidth. This may introduce a bottleneck when transfering large trees in and out of the node. Fortunately, in most cases a node starts with just a few tokens, which expand until a reduced result is obtained; only rarely is it necessary to load and unload large subtrees into a node.

[4]This occurs when two or more messages are routed through the same intermediate cell or bus, where they contend for limited retransmission resources to the destination cell.

Another problem at the node level is loading rule sets. Hopefully one can start with just a very few rules, and then load the rest while the chip works with the initial ones. The compiler should be smart enough to order rules for loading according to the highest probability of a successful match; [5] gives some suggestions on how this could be done.

4 Cluster Architecture

This architectural level is concerned with problems of coordinating many rewrite nodes, including

- node communication needs,
- node interconnection structures, and
- rule distribution methods.

In addition to a quite large number of nodes, a cluster also contains backup memory and facilities for connecting to a minicomputer storing the original total rule set.

4.1 Interaction Between Rewrites

RRM nodes are relatively independent entities. Since the resources available at each node are limited, nodes must cooperate to solve non-trivial problems. The tree being rewriten can thus span several nodes, and the pattern matching algorithm must take this into account.

It is of great importance to partition the tree being rewritten in such a way that internode communication is reduced. The partitioned concurrent rewriting computation model [5] will be used to discover suitable tree "articulation points"[5] where activity should be continued onto another node. To complicate matters further, the underlying tree is a very dynamic object, and could expand or shrink greatly as the result of even one successful rewrite.

4.1.1 Split-Pattern Matching

Two nodes containing adjacent tokens must cooperate in the pattern matching process. The worst conceivable situation is that of *split-pattern matching*, which occurs when a pattern must match a subtree located across a node communication link. The problem is that each node is independently (and asynchronously) controlled, so that negotiations to determine the success of matching a pattern across nodes will significantly slow down execution.

4.2 Communication Methods

Since we assume that nodes are each on a VLSI chip, internode communication is very slow compared with operations inside nodes. Still, fast node control dialogs are needed for split-pattern matching. The major options are *duplicated memory* and *message passing*.

[5] A tree token that is predicted to generate a lot of rewrite activity unrelated to the tokens above it in the tree is called an articulation point.

4.2.1 Duplicated Memory

The duplicated memory method of node communication allows a few cells in each of the two nodes to contain duplicated information, so that pattern matching is not hindered by node borders. This comes at the expense of more complex node coordination to avoid inconsistency. Following a tree rewrite involving the duplicated region, the node must make sure that the change is also recorded in the other node storing the duplicated subtree. There is also the unpleasant possibility that two nodes might modify different copies of the same data simultaneously, so that one of the two changes must be undone.

4.2.2 Message Passing

The message passing method relies on nodes sending matching requests and result information in messages to other nodes. This requires more complex pattern matching operations:

- A node initiates a split-pattern match by successfully matching its part of the tree. It then suspends its operation and sends a request message to the node holding the rest of the subtree.
- This second node must match the requested pattern fragment against the local subtree cells. If the partial match is successful, it must also prevent any changes from being done in this region until the requestor node completes the operation.
- Upon a successful reply, the first node may proceed with the replacement.
- The replacement could again involve the second node. In this case the first node would issue a rewrite request. In the worst case there might even be a need to redistribute tokens between the two nodes.
- Only after the whole activity has ended can the second node unlock the data subtree on which the fragment pattern match succeeded.

4.3 Node Connectivity

Another important issue is *cluster topology*. We need a node interconnection structure that will support the underlying tree topology. As the number of nodes in a cluster is quite large, we should try to avoid a fully general routing facility, since it will be slow, or else very expensive, or even impossible. To take advantage of the locality of interconnection that is characteristic of tree-structured data, we are considering a variety of interconnection structures, including a hexagonally tesselated sphere (or torus) and a hypercube.

4.4 Load Management

Simulation results [2] show that some problems may display an exponential increase in cell demands. Some of these could be satisfied by going to nodes in outside clusters, but in general such a problem can overwhelm any available resources. A node that needs

to send part of its activity to another node must find a free node that can be accessed cheaply. Otherwise the activity request must wait for some working nodes to become free. This could be achieved by using the backup memory to store a "waiting" request instead of directly forwarding it to a node. The cluster controller manages available resources and decides which waiting requests to dispatch to free nodes.

It is also important to manage expensive connections. The controller must keep track of available node and connection resources and find the best way to resume a waiting activity. Dually, the controller could also decide to put a request on wait, even if there are free nodes, in order to keep available nodes connected to other very active nodes.

4.5 Rule Distribution

Rule sets are compiled and distributed from a central minicomputer. Since we envision a large number of RRM nodes, each with a different active rule set, the architecture must avoid the minicomputer bottleneck. One solution currently being explored is the *inheritance of rule sets*. Preliminary simulation results show that when a node spawns another active node, its rule set is often relevant to that new node. Therefore the rule memory can be copied from the old node, instead of waiting for the minicomputer; only rules that are specific to the new node need to be requested from the central minicomputer.

5 Network Architecture

Several clusters participate in a network for solving larger problems. Tokens are known outside clusters by long names, since the 8 to 10 bits used inside a cluster will not be sufficient to identify all tokens participating in a larger problem. This requires translations whenever two cluster interact.

Due to technological limitations, we must also assume that inter-cluster connections are very slow compared to intra-cluster connections. We envision a reasonably small number of clusters in the network, so that a general-purpose interconnection switch would be appropriate.

As a consequence of the limitations described above, split-pattern matching should not be allowed accross networks.

At this level we can also study higher grain concurrency problems, such as executing several reduction problems simultaneously. Another network level problem is providing microcomputer storage and retrieval for large data sets, for example, the symbol tables that might be needed by a compiler running on the RRM. The implementation of objects, as needed for FOOPS [4], is another issue at this level.

6 Modularity Features

The proposed RRM organization combines two concurrent architectural paradigms:

- Cells within a node share a single controller, and therefore operate in SIMD mode.

- Nodes are independent MIMD processors that cooperate by exchanging messages or by maintaining duplicated data.

The combination of SIMD and MIMD allows both the speed advantage of cells executing rewrites in lock-step, as well as the flexibility of independently controlled processors.

6.1 Scalability

A particular advantage stemming from independent execution at the node level is *scalability*. In the proposed RRM architecture, whenever existing resources are exhausted by a particular rewrite activity, some nodes may have to stop and wait for other activities to finish. Requests to spawn new nodes are kept in the backup memory for later processing.

If the node interconnection structure is a regular tesselation (e.g., a sphere or torus), more nodes can be added just by expanding the surface, and these additional nodes could take care of pending node generation requests. Scalability at the network level is even more appealing. Assuming a general interconnection network among clusters, new resources can be added by just "dropping in" a new cluster.

Thus, a computation which can use more resources than currently available can improve its performance by a factor proportional to the number of additional nodes. Of course, the node interconnection structure should allow unlimited embedding of new tree connections into the existing node communication channels, but this does not seem especially challenging given the inherent limited connectivity requirements of trees.

6.2 Fault Tolerance and Reconfigurability

Another important issue is *fault tolerance*. In common with most MIMD architectures, the RRM can tolerate faulty nodes, since the node connection structure provides more than sufficient connectivity. More relevantly, the concurrent rewriting computational model can survive failed nodes simply by re-requesting action from other healthy nodes. This feature, common to functional programming languages, assures that no partial computation result can be lost due to some nodes failing to deliver their results. Again, backup memory will be used to store requests to spawn new nodes. For fault tolerance, all node requests should be logged, not just the ones waiting for free nodes. Then the cluster controller can restart any failed computation from the backup request information.

7 Estimated Performance

Although it is early to evaluate the RRM architecture, some educated guesswork is possible, based on various assumptions.

7.1 Technological Assumptions

The RRM is to be built from custom VLSI chips, and each rewrite node is contained on a single chip. A cluster encompasses several rewrite nodes plus additional memory and control. Such a cluster can be placed on a single printed circuit board, also including

access to a minicomputer storing the total rule set. Since we plan initial experimental chip designs using DARPA's MOSIS facilities, several constraints are more or less clear:

- Power consumption demands CMOS technology; previous MOSIS experience shows that the largest standard chip can accomodate up to about 250,000 transistors.
- Standard sized printed circuit boards could accomodate about 128 custom VLSI chips, still leaving enough space for backup memory and a general purpose microcomputer.
- The custom VLSI chips could operate at 20 Mhz, since there are no complex or lengthy operations involved in tree rewriting.

7.2 Computational Resources

As already noted, tokens have 8 to 10 bits each. Each cell stores one token and can perform normal arithmetic-logic operations on it, as well as comparing its token, against a broadcast pattern.

- A reasonable guess at cell complexity yields about 200 logic gates, corresponding to approximately 1,500 transistors for each RRM cell.
- Assuming an efficient solution to the cell connectivity problem, one could pack 128 cells in each chip and still leave enough space for the node controller.
- One could build each cluster on a single printed circuit board, so that each cluster will contain 128 nodes.
- A reasonable demonstration RRM network could contain 16 such clusters connected through a general-purpose switch.

This discussion sets a target of 128 rewrite cells per node and 16,384 cells per RRM cluster. The proposed RRM network will then contain 16 clusters for a total of about 256 thousand cells.

7.3 Computational Power

For estimating the cell execution speed, let us assume that:

- A rewrite, consisting of a pattern match and a subtree replacement, takes an average of 5 clock cycles.
- As the clock cycle is 20 Mhz, each cell will be able to perform 4 million rewrites per second. Considering that a rewrite is the RRM equivalent of an instruction, each cell is assumed to have a computational power of 4 MIPS (million instructions per second).

The total computational power for the prototype RRM is then (by straight multiplication of the cell power and the available number of cells) *one million MIPS!*

7.4 Discussion

Of course, this result is utterly misleading[6]. First of all, only a fraction of the pattern matches will succeed, so most cells will not actually perform replacements. Second, internode communication will slow down the computation whenever a pattern is matched across a border between chips. Third, the elementary instruction is considered to be a single replacement, which could be simpler than the rules in a real program, due to compilation.

However, the performance estimate of one million MIPS is still valuable. If unsuccessful matches and communication problems permit only 0.75% of matches to yield replacements[7], the RRM prototype would still have a performance of 8,000 MIPS. Although more simulations are needed to refine these figures, it seems clear that the amount of concurrency available in the RRM promises a tremendous breakthrough in performance.

It is also interesting to compare this performance with the measures of *Logical Inferences per Second (LIPS)* used in "Logic Programming Machines". Roughly speaking, an attempt to match a pattern in the RRM is equivalent to an attempt to unify two clauses, i.e., to one logical inference. Since LIPS do not count failed unification attempts, we should not use our pessimistic approach to performance evaluation. As a first approximation we could also omit performance penalties due to split-pattern matching; therefore, in a fair comparison with Prolog architectures, the RRM architecture should be rated at one million LIPS.

The Connection Machine [6], with *incredibly high MIPS performance estimates*, is a modern concurrent architecture currently enjoying great popularity. The RRM is expected to achieve similar degrees of concurrency and MIPS performance. In comparing these two architectures, it is important to note that (as far as we can tell) Connection Machine instructions manipulate one-bit operands, whereas RRM cells manipulate whole bytes. If this is the case, allowing for overhead while the Connection Machine propagates bits, one MIPS on the RRM should be worth at least 10 MIPS on the Connection Machine.

8 Architectural Testbed

Because of the novelty of the RRM architecture, it would seem wise to spend considerable time thinking and experimenting before finalizing the architecture and building a prototype.

The architectural levels of the RRM organization should be treated in a unified testbed, able to model mixtures of description levels. This has the advantage of allowing the independent exploration of implementation decisions by combining components in various stages of development.

---

[6] However, it is no more misleading than the usual computer manufacturers' use of MIPS performance estimates.

[7] This is a very pessimistic assumption - it means that on the average only one cell within each node performs a replacement.

8.1 Simulation Requirements

The large decision space motivates some requirements on a simulation facility for the RRM project. The following features are considered critical for a testbed environment to evaluate RRM architectures:

- Modularity is needed to facilitate experimenting with small model changes.
- Our multiple levels of decision need multiple levels of representation and simulation.
- Model development requires separating tasks into
  1. module functionality,
  2. module interconnection structure, and
  3. control and coordination of module activity.
- Easy to use graphics interface.

We will carefully evaluate existing simulators, such as SARA, Palladio, and N.2, but it seems likely that none will satisfy our requirements. Therefore, part of our effort should be devoted to building a simulator that can cope with our rich decision space.

8.2 Model Components

Simulation models are basically composed of:

1. interconnections allowing element communication,
2. functional description for each element in the architecture, and
3. means for controlling the timing of element interactions.

8.2.1 Interconnections

Ideally one should be able to specify graphically the topology of the modeled system. Simulated modules should be represented by boxes with ports for allowable interactions, and connected lines should show admissible communication paths.

8.2.2 Functionality

Each simulated component should be described in a high level language. As more architectural details are added, the description would be refined to include the new details. Some form of element "version control" will probably be needed to make sure that the history of refinements is preserved.

Another possibility is replacing element functionality by actual hardware. As we envision designing some experimental VLSI chips, we propose to insert these into the simulator. The advantage will be two-fold: the hardware prototypes will be thoroughly debugged, and the simulation will get a tremendous boost in speed.

8.2.3 Coordination

The issue of controlling when simulated components may exchange information is more complex. Since the functional description of each component does not reflect the actual delays involved in execution, special care must be taken to ensure that the occurence of simulated events reflects their real order. The following synchronization methodology is proposed:

- Elements may define rendezvous points.
- An element reaching such a point waits until the coordination constraints have been satisfied.
- To decouple the functional definition of elements from this synchronization problem, the ability to insert rendezvous points in an element's functionality will be needed.
- For simulation efficiency one needs low overhead implementation. For example, implementing rendezvous checks at each clock cycle will be overly expensive.

8.3 Simulation Development

Since we are dealing with an extreme level of concurrency, the simulation model should be developed gradually. Once a working model is achieved we will start changing its parameters and collecting the results.

8.3.1 Development Strategy

We suggest the following approach to building a model of a concurrent architecture from scratch:

- First attempt to achieve correct serial functionality.
- Then gradually start to insert concurrency by adding rendezvous points while maintaining functionality.
- After the model works correctly in a concurrent simulation, modify the architectural parameters and check the effects.

8.3.2 Result Collection

We expect to generate a large amount of data showing:

- critical states in the simulated cells and the data tree,
- a record of rendezvous situations together with the cells involved,
- a record of cells executing concurrently (together with what operations were performed), and
- statistics and patterns of interconnection use.

9 Summary

From an architectural viewpoint, the RRM provides a unique opportunity to execute ultra high level programs with unprecedented concurrency. In contrast to most current investigations of advanced computing structures, the RRM project is a symbiosis of software and hardware concerns. Therefore we expect to avoid the sad outcome of a "concurrent computer architecture in search of suitable problems" or its dual, a "modern programming paradigm that is woefully slow on traditional von Neumann computers." Our group is excited by the prospect of a long-awaited breakthrough in concurrent computation.

References

[1] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud and José Meseguer. Principles of OBJ2. In *Proceedings, Symposium on Principles of Programming Languages*, page 52, 1985.

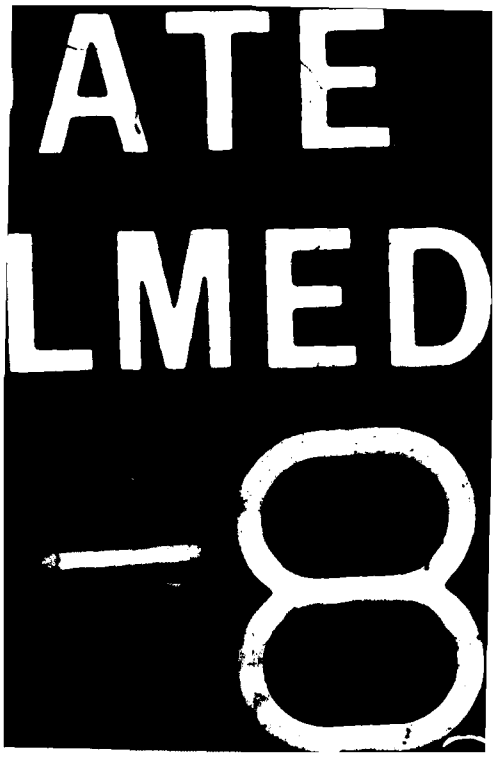
[2] Timothy Winkler, Sany Leinwand and Joseph Goguen. *Simulation of Concurrent Rewriting for the Rewrite Rule Machine.* In preparation, SRI International, 1986.

[3] Joseph Goguen and José Meseguer. Eqlog: Equality, Types and Generic Modules for Logic Programming. In *Functional and Logic Programming*, Doug DeGroot and Gary Lindstrom, editors, page 295, Prentice-Hall, 1986.

[4] Joseph Goguen and José Meseguer. *Object-Oriented Programming as Reflective Equational Programming.* In preparation, SRI International, 1986.

[5] Joseph Goguen, Claude Kirchner and José Meseguer. *Models of Computation for the Rewrite Rule Machine.* In preparation, SRI International, 1986.

[6] W. Daniel Hillis. *The Connection Machine.* MIT Press, 1985.


ATE
LMED
-8