Decomposition Abstraction in Parallel Rule Languages

Shiow-yang Wu, Daniel P. Miranker, James C. Browne

Abstract— **Decomposition abstraction is the process of organizing and specifying decomposition strategies for the exploitation of parallelism available in an application. In this paper we develop and evaluate declarative primitives for rule-based programs that expand opportunities for parallel execution. These primitives make explicit, implicit relations among the data and similary among the rules. The semantics of the primitives are presented in a general object-based framework such that they may be applied to most rule-based programming languages.**

We show how the additional information provided by the decomposition primitives can be incorporated into a semantic-based dependency analysis technique. The resulting analysis reveals parallelism at compile time that is very difficult, if not impossible, to discover by traditional syntactic analysis techniques. Simulation results demonstrate scalable and broadly available parallelism.

Keywords—**decomposition abstraction, parallel production systems, multiple rule firing systems, parallel decomposition, explicit parallel languages, decomposition mechanisms, set-oriented constructs, semantic-based dependency analysis, functional dependency.**

I. Introduction

Production systems, also known as rule-based systems or simply rule systems, have been shown to be a powerful architecture for intelligent systems [1], [2]. Initial implementations of production systems suffered from poor performance which prohibited their use in large scale applications [3]. Nevertheless, applications of rule-based programming have continued to expand. Recent interest in data intensive rule-based applications [4] has further fueled the need for high performance execution environments for production systems.

Intuition suggests that languages based on the production system model admit a high degree of parallelism [5]. Efforts to exploit parallel processing to increase production system performance have been ongoing for over a decade [6], [7], [8]. However, the maximum speedup achieved by actual implementation rarely exceeds tenfold and has never done so over a general suite of applications no matter how many processors are used.

Decomposition abstraction is the process by which programmers specify decomposition strategies for the exploitation of parallelism embodied by an application. Most parallel programming languages provide constructs for decomposition. This is almost a must because parallelizing compilers for sequential languages can not completely resolve name ambiguity and dependencies at compile-time. Numerous examples of language constructs for parallel decomposition can be found in both imperative and declarative parallel languages. The PARTITION statement in Refined Fortran [9], the DECOMPOSITION and related statements in Fortran D [10], the Ada tasking mechanism [11], the `process` type in Concurrent C [12], the `pcall` and `future` in Multilisp [13], and the notions of *blackboard* and *theory* in Shared Prolog [14], just to name a few, are all examples of language mechanisms or constructs for data or function decomposition to facilitate the expression of parallelism.

The need for decomposition abstraction has not been well recognized by the research community of production system languages [15], [16], [17], [7], [18], [19]. In a recent paper [20], we analyzed several commonly used benchmark programs and pointed out that the rather limited success in the past was primarily due to the failure to properly exploit the parallelism inherent in the application. Contrary to the conclusions drawn by previous work that the true performance gain from parallelism is quite limited [21], we showed that massive and scalable speedup was indeed achievable.

In this paper we develop methods of decomposition abstraction suitable for rule-based programming. The methods are declarative. Further, the methods capture intrinsic properties more closely related to the application than to issues of parallel computing. Thus we say that the abstraction mechanisms capture, normally unexpressed, semantic properties of the application. After detailing the semantics of these primitives we develop a semantic-based analysis technique for detecting, at compile-time, rule interference such that, at run-time, multiple instantiations of rules may be executed concurrently. Simulation results demonstrate that decomposition abstraction provides us with a key to scalable and massive parallelism in production systems.

The main results in this paper can be summarized as follows.

- A new general formulation of production systems in an object-based framework together with a rule notation that abstracts away unnecessary details while characterizing all essential features of production systems.
- A set of declarative decomposition abstraction mechanisms for rule languages.
- A semantic-based interference analysis technique.
- Simulation results integrating a full implementation of the communication synchronization aspects of the system on a Sequent Symmetry multiprocessor.
- Analysis of effective implementation strategies based on the experimental results.

Shiow-yang Wu is with the Institute of Computer Science and Information Engieering, National Dong Hwa University, Hualien, Taiwan, ROC. E-mail: `showyang@csie.ndhu.edu.tw`.

Daniel P. Miranker is with the Department of Computer Sciences, University of Texas at Austin. E-mail: `miranker@cs.utexas.edu`.

James C. Browne is with the Department of Computer Sciences, University of Texas at Austin. E-mail: `browne@cs.utexas.edu`.

II. A General Object-Based Framework of Production Systems

The principles of decomposition abstraction are language independent. Thus, we develop a general object-based framework within which to define the decomposition mechanisms.

A. Production Systems

We review the structure and operation of production systems. This will serve to introduce both the terminology used throughout this paper and the essential features that must be accounted for when we develop our object-based framework. More information about production systems in general and about the OPS5 production system language in particular, can be found in [22], [23].

A typical production system is composed of three components: a data store called *working memory*, a set of *rules*, and an *inference engine*. Working memory is a global database composed of data objects called *working memory elements* (WME's) representing the state of the system. A rule embodies a condition-action pair. The inference engine implements the three-phase cyclic execution model of matching, conflict-resolution and firing, which is also known as the *recognize-act* cycle. In a cycle, the conditions of each rule are matched against the working memory. A rule with a set of WME's matching the conditions is called an *instantiation*. The set of all instantiations constitutes the *conflict set*. In a sequential environment, conflict-resolution aims to select one instantiation from the conflict set for firing. In a parallel environment, multiple rule instantiations can be selected for firing simultaneously subject to proper correctness constraints such as serializability [19]. Firing an instantiation simply means to execute the actions which may add, delete, or modify WME's in the working memory. The cycle repeats until no rule can be fired, i.e. no instantiations are computed by the match.

B. Object Model and the Abstract Rule Notation

Our object model is inspired by [24], [25] and is comprised of the following sets of symbols:

- $\mathcal{A}$: attribute names,
- $\mathcal{C}$: class names,
- $\mathcal{I}$: identifiers,
- $\mathcal{M}$: method names,
- $\mathcal{R}$: rule names,
- $\mathcal{V}$: variable names.

DEFINITION **1 (Methods)** *A* **method definition** *is a triple (M, P, B) where M is a method name, P is a set of parameter specifications, and B is the definition of operations performed by the method (usually called the* **body** *or* **implementation** *of the method). A* **method invocation** *is a method name with necessary parameters fully supplied.*

We have deliberately left out the details of how a parameter or body of a method is actually specified. No restriction is placed on the way actual arguments are passed in a method invocation. These issues are not essential to our discussion and thus our results are independent of any specific method definition or invocation mechanism.

DEFINITION **2 (Classes)** *A* **class** *defines a set of objects with similar structure and behavior.*

- **INT**, **FLOAT**, *and* **STRING** *are* **primitive classes** *representing the set of integers, floats, and character strings, respectively.*
- *An* **attribute definition** *is a pair (a, C) where a is an attribute name and C is a class name.*
- *A set-valued attribute can be defined by adding a "*" at the end of an attribute name.*
- *A class is a triple (C, A, M) where C is a class name, A is a set of attribute definitions, and M is a set of method definitions.*

In terms of our abstract rule notation, a class C with attribute definitions $(a_1, C_1), \ldots, (a_n, C_n)$ and method definitions $M_1, \ldots, M_k$ is defined as follows.

> **class** C {
>
> **attributes** ($a_1 : C_1, \ldots, a_n : C_n$)
>
> **methods** ($M_1, \ldots, M_k$)
>
> }

The sets of attributes and methods of C are denoted by $A(C)$ and $M(C)$ respectively.

DEFINITION **3 (Objects and WME's)** *Objects are defined to model WME's. They are the basic units of information and behavior encapsulation. Inheritance is not considered since it is not an essential part of the production system model.*[1]

- *Integers, floats, and character strings are* **primitive objects**.
- *If $a_1, a_2, \ldots, a_n$ are the attribute names of a class C and $O_1, O_2, \ldots, O_n$ are objects, then:*
  $$O = (\ a_1 : O_1,\ a_2 : O_2,\ \ldots,\ a_n : O_n\ )$$
  is a **structural object**. *The object is an* **instance** *of the class C.*
- *Objects are the generalization of WME's. Each object has a unique identifier associated with it.* **Working memory** *is a set of objects.*
- *If $O_1, O_2, \ldots, O_n$ are objects of a class C, then*
  $$\{\ O_1, O_2, \ldots, O_n\ \}$$
  is a **set object**. *O_i's are* **elements** *of the set object.*

Note that the identifier of an object must be globally unique.[2] Objects with identical set of attribute values are treated as different objects because of the unique identifiers. Working memory is therefore a set rather than a multiset. Also note that elements of a set object must be instances of the same class.

DEFINITION **4 (Rules)** *A* **rule** *has a triggering condition and an action component. Conditions may be positive or negative.*

[1] However, our formalization is general enough to be extended later to include inheritance.

[2] The time tag of a WME in OPS5 is an example of a globally unique identifier.

- *An* **expression** *is a quantifier-free first order formula which may contain side-effect free method invocations.*
- *If v is a variable name, C is a class name and E is an expression, then $(v : C :: E)$ is a* **positive condition** *and $-(v : C :: E)$ is a* **negative condition**.
- *If P is a condition, then $v(P)$, $C(P)$, and $E(P)$ denote the variable, class, and expression components, respectively, of the condition.*
- *A* **rule** *is a triple (P, N, M) where P is a set of positive conditions, N is a set (possibly empty) of negative conditions, and M is a sequence of method invocations.*[3]
- *A positive or negative condition is termed a* **condition element**. *The set of all condition elements is called the* **antecedent**. *The set of method invocations is called the* **consequent**.

Apparently a rule with an empty consequent has no effect, therefore M is usually non-empty. Since methods can only be invoked on objects selected by the positive conditions, P must be non-empty as well.

In the rule notation, if r is a rule name, $P_1, P_2, \ldots, P_n$ $(n \geq 1)$ are positive conditions, $N_1, N_2, \ldots, N_m$ $(m \geq 0)$ are negative conditions, and $M_1, M_2, \ldots, M_k$ $(k \geq 1)$ are method invocations, then a rule is defined as follows with $\rightarrow$ delimiting the antecedent and consequent.

$$
\begin{array}{l}
\textbf{rule } r \ \{ \\
\quad P_1, P_2, \ldots, P_n, \\
\quad N_1, N_2, \ldots, N_m \\
\rightarrow \\
\quad M_1, M_2, \ldots, M_k \\
\}
\end{array}
$$

DEFINITION **5 (Program and System)** *A* **program** *is a pair* **(C, R)** *where* **C** *is a set of class definitions and* **R** *is a set of rule definitions. A* **rule system** *is also a pair* **(O, P)** *where* **O** *is a set of objects and* **P** *is a rule program.*

C. Execution Model and Semantics

We specify the semantics of rules by considering rule antecedents as queries to the working memory for selecting a consistent set of objects. The execution of a rule system is defined in terms of state transitions between working memory states.

DEFINITION **6 (State)** *The* **state** *of a rule system is the set of objects in working memory.*

DEFINITION **7 (Selection and Instantiation)** *Pattern matching is modeled by object selection. The following definitions are defined assuming a given state S.*

- *A positive condition element $(v : C :: E)$ is* **satisfied** *in S if there exists an object of class C such that E is evaluated to true. The object (which can be referenced by the variable v) is said to be* **selected** *by the condition element.*
- *A negative condition $-(v : C :: E)$ is satisfied in S if there does not exist any object of class C such that E is evaluated to true.*
- *A rule is satisfied in S if there exists at least one set of objects in S such that all condition elements in the antecedent are satisfied. The set of objects selected by the positive condition elements is called an* **instantiation** *of the rule.*

Formally, a rule as defined in Definition 4 is satisfied if the following formula is true.

$$
\begin{array}{l}
\exists\, v(P_1), \ldots, v(P_n)\ (\\
\quad v(P_1) \in C(P_1)\ \wedge\ \ldots\ \wedge\ v(P_n) \in C(P_n)\ \wedge \\
\quad E(P_1)\ \wedge\ \ldots\ \wedge\ E(P_n)\ \wedge \\
\quad \nexists\, v(N_1), \ldots, v(N_m)\ (E(N_1)\ \vee\ \ldots\ \vee\ E(N_m)))
\end{array}
$$

Each set of n objects satisfying the formula is an instantiation of the rule. Note that it is possible for the same set of objects to satisfy the formula in more than one ways. All of them are instantiations of the rule.

Operationally, a rule can be considered as a query to the working memory. The result of the query is a class whose instances are instantiations of the rule. In other words, the set of all instantiations of a rule r, denoted $\mathbf{Inst}(r)$, can be formally characterized by the set

$$\mathbf{Inst}(r) \equiv \{\ t \mid t \in Inst\_of\_r\ \wedge\ A\theta\ \}$$

where $Inst\_of\_r$ is the class

$$
\begin{array}{l}
\textbf{class } Inst\_of\_r\ \{ \\
\quad \textbf{attributes}\ (v(P_1) : C(P_1),\ \ldots\ , v(P_n) : C(P_n)) \\
\quad \textbf{methods}\ (\ \bigcup_{i=1}^{n} M(C(P_i))\ ) \\
\}
\end{array}
$$

and θ is the variable substitutions and A is a formula representing the tests in the antecedent. That is,

$$
\begin{array}{rcl}
\theta & = & \{\ v(P_1)/t.v(P_1), \ldots, v(P_n)/t.v(P_n)\ \} \\
A & = & E(P_1)\ \wedge\ \ldots\ \wedge\ E(P_n)\ \wedge \\
 & & \nexists\, v(N_1), \ldots, v(N_m) \\
 & & \quad (E(N_1)\ \vee\ \ldots\ \vee\ E(N_m)).
\end{array}
$$

Note that the value of each attribute in the class is an object selected by the corresponding positive condition element, and the methods are the union of all methods that can be invoked on the selected objects.

DEFINITION **8 (Rule Firing)** *Let S be a state, r be a rule which is satisfied in the state, and i be an instantiation of r. The result of* **firing** *the rule instantiation is a new state S' obtained from S by invoking the methods in the consequent of r on the set of objects in i. We denote such a rule firing by $S' = S(i)$.*

[3]Note that it is possible to weaken the definition to allow concurrency among method invocations. Since our experiences show that this is not a primary source of parallelism, we choose to avoid the potential complexity and overhead by requiring methods to be invoked in sequence.

DEFINITION **9 (Execution)** *An* **execution** *of a rule system is a sequence of rule firings that transforms the system from a state to another state. A state is a* **terminal state** *if no rule is satisfied under that state. An execution is a* **terminal execution** *if the last state in the sequence of rule firings is a terminal state.*

It is important to note that in the definitions of rule firing and execution, no restriction is placed on how objects are selected or on which rule instantiation to pick. In other words, no matching technique or conflict resolution strategy is assumed. An execution is not required to be a terminal execution. This allows systems that do not terminate.

D. Parallel Rule Firing

We extend the model to allow simultaneous firing of multiple rule instantiations.

DEFINITION **10 (Interference)** *If* i_1 *and* i_2 *are instantiations of two (possibly the same) rules that are satisfied in a state* S*, then* i_1 **interferes with** i_2 *if any one of the following conditions is true:*

1. *The execution of* i_1 *prevents* i_2 *from being an instantiation in the new state resulting from* i_1*'s execution, or vice versa.*
2. *There exist methods invoked by* i_1 *and* i_2 *that modify the same object.*

Since a newly created object is always assigned a unique identifier, object creations do not contribute to any interference except when Condition 1 is true. Identical objects with different identifiers are allowed to coexist in our model, which is consistent with most rule languages. It is possible to weaken Condition 2 above since we need only to avoid conflicting methods to be invoked on the same object. However, such fine-grained parallelism can be easily overwhelmed by the potential complexity. We reserve this issue for future research.

DEFINITION **11 (Compatibility)** *Two instantiations are said to be* **compatible** *if they do not interfere with each other. A set of instantiations is compatible if the instantiations are pair-wise compatible.*

Since compatible instantiations do not interfere with each other, they can be executed in parallel. Our definitions of interference and compatibility are similar to the corresponding definitions in [15], [16], [17], [19] all of which originated from Bernstein's conditions [26] and database concurrency control theory [27], [28]. However, we formalize it in a general object-based context which allows any type of method instead of just the add, delete, and modify operations as in most previous work on parallel production systems.

DEFINITION **12 (Parallel Rule Firing)** *The result of* **parallel firing** *of two compatible instantiations in a state is a new state obtained by invoking all methods on corresponding objects of the two instantiations. Likewise, the parallel firing of a set of compatible instantiations* I *in a state* S *is to invoke all methods on corresponding objects of all instantiations. The parallel firing is denoted by* $S' = S(I)$*.*

Because of the non-interference requirement between parallel executable instantiations, the resulting state of the parallel firing is the same as the result of execution of the set of instantiations in sequence following any order. To state it more precisely, if $I = \{i_1, \ldots, i_n\}$ is a set of parallel executable instantiations in a state S, then

$$S(I) \;=\; S(i_{j_1})(i_{j_2})\ldots(i_{j_n})$$

where $j_1, j_2, \ldots, j_n$ is any permutation of n.

III. Abstract Mechanisms for Parallel Decomposition

In this section, we first present a systematic analysis of the types of parallelism in production systems to derive a set of criteria for assessing the quality of decomposition abstraction mechanisms. We then introduce a small but powerful set of language-independent abstract mechanisms for parallel decomposition followed by our semantic-based interference analysis technique. Actual constructs for parallel decomposition in any rule language can be designed by adopting these mechanisms.

A. Parallelism in Multiple Rule Firing Production Systems

When rule instantiations are allowed to fire in parallel, various opportunities for parallelism arise at different levels of the production system model. This analysis of parallelism is different from Gupta's analysis [5] in that we examine this issue from a semantic point of view. In particular, we focus on the patterns of computation and programming style that naturally map to the familiar notions of data and function decomposition. Unlike such notions like *node parallelism* from Gupta's analysis, our analysis is independent of any match or rule evaluation scheme.

Data Level Parallelism. In a sequential environment, WME's are processed one at a time. By *data level parallelism* we mean that different sets of WME's can be processed in parallel similar to the SPMD or data parallel systems [29], [30]. However, unlike other languages where data resides in regular data structures, this type of parallelism in the context of production systems usually manifests in the form of multiple (either all or subset of) instantiations of the same rule. Specifying this type of parallelism with data declarations is not likely to work since the same set of WME's may need to be processed sequentially for some rules while they may be fully decomposable for some other rules. We will show in later sections that declarations on a per rule basis turn out to be the most natural way for exposing this level of parallelism.

Rule Level Parallelism. This is the concurrency observed when instantiations of multiple rules are fired in parallel. It can be the result of both data and function decomposition depending on whether the rules are designed for similar or different functionalities. For exploiting this level of parallelism, the main issues are the possible interference between different instantiations and the correctness of parallel execution. Straightforward specification is inappropriate due to the burden and complexity of reasoning about concurrency placed on the programmers. The preferred way is to have the language system derive the concurrency, possibly with the help of a minimum amount of semantic information provided by the programmers, and maintain the correctness of parallel execution. We will show that relationships between data objects provide the key gateway to the exploitation of rule level parallelism in production systems.

Program Level Parallelism. Finally, a problem can often be decomposed into subproblems such that part or all of them may be processed in parallel. This corresponds to the program level parallelism where the structuring of program provides valuable hints for function decomposition. However, the basic production system model does not have any notion of modules or rule groups, which is certainly a disadvantage from this point of view. Mechanisms for rule structuring would certainly help the programmers in program development and the system in uncovering this type of parallelism.

B. Criteria

From the analysis of the types of parallelism presented in the previous section, we derive a set of criteria that must be met by any proper decomposition abstraction mechanism for a parallel rule language.

What vs. How. Since rule languages are declarative languages, the mechanisms must be declarative as well. Specify what type of decomposition naturally exhibits the parallelism in the application independent of how the decomposition is actually achieved.

Consistent with Pattern Matching Paradigm. The whole idea of production system is centered around pattern matching. The mechanisms for parallel decomposition must also be expressed under the pattern matching paradigm.

Conciseness. The mechanisms must be conceptually simple and intuitively appealing. The burden of reasoning about concurrency placed on the programmers should be minimized to the extent that only natural parallelism in the application semantics need be considered.

Versatility. The set of mechanisms must be semantically rich and powerful enough to express as many types of parallelism as possible.

Compatibility. The mechanisms should be compatible with sequential semantics. All sequential programs should still run correctly. Parallel programs should be able to run correctly even if executed sequentially.

Effective Implementation. Any set of mechanisms for parallel decomposition should be effectively implementable.

C. Parallel Structuring Mechanisms

Following the design criteria, we propose a set of abstract mechanisms for expressing decomposition strategies. The formal semantics of the mechanisms are specified under our object-based model with illustrative examples using the abstract rule notation.

Set Selection Condition

In most (if not all) sequential rule languages, each positive condition element matches a single data object from a specified class. Then actions in the consequent are applied on the instantiation composed of selected objects, one from each positive condition element. This implies that only one object from each class can be processed at a time. On the other hand, it is quite natural for an application to adopt a basic problem solving strategy such that all objects satisfying certain conditions in a specified class need to be processed. This apparent mismatch between the language model and the application semantics is almost always circumvented by firing the same rule repeatedly until all qualified objects have been processed. For example, the rule

$$
\begin{array}{l}
\textbf{rule}\ \mathit{Raise\_Poor\_Employee}\ \{ \\
\quad (\ d : \mathit{Department}\ ), \\
\quad (\ e : \mathit{Employee} :: \ e.\mathit{dept} == d.\mathit{name} \wedge \\
\qquad\qquad\qquad\quad e.\mathit{salary} < 10000\ ) \\
\rightarrow \\
\quad e.\mathit{salary} = e.\mathit{salary} + e.\mathit{salary}/10 \\
\}
\end{array}
$$

will fire repeatedly on each "poor" employee in all departments to raise his/her salary by 10%. This type of rule can be found in almost all rule programs. If each employee belongs to exactly one department, the rule actually represents a perfect case of DOALL loop [31] in which parallelism can be fully exploited. On the other hand, this is not at all obvious for many parallelizing compilers of sequential rule languages since the possibility of interference can not be ruled out at compile-time. If the application semantics implies that no interference can occur, then there is no reason to be so conservative. What we need here is a mechanism for specifying the exact semantics of the application as to whether a rule is to be applied on all or just the selected object one at a time.

For achieving the purpose above, we found that enriching the semantics of the rule antecedent to allow a positive condition element to match not just one but all satisfying objects solves the problem naturally and elegantly. Using the notation and the example above, the rule below specifies that for a department d, select all poor employees and raise the salary of each one of them by 10%.

$$
\begin{array}{l}
\textbf{rule}\ \mathit{Raise\_All\_Poor\_Employees}\ \{ \\
\quad (\ d : \mathit{Department}\ ), \\
\quad \textbf{[}\ e : \mathit{Employee} :: \ e.\mathit{dept} == d.\mathit{name} \wedge \\
\qquad\qquad\qquad\quad e.\mathit{salary} < 10000\ \textbf{]} \\
\rightarrow
\end{array}
$$

$e.salary = e.salary + e.salary/10$

}

A positive condition element enclosed in square brackets is a *set selection condition* denoting that all qualified objects should be processed by the consequent and that they can be processed independently. In other words, the selected set of data objects is fully decomposable and can be processed in parallel.

The square bracket notation is adopted from [32] for its conciseness. However, the semantics is rather different. A relational semantics was taken in their set-oriented constructs to facilitate the integration with database systems. Our set selection condition is a mechanism for expressing parallelism. The implication that objects in the selected set can be processed in parallel is not in their relational semantics.

Formally, a rule r with positive conditions $P_1, \ldots, P_i$, set selection conditions $P_{i+1}, \ldots, P_n$, and negative conditions $N_1, \ldots, N_m$, defines the following class.

$$
\begin{array}{l}
\textbf{class } Inst\_of\_r \; \{ \\
\quad \textbf{attributes} \; (\; \begin{array}[t]{l} v(P_1) : C(P_1), \\ \vdots \\ v(P_i) : C(P_i), \\ v(P_{i+1})* : C(P_{i+1}), \\ \vdots \\ v(P_n)* : C(P_n) \;) \end{array} \\
\quad \textbf{methods} \; (\; \bigcup_{i=1}^{n} M(C(P_i)) \;) \\
\}
\end{array}
$$

Each instance of the class represents a *set instantiation* composed of objects (one from each regular condition) and set objects (one for each set selection condition). Set instantiations are, as the name suggests, representations of sets of *ground instantiations*, which are instantiations of the *ground rule* of r obtained by treating all set selection conditions as regular conditions. The set of all ground instantiations can be characterized by the set

$$
\begin{array}{l}
\{ \; (g_1, \ldots, g_n) \mid \exists \, t \; (\\
\quad t \in Inst\_of\_r \;\; \wedge \\
\quad g_k = t.v(P_k), \; 1 \leq k \leq i \;\; \wedge \\
\quad g_k \in t.v(P_k)*, \; i+1 \leq k \leq n \;\; \wedge \\
\quad A\theta \;) \; \}
\end{array}
$$

in which

$$
\begin{array}{rcl}
\theta & = & \{ \; v(P_1)/g_1, \ldots, v(P_n)/g_n \; \} \\
A & = & E(P_1) \; \wedge \; \ldots \; \wedge \; E(P_n) \; \wedge \\
 & & \nexists \; v(N_1), \ldots, v(N_m) \\
 & & \quad (E(N_1) \; \vee \; \ldots \; \vee \; E(N_m)).
\end{array}
$$

We note that when more than one set selection condition coexists in a rule, a *join* semantics similar to a relational join is implied. This is consistent with sequential semantics in that the set of ground instantiations is exactly the same as that of the rule with set selection conditions treated as regular conditions. The difference is that the former is fully parallel decomposable while the later can only be processed one at a time.

Aggregate Operators

The set of objects selected by a set selection condition can also be processed as a whole by *aggregate operators* such as **count**, **sum**, **max**, **min**, and **avg**. This provides a new dimension of language constructs that greatly simplify rule-based programming. The fact that efficient parallel algorithms can be used to implement these operators further increase the value of set selection mechanism in rule languages.

Using our abstract rule notation, in a set selection condition $[\; v : C :: E \;]$, v denotes an individual and $v*$ the whole set of selected objects, respectively. For example, if the number of poor employees in a department is desired, the following rule does exactly what we want.

$$
\begin{array}{l}
\textbf{rule } Count\_Poor\_Employees \; \{ \\
\quad (\; d : Department \;), \\
\quad [\; e : Employee :: \; \begin{array}[t]{l} e.dept == d.name \; \wedge \\ e.salary < 10000 \;] \end{array} \\
\rightarrow \\
\quad d.poor\_emps = \textbf{Count}(e*) \\
\}
\end{array}
$$

Without set selection conditions and aggregate operators, the same effect would require two rules, where one rule fires repeatedly on each poor employee to increment a counter and another semantically redundant rule is used purely for testing whether all poor employees have been counted.

ALL Combinators

Examining the *Count_Poor_Employees* rule above reveals an additional, unexploited level of parallelism. Using only the set selection condition and aggregate operators, all poor employees in a department can be processed as a whole using an efficient parallel algorithm. However, different departments are still processed sequentially. To specify that both conditions are set selection conditions does not work since the join semantics would mean that the total number of poor employees in the company, not any department, is set to the *poor_emps* attribute of each department. Instead, a new mechanism is needed here to specify the intended patterns of decomposition among selected sets of objects. In the example above, we want to specify that not only poor employees in a department need to be considered, but that the same thing can be applied on all departments independent of each other. In other words, if the selected employees are decomposed or grouped by the department, then different groups of objects can be processed in parallel since no interference can occur.

For the purpose above, we found that a natural way to specify the desired semantics is to group several condition elements together which represents the desired patterns of decomposition. To illustrate this, the following rule specifies that the number of poor employees of each departments can be computed in parallel.

$$
\begin{array}{l}
\textbf{rule}\ Count\_All\_Poor\_Employees\ \{ \\
\quad \textbf{All}\ (\ (\ d : Department\ ), \\
\quad\quad [\ e : Employee :: \ e.dept == d.name \wedge \\
\quad\quad\quad e.salary < 10000\ ]\ ) \\
\rightarrow \\
\quad d.poor\_emps = \textbf{Count}(e*) \\
\}
\end{array}
$$

The **All** *combinator* groups together several condition elements into an **All** *condition* to denote that any consistent collection of objects and set objects (for set selection conditions) can be considered independent, and therefore all of them can be processed in parallel without worrying about interference.

Formally, a rule r with positive conditions $P_1, \ldots, P_i$, set selection conditions $P_{i+1}, \ldots, P_n$, negative conditions $N_1, \ldots, N_m$, and an **All** condition combining positive conditions $AP_1, \ldots, AP_j$, set selection conditions $AP_{j+1}, \ldots, AP_k$, and negative conditions $AN_1, \ldots, AN_l$, defines a class:

$$
\begin{array}{l}
\textbf{class}\ Inst\_of\_r\ \{ \\
\quad \textbf{attributes}\ (\ v(P_1) : C(P_1), \\
\quad\quad\quad \vdots \\
\quad\quad\quad v(P_i) : C(P_i), \\
\quad\quad\quad v(P_{i+1})* : C(P_{i+1}), \\
\quad\quad\quad \vdots \\
\quad\quad\quad v(P_n)* : C(P_n), \\
\quad\quad\quad all* : AllClass\_of\_r\ ) \\
\quad \textbf{methods}\ (\ \bigcup_{i=1}^{n} M(C(P_i))\ \cup\ \bigcup_{i=1}^{k} M(C(AP_i))\ ) \\
\}
\end{array}
$$

where $AllClass\_of\_r$ is the class

$$
\begin{array}{l}
\textbf{class}\ AllClass\_of\_r\ \{ \\
\quad \textbf{attributes}\ (\ v(AP_1) : C(AP_1), \\
\quad\quad\quad \vdots \\
\quad\quad\quad v(AP_j) : C(AP_j), \\
\quad\quad\quad v(AP_{j+1})* : C(AP_{j+1}), \\
\quad\quad\quad \vdots \\
\quad\quad\quad v(AP_k)* : C(AP_k)\ ) \\
\quad \textbf{methods}\ (\ \bigcup_{i=1}^{k} M(C(AP_i))\ ) \\
\}
\end{array}
$$

The set of all ground instantiations can be characterized by the set

$$
\begin{array}{l}
\{\ (g_1, \ldots, g_n, a_1, \ldots, a_k)\ |\ \exists\ t, u\ (\\
\quad t \in Inst\_of\_r\ \wedge\ u \in t.all*\ \wedge \\
\quad g_h = t.v(P_h),\ 1 \le h \le i\ \wedge \\
\quad g_h \in t.v(P_h)*,\ i+1 \le h \le n\ \wedge \\
\quad a_h = u.v(AP_h),\ 1 \le h \le j\ \wedge \\
\quad a_h \in u.v(AP_h)*,\ j+1 \le h \le k\ \wedge \\
\quad A\theta\ )\ \}
\end{array}
$$

where

$$
\begin{array}{rcl}
\theta & = & \{\ v(P_1)/g_1, \ldots, v(P_n)/g_n, \\
& & \quad v(AP_1)/a_1, \ldots, v(AP_k)/a_k\ \} \\
A & = & E(P_1)\ \wedge\ \ldots\ \wedge\ E(P_n)\ \wedge \\
& & E(AP_1)\ \wedge\ \ldots\ \wedge\ E(AP_k)\ \wedge \\
& & \nexists\ v(N_1), \ldots, v(N_m),\ v(AN_1), \ldots, v(AN_l)\ (\\
& & \quad E(N_1)\ \vee\ \ldots\ \vee\ E(N_m) \vee \\
& & \quad E(AN_1)\ \vee\ \ldots\ \vee\ E(AN_l)\ ).
\end{array}
$$

We note that there is no need to have more than one **All** condition in a rule. It is not difficult to prove that for a rule with multiple **All** conditions, the set of ground instantiations is exactly the same as the rule with all condition elements in each combinator placed under one **All** condition. In other words, if $C_1, \ldots, C_n$ are sets of condition elements, then

$$\textbf{All}(C_1) \wedge \ldots \wedge \textbf{All}(C_n)\ \equiv\ \textbf{All}(C_1 \wedge \ldots \wedge C_n).$$

On the other hand, while nested **All** conditions may seem to provide more expressive power than a single level one, they unnecessarily complicate the semantics. This is certainly against our conciseness criterion. We will show in later sections that the combination of set selection conditions, aggregate operators, and combinators (including the **Disjoint** combinator to be introduced next) is versatile enough to express all sources of data level parallelism discussed in Section III-A.

Disjoint Decomposition

The semantics of both set selection conditions and the **All** combinator imply that any consistent set of objects satisfying the conditions is a valid unit of decomposition and all such units can be processed in parallel. This is desirable when there is no worry about the repetition of selected objects between different sets as in all example rules above. However, when it is possible to have the same object selected to different units, the semantics above may not be exactly what we want as demonstrated in the following rule.

$$
\begin{array}{l}
\textbf{rule}\ Team\_Employees\ \{ \\
\quad (\ e1 : Employee :: \ e1.dept == \text{“}research\text{''} \wedge \\
\quad\quad e1.team == \textbf{unknown}\ ), \\
\quad (\ e2 : Employee :: \ e2 \ne e1\ \wedge \\
\quad\quad e2.dept == \text{“}research\text{''} \wedge \\
\quad\quad e2.team == \textbf{unknown}\ ), \\
\quad (\ e3 : Employee :: \ e3 \ne e2\ \wedge\ e3 \ne e1\ \wedge \\
\quad\quad e3.dept == \text{“}research\text{''} \wedge \\
\quad\quad e3.team == \textbf{unknown}\ ) \\
\rightarrow
\end{array}
$$

$e1.team = \textbf{new}\ Team(e1, e2, e3),$
$e2.team = e1.team,$
$e3.team = e1.team$
}

The purpose of the rule is to team up all employees in the research department such that each employee is in only one team and each team has exactly three employees. We can not use set selection conditions or the **All** combinator here since the same employee could be assigned to multiple teams. The key to the decomposition in this case is the disjointness of selected employees between different sets. Indeed, selecting disjoint sets of data objects for processing is a commonly used strategy in rule-based problem solving. This entitles a new mechanism for specifying the disjoint decomposition, which we call **Disjoint** *combinator*. For the example above, the rule below specifies that all teams can be formed at the same time as long as the selected employees are mutually disjoint (i.e., no two sets of selected objects have employees in common).

rule *Team_All_Employees* {
Disjoint (
($e1 : Employee ::\ e1.dept == \text{“}research\text{''} \wedge e1.team == \textbf{unknown}$),
($e2 : Employee ::\ e2 \neq e1 \wedge e2.dept == \text{“}research\text{''} \wedge e2.team == \textbf{unknown}$),
($e3 : Employee ::\ e3 \neq e2 \wedge e3 \neq e1 \wedge e3.dept == \text{“}research\text{''} \wedge e3.team == \textbf{unknown}$))
$\rightarrow$
$e1.team = \textbf{new}\ Team(e1, e2, e3),$
$e2.team = e1.team,$
$e3.team = e1.team$
}

Similar to the **All** combinator, the **Disjoint** combinator is used to combine several condition elements into a **Disjoint** *condition* for denoting that objects matching the enclosed conditions are to be decomposed in a disjoint pattern. In other words, for any two instantiations of a rule with **Disjoint** combinator, as long as the selected set of objects for the enclosed conditions are disjoint (i.e., no object in common), they are parallel executable. The true power of this mechanism is to reduce combinatorial explosive number of possibly interfering and mostly redundant instantiations into an exact set of all necessary and parallel executable instantiations. As an example, for a rule with n condition elements similar to the rule above, traditional methods will generate $n!$ instantiations as opposed to only one instantiation using the **Disjoint** combinator. The $n! - 1$ redundant instantiations will either have to be detected by interference analysis or, as in the PARULEL system [33] discussed in the Related Work section, removed by meta rules. In both cases the overhead is excessive. It may seem that to differentiate the wanted instantiation(s) from the redundant instantiations, we still need to pay the price either at match time or conflict-resolution time. We are developing an intelligent search technique to generate the desired instantiation(s) directly with much less work than what would be required to generate all the instantiations.

The formal semantics of the **Disjoint** combinator can be defined in a similar way as the **All** combinator except that the disjointness property must be clearly specified. For this purpose, we define an additional notation $\textbf{Objs}(a)$, for an object a, to denote the set of objects which are values of non-set attributes of a or elements of set-valued attributes of a. The semantics of the **Disjoint** combinator can now be defined as follows. A rule r with positive conditions $P_1, \ldots, P_i$, set selection conditions $P_{i+1}, \ldots, P_n$, negative conditions $N_1, \ldots, N_m$, and a **Disjoint** condition combining positive conditions $DP_1, \ldots, DP_j$, set selection conditions $DP_{j+1}, \ldots, DP_k$, and negative conditions $DN_1, \ldots, DN_l$, defines a class:

class *Inst_of_r* {
attributes ($v(P_1) : C(P_1),$
$\vdots$
$v(P_i) : C(P_i),$
$v(P_{i+1})* : C(P_{i+1}),$
$\vdots$
$v(P_n)* : C(P_n),$
$disjoint* : DisjointClass\_of\_r$)

methods ($\bigcup_{i=1}^{n} M(C(P_i)) \cup \bigcup_{i=1}^{k} M(C(DP_i))$)
}

where *DisjointClass_of_r* is the class

class *DisjointClass_of_r* {
attributes ($v(DP_1) : C(DP_1),$
$\vdots$
$v(DP_j) : C(DP_j),$
$v(DP_{j+1})* : C(DP_{j+1}),$
$\vdots$
$v(DP_k)* : C(DP_k)$)

methods ($\bigcup_{i=1}^{k} M(C(DP_i))$)
}

The set of all ground instantiations can be characterized by the set

$\{\ (g_1, \ldots, g_n, d_1, \ldots, d_k) \mid \exists\, t, u\ ($
$t \in Inst\_of\_r \ \wedge\ u \in t.disjoint* \ \wedge$
$\forall x \in t.disjoint*$
$(x \neq u \Rightarrow \textbf{Objs}(x) \cap \textbf{Objs}(u) = \emptyset) \ \wedge$
$g_h = t.v(P_h),\ 1 \leq h \leq i \ \wedge$
$g_h \in t.v(P_h)*,\ i+1 \leq h \leq n \ \wedge$

$$d_h = u.v(DP_h),\ 1 \leq h \leq j\ \wedge$$
$$d_h \in u.v(DP_h)*,\ j+1 \leq h \leq k\ \wedge$$
$$A\theta\ )\ \}$$

where

$$\begin{aligned}\theta &= \{\ v(P_1)/g_1, \ldots, v(P_n)/g_n, \\ &\quad\ v(DP_1)/d_1, \ldots, v(DP_k)/d_k\ \} \\ A &= E(P_1)\ \wedge\ \ldots\ \wedge\ E(P_n)\ \wedge \\ &\quad E(DP_1)\ \wedge\ \ldots\ \wedge\ E(DP_k)\ \wedge \\ &\quad \nexists\ v(N_1), \ldots, v(N_m), v(DN_1), \ldots, v(DN_l)\ (\\ &\qquad E(N_1)\ \vee\ \ldots\ \vee\ E(N_m)\ \vee \\ &\qquad E(DN_1)\ \vee\ \ldots\ \vee\ E(DN_l)\ ).\end{aligned}$$

Analogous to the **All** combinator, a single **Disjoint** combinator is enough since

$$\begin{aligned}&\mathbf{Disjoint}(C_1) \wedge \ldots \wedge \mathbf{Disjoint}(C_n) \\ \equiv\ &\mathbf{Disjoint}(C_1 \wedge \ldots \wedge C_n).\end{aligned}$$

Nesting is not recommended either because of the conciseness criterion. We can also have both **All** and **Disjoint** combinators in the same rule with well-defined semantics. However, the resulting decomposition patterns seem to be too complex and not intuitively appealing enough for a programmer to conceive and use.

Contexts

A common programming style for rule programs is to decompose the problem solving process into groups of rules where a set of rules is only satisfiable in a *context*. In rule languages without an explicit context mechanism, typically programmers create a special pattern and include it as a condition of all the rules of that context. A context can be considered as a function unit where the set of rules are written to serve the functionality of that context. Opportunities for parallelism are presented when this level of application semantics is taken into consideration [17]. In particular, causally independent contexts can often be processed in parallel. There is a catch if we are to be consistent with our goals. We should avoid providing any procedural-oriented or control-oriented mechanism. Thus, we provide mechanisms that designate the context for which a rule is intended and that specify the causal dependency between different contexts. The actual control dependency implied by the semantic information above should be left for the language system to derive.

A rule of the form

rule r **in context** $T\ \{\ \ldots\ \}$

denotes that the rule r is designed for context T. All rules designated to the same context are for solving the same subproblem. A *context rule* of the form

$$T\ \vdash\ T_1, T_2, \ldots, T_n$$

specifies that context T is causally dependent on contexts $T_1, T_2, \ldots, T_n$ which means, to solve the problem for which T is designed, all the subproblems for which $T_1, T_2, \ldots, T_n$ are designed must be solved first. Note that a context rule specifies a causal dependency rather than an implication. The subproblem represented by context T must still be solved after solving all dependent contexts. The following example denotes that the rule *Raise_All_Poor_Employee* belongs to the context *Salary_Adjustment* which is a collection of rules for adjusting salary.

rule *Raise_All_Poor_Employees*
in context *Salary_Adjustment* $\{$
$(\ d : Department\ ),$
$\mathbf{[}\ e : Employee :: \ e.dept == d.name\ \wedge$
$\qquad e.salary < 10000\ \mathbf{]}$
$\rightarrow$
$e.salary = e.salary + e.salary/10$
$\}$

The context rule below specifies that before working on salary adjustment, we must perform profit evaluation and salary survey.

$$Salary\_Adjustment\ \vdash\ Profit\_Evaluation,\ Salary\_Survey$$

The context mechanisms proposed above provide a simple way to specify function decomposition of a problem into subproblems. A set of context rules specifies a partial order that must be observed between subproblems to correctly solve the entire problem. Parallelism at the problem solving level can then be exploited by processing independent contexts in parallel.

The parallel structuring mechanisms presented in this section are much more powerful than the examples can demonstrate when it comes to real programming. Programmers can easily pick up the ideas and use them effectively. Later in this paper we will show that just this simple set of mechanisms can significantly improve the performance of parallel rule programs.

D. Semantic-Based Interference Analysis

An equally important technique in our decomposition abstraction approach is a semantic-based interference analysis technique which derives information about run-time parallel structure from associative relationships among data objects. We present the technique in this section.

More often than not, class relationships provide valuable hints on data decomposition patterns that actually happen at run time but are not necessarily clear at design or compile time. In particular, this information can often be used in determining the semantic compatibility (i.e., parallel executability) of instantiations of the same rule or between different rules.

As an intuitive example, consider the following rule from our corporation application domain which is to raise the salary of all under-paid employees in a team.

rule *Team_Fairness* $\{$
$(\ t : Team\ ),$
$[\ e : Employee :: \ e.team == t.name\ \wedge$
$\qquad e.salary < t.min\_wage\ ]$

$\rightarrow$

$e.salary = t.min\_wage$

}

In general, different instantiations of this rule can not be executed in parallel because the same employee may be a member of different teams. On the other hand, if each team is associated with a distinct set of employees, then different instantiations will select different teams with disjoint set of employees. Apparently, all such instantiations can be fired in parallel. The key point is on the relationship between instances of the *Team* and the *Employee* class. We call this relationship a *functional dependency* which is formally specified in the following definitions.

DEFINITION **13 (Class Relations and Schemes)** *A* **class relation scheme** *(or simply* **scheme***) is an ordered set of class names. A* **class relation** *on a class relation scheme with n class names is an n-ary relation among instances of the corresponding classes.*

It helps in understanding the definition by comparing the class names, classes, and schemes with the attribute names, domains, and relation schemes in relational database.

For a class relation A, we denote the scheme on which A is defined by $Sch(A)$. A class relation can be considered as a collection of classes with a certain relationship. Note that an element of an n-ary class relation is an ordered set of n objects, one from each corresponding class in the scheme. An object here can be either a structural object or a set object. For an element $a \in A$ and a scheme $X \subseteq Sch(A)$, the notation $a(X)$ denotes the ordered collection of objects in a which are from classes in X. We note that from the definition above, $a(X) \subseteq a$ and $a(Sch(A)) = a$.

DEFINITION **14 (Functional Dependency)** *Let X and Y be the schemes of two class relations R_x and R_y. The* **functional dependency**

$$X \rightarrow Y$$

holds on R_x and R_y if

1. *Each element in R_x is associated with a unique element in R_y.*
2. *For all a_1, a_2 in R_x and the associated b_1, b_2 in R_y,*
$$a_1 \neq a_2 \Rightarrow b_1 \cap b_2 = \emptyset.$$

As an example from the corporation application domain discussed earlier, the functional dependency $\{Team\} \rightarrow \{Employee\}$ holds when each team is associated with a unique and disjoint set of employees.

Since an instantiation can also be considered as an ordered set of objects (one for each positive or set selection condition), a rule r actually defines a class relation whose elements are exactly the set of instantiations of the rule, i.e., $\mathbf{Inst}(r)$. The scheme of $\mathbf{Inst}(r)$, denoted by $\mathbf{Scheme}(r)$, is the ordered set of class name components of positive and set selection conditions of r. For example,

$$\mathbf{Scheme}(Team\_Fairness) = \{Team, Employee\}.$$

DEFINITION **15** *Let X and Y be two class relation schemes, and r be a rule. The* **rule specific functional dependency**

$$X \rightarrow Y \text{ in } r$$

holds if both $X \subseteq \mathbf{Scheme}(r)$ and $Y \subseteq \mathbf{Scheme}(r)$ and for all i, j in $\mathbf{Inst}(r)$,

$$i(X) \neq j(X) \Rightarrow i(Y) \cap j(Y) = \emptyset.$$

It is *rule specific* because the dependency only needs to hold on all instantiations of r. It may or may not hold on collections of objects that are not instantiations of r.

DEFINITION **16** *Let R be a set of rules, X and Y be two class relation schemes, then*

$$X \rightarrow Y \text{ in } R$$

holds if $X \rightarrow Y$ in r holds for each rule r in R.

Except for borrowing the terminology, functional dependency as defined here is quite different than in databases [34]. In database systems, the notion of functional dependency is defined at the attribute level and is used primarily in the normalization process. We generalize the concept to the class level and use it to identify the parallelism in rule systems. Functional dependencies are considered as specifications of data decomposition across class boundaries, which are shown below to play a crucial role in determining the compatibility between instantiations of the same or different rules.

DEFINITION **17** *Let r be a rule. The* **access set** *of r, denoted by $\mathbf{Access}(r)$, is the set of all class names referenced in the antecedent of r. The* **write set** *of r, denoted by $\mathbf{Write}(r)$, is the set of class names with objects that are modified (including creation and deletion) in the rule.*

DEFINITION **18 (Dominant Set)** *Let r be a rule and C be a class relation scheme. C is a* **dominant set** *of r if:*

1. *$C \subseteq \mathbf{Scheme}(r)$,*
2. *for all $i, j \in \mathbf{Inst}(r)$ $(i \neq j \Rightarrow i(C) \neq j(C))$.*

A dominant set of a rule is simply a set of class names sufficient to discriminate between different instantiations of the rule.

THEOREM 1 (Self Compatibility) *Let r be a rule and A, B, C be three class relation schemes that are subsets of $\mathbf{Scheme}(r)$ satisfying the following conditions:*

1. *C is a dominant set of r and $C \subseteq A$*
2. *$A \rightarrow B$ or $A \rightarrow B$ in r*
3. *$\forall c \in \mathbf{Write}(r)(c \in B \vee c \notin \mathbf{Access}(r))$*

then all instantiations of r are compatible (i.e., parallel executable).

Proof sketch: In any given state, let i and j be instantiations of r such that $i \neq j$.

$$\begin{array}{ll} & i \neq j \\ \Rightarrow & i(C) \neq j(C) \\ \Rightarrow & i(A) \neq j(A) \\ \Rightarrow & i(B) \cap j(B) = \emptyset \\ \Rightarrow & i(\mathbf{Write}(r)) \cap j(\mathbf{Write}(r)) = \emptyset \\ \Rightarrow & i \text{ and } j \text{ do not interfere with each other} \\ \Rightarrow & i \text{ and } j \text{ are compatible} \end{array}$$

■

The central idea of this theorem is that functional dependency implies disjoint decomposition of objects selected by the instantiations of a rule. As long as the objects modified in the consequent belong either to the decomposition or to classes which do not affect the satisfiability of the rule, no instantiations will interfere with each other. There will be examples later in this section. We first generalize this idea to the analysis of interference between multiple rules.

DEFINITION **19 (Partially Mutual Exclusion)** *Let p, q be rules and C be a class relation scheme. We say that p and q are* **partially mutual exclusive on** C, *denoted by $p \bowtie_C q$, if*

1. $C \subseteq \mathbf{Scheme}(p)$ *and* $C \subseteq \mathbf{Scheme}(q)$
2. *For any two instantiations i, j of p and q respectively, $i(C) \neq j(C)$.*

Partially mutual exclusion simply means that p and q can not have instantiations containing the same set of objects of classes in C. The simplest and most common case is when C contains a single class referenced in both p and q but tested on disjoint values of the same set of attributes. Since the values are disjoint, p and q can not select the same object in C.

Note that no requirement is placed on selected objects that are not of the classes in C. Therefore, partially mutual exclusive rules may still interfere with each other. However, in many cases, partially mutual exclusive rules can be determined to be parallel executable with the help of functional dependencies as indicated by the following theorem.

THEOREM 2 (Pair-Wise Compatibility) *If p, q are two distinct rules, and A, B, C are class relation schemes that are subsets of both* **Scheme***(p) and* **Scheme***(q) such that the following conditions are satisfied:*

1. $p \bowtie_C q$
2. $C \subseteq A$
3. $A \rightarrow B$ *or* $A \twoheadrightarrow B$ *in* $\{p, q\}$
4. $\forall c \in \mathbf{Write}(p)(c \in B \vee c \notin \mathbf{Access}(q))$
5. $\forall c \in \mathbf{Write}(q)(c \in B \vee c \notin \mathbf{Access}(p))$

then p and q are compatible and therefore parallel executable.

Proof sketch: In any given state, let i be an instantiation of p and j be an instantiation of q.

$$\begin{array}{ll} & p \bowtie_C q \\ \Rightarrow & i(C) \neq j(C) \\ \Rightarrow & i(A) \neq j(A) \\ \Rightarrow & i(B) \cap j(B) = \emptyset \\ \Rightarrow & i(\mathbf{Write}(p)) \cap j(\mathbf{Write}(q)) = \emptyset \\ \Rightarrow & p \text{ and } q \text{ are compatible} \end{array}$$

■

Again, the central idea of this theorem is that as long as objects modified in p and q can be determined as non-overlapping with the help of functional dependency, instantiations of p and q do not interfere with each other.

Even with their general applicability to many cases, the two theorems above are less complicated than they appear. Continuing with our examples in the corporation application domain, if a team is associated with a set of disjoint employees as team members, then the functional dependency $\{Team\} \twoheadrightarrow \{Employee\}$ holds. We note that this semantic information can be easily supplied by the programmer (similar to the identification of key attributes in database systems). With functional dependency and the fact that a team can be uniquely identified by its name, we can immediately determine that all instantiations of the *Team_Fairness* rule can be fired in parallel using Theorem 1.

As another example, the following two rules can be determined to be parallel executable by Theorem 2.

rule *Facilities_Research* {
  ($t : Team :: t.dept ==$ "*research*")
  [$e : Employee :: e.team == t.name$]
$\rightarrow$
  $e.equipment =$ "*AXP500X(Alpha)*"
}

rule *Facilities_Sales* {
  ($t : Team :: t.dept ==$ "*sales*")
  [$e : Employee :: e.team == t.name$]
$\rightarrow$
  $e.equipment =$ "*PowerBook*"
}

In this case, the two rules are partially mutual exclusive on *Team*. With the help of functional dependency, they can be statically determined to be parallel executable.

As simple and natural as it may seem to be, without the knowledge of functional dependency between the *Team* and the *Employee* classes, it is very difficult, if not impossible, for a parallelizing compiler or any other static transformation technique to identify the parallelism underlying these rules.

In general, any type of class relationship which implies certain patterns of association or partitioning in the application domain is of great help in the determination of proper decomposition for parallel processing. Mechanisms for expressing these relationships are therefore of great value to the decomposition abstraction process in program development.

We can now refine the definition of program in our framework to include semantic specifications expressed by the decomposition abstraction mechanisms.

DEFINITION **20 (Program)** *A program is a quadruple* $(\mathbf{C}, \mathbf{R}, \mathbf{D}, \mathbf{T})$ *where* $\mathbf{C}$ *and* $\mathbf{R}$ *are class and rule definitions, respectively.* $\mathbf{D}$ *is a set of* **class relationship definitions** *which specify how objects are related to each other between different classes. And* $\mathbf{T}$ *is a set of* **context rules** *that specify the causal dependencies between different contexts.*

As a summary, we have proposed a set of mechanisms as the basis for decomposition abstraction in parallel rule languages. These are set selection, aggregate operators, combinators, context, and class relationships specifications (functional dependency in particular). Together they form a powerful set of tools for parallel rule-based programming.

IV. Evaluation

To assess the quality of the abstract mechanisms, we evaluate them against the design criteria listed in Section III-B. The expressive power of the mechanisms is judged by evaluating how well they cover different types of parallelism discussed in Section III-A.

What vs. How. All proposed mechanisms are declarative. They represent core decomposition concepts that can be used to specify various types of parallelism. How they are actually implemented is independent of the decomposition strategies they stand for. Programmers can use them at the conceptual and problem level without worrying about implementation details.

Consistent with Pattern Matching Paradigm. A set selection condition element selects objects in exactly the same way as a normal condition element except that all matching objects are selected. An aggregate operator derives information on a set of objects selected by pattern matching. Both **All** and **Disjoint** combinators simply group together several condition elements to partition the objects selected by pattern matching in the enclosed condition elements. The pattern matching paradigm is left intact. The additional power added by the decomposition abstraction mechanisms includes the abilities to partition the objects selected by pattern matching and to organize the program at the problem solving level.

Conciseness. The set of mechanisms is conceptually simple and easy to use in the sense that all of them correspond naturally to the patterns of decomposition commonly seen in rule-based systems. When specified using our mechanisms, the decomposition patterns map naturally to parallelization strategies which are then exploited by the system. Programmers are not required to specify synchronization or communication constraints explicitly. There is no need to worry about interference beyond the level of application semantics.

Versatility. This criterion is about the expressive power of the mechanisms. Under the context of rule languages, it can be evaluated by considering how well these mechanisms support the different types of parallelism discussed in Section III-A. The combination of set selection conditions, aggregate operators, **All** and **Disjoint** combinators is capable enough to specify all sources of data level parallelism we found. Using pattern matching to specify decomposition also seems to be more flexible than partitioning merely on array index as in most explicit parallel imperative languages.

For rule level parallelism, the use of class relationships to discover semantic compatibility between rules is very effective, especially when used in conjunction with the existing techniques on syntactic dependency analysis [15], [16], [17], [18], [19]. In particular, all of the parallel executable rules in our benchmark programs can be successfully identified.

For program level parallelism, the context mechanism provides a convenient way to specify any causal relationship or partial order among the problem solving stages. Independent contexts representing independent subproblems can then be identified by a topological sort on the partial order.

The set of mechanisms covers all three levels of parallelism found in production systems, and is certainly versatile and expressive enough for our purpose.

Compatibility. The mechanisms are designed in such a way that programs not using any of these mechanisms are still correct programs. In particular, all sequential programs are legal programs. Many sequential programs can be transformed directly into parallel programs as indicated by the examples in Section III-C and in [20]. Above all, by simply removing the mechanisms in a parallel program, we get a legal sequential one back. The set of mechanisms is therefore compatibility with sequential semantics.

Effective Implementation. The set of mechanisms can indeed be effectively implemented on shared memory multiprocessors. Several alternative implementation strategies and their effectiveness are discussed in Section V with performance results.

V. Implementation Strategies

With the proposed set of decomposition abstraction mechanisms, programmers can specify the parallelism inherent in the problem domain, thereby increasing the concurrency that can be exploited by the language system. However, just like any explicit parallel language, the actual performance gain depends heavily on the implementation strategy. This is especially the case in rule systems because rules tend to have large variation of processing requirements [21] and irregular patterns of decomposition. Systems that fire multiple instantiations in parallel tend to incur large run-time overhead [35]. Granularity control and proper scheduling strategy are therefore of crucial importance to a multiple rule firing production system with decomposition abstraction mechanisms. In this section, we discuss several alternative implementation strategies and their performance on a parallel rule execution engine whose purpose is to provide a test bed and a guide to our final implementation of the proposed decomposition mechanisms.

A. A Parallel Rule Execution Engine

To be as close as possible to a full implementation, we developed a *parallel rule execution simulation engine* that comprises a complete implementation of the communication and synchronization components of a full implementation. The behavior of the remainder of the inference engine is extracted by a *work-load generator* from recorded traces of sequential executions of each of the benchmarks. Rule actions are executed as they would be in a full parallel implementation and an accurate representation of the working memory is maintained. Thus, the communication and synchronization operations needed to maintain the correct parallel execution are *not simulated* they are actually performed. In other words, the only factor which is not implemented is the matcher. We justify simulating this component as follows. The work-load generator extracts sequential execution behavior for the simulated component. A full implementation would incorporate extensive results on both sequential and parallel algorithms for matching [36], [37], [38]. Thus, the simulation results are a conservative estimate performance of a full implementation. The target machine is a Sequent Symmetry shared memory multiprocessor.

Fig. 1 depicts the simulation method we developed. The parallel rule execution engine is a set of programs built on top of a C++ based object-oriented light weight thread package called PRESTO [39]. A sequential rule program[4] and its execution trace are translated into a PRESTO program which is the parallel version of the program integrated with the PRESTO run-time libraries. More specifically, each data definition[5] is translated into a C++ class definition which is called a *WME class*. Each rule definition is translated in to a C++ function which is termed a *rule function*. The work load generator generates a call to a rule function(with the WME's accessed by the rule as arguments) for each rule firing in the execution trace. Successive rule firings are translated into a sequence of asynchronous thread invocations to execute the corresponding rule functions if the rules are compatible as determined by the decomposition abstraction specification. Synchronization points are inserted whenever it is necessary to synchronize incompatible rules. With different scheduling strategies to be discussed later, one or more rule function calls can be assigned to a thread for executing sequentially within the thread.

To experiment with the effect of granularity of rules on system performance, a controllable dummy loop (a for loop with empty body and a parameter controlling the number of iterations) is added to the action part of each parallel rule in the PRESTO program. The time to execute the action part of a rule can then be controlled by varying the number of iterations for the dummy loop. Also, by varying the number of threads created for processing parallel instantiations and the granularity of work (number of instantiations) assigned to each thread, we have measured the performance of four alternative scheduling strategies.

- **Strategy 1: Maximal Parallelism** Create a new thread for each parallel instantiation.
- **Strategy 2: Fixed Granularity** Same as Strategy 1 except that each thread is assigned a fixed number of instantiations to execute sequentially. This is to increase the granularity of work assigned to each thread so as to reduce the total number of threads.
- **Strategy 3: Supervisor/Worker** Create a fixed number of worker threads and a supervisor thread. The supervisor thread keeps dispatching instantiations, one instantiation for each worker, as long as there are idle workers. If no idle worker exists, the supervisor executes the instantiation itself.
- **Strategy 4: Supervisor/Worker with Packing** Same as Strategy 3 except that each time an idle worker is given a fixed number of instantiations to work on sequentially. The number is called the *grain size* [40] whose purpose is to increase the granularity of work assigned to a worker thereby reducing thread management and scheduling overhead.

Even though previous work on task scheduling shows that the performance of the supervisor/worker model can be significantly affected by the communication cost [41], this model can still benefit from parallel execution when the computation cost is high enough. We have demonstrated this in the set of experiments on rule granularity.

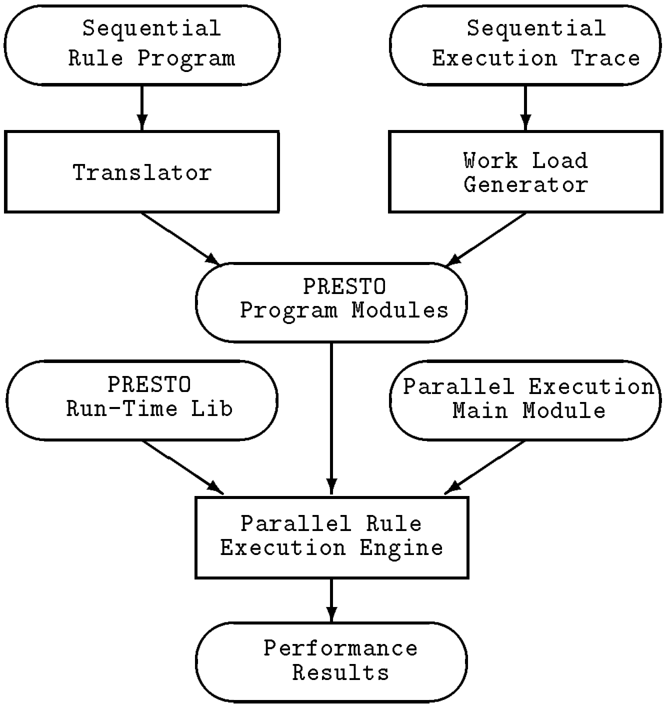

Fig. 1. Parallel Rule Execution Engine and the Simulation Method.

[4] At this moment, the parallel rule execution engine takes only OPS5 programs. However, the same approach can be applied to any sequential rule language.

[5] In the case of OPS5, it is an *element class* defined by the **literalize** command.

To evaluate the effectiveness of different scheduling strategies, we collect the performance results from the execution of three benchmark programs drawn from the Texas OPS5 Benchmark Suite [42] and listed in Table I. All three programs are executed with increasing numbers of processors on different problem sizes and different grain sizes. We then compare their relative performance, scalability, as well as sensitivity to granularity change. The actual time to execute the dummy loop for various number of iterations are also measured and listed in Table II.

B. Characteristics of the Benchmark Programs

Identifying the characteristics of an application is of crucial importance in the process of decomposition abstraction. In this section, we analyze the concurrent behavior of each benchmark program and point out key issues to the successful application of proposed mechanisms.

MANNERS

MANNERS was derived from an example program in [44] which employed a combinatorial search for solving a seat assignment problem among a number of guests. The seats must be assigned such that neighbors are of opposite sex and share at least one common hobby. This simple program, containing just 8 rules, is a very good test program for evaluating how effective a parallel production system is. It consists of a hot spot rule that fires repeatedly and consumes over 90% of sequential execution time for problem size of 64 guests or more. The larger the problem size, proportionally more time is consumed by this rule, which is used to maintain partial solutions. Although all instantiations of this rule can be fired in parallel, it is very difficult to recognize this opportunity by pure syntactic techniques at compile-time without the information provided by our mechanisms. The concurrency exhibited is also quite exceptional. It is regular but not evenly distributed. The execution starts with a very low degree of parallelism which then gradually increases toward the end. More specifically, the program starts with only 2 instantiations that can be executed in parallel, then 3 instantiations, then 4, 5, ... etc. Fig. 2 is the concurrency profile of MANNERS with 16 guests. This is highly challenging since a parallel production system must not only detect the hot spot rule, but also exploit effectively a rather peculiar pattern of parallelism.

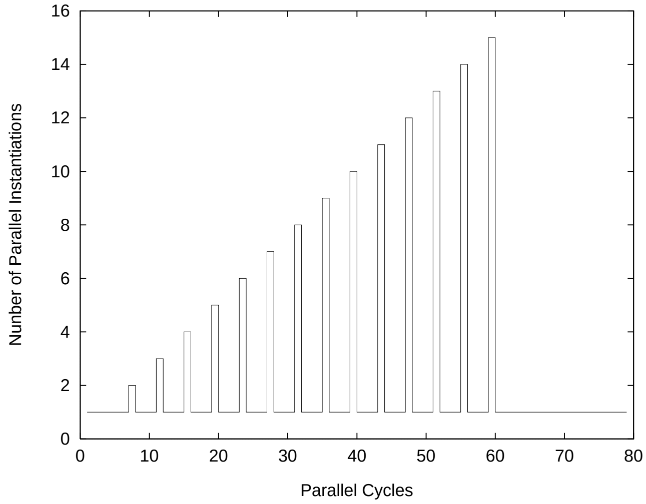

Fig. 2. MANNERS16 Concurrency Profile.

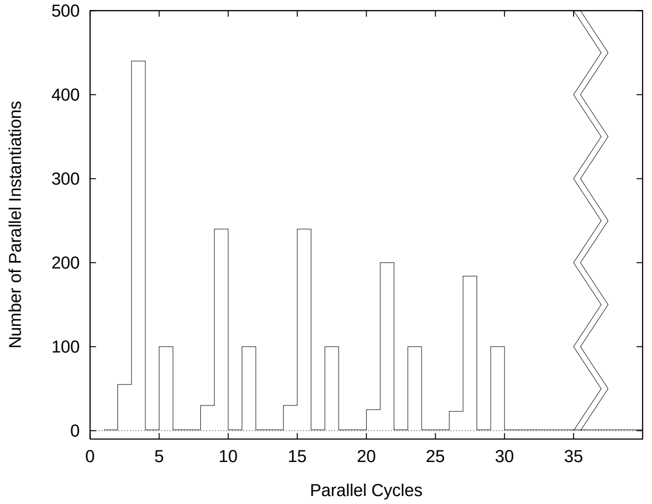

Fig. 3. LIFE10 Concurrency Profile (with Sequential Printing at the End trimmed by the zigzag line).

LIFE

LIFE is a simulation program that simulates the existence of bacteria in a rectangular grid of cells for a specified number of generations. Whether a cell stays alive across a generation is determined by the number of neighbors it has. A living cell is born in an empty cell if it has exactly 3 neighbors. Since all decisions can be made locally, LIFE exhibits a high degree of data level parallelism. However, the available concurrency has not been effectively exploited in previous research. The difficulty of detecting it by syntactic analysis alone is again the key reason. Another probably even more important reason is that a parallel production system must have the ability to perform set-oriented and aggregate operations to exploit the available concurrency. Fig. 3 is the concurrency profile of a 10x10 LIFE execution trace without showing the sequential printing at the end of execution. Because of the rather evenly distributed pattern of parallelism, keeping processors busy doing useful work at all time is the primary issue.

WALTZ

The frequently studied WALTZ program is also selected here to serve both as a test program to evaluate the effectiveness of our mechanisms and as a benchmark program to compare our results with others. This is a constraint satisfaction problem that implements Waltz's algorithm for labeling of line drawing scenes. The algorithm propagates labels based on local decisions and therefore exhibits both SPMD- and MIMD-style of parallelism (i.e., parallel instantiations of the same or different rules working on different parts of the scene). The available parallelism is depicted in Fig. 4, which is the concurrency profile of a 10 regions execution trace with sequential printing of results at the

Program	No. Rules	Description
LIFE	16	A simulation program implements Conway's LIFE.
WALTZ	33	A constraint satisfaction problem using Waltz's algorithm for scene labeling [43].
MANNERS	8	A combinatorial search problem for seat assignment.

TABLE I
BENCHMARK PROGRAMS USED IN THE EXPERIMENTS.

No. Iterations	Time (ms)
0	0.00
1000	2.57
3000	7.64
5000	12.71
7000	17.78
10000	25.38
20000	50.75

TABLE II
TIME TO EXECUTE THE DUMMY LOOP FOR SPECIFIED NUMBER OF ITERATIONS.

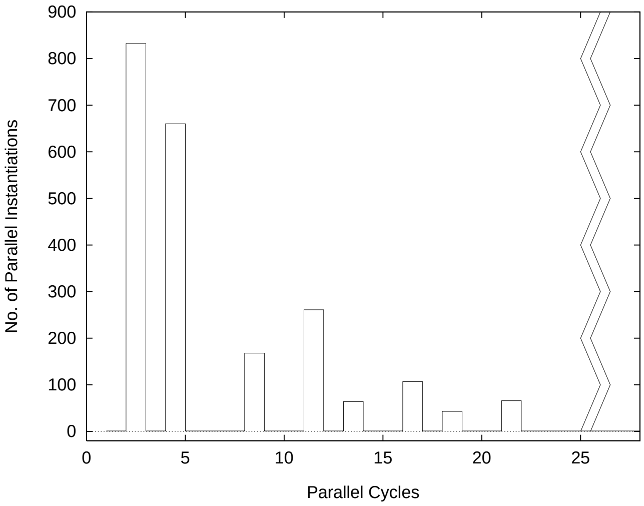

Fig. 4. WALTZ10 Concurrency Profile (with Sequential Printing at the End trimmed by the zigzag line).

end excluded.

Because, syntactically, a number of similar constraints appear in many rules, the rules seem to be highly interfering with each other. Most of the existing parallel production systems can only handle this at run-time resulting in excessive overhead. However, the constraints are disjoint, most of the run-time overhead is actually superfluous. The functional dependency declaration and **Disjoint** combinators express this to reveal a compile-time parallel structure. A system capable of forming disjoint partitions of consistent data objects can then effectively exploit this available concurrency without the overhead of run-time interference detection.

C. Performance Results

In this section, we demonstrate and analyze the performance results of three OPS5 benchmark programs on our parallel rule execution engine. Experiments are conducted on various dimensions affecting the selection of implementation strategies. The speedup is measured against the execution time of the execution engine with a single processor instead of an optimized uniprocessor OPS5 compiler such as OPS5c [45] because the former is two to three orders of magnitude faster than the latter. The difference is primarily due to the additional matching performed by the uniprocessor compiler. As a remedy to this lack of match phase, we artificially increase the per rule processing time using dummy loop as described earlier.

The Effect of Rule Granularity

To understand the effect of rule granularity (i.e., the time to process a rule) on the system performance, we select the supervisor/worker with packing scheduling strategy while varying the granularity of a rule by changing the number of iterations in the dummy loop. Fig. 5 shows the results on MANNERS512 (i.e., 512 guests). The performance improves significantly with larger granularity. Nearly ideal speedup is achieved when the granularity per rule is increased to 20000 (i.e., 50.75ms). As a comparison, the average cycle time of the same program and data set running under OPS5c on a much faster CPU (SUN4 workstation vs. the Intel 80386 on Sequent Symmetry) is about 210ms. This implies that the overhead of scheduling and thread management is very low, and as long as we can keep it low in a full implementation it is very likely to get even better results since the granularity per rule is expected to be much higher than 50ms when matching is included.

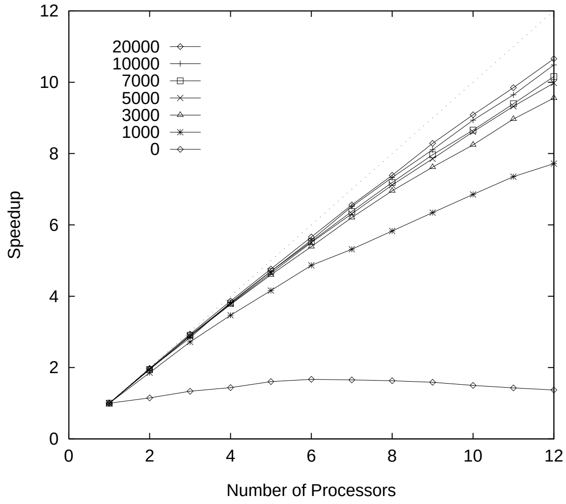

Fig. 5. MANNERS512 Speedup with Varying Granularities.

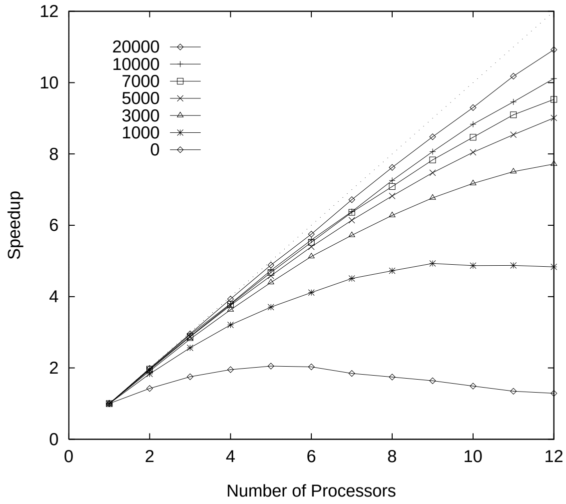

Fig. 6. LIFE40 Speedup with Varying Granularities.

Fig. 6 and Fig. 7 are the results of similar experiments on LIFE (40x40) and WALTZ (30 regions), respectively. In both cases, performance improvement is observed with increasing granularity.

Among all three test cases, LIFE achieves the highest speedup with granularity 20000. This is plausible because the run-time behavior of LIFE exhibits the highest and the most regular pattern of concurrency as depicted earlier in Fig. 3. On the other hand, WALTZ requires larger granularity to achieve the same level of performance. We were puzzled by this unexpected result at first since from the characteristics of the Waltz's algorithm, there should not be that much difference. Later on we found that the available parallelism of a WALTZ program execution depends heavily on the data set (i.e., the scene to be label). The data generator we use (and used by other researchers as well) introduces a sequential factor that severely restricts the available parallelism. The generated scene consists of two arrays of rectangular blocks growing linearly according to the given problem size parameter. This linear factor contributes to the performance difference between WALTZ and the other two programs. We plan to develop a new

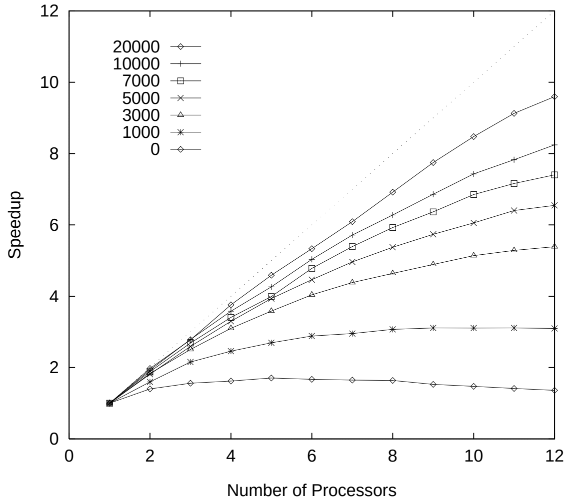

Fig. 7. WALTZ30 Speedup with Varying Granularities.

data generator that generates scenes without this linear factor. Note that this should not be mistaken as a deficiency of our approach since it is caused by the way the WALTZ program works (i.e. starting from two opposite corners) and the peculiar data generator. In other words, if the inherent concurrency is low, there is little we can do with parallelism.

Scalability: The Effect of Problem Size

An important criterion when evaluating the effectiveness of a parallel system is its scalability. When the available parallelism increases, a parallel processing system must be able to effectively exploit it and achieve better performance. For the three benchmark programs, a common characteristic is that available parallelism increases with the problem size. Therefore, we tested our mechanisms and system on three programs with increasing problem sizes where the problem size to MANNERS, LIFE and WALTZ are the number of guests to be assigned, the grid size, and the number of regions[6], respectively. All programs were tested under the supervisor/worker with packing scheduling strategy. The grain size was set to 5 with rule granularity fixed at 20000 (i.e., 50.75ms).

Fig. 8 illustrates the performance results of MANNERS on different problem sizes and graphically displays scalable speedup of our scheme. Fig. 9 and Fig. 10 are the results of similar experiments on LIFE and WALTZ. The speedup achieved on smaller problems is lower because the available parallelism is not enough to keep all processors busy. When problem size becomes larger, processor utilization increases and so does the speedup achieved.

Controlled vs. Unrestricted Parallelism

Too much water drowned the miller. If the available parallelism is not exploited appropriately, the benefit of parallel processing can easily be overwhelmed by the scheduling and synchronization overhead. In our case, since the

[6] In the scene generated by the data generator for WALTZ, a region consists of 72 line segments.

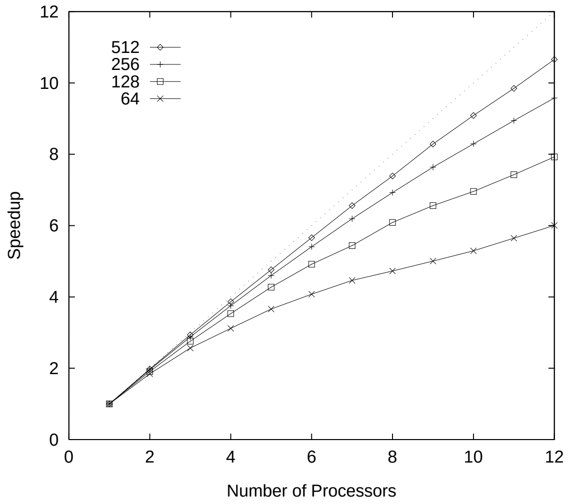

Fig. 8. MANNERS Speedup on Different Problem Sizes.

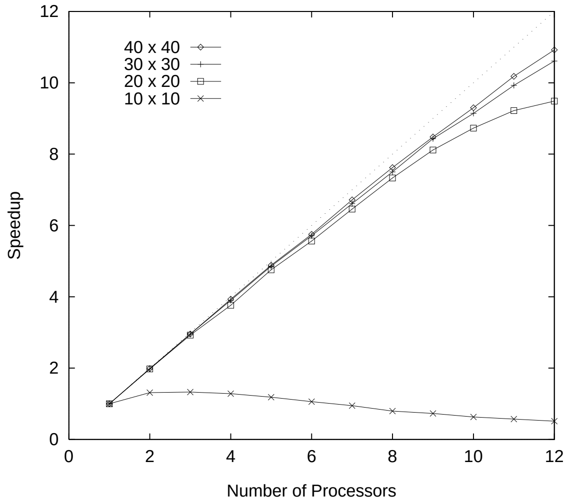

Fig. 9. LIFE Speedup on Different Problem Sizes.

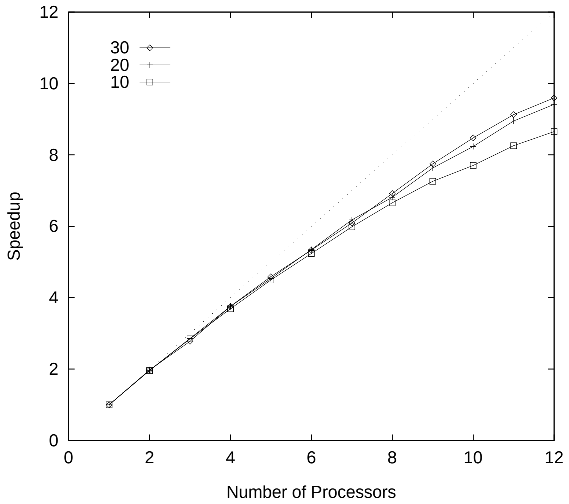

Fig. 10. WALTZ Speedup on Different Problem Sizes.

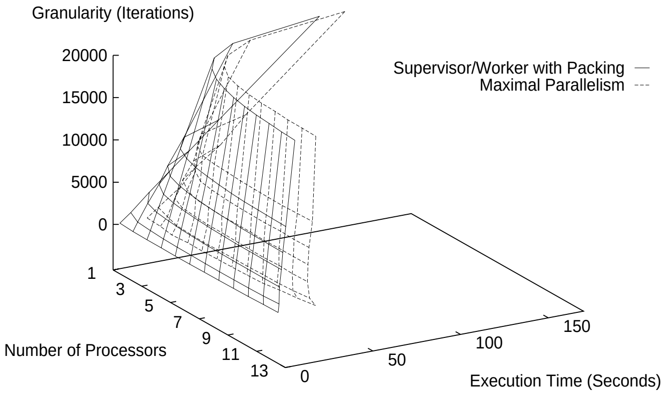

Fig. 11. 3-D Display of Controlled vs. Maximal Parallelism on WALTZ10.

embedded parallelism in the application is explicitly expressed, the key issue comes down to efficiently processing the collection of parallel instantiations on available computation resources. For a thread-based implementation like ours, this issue manifests itself in a tradeoff between parallel processing of as many instantiations as possible and controlling the number of concurrent threads. If a new thread is created for each parallel instantiation (i.e., Strategy 1), we get maximal parallelism on the one hand but highest thread management overhead on the other hand. Using the supervisor/worker scheduling strategy, the number of threads is fixed but the communication and synchronization cost increase because of the need to partition and dispatch parallel instantiations to the worker threads.

To understand the effect of thread management overhead on system performance, we compare the results between applying Strategy 1 (maximal parallelism) and Strategy 4 (supervisor/worker with packing). Fig. 11 is a 3-D display of two sets of experiments on WALTZ10. The B-spline surfaces are to demonstrate the performance differences. It is quite evident that supervisor/worker with packing outperforms maximal parallelism by a substantial margin. Fig. 12 presents the results of similar experiments on LIFE30. The difference is smaller but still perceptible. This suggests that throttled parallelism is much better than unrestricted parallelism.

Just when we expect to observe a similar type of performance difference on MANNERS program, it does not happen to be the case. Fig. 13 is the 3-D graph of the similar experiments as above. The performance of maximal parallelism is not only comparable to that of supervisor/worker with packing, it actually performs better when the granularity of rules and the number of processors increases. To more clearly show this phenomenon, we demonstrate the timing curves on a 2-D diagram in Fig. 14. A closer look at the problem reveals a peculiar pattern of concurrent behavior of the MANNERS program. As depicted in Fig. 2, the number of parallel executable instantiations is quite small in the early stage of the execution. When packing is

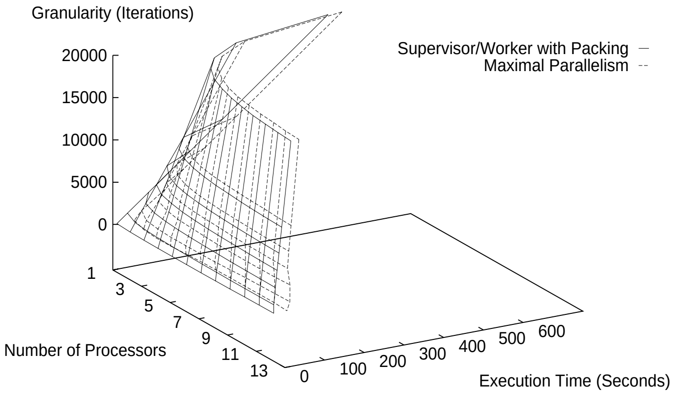

Fig. 12. 3-D Display of Controlled vs. Maximal Parallelism on LIFE30.

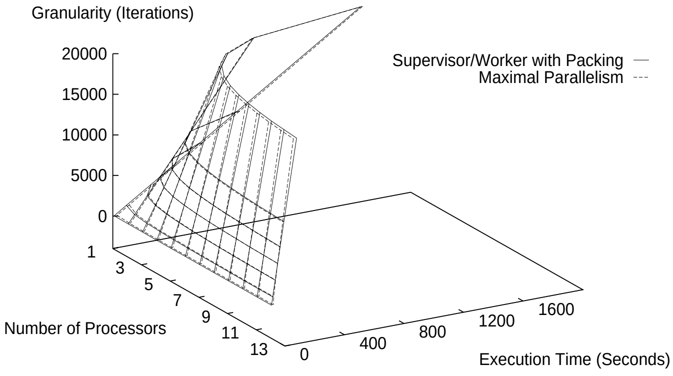

Fig. 13. 3-D Display of Controlled vs. Maximal Parallelism on MANNERS256.

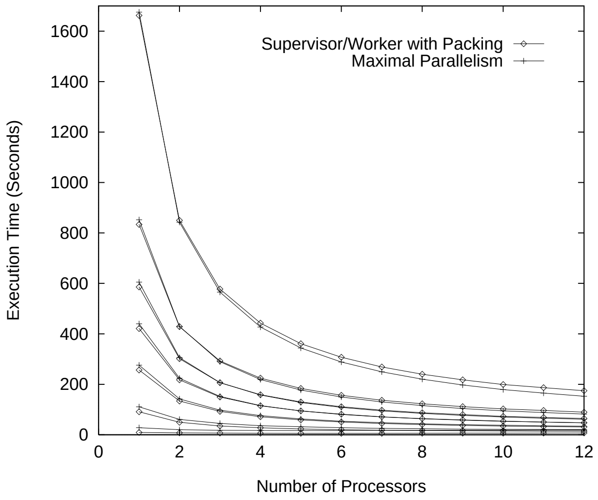

Fig. 14. 2-D Display of Controlled vs. Maximal Parallelism on MANNERS256.

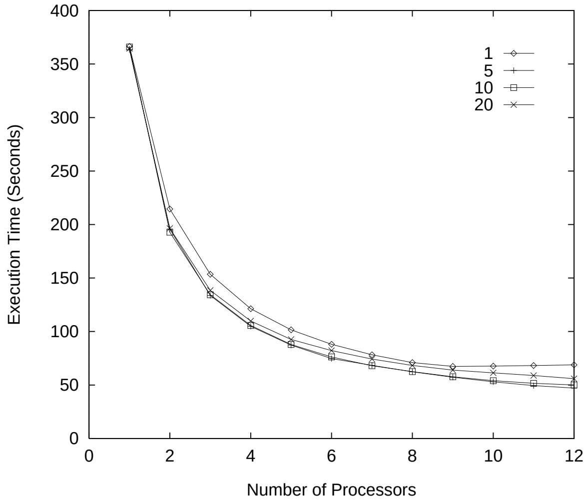

Fig. 15. MANNERS512 Execution Time on Different Grain Sizes with Rule Granularity Fixed at 1000.

applied, the available parallelism is left unexploited while under the maximal parallelism strategy these instantiations are always processed in parallel.

As a summary, the supervisor/worker strategy which creates only a limited number of threads is, in general, better than the maximal parallelism strategy which creates as many threads as the number of parallel instantiations. However, the actual performance gain still depends on the characteristics of the underlying program. When the degree of concurrency in an application is low, the maximal parallelism strategy is likely to be better.

The Effect of Grain Size

In last section, supervisor/worker with packing appeared to be the winner in overall performance. The question of determining appropriate grain size follows which is known as the *grain-size determination* problem [40]. It is unlikely that a single grain size is optimal for every program. We need, at a minimum, to determine if performance as a function of grain size is well behaved enough to offer a system default. We test the benchmark programs with grain sizes of 1 (i.e., no packing), 5, 10, and 20. Each one of them is tested with a fixed level of rule granularity. To understand the correlation of packing with respect to rule processing time, the same set of experiments is carried out with different levels of rule granularity.

Fig. 15 is the results on MANNERS512 with rule granularity set to 1000. We can observe quite clearly that packing is always better than no packing at this level of granularity. A grain size of 5 provides the best performance. On the other hand, with the granularity of a rule raised to 20000, the best performance is obtained when packing is not applied as shown in Fig. 16. Similar results can be observed on both LIFE (Fig. 17, Fig. 18) and WALTZ (Fig. 19, Fig. 20) except that grain size of 5 is not necessarily a clear winner over other grain sizes.

In summary, reducing the granularity of rules improves the results we get from packing. This is primarily because of the reduction in thread management, communication, and synchronization overhead. However, when the average

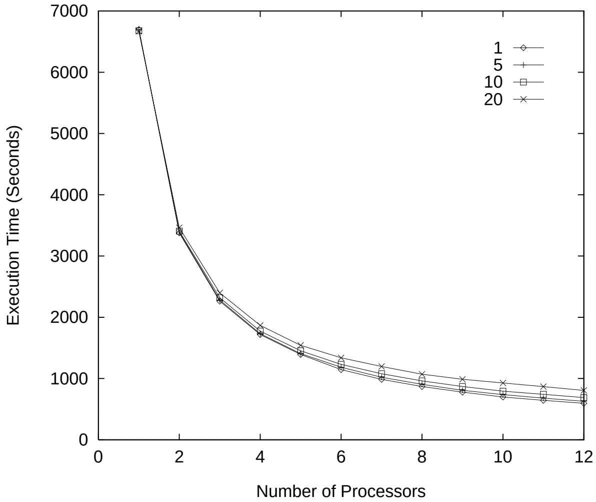

Fig. 16. MANNERS512 Execution Time on Different Grain Sizes with Rule Granularity Fixed at 20000.

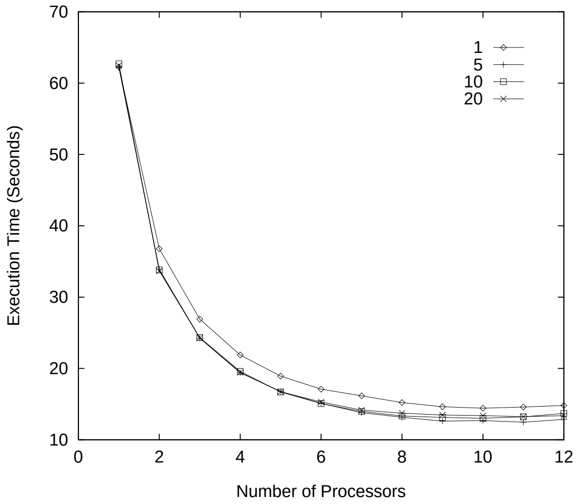

Fig. 17. LIFE40 Execution Time on Different Grain Sizes with Rule Granularity Fixed at 1000.

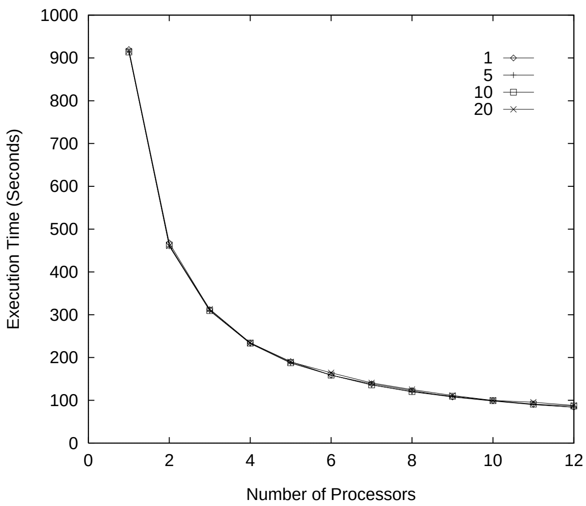

Fig. 18. LIFE40 Execution Time on Different Grain Sizes with Rule Granularity Fixed at 20000.

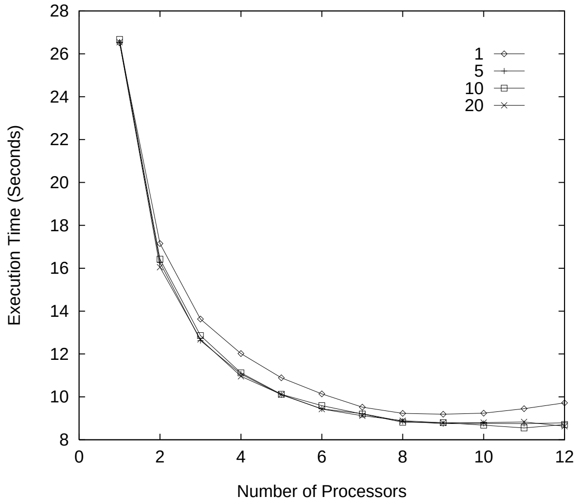

Fig. 19. WALTZ30 Execution Time on Different Grain Sizes with Rule Granularity Fixed at 1000.

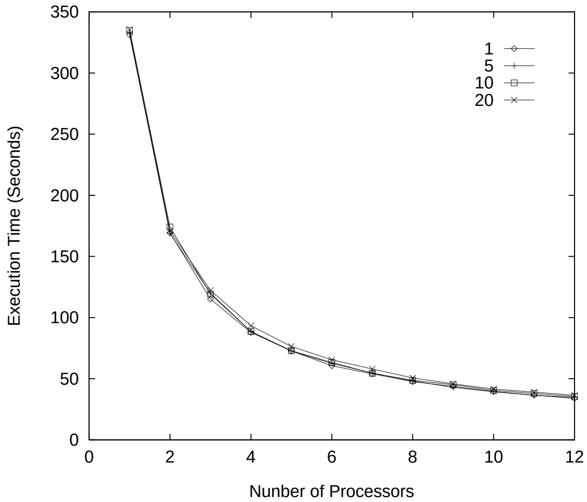

Fig. 20. WALTZ30 Execution Time on Different Grain Sizes with Rule Granularity Fixed at 20000.

granularity of the rules or the grain size becomes larger, the loss of parallelism offsets the benefit of packing. In general, packing with grain size 5 provides the best performance when the rule granularity is smaller than 5000. When the granularity is larger than 5000, it is better to do without packing. In other words, packing is good for cases where the per rule scheduling overhead is comparable or larger than that of the granularity of rule. This result suggests a dynamic scheduling strategy with either user-specified granularity assignment to each rule or an automatic estimation done by the system. We are investigating this issue in our full implementation of an object-based parallel rule language equipped with decomposition abstraction mechanisms. Similar issue has also been studied in other work such as [46].

VI. Related Work

Although there is no work speaking explicitely to decomposition abstraction in rule-based programming elements of our abstractions have appeared before. Stolfo and his colleagues [47], [48] were probably the first to point out the issue and provide constructs for set-oriented processing in rule-based programming. Our set selection condition is similar to their universal quantification. However, the exact semantics of the universal quantification has not been formally specified as we have done in this paper. Without care, the introduction of universal quantification can easily result in ambiguous and complicate rules.

Set-oriented constructs also appear in rule languages for data base systems. Examples include the Relational Production Language (RPL) [49] and the Starburst rules [50]. Our approach differs in that our concern is concurrency rather than relational completeness. To the best of our knowledge, the semantic-based interference analysis technique we developed has no direct counterpart in previous work.

Although absent in OPS5, most commercially developed rule languages include some explicit grouping mechanism similar to our context primitive.

There is a rich history of efforts toward parallel execution of rule languages. Most efforts fall into two broad categories, the improvement of match times through parallelism [5], [21], [51], [37], [52] and syntactically driven concurrent firing of rules. Initially a critical bottleneck, improvements in sequential match algorithms and advances in compilation techniques [36], [38], [45], [53] drastically reduced the proportion of time spent in the matching such that, the effects of Amdahl's law prevent significant speed-up by parallel matching alone. Multiple rule firing systems parallelize both match phase and other phases of the recognize-act cycle by identifying disjoint behavior at the level of rules or rule instantations [15], [54], [16], [17], [19]. The special challenge in these systems is that due to the free variables in the rules, syntactic analysis can not perform a complete analysis at compile time. Therefore, some calculations concerning the interference among rule instantiations must be deffered to the run-time. Neiman has shown through extensive analysis of design alternatives that, again due to the effect of Amdahl's law, even very modest overhead of this form will limit the total effectiveness of a system [35].

There have been two projects in parallel rule systems that fall outside these broad categories, PARULEL [33] and SPAM/PSM [55]. Both of which relate closely to the work in this paper.

PARULEL employs a two-level system where programmers are required to write *meta rules* that specify which instantiations represent incompatible rule instantiations. Using PARULEL a skilled programmer knowledgable in the details of the application can specify the concurrent execution of rule instantiations similar to those developed in Venus/DA[7]. However, in PARULEL, only explicit parallelism is pronounced. In Venus/DA the programmer need not be as aware of the nuances of parallel execution. A further weakness of the PARULEL system is that the determination of concurrently firable instantiations is determined at run-time by a rule system. In Section III-C, in the context of the **Disjoint** combinator, we related how PARULEL may have to examine $n!$ rule instantiations while Venus/DA needs to generate only one. Although parallelism may be applied recursively to speed the execution of the meta-level system, significant run-time overhead may still occur. Neiman's results apply to this architecture as well.

[7] Venus/DA is an object-based parallel rule language we are developing for the evaluation of the decomposition abstraction approach.

The SPAM/PSM system demonstrated large-scale parallel speedup on an image processing application [55]. Much of the effectiveness of this system was the result of a task level decomposition of the problem. Unfortunately the architecture of this system was a direct consequence of the detailed understanding of the application. Though this system contained little explicit development of general purpose decomposistion strategies it did demonstrate, early, that a good understanding of the application can bring about success on a parallel computer.

A. Comparative Evaluation

We compare the performance of our simulation results with the performance results reported by other work. Detail and fair comparison is not always possible because of a number of practical issues:

- The version of the benchmark program may not be the same.
- Some reported speedup figures are the ratio of sequential cycles and parallel cycles instead of the real execution time. This is known to bias toward the parallel version.
- Many performance figures reported did not include the size of the data been used.
- Most previous work did not provide scaling results.

In Table III, we compare the best reported speedup on the Waltz, Toru-Waltz, and Life programs. Toru-Waltz is a dialect of Waltz designed to be easier to parallelized. Some of the numbers are taken from the figures of the cited papers since no exact number is reported.

We note that because of hardware limitation, we can only have performance results up to 12 processors. Judged from the trend in the performance figures shown in earlier sections, the DA system has the potential of achieving higher speedup with more computation resources.

Even though the number of test programs and the results are not enough to show that DA approach significantly improves upon previous work on a wide range of applications, they do indicate certain degree of consistency and are encouraging enough for the purpose of our simulation, i.e. to have an early evaluation of the potential of DA approach and to experiment with alternative scheduling strategies for the full implementation.

VII. Conclusions and Future Work

We have shown in this paper that production systems and rule-based programs do encompass high degrees of concurrency, and it is possible to effectively exploit such levels of parallelism. These results are obtained when a programmer is provided with the mechanisms we proposed to specify the natural parallelism inherent in the application

Systems	#PE's	Waltz Speedup	Toru-Waltz Speedup	Life Speedup
DA	12	9.60	–	10.92
Neiman [35]	20	–	9.00	–
PARULEL [56]	6	2.66	–	–
CREL [16]	12	12.00	8.50	6.50
Kuo & Moldovan [17]	8	–	3.18	–
Schmolze [19]	16	3.00	–	–

TABLE III
Speedup Comparison

domain. The semantics of the mechanisms is well defined allowing a certain independence of how they are actually implemented. Programmers are free from the need to worry about most of the synchronization and communication issues. The approach of using data relationships (functional dependency in particular) in the derivation of information for parallelism is a promising direction for exposing hidden concurrency that are not explicit specified. Using decomposition abstraction, sufficient parallelism reveals the standard problems of granularity and scheduling strategies. Based on a new technique of rapid system development for early evaluation and experimentation, we show that the proposed set of mechanisms can be efficiently implemented on shared-memory multiprocessors.

Even with effective implementation, the granularity of rules is still critical to the actual performance gain. If this information is available to the run-time scheduler, then a strategy of supervisor/worker with dynamic grain size is very likely to perform better than the strategy with fixed grain size or no packing at all. Obviously, a user-specified granularity level assigned to each rule (similar to a priority assignment or certainty factor) is an immediate solution. We plan to provide this feature in our language implementation. However, unlike other proposed mechanisms in this paper, proper granularity assignment is in general not readily available from the application semantics since it involves an estimation of processing time for each rule. Another way is to have the system estimate the right granularity for a rule either from its syntactic structure or from an execution profile.

We are now embodying our decomposition abstraction mechanisms on a C/C++-based modular rule language called Venus [57] on the Sequent Symmetry shared-memory multiprocessor. We choose Venus because it is probably the first sequential rule-based programming language to provide both a declarative syntactic and semantic mechanism to support top-down modular design of rule-based programs. The embodiment of our mechanisms instantly converts the language into a parallel rule language, which is called Venus/DA. The language system is not only intended to serve as a test-bed for semantic-based exploration of parallelism in production systems, but also to be used for the planned extension of rule-based technology into database applications.

Acknowledgments

This work is partially supported by DARPA under grant DABT63-92-C-0042.

We would like to thank the anonymous referees for their invaluable suggestions on the improvement and presentation of this paper. Special thanks to Professor Salvatore J. Stolfo at the Columbia University, for his in-depth discussion and exchange of ideas related to this work, and to Lance Obermeyer at Applied Research Lab, for his help on the system and implementation issues.

References

[1] B. G. Buchanan and E. H. Shortliffe, *Rule-Based Expert Systems*, Addison-Wesley, Reading, MA, 1984.

[2] J. McDermott, "R1: A rule-based configurer of computer systems", *Artificial Intelligence*, vol. 19, pp. 39–88, 1982.

[3] C.L. Forgy, Anoop Gupta, A. Newell, and R. Wedig, "Initial assessment of architectures for production systems", in *Proc. 4th National Conference on Artificial Intelligence (AAAI-84)*, Austin, TX, Aug. 1984, pp. 116–120.

[4] *Proceedings of RIDE'94: Workshop on Research Issues in Data Engineering/Active Database Systems.* IEEE Computer Society Press, Feb. 1994.

[5] Anoop Gupta, *Parallelism in Production Systems*, PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Mar. 1986.

[6] Steve Kuo and Dan Moldovan, "The state of the art in parallel production systems", *Journal of Parallel and Distributed Computing*, vol. 15, no. 1, pp. 1–26, May 1992.

[7] Daniel P. Miranker, "Special issue on the parallel execution of rule systems: Guest editor's introduction", *Journal of Parallel and Distributed Computing*, vol. 13, no. 4, pp. 345–347, Dec. 1991.

[8] A. Sohn and J-L Gaudiot, "A survey on the parallel distributed processing of production systems", *International Journal on Artificial Intelligence Tools*, vol. 1, no. 2, pp. 279–331, June 1992.

[9] D. Klappholtz, X. Kong, and A. D. Kallis, "Refined Fortran: An update", in *Proc. of Supercomputing '89*, Santa Clara, CA, Nov. 1989.

[10] S. Hiranandani, K. Kennedy, C. Koebel, U. Kremer, and C. Tseng, "An overview of the Fortran D programming system", in *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, Aug. 1991.

[11] David A. Mundie and David A. Fisher, "Parallel processing in Ada", *IEEE Computer*, vol. 19, no. 8, pp. 20–25, Aug. 1986.

[12] N.H. Gehani and W.D. Roome, "Concurrent C", *Software - Practice and Experience*, vol. 16, no. 9, pp. 821–844, 1986.

[13] Robert H. Jr. Halstead, "Multilisp: A language for concurrent symbolic computation", *ACM Trans on Programming Languages and Systems*, vol. 7, no. 4, pp. 501–538, Oct. 1985.

[14] A. Brogi and P. Ciancarini, "The concurrent logic language Shared Prolog", *ACM Trans on Programming Languages and Systems*, vol. 13, no. 1, pp. 99–123, 1991.

[15] T. Ishida and Salvatore J. Stolfo, "Toward the parallel execution of rules in production system programs", in *Proc. IEEE Intl. Conf. on Parallel Processing*, 1985, pp. 568–574.

[16] Chin-Ming Kuo, Daniel P. Miranker, and James C. Browne, "On the performance of the CREL system", *Journal of Parallel and Distributed Computing*, vol. 13, no. 4, pp. 424–441, Dec. 1991.

[17] Steve Kuo and Dan Moldovan, "Implementation of multiple rule firing production systems on hypercube", *Journal of Parallel and Distributed Computing*, vol. 13, no. 4, pp. 383–394, Dec. 1991.

[18] Daniel P. Miranker, C. Kuo, and J. Browne, "Parallelizing compilation of rule-based programs", in *Proc. IEEE Intl. Conf. on Parallel Processing, Vol. II*, 1990, pp. 247–251.

[19] James G. Schmolze, "Guaranteeing serializable results in synchronous parallel production systems", *Journal of Parallel and Distributed Computing*, vol. 13, no. 4, pp. 348–365, Dec. 1991.

[20] Shiow-yang Wu and James C. Browne, "Explicit parallel structuring for rule based programming", in *Proc. 7th International Parallel Processing Symposium*, 1993, pp. 479–488.

[21] Anoop Gupta, Charles Forgy, and Allen Newell, "High-speed implementation of rule-based systems", *ACM Trans on Computer Systems*, vol. 7, no. 2, pp. 119–146, May 1989.

[22] L. Brownston, R. Farrel, E. Kant, and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, MA, 1985.

[23] T. A. Cooper and N. Wogrin, *Rule Based Programming with OPS5*, Morgan Kaufmann, Palo Alto, 1988.

[24] Francois Bancihon and Setrag Khoshafian, "A calculus for complex objects", in *Proc. 15th ACM Symp. on Principles of Database Systems*, 1986, pp. 53–59.

[25] Elisa Bertino, Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella, "Object-oriented query languages: The notion and the issues", *IEEE Trans. on Knowledge and Data Engineering*, vol. 4, no. 3, pp. 223–237, June 1992.

[26] A.J. Bernstein, "Analysis of programs for parallel processing", *IEEE Trans. Electronic Computers*, vol. 15, pp. 757–762, 1966.

[27] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987.

[28] C. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

[29] F. Darema-Rogers, V.A. Norton, and G.F. Pfister, "Using a single-program-multiple-data computation model for parallel execution of scientific applications", Research Report RC 11552, IBM T.J. Watson Research Center, Yorktown Heights, 1985.

[30] D. Hillis and G. Steele, "Data parallel algorithms", *Comm ACM*, vol. 29, pp. 1170–1183, 1986.

[31] Hans Zima and Barbara Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, New York, 1991.

[32] Douglas N. Gordin and Alexander J. Pasik, "Set-oriented constructs: From Rete rule bases to database systems", in *ACM SIGMOD Intl. Conf. on Management of Data*, 1991, pp. 60–67.

[33] 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*, vol. 13, no. 4, pp. 366–382, Dec. 1991.

[34] David Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, Maryland, 1983.

[35] Daniel E. Neiman, "Issues in the design and control of parallel rule-firing production systems", *Journal of Parallel and Distributed Computing*, vol. 23, no. 3, pp. 338–363, Dec. 1994.

[36] Ho Soo Lee and Marshall I. Schor, "Match algorithms for generalized Rete networks", *Artificial Intelligence*, vol. 54, no. 3, pp. 249–274, Apr. 1992.

[37] Daniel P. Miranker, *TREAT: A New and Efficient Match Algorithm for AI Production Systems*, PhD thesis, Department of Computer Science, Columbia University, 1987, Also published by Morgan-Kaufmann, 1990.

[38] Daniel P. Miranker, David A. Brant, B.J. Lofaso, and David Gadbois, "On the performance of lazy matching in production systems", in *National Conference on Artificial Intelligence*, July 1990, pp. 685–692.

[39] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, "PRESTO: A system for object-oriented parallel programming", *Software - Practice and Experience*, vol. 18, no. 8, pp. 713–732, Aug. 1988.

[40] B. Kruatrachue and T. Lewis, "Grain size determination for parallel processing", *IEEE Software*, vol. 5, no. 1, pp. 23–32, Jan. 1988.

[41] Hesham El-Rewini, Theodore G. Lewis, and Hesham H. Ali, *Task Scheduling in Parallel and Distributed Systems*, PTR Prentice Hall, Prentice-Hall, Inc., 1994.

[42] David A. Brant, Timothy Grose, Bernie Lofaso, and Daniel P. Miranker, "Effects of database size on rule system performance: Five case studies", in *Proc. 17th Intl. Conf. on Very Large Data Bases*, 1991, pp. 287–295.

[43] D. L. Waltz, "Understanding line drawings of scenes with shadows", in *The Psychology of Computer Vision*, P Winston, Ed., pp. 19–91. McGraw Hill, New York, NY, 1975.

[44] G. Kiernan, C. de Maindreville, and E. Simon, "Making deductive database a practical technology: A step forward", in *ACM SIGMOD Intl. Conf. on Management of Data*, 1990, pp. 237–246.

[45] Daniel P. Miranker and B.J. Lofaso, "The organization and performance of a TREAT based production system compiler", *IEEE Trans. on Knowledge and Data Engineering*, vol. 3, no. 1, pp. 3–10, Mar. 1991.

[46] Hasanat M. Dewan, Salvatore J. Stolfo, , Mauricio Hernández, and Jae-Jun Hwang, "Predictive dynamic load balancing of parallel and distributed rule and query processing", in *Proceedings of the 1994 ACM SIGMOD International Confernece on Management of Data*, 1994, pp. 277–288, Also appears in SIGMOD RECORD, 23(2):277–288, June 1994.

[47] Salvatore J. Stolfo and D.E. Shaw, "DADO: A tree-structured machine architecture for production systems", in *Proc. 2nd National Conference on Artificial Intelligence (AAAI-82)*, 1982.

[48] Michael van Biema, Daniel P. Miranker, and Salvatore J. Stolfo, "The do-loop considered harmful in production system programming", in *Expert Database Systems*, Larry Kerschberg, Ed., pp. 177–190. The Benjamin/Cummings Publishing Company, Inc., 1986.

[49] Lois M.L. Delcambre and James N. Etheredge, "The relational production language: A production language for relational databases", in *Expert Database Systems, Proc. 2nd Intl. Conf. on Expert Database Systems*, Larry Kerschberg, Ed., pp. 333–351. Benjamin/Cummings, Redwood City, CA, 1988.

[50] Jennifer Widom and Sheldon J. Finkelstein, "A syntax and semantics for set-oriented production rules in relational database systems", *SIGMOD RECORD*, vol. 18, no. 3, pp. 36–45, Sept. 1989.

[51] M.A. Kelly and R.E. Seviora, "An evaluation of DRete on CUPID for OPS5 matching", in *Proc. 11th International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, Aug. 1989.

[52] Salvatore J. Stolfo, "Five parallel algorithms for production system execution on the DADO machine", in *National Conference on Artificial Intelligence*, 1984, pp. 300–307.

[53] M.I. Schor, T.P. Daly, H.S. Lee, and B.R. Tibbitts, "Advances in Rete pattern matching", in *AAAI-86*, 1986, pp. 226–232, YES/RETE.

[54] "Special issue on the parallel execution of rule systems", *Journal of Parallel and Distributed Computing*, vol. 13, no. 4, Dec. 1991.

[55] Wilson Harvey, Dirk Kalp, Milind Tambe, David McKeown, and Aleen Newell, "The effectiveness of task-level parallelism for production systems", *Journal of Parallel and Distributed Computing*, vol. 13, no. 4, pp. 395–411, Dec. 1991.

[56] Hasanat M. Dewan, Salvatore J. Stolfo, and Leland Woodbury, "Scalable parallel and distributed expert database systems with predictive load balancing", *Journal of Parallel and Distributed Computing*, vol. 22, no. 3, pp. 506–522, Sept. 1994.

[57] James C. Browne and et. al., "A new approach to modularity in rule-based programming", in *ICTAI'94: IEEE Intl. Conf. on Tools for Artificial Intelligence*, 1994.

Shiow-yang Wu is an assistant professor in the Institute of Computer Science and Information Engineering of National Dong Hwa University, Hualien, Taiwan, ROC. He received the BS and MS degrees in computer engineering from National Chiao Tung University, Hsinchu, Taiwan, ROC, and the PhD degree in computer science from the University of Texas at Austin in 1984, 1986, and 1995, respectively. His research interests are parallel/distributed processing, databases, artificial intelligence, mobile computing, and distance learning technologies.

Daniel P. Miranker is an Associate Professor of Computer Science at the University of Texas at Austin. He earned a BS (1979) in Mathematics at MIT and a PhD (1987) in Computer Science at Columbia University with a thesis on the parallel execution of expert-system programs on DADO, a 1,023 processor multicomputer using the TREAT match algorithm. He is copatent holder on the DADO architecture. Prof. Miranker has been at the University of Texas since 1986. His work encompasses the integration of AI knowledge-base methodologies with database systems. These ideas materialize as efforts in Active-Databases, Data Warehousing and Query Optimization, often targeting both parallel and distributed computer systems.

James C. Browne earned his PhD in Chemistry at the University of Texas in 1960. He is currently professor of computer science and physics at the University of Texas at Austin. He is an internationally recognized expert in parallel architectures and programming. He leads a major project in the development of graph-based environment (CODE) for parallel programming. Browne's research interests span parallel architectures and parallel programming, performance measurement and analysis, operating systems and VLSI synthesis. His early research career in computational physics adds familiarity with large scale computational problems. Browne has been a member of the Technical Advisory Committee for Lawrence Livermore Laboratories, Los Alamos National Laboratories, the National Bureau of Standards, the National Science Foundation–Computer Research Section and the DARPA Information Science and Technology Office. He is a fellow of the British Computer Society, the American Physical Society and the American Association for the Advancement of Science. He has been Chairman of the ACM Special Interest Group on Operating Systems and has been in the past an Associate Editor of several journals. Browne has published approximately 100 papers in computational physics and 150 papers in Computer Science. He has given approximately 200 invited lectures to most major Computer Science and Physics Universities in the United States.

List of Tables

List of Figures