Explicit Parallel Structuring for Rule-Based Programming

Shiow-yang Wu and James C. Browne

Department of Computer Sciences,
The University of Texas at Austin,
Austin, TX 78712-1188

Abstract

This paper presents semantically-based explicit parallel structuring for rule-based programming systems. Explicit parallel structuring appears to be necessary since compile-time dependency analysis of sequential programs has not yielded large scale parallelism and run-time analysis for parallelism is restricted by the execution cost of the analysis. Simple language extensions specifying semantics of rules are used to define parallel execution behavior at the rule level. Type definitions for working memory elements are extended to include relationships within and among objects which define the parallelism allowed on instances of object types.

The first result presented is that the algorithms implemented by commonly used benchmark rule-based programs contain scalable parallelism. The second result is that much of that parallelism can be captured by simple and modest extensions of rule-based languages which are analogies of models and constructs used for specification of parallel structures in imperative programming languages. A sketch is given for a comprehensive language system which exploits specification of semantics defining parallel structures in both object-definition and executable segments of rule-based programs.

1. Introduction

This paper demonstrates that the rule-based programming paradigm can be formulated so as to yield scalable and potentially large-scale parallelism well beyond the degree of parallelism attained in previous work. The first step was to analyze common benchmark problems and the rule-based programs written to realize them for potential parallelism. While the amount of parallelism found in previous work has typically been modest and not necessarily scalable, we found that several of these programs had the potential for massive and scalable parallelism. The second step was to determine the minimum degree of additional semantic information which had to be added to these programs in order to allow compile time detection and realization of this potential parallelism. This was done through systematic analysis of the algorithms and determining how parallel semantics for the algorithm could be embedded in the rule-based program. The models of parallel structuring used for imperative programs (particularly data partitioning [7, 5] and data parallel [4] models of parallel structuring) suggested analogies for rule-based programs.

Section 2 gives the results of simulated parallel execution for several programs used as benchmarks in previous studies and defines the sources of parallelism in these programs and algorithms. Section 3 gives the parallel structuring constructs and draws the analogies with parallel imperative programming and other models of parallel computation. Section 4 surveys previous approaches to parallel structuring of rule-based programs. Section 5 gives a brief sketch of the full approach which we are undertaking.

2. Potential for Parallelism in Rule-Based Programs

As a first step to explore the potential of large-scale and scalable parallelism in production systems, we analyzed several benchmark problems and the corresponding rule-based programs implemented in OPS5. These programs have been widely used in previous studies to evaluate the effectiveness of language extensions and compilation techniques [8, 11, 9, 13]. While the amount of parallelism found in previous research has typically been modest and not necessarily scalable, we found, contrary to previous expectations, that most
of these programs had the potential for massive and scalable parallelism. In this section, we give the results of simulated parallel execution of three of the benchmark programs and identify the sources of parallelism in the algorithms which strongly suggest the need for explicit parallel structuring mechanism in rule-based programs.

The programs are Life, Waltz, and Manners as listed in Table 1. All simulation results are obtained by going through the following steps:

- First, OPS5 benchmark programs and their sequential execution traces are carefully studied and analyzed to identify the potential parallelism in the problems and the algorithms.
- Then, all programs are reformulated such that inherent parallelism can be effectively exploited with the explicit parallel structuring constructs proposed in Section 3.
- Both the results of sequential and parallel executions are collected in terms of number of execution cycles.

The speedup is measured by comparing the number of cycles between sequential and parallel executions. Sequential cycles are obtained by actually running the OPS5 programs using ops5c [11] on SUN SPARCs and HP 9000 workstations. Parallel cycles are calculated by hand with the assumption of unlimited resources, no overhead and no contention. To see whether our approach scales up, the performance results of increasing problem size are collected for each program.

2.1 Life

There are three sets of results on the Life program. The original program contains a sequential print context to print out intermediate and final results. Since this process is inherently sequential, according to Amdahl's law, the speedup is limited by the sequential part. The simulation result on this version of the program is given in Figure 1. Because of the limit imposed by the sequential printing, the potential speedup is quite small but close to the theoretical maximum speedup calculated following Amdahl's law. To measure the actual speedup in the computation part, we have obtained the results on two slightly modified versions of the program. The first one is the version without printing intermediate results. This

\[ S \leq \frac{1}{f + (1 - f)/p}. \]

1Amdahl's law says that if \( f \) is the fraction of a computation that must be performed sequentially, where \( 0 \leq f \leq 1 \), then the maximum speedup \( S \) achievable by a parallel computer with \( p \) processors is \( S \leq \frac{1}{f + (1 - f)/p} \).

Figure 1: Life Speedup (with printing).

Figure 2: Life Speedup (without printing intermediate results).

is presented in Figure 2 together with the theoretical speedup limits. The second set of results, which is in Figure 3, is to measure the computation part alone without any printing.

The key reason for such impressive results resides in the identification of the following sources of inherent parallelism in the Life program:

- All live cells can generate neighbors at the same time.
- All cells can compute their neighbors simultaneously.
- The generation update of all cells can be executed in parallel.

The consequence is that it takes a constant number of parallel cycles to do the computation part while sequential cycles increase dramatically with the problem size. While these sources of parallelism are quite difficult, sometimes impossible, to detect at compile-time using dependency analysis techniques developed in previous research, they are straightforward to specify using the constructs proposed in Section 3.
<table>
<thead>
<tr>
<th>Program</th>
<th>No. rules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Life</td>
<td>16</td>
<td>A simulation program implements Conrad's LIFE.</td>
</tr>
<tr>
<td>Waltz</td>
<td>33</td>
<td>A constraint satisfaction problem using Waltz's algorithm for scene labeling.</td>
</tr>
<tr>
<td>Manners</td>
<td>8</td>
<td>A combinatorial search problem for seat assignment.</td>
</tr>
</tbody>
</table>

Table 1: Benchmark programs used in the simulation.

![Graph showing Life Speedup](image)

Figure 3: Life Speedup (without printing).

![Graph showing Waltz Speedup](image)

Figure 4: Waltz Speedup.

2.2 Waltz

Because of the time taken to simulate large data sets, we only performed simulations on small data sets for the Waltz program. Nevertheless, the results are still quite encouraging as depicted in Figure 4.

The potential parallelism in the Waltz program rests on the important semantic information in the problem and the organization of the data:

- Each line is associated with exactly two edges with opposite end points.
- The type of a junction is unique and each junction is associated with a unique set of edges.

From this information, we can identify the following additional parallelism in the program:

- All junctions can be made concurrently without interfering with one another.

- Since each labeling rule matches a single junction and its associated edges, all enabled labeling rules with disjoint matching working memory elements can be fired in parallel.

This is clearly scalable parallelism because the larger the problem (in terms of number of line segments in the input drawing), the more junctions and edges a drawing has, which results in more rules being eligible for firing in parallel. Like the case for the Life program, the additional parallelism is a characteristic of the problem and is derived from the implicit design decisions of the Waltz program. Without explicit parallel structuring mechanisms to express this knowledge, a general dependency analysis technique can only detect problem independent parallelism which is quite modest and not necessarily scalable.

2.3 Manners

The Manners program is a good example of the advantage of explicit parallel structuring over that of the compile-time dependency analysis approach. It contains a hot-spot rule which fires repeatedly during the execution of the program. The number of repetitions increases dramatically with the problem size. After carefully analyzing the program and traces, it turns out that all instantiations of this rule can be fired in parallel. However, this can not be determined at compile time using dependency analysis alone. With the explicit parallel structuring mechanisms to be discussed in the following section, it simply takes an ALL combinator to express this semantic which results in an impressive linear speedup as shown in Figure 5.

As a summary, the examples indicate that application specific information is the key to the effective exploitation of inherent parallelism in the problems and the rule-based programs. The simulation results further exhibit the potential of explicit parallel structuring for large-scale and scalable parallelism. In [16], we have made an effort to systematically explore the potential and sources of this level of parallelism beyond that of compile-time dependency analysis techniques developed by previous research. Because of the use of application specific knowledge and the semantic nature of this approach, we call it semantic level parallelism in production systems.
3 Explicit Parallel Structuring of Rule-Based Programs

For the exploitation of semantic level parallelism in production systems, we show that by simple but semantically rich explicit parallel structuring constructs, much of the application specific parallelism identified in the previous section can be effectively expressed.

Instead of a flat working memory as in OPSS, we have adopted an object-based model for a new parallel rule language called OPRL (Object-based Parallel Rule Language). Under this model, the role of working memory elements is replaced by objects and the condition part of a rule is to select qualified objects for the operations in the action part. A brief sketch of the full approach will be discussed in Section 5. Here we introduce only some explicit parallel structuring constructs for the production rules by a series of examples.

Example 1. Object Selection and Set Selection Conditions Figure 6 is an OPRL version of the make-neighbors rule in the Life program. This example demonstrates the format of a rule and how object and set of objects are selected for processing in parallel.

A rule in OPRL has the form

```plaintext
rule (rule_name) {
  (enabling_conditions)
  --> (actions)
}
```

where the (enabling_conditions) is a set of conditions for selecting qualified objects and (actions) are operations on the selected objects. A rule is said to be enabled if there exists a consistent set of objects such that all enabling conditions are satisfied. A condition

```plaintext
rule make_neighbors {
  (p:Priority :: state == 0)
  [c:Cell :: alive == "alive",
   n_made == "no"]
  (g:Goal :: value == "do_neighbors")
  -->
  c.n_made := "yes" ||
  New_Neighbor(c.x - 1, c.y - 1) ||
  New_Neighbor(c.x - 0, c.y - 1) ||
  New_Neighbor(c.x + 1, c.y - 1) ||
  New_Neighbor(c.x + 1, c.y + 0) ||
  New_Neighbor(c.x - 1, c.y + 1) ||
  New_Neighbor(c.x - 0, c.y + 1) ||
  New_Neighbor(c.x - 1, c.y - 0)
}
```

Figure 6: Object and set selection conditions.

of the form

```
((obj_var) : (class_name) :: (restrictions))
```

is called an object selection condition the purpose of which is to select an object, denoted by the object variable (obj_var), from the class (class_name) such that the constraints specified in (restrictions) are satisfied. When square brackets are used instead of parentheses, the condition is called a set selection condition with the semantics of selecting the set of all objects, satisfying the restrictions, from the specified class. The condition is considered not satisfied if the resulting set is empty. Any reference to the object variable of a set selection condition is to be applied to all objects in the set. As in the example above, all objects of the class Cell with attributes alive and n_made equal "alive" and "no" respectively are selected and all neighbors are made in parallel, as denoted by the parallel bar || for parallel composition of operations. Clearly, this is a SIMD or SPMD (Single Program Multiple Data) style of computation. By assuming this, there is no need to provide any construct for iterating through the objects of the selected set in the action part. This is a deliberate design decision because we want our language to be concise and clean. Besides, if the purpose of iterating through the set is to do different things on different objects, some form of testing must be done to discriminate between objects that are to be processed in different ways. This can be accomplished just as well by having more specific rules such that all objects of the selected set are to be processed in the same way.
rule compute_neighbors {
  (p:Priority :: state == 0) (c:Cell)
  {[n:Neighbor :: x == c.x, y == c.y]} (g:Goal :: value == "do_neighbors")
  --> c.neighbors := Count(n) || n.remove /* or Remove(n) */
}

Figure 7: Aggregate operator on set.

A set selection condition can be enclosed in braces with a set variable denoting the set. Then aggregate operators such as Count, Max, Min, etc. can be applied on the selected set as demonstrated in the next example.

Example 2. Set Variable and Aggregate Operators In Figure 7, all neighbors having the same x and y values as the selected cell object are collected in a set and denoted by the set variable N. Count(N) then counts the number of objects in the set and thus computes the number of neighbors of a cell in a single rule firing.

In fact, there is another source of parallelism which is left unexplored in the example. Namely, not only can the number of neighbors of a single cell be computed in a single rule firing, the same can be done on all cells at the same time. This can be accomplished by using the ALL combinator.

Example 3. ALL Combinator Figure 8 is another version of the compute_neighbors rule demonstrating the ALL combinator used with set selection condition and aggregate operator. In this rule, the number of neighbors of each cell is computed in parallel.

Any number of object and set selection conditions can be grouped together by the ALL combinator with the semantics of forming all sets of consistent selections together with other regular conditions and firing all of them in parallel. As in the example above, an instantiation is generated for each consistent pair of Cell object and Neighbor set. Then all instantiations are fired in parallel which means the number of neighbors is computed for all cells at the same time.

In addition to the ALL combinator, OPRL provides another two combinators, namely, ANY and DISJOINT, for the specification of the other most commonly encountered patterns of rule firing.

rule compute_neighbors {
  (p:Priority :: state == 0) (c:Cell)
  {[n:Neighbor :: x == c.x, y == c.y]} (g:Goal :: value == "do_neighbors")
  --> c.neighbors := Count(n) || n.remove /* or Remove(n) */
}

Figure 8: ALL combinator.

rule make_3_junction {
  (s:Stage :: value == "detect_junction")
  (ANY)
  (e1:Edge :: joined == FALSE) (e2:Edge :: pi == e1.p1, p2 <> e1.p2, joined == FALSE)
  (e3:Edge :: pi == e1.p1, p2 <> e1.p2 <> e2.p2, joined == FALSE))
  --> New_Junction(
    "type mk_3_junc(e1.p1, e2.p2, e3.p2),"
    "base_point e1.p1" ||
    e1.joined, e2.joined, e3.joined := TRUE, TRUE, TRUE
  )
}

Figure 9: ANY combinator.

Example 4. ANY Combinator Figure 9 is an OPRL rule for demonstrating the ANY combinator which is used to specify that one and only one instantiation need to be generated and fired with any consistent combination of objects.

The semantics of the ANY combinator is to select any consistent set of objects satisfying all the conditions grouped together. Any one, and only one, consistent set of objects can be selected to form an instantiation. If no consistent binding can be found, the condition is considered not satisfied. This example also demonstrates the use of multiple assignment statements in the action part as a shorthand for parallel compositions of several assignment operations.

The purpose of the ANY combinator is to allow more precise specification of rule behavior so as to reduce run-time overhead of generating unnecessary instantiations. Without it, six instantiations will be generated.
rule make_3junction {
    (s:Stage :: value == "detect_junction")
    (DISJOINT
     (e1:Edge :: joined == FALSE)
     (e2:Edge :: p1 == e1.p1, p2 <> e1.p2, 
       joined == FALSE)
     (e3:Edge :: p1 == e1.p1, 
       p2 <> e1.p2 <> e2.p2, 
       joined == FALSE)
    )
    -->
    Neu_junction(
        "type mk_3_junc(e1.p1, e2.p2, e3.p2),
        "base_point e1.p1" ||
        e1.joined, e2.joined, e3.joined := 
        TRUE, TRUE, TRUE
    )
}

Figure 10: DISJOINT combinator.

for any three consistent Edge objects. Only one (any one of the six) need to be fired. Note, however, that the programmer is responsible for preventing other instantiations from being generated in the next cycle if this is the intended meaning. The assignments of the value TRUE to the joined attribute of all three edge objects are to prevent the same set of edges from being selected again.

Even though the use of ANY combinator often results in a significant amount of saving in processing time, it does not provide any hint for parallel execution. For the example above, only one 3-junction is made at a time. As discussed previously in Section 2.2, the design of the Waltz program actually implies that all 3-junctions can be made at the same time. For this purpose, we need the DISJOINT combinator.

Example 5. DISJOINT Combinator The use of DISJOINT combinator is illustrated in Figure 10. Everything else is exactly the same as previous example except that ANY is replaced by DISJOINT.

As indicated by its name, the DISJOINT combinator is used to form a set of instantiations with disjoint selection of the enclosed objects. For the example above, an instantiation is generated for each set of three edges that satisfies the constraints. Different instantiations must not have edges in common, i.e., the selected edges must be disjoint. This prevents redundant instantiations for the same set of edges from being generated while allows instantiations on different set of edges to be formed and fired in parallel. In this case, all 3-junctions are made at the same time.

In general, combinators can be nested. At this point of our research, however, we will not go into this in any detail since we believe that three combinators without nesting, together with set-oriented constructs, are rich enough for most practical purposes. It is also possible to study the algebraic properties of these combinators. Again, this is not within our current research goal.

In contrast to the enabling conditions, the action part of a rule is simply a collection of primitive operations composed by sequential ("," ) and parallel ("||") compositions. A primitive operation is in one of the following forms:

- object creation: Neu_Neighbor(1, 1).
- object and set removal: c.remove, Remove(N).
- operation invocation: Such as, s.pop where s is a stack object.
- assignment: A multiple assignment of the form
  \( (variable)\{, (variable)\} := (exp)\{, (exp)\}\).  
- output action: write("plot" e.p1 e.p2).
- halt action: halt.

They are designed to be simple and accordant with the RHS of traditional production system while expressive enough to be consistent with the object framework of OPRL and for explicit parallelism. Other features can be found in [18].

As a summary, the design of OPRL rules is to enable the programmers to explicitly specify, in a form that is intuitively appealing and easy to understand, exactly how each rule is supposed to behave in a multiple rule firing environment. OPRL promotes the design of rules in such a way that no redundant instantiations need to be generated and all generated instantiations in a cycle are to be fired in parallel.

4 Related Work

In this section, related work on multiple rule firing production systems is briefly surveyed. The purpose is to compare our approach with those in the literature that are closely related. For a more complete survey of parallel production systems in general, see [10].
4.1 CREL

CREL [8] is a representative language that takes the compile-time dependency analysis approach. The syntax of CREL is identical to OPS5 but the semantics is different. Rules are executed asynchronously. A parallel execution is correct if it is serializable. A bipartite data dependency graph adopted from [6] is used for dependency analysis of rules. Two types of interference between rules are identified as special properties of the dependency graph. An algorithm is provided to find the mutual exclusion sets in a program, which are defined to be sets of rules that cannot be statically determined to be executable in parallel and thus require synchronization. Parallel execution is thus guaranteed to be serializable if multiple rules selected from the same mutual exclusion set do not form a cycle with conflicting interferences. For asynchronous execution, the synchronization set is defined to be a set of rules where global synchronization is needed to ensure serializability. It is shown that a synchronization set is actually the maximum cycle among mutual exclusion sets where global synchronization is needed. The partition of rules by synchronization sets then has the desired property that global synchronization is no longer needed among different partitions for correct execution. A program can thus be executed completely asynchronously.

Another contribution of this research is the optimizing transformations performed on programs to further increase the available parallelism. Several transformation techniques are developed and proved to be quite effective.

While CREL is a comprehensive language system that exploits potential parallelism in all phases of a production system cycle, it is limited by the conservative requirement of the compile-time dependency analysis approach. On the contrary, the explicit parallel structuring approach taken herein does not have this limitation. And the advantages of both semantic and syntactic approaches can be combined under our framework.

4.2 PARULEL

Among all the multiple rule firing production systems, PARULEL [15] is probably the only one that makes use of meta-level knowledge in forming parallel executable rule instantiations. It is an example of using semantic level knowledge in multiple rule firing production systems similar to ours. However, the approach taken is completely different.

The most distinctive feature of PARULEL is that it is a two level system. Domain rules are for encoding domain knowledge while meta rules (or redaction rules), on the other hand, are used to select parallel executable rule instantiations. The way meta-rules are used in PARULEL is quite unique in the literature. Programs are executed through the match-redact-fire cycles until a fixpoint is reached. Meta-rules are used in the redact phase to eliminate incompatible rule instantiations such that all remaining instantiations can be fired in parallel. Application specific control knowledge is encoded in a similar way as domain knowledge, i.e. by way of using rules, which is both uniform and flexible. However, the responsibility of writing correct meta-rules to guarantee correctness is completely on the programmers. The run-time overhead of matching and executing meta-rules can be substantial.

4.3 Set-Oriented Extensions to Rule-Based Languages

While set-oriented constructs for production systems have been proposed years ago [16, 14], the proposed extensions are either not developed in detail or limited to procedural matching done on the right-hand-side only. It was not until recent years, when the interest on expert database systems growing rapidly, that the set-oriented or similar extensions to production systems became a focus of attention (e.g. [1, 17, 3]). However, the design goal of most of the database production languages or systems is to integrate databases and expert system shells. Performance and parallel processing are considered as secondary or even not considered at all. Here instead, we consider set-oriented constructs as a formalism for expressing semantic knowledge for increasing concurrency. We discuss Gordin and Pasik's work [3].

Gordin and Pasik have developed several set-oriented constructs and showed how these constructs can be added to a DBMS implementation of OPS5. The new constructs are implemented by an extended version of the Rete algorithm [2].

A condition element (CE) is set-oriented if it is enclosed in square brackets. The semantics is to match with all consistent WMEs to be associated with a single rule instantiation. If a rule contains only set-oriented CEs, the entire relation with tuples satisfying the CEs is generated with the instantiation when the rule is matched against the database. For a LIHS containing both set-oriented and regular CEs, the regular CEs can be considered as partitioning the relation into smaller relations, or equivalently, the set-oriented CEs can be seen as combining the tuples forming the regular instantiations into aggregated instantiations.

Similarly, a pattern variable (PV) is set-oriented if it occurs within a set-oriented CE. The domain of val-
ues of a set-oriented PV is the set of values occurring in the WMEs satisfying the corresponding CE. When a set-oriented PV occurs in more than one CE's, a join is performed. When a PV occurs in both a set-oriented CE and a regular CE, it is bound to the value in the WME matching the regular CE. A PV in a set-oriented CE can be forced to be non-set-oriented by listing it in the :scalar clause. The effect is to partition the relation induced by the LHS into separate instantiations. Aggregate operators such as count, min, max, sum, and avg can be applied on set-oriented PV.

For the RHS actions, aggregate operations such as set-remove and set-modify are added to operate on an entire set. A foreach iterator construct is provided to execute its body on each subset of the instantiation, having a distinct value for a specified set-oriented PV. The foreach operator can be nested to have compositional effect. The operator can also be applied on set-oriented CE's with the semantics of iterating through the matching WME's rather than values.

As a comparison, our approach differs in that we take an object-based approach rather than a relational database approach. Our design is to have a more expressive LHS and a SPMD semantics for RHS instead of the iterator approach which is contrary to the declarative nature of production systems. As an example of more expressive LHS, the semantics of DISJOINT construct in OPRL can not be expressed by their corresponding LHS set-oriented constructs. Most importantly, our major concern is concurrency which is not the design goal of Gordin and Pasik's work. No discussion was given about the interference analysis of rule and multiple rule firing.

5 General Object-Based Model for Parallel Rule-Based Systems

OPRL is a comprehensive language system we are developing for the fully exploitation of semantic level parallelism in production systems. A brief sketch of the OPRL language and system model is given in this section with examples.

A system consists of a set of rules and a set of objects. Passive working memory elements are replaced by concurrent objects which are units of data encapsulation and information hiding. Information encapsulated in an object can only be accessed through explicitly defined operation interfaces. Multiple operation requests to a single object can be serviced in parallel as long as the concurrent execution follows the behavior specification of the object. Concurrent objects are used because they provide the benefits of data organization, encapsulation, object identity, and modularity. More importantly, objects provide a good basis for specifying type-specific information for concurrent access. They are natural entities for the integration of rule-expressed and data-expressed parallelism. The behavior of an object is defined in its class definition which also serves as a template for a set of objects with similar structure and behavior. A class definition consists of an interface specification and a structure specification. The class interface defines the externally observable attributes, the syntax of operations, and the interface behavior of objects defined by the class. The class structure defines the internal structure of the objects and the implementation of operations in terms of operations of the component objects. A class structure may also contain an optional body to model active objects.

Depending on whether entities are active or not, we have identified four different execution models for production systems:

- passive objects passive rules (POPR) model,
- passive objects active rules (POAR) model,
- active objects passive rules (AOPR) model, and
- active objects active rules (AOAR) model.

The POPR model is the model used by traditional and most parallel production systems. Both rules and data objects are actually data to the inference engine(s). The actual realization of the parallelism is exclusively done in the inference engine(s).

In the POAR model, each rule is a process by itself that keeps trying to apply its actions whenever its enabling conditions is satisfied. Data objects are passively matched and manipulated by rule processes. Higher level of concurrency can be achieved in this model than the POPR model because active rules are naturally working in parallel. However, to avoid inconsistent updates to data objects, proper synchronization mechanism must be provided for the rule processes.

In contrast to the POAR model, data objects are active entities in the AOPR model while rules no longer correspond to processes. It is useful to think of rules as scripts with LHS being the qualifications of data objects that are eligible for participating the scripts and RHS being the actions to be performed by each qualified object. Data objects keep searching for the right script (i.e. rule) to participate in and a script is played (i.e. the RHS being executed) whenever all rules (i.e. condition elements) have qualified objects to serve. This model is attractive because it exhibits a
data flow style of parallelism. A rule is fired whenever
the data satisfying the conditions are ready.

To allow even more parallelism, we can have concurrent
objects together with active rules in the AOAR
model which is the selected execution model for the
OPRL language. In this model, both objects and rules
are processes communicating through message pass-
ing. Each rule process proceeds by first sending mes-
sages to related objects in order to satisfy the enabling
conditions. When the conditions are finally satisfied,
actions are executed by asking selected objects to per-
form the specified operations. When all operations are
serviced and returned, the rule continues by trying to
satisfy its enabling conditions again. On the other
hand, object processes keep receiving messages and
perform the specified operations. Multiple requests
can be serviced in parallel under the constraint that
the correct interface behavior of each individual ob-
ject must always be maintained. When servicing a re-
quest, an object can communicate with other objects
by sending messages.

In practice, the execution can be carried out ei-
ther synchronously or asynchronously. In synchronous
mode, a system is executed through cycles where each
cycle consists of three phases:

- **Condition evaluation**: The enabling conditions
  of all rules are evaluated in parallel. A rule is
  *enabled* if its enabling conditions are satisfied.

- **Selection**: All or a subset of the enabled rules
  are selected for parallel execution.

- **Action evaluation**: The actions of the selected
  rules are evaluated in parallel.

The cycle then repeats itself until no rule is enabled.

In asynchronous mode, rules are partitioned into
disjoint *clusters* such that rules of different clusters do
not interfere with each other. Then the rules in each
cluster are executed in synchronous mode while differ-
ent clusters are executed completely asynchronously.

For both synchronous and asynchronous parallel execu-
tions, the correctness criteria is no longer the seri-
alisability of multiple rule firing. Since the behaviors
of objects are precisely specified in the corresponding
classes, the correctness of parallel execution can be
verified in terms of the behavior specifications.

Examples of OPRL rules have been given in Sec-
tion 3. The concurrent behavior of an object is speci-
fied in its class with an event-based partial order spec-
ification formalism [12]. Because of the space limit, we
give only the interface specification of a concurrent
FIFO queue in Figure 11 to demonstrate the specifica-
tion style of OPRL.

```plaintext
class_interface Q(itentype) {
  operations
  Enq: itentype ->;
  Deq: -> itentype;

  behavior
  proper(Enq);
  proper(Deq);
  WIR1(Enq, Deq);
  \forall i, j, k, x, y
  \([Enq(x.i), i] \leq [Enq(y.j), j] \wedge
  \([Enq(y.j), j] \leq [Deq(y.j), k] \Rightarrow
  \exists l ([Deq(x.i), l] \leq [Deq(y.j), k])
}
```

Figure 11: The interface specification of a concurrent
FIFO queue.

The built-in predicates *proper* and *WIR1* are part
of a rich specification library that can be reused in any
class specification. An operation is *proper* if its invo-
cation always precedes its termination or return in the
partial order that represents the concurrent behavior
of the object on which the operation is defined. *WIR1*
is to characterize a pair of read/write like operations
with the semantics that a data entity can only be re-
trieved (read) after it has been stored (written) and
that any stored item can only be retrieved once. The
Enq/Deq operations here and the push/pop operations
of a stack are both good examples of such an access
pattern. The last assertion is the FIFO rule which says
that if the data item \(x.i\) is enqueued before data
item \(y.j\) then \(x.i\) must be dequeued before \(y.j\)
can be dequeued.

On the control level, OPRL allows explicit grouping
of rules into rule sets and provides a context formal-
ism for explicit specification of a context graph as the
control flow diagram. The system can then exploit the
parallelism induced by executing multiple contexts at
the same time, by overlapping the execution of adja-
cent contexts, and by pipeline parallelism between
contexts.

For a detailed account of our language and model
for parallel rule-based programming, see [18].

6 Conclusions and Future Work

We have made the observation that, contrary to
previous expectations, massive and scalable parallel-
elism does exist in production system programs. A
series of careful studies and analysis leads to the recognition that domain knowledge and explicit structuring are the key to the realization of such level of parallelism. We then identify the various sources of this level of parallelism which we called semantic level parallelism. A comprehensive language system OPRL supporting semantic level parallelism is proposed. Preliminary simulation results clearly exhibit the potential of our approach. Both the language and run-time system are under design and implementation. We plan to provide a library of behavior specification predicates, to design a set of reusable classes, to enhance the language with modular constructs and name scoping, possibly with specification inheritance, and to develop a methodology for the design of concurrent objects as well as the OPRL programs.

References


