Set-Oriented Constructs: From Rete Rule Bases to Database Systems

Douglas N. Gordin
AT&T Bell Laboratories

current affiliation:
IBM T. J. Watson Research Center
PO Box 704
Yorktown Heights, NY 10598

Alexander J. Pasik
Department of Computer Science
Columbia University

current affiliation:
Citicorp Technology Office
909 Third Avenue, 32 Floor
New York, NY 10022

Abstract

Set-oriented constructs for forward chaining rule-based systems are presented in this paper. These constructs allow arbitrary amounts of data to be matched and changed within the execution of a single rule. Second order tests on the data can be included in the match. The ability of a single rule to directly access all of the data to be manipulated eliminates the need for unwieldy control mechanisms and marking schemes. Adding this expressivity to rule-based languages enhances their value and capabilities as database programming languages since operations on entire relations can now be clearly specified, thus providing the database management system an opportunity to use its ability to update large amounts of data. Additionally, these set-oriented constructs can provide a basis for more efficient implementations of rule-based systems, for both the traditional memory-based systems and the emerging disk-based ones. The work described has been implemented using an extended version of the Rete network algorithm.

1. Introduction

Set-oriented constructs for forward chaining rule-based systems allow arbitrary amounts of data to be matched and changed within the execution of a single rule. Second order tests on the data can be included in the match. These enhancements provide a succinct method to perform commonly required operations while adding new opportunities for efficiency optimizations.

The ability of a single rule to access all of the relevant data eliminates the need for unwieldy control mechanisms and marking schemes. Similarly, the direct specification of second order tests eliminates the need to compute these values. Such abilities are standard within database systems, but were formerly not integrated with rule-based programming. Adding this expressivity to rule-based languages enhances their value to expert system developers and their capabilities as database programming languages.

It is hoped that these set-oriented constructs will provide a basis for more efficient implementations of rule-based systems, both for the traditional memory-based systems and the emerging disk-based ones. We demonstrate how set-oriented constructs can be added to a DBMS implementation of OPS5, thereby aiding in the solution of several unresolved problems with it. In addition, the direct expression of set operations allows compilers to optimize these operations. For example, a parallel architecture could perform an operation on the members of a set in parallel. Furthermore, research has shown that a limiting factor for parallelization of the Rete network is the number of operations done per rule firing [Gupta 1984, Miranker 1986, Pasik 1989]. The number of actions in a set-oriented rule should be substantially greater, providing the ability to increase parallelism.

All of the work described has been implemented within C5 [Vesonder 1988], a language that is an upwardly compatible superset of OPS5 [Forgy 1981][1]. The introduction of the set-oriented changes were made in a way that does not degrade the performance when executing regular OPS5 programs, and has been implemented using an extended version of the Rete algorithm presented herein [Forgy 1982]. This paper expands on the initial work presented in [Gordin and Pasik 1991].

2. Related Literature

The fields of rule-based systems, databases, and programming languages have been converging. This work relates both to rule-based programming languages and adding rules to databases.

2.1. OPS Languages

In research related to OPS rule languages, matching sets of working memory elements (WMEs) on the left-hand-side (LHS) has been proposed before to increase the amount of parallelism in production systems [van Biema *et al.* 1986]. However, this proposal was never developed in detail. Iterating over sets on the right-hand-side (RHS) is proposed as an addition to OPS5 in the YES/OPS work at IBM [Schor *et al.* 1986]. There work differs in that the matching is performed on the RHS rather than

[1]This paper assumes knowledge of OPS5 as described in [Forgy 1981].

2.2. Rule-based Systems as Database Management Systems

Some investigators are attempting to build rule-based systems that are disk based rather than memory based [Sellis *et al.* 1988, Sellis *et al.* 1989]. This system, DIPS, attempts to provide facilities for handling much larger amounts of data by matching on disk. However, this approach is undermined by the tuple-orientation of most rule-based systems. Greater optimization could be performed for set-oriented rules since they can cause large aggregate changes to the database. We propose a method of integrating our work into the DIPS system that helps resolve this problem.

Another proposal has been to use SQL queries for the LHS and having the RHS actions apply uniformly to the entire relation matched [Delcambre and Etheredge 1988]. This proposal, RPL, is similar to ours in that the LHS matches an entire relation rather than just one tuple of the relation. Moreover, their LHS expressivity is greater since SQL is relationally complete. However, our set-oriented constructs provide a natural extension of OPS5 that maintains programming style and can be implemented in an efficient manner. In addition, the RHS constructs presented below are more flexible that that of RPL. Other work extends SQL databases with rule-based functions [Widom and Finkelstein 1990, Stonebraker *et al.* 1990].

3. OPS5 as a Database Language

A number of researchers have chosen to describe OPS5 as a database programming language [Miranker 1986, Delcambre and Etheredge 1988, Tzvieli 1988]. This analogy is established by noticing:

- Working memory (WM) is a relational database with one important difference: Each WME has a time tag that uniquely identifies it.
- The LHS of productions are relational queries that perform selections and joins. Each comparison to a constant performs a selection. The appearances of a pattern variable (PV) in multiple condition elements (CEs) causes a join. Thus, each LHS generates a relation, possibly empty, of joined WMEs that satisfy the conditions.
- Each tuple in the relation is an instantiation.
- RHS actions update the database.

The concepts of rules, WM, and the conflict set of instantiations are illustrated in Figure 1. WM contains team members on two different teams. The rule **compete** generates all possible competitions between members of the two teams. Note that in a relational database system, the six instantiations of the **compete** rule would be contained in a single relation.

4. Adding Set-Oriented Constructs to the Left-Hand-Side

Set-oriented constructs add the ability to specify set-oriented variables and the ability to specify aggregate tests to be performed on those variables.

4.1. Set-Oriented Condition Elements and Pattern Variables

A set-oriented CE (designated by square brackets) directs the interpreter to include all of its consistent matches within the instantiation. In contrast, regular instantiations only contain a single matched WME for each CE. Hence, a LHS that only contains set-oriented CEs will only produce *one* instantiation (see Figure 2, **compete1**). This instantiation will contain the entire relation generated by the LHS. However, a LHS can contain both set-oriented and regular CEs. In this case, the regular CEs can be thought of as partitioning the relation induced by the LHS into smaller relations. Alternatively, the set-oriented CEs can be seen as combining the regular instantiations into aggregated instantiations (see Figure 2, **compete2**).

```
Attribute  Definitions
(literalize player name team)
Rule  Definition
(p compete
    (player ^name <n1> ^team A)
    (player ^name <n2> ^team B)
    -->
    (write Player A: <n1>, Player B: <n2>))
Working  Memory
1: (player ^team A ^name Jack)
2: (player ^team A ^name Janice)
3: (player ^team B ^name Sue)
4: (player ^team B ^name Jack)
5: (player ^team B ^name Sue)
6  Instantiations:
1: player A Jack   3: player B Sue

1: player A Jack   4: player B Jack

1: player A Jack   5: player B Sue

2: player A Janice 3: player B Sue

2: player A Janice 4: player B Jack

2: player A Janice 5: player B Sue
```

Figure 1: Rule, Working Memory, and Conflict Set

A PV is set-oriented if it occurs within a set-oriented CE. These PVs do not have a scalar binding as do regular PVs, rather their domains are specified by the set of values occurring in the WMEs satisfying their CEs (see Figure 2, **compete1**). When a set-oriented PV occurs in two set-oriented CEs, the domain is reduced to the consistent values of the domains (in database terms a join is performed).

When a PV occurs in both a set-oriented CE and a regular CE it is bound to a scalar value, namely the value occurring in the WME matching the regular CE. Additionally, it may be necessary to specify that a PV only occurring in set-oriented CEs be non-set-oriented. The **:scalar** clause lists PVs that should be non-set-oriented even though they only occur in set-oriented CEs. The effect of this is to partition by value the relation induced by the LHS into separate instantiations.

4.2. Aggregate Operators on the Left Hand Side

The addition of aggregate operators on the LHS eases programming by adding expressive power and allows for efficiency optimizations. If OPS5 program needs to act based on the cardinality of a set or an average of a value, it needs to cycle through all the members of that set calculating the second order value. With aggregate operators, this value can be directly accessed. Efficiency optimizations can be performed since the aggregate operation has been explicitly specified.

```
(p compete1
    [player ^name <n1> ^team A]
    [player ^name <n2> ^team B]
    -->
```

1 Instantiation:

1: player A Jack	3: player B Sue
1: player A Jack	4: player B Jack
1: player A Jack	5: player B Sue
2: player A Janice	3: player B Sue
2: player A Janice	4: player B Jack
2: player A Janice	5: player B Sue

```
(p compete2
    [player ^name <n1> ^team A]
    (player ^name <n2> ^team B)
    -->
```

3 Instantiations:

1: player A Jack	3: player B Sue
2: player A Janice	3: player B Sue

1: player A Jack	4: player B Jack
2: player A Janice	4: player B Jack

1: player A Jack	5: player B Sue
2: player A Janice	5: player B Sue

Figure 2: Set-Oriented LHSs and Instantiations

The aggregate operators on the LHS include the standard ones from SQL namely, *count, min, max, sum,* and *avg*. Currently, only the count operator has been implemented.

5. Changes to Rete

The approach taken to implement set-oriented pattern-matching was to extend Rete while making as few changes to the Rete network structure as possible. This is achieved by leaving the network untouched, except at the end of the network for each set-oriented rule. All of the advantages of Rete such as shared tests remain, even between set-oriented and non-set-oriented rules. The key insight is that set-oriented instantiations (SOIs) are made up of aggregations of regular instantiations.

The aggregation of candidate instantiations into SOIs occurs in a new type of node in the Rete network called an S-node. An S-nodes is placed after the last test node of a rule containing set clauses. Each S-node contains a γ-memory that maintains the candidate SOIs. The γ-memory also contains additional state information used to incrementally evaluate the test expression and determine whether a candidate SOI should flow into the conflict set.

The function of an S-node is given by its static data which reflects the rule, the contents of its γ-memory, and the algorithm the S-node implements.

The static, rule-derived data of an S-node is a five-tuple: (C, P, APVs, ACEs, T) where

- C is the non-set-oriented CEs of the rule,
- P is the set-oriented variables in the scalar clause,
- APVs are the set-oriented PVs on which aggregate operations are performed,
- ACEs are the set-oriented CEs on which aggregate operations are performed, and
- T is the test expression.

```
Find the SOI and place within it
if sign = + then
    for i in candidate SOIs
        if ∀x∈C i[x] = token[x]
            and ∀x∈P i[x] = token[x]
        then S := i; found := true
    if not found
    then    create new SOI called S with token;
            chg := new; S.status := inactive
    else    insert token at correct place
            (ordered like conflict set);
            if inserted at head
            then    chg := new-time
            else    chg := same-time
if sign = - then
    for i in candidate SOIs
            if ∀x∈C i[x] = token[x]
                and ∀x∈P i[x] = token[x]
            then S := i
    for t in S.Tokens
        if t = token remove t
    if S.Tokens is empty
    then    chg := delete
    else    if t was at head
            then    chg := new-time
            else    chg := same-time
Update the aggregates and reevaluate
if chg ≠ delete
then    update APVs and ACEs;
        if not eval(T) chg := fail
Decide flow of SOI
if chg = new
then    S.Status := active; send <S,+>
if chg = delete
then    send <S,->
if chg = fail and S.Status = active
then    S.Status := inactive; send <S,->
if chg = new-time
then    if S.Status = active
        then    send <S,time>
        else    S.Status = active; send <S,+>
```

Figure 3: The S-node algorithm

The γ-memory of an S-node is a list of triples, one for each candidate SOI. Each triple is (Tokens, Status, AV) where Tokens is the candidate SOI, Status is **active** or **inactive**, and AV is a list with one entry for each aggregate operation. Each aggregate operation is stored as the aggregate's current value followed by a list of (value, counter) pairs representing the values in the WMEs used in the computation of the aggregate operation.

The S-node algorithm is divided into three stages and is triggered by the arrival of a token.

- Find the relevant SOI and find the correct place within it.
- Update the aggregates and reevaluate the test expression.
- Decide whether to flow the candidate SOI to the P-node.

Variables in the algorithm are

- *sign* of the token which is + or - indicating whether the action is a make or remove, and
- *chg* which is set to fail, new, delete, new-time, or same-time (see Figure 3).

If a candidate SOI in the S-node succeeds and thus flows to the P-node, only a pointer is passed; updates to an active SOI in the S-node's γ-memory transparently updates the SOI in the conflict set.

P-nodes connected to S-nodes receive tokens with marks being +, -, or time. As with regular P-nodes, + tokens are added to the conflict set and - tokens are removed. Time tokens, however, represent SOI that are currently in the conflict set, but must be repositioned in the conflict set.

6. Adding Set-Oriented Constructs to the Right-Hand-Side

The division of a rule into LHS and RHS breaks up the specification of a relation from the actions to be performed on it. Formerly, this division was obscured by the inability of a rule to access the entire relation that its LHS defined. The addition of set-oriented constructs allows that relation or any part of it to be accessed on the RHS.

The interpreter's control strategy is also affected by the set-oriented constructs. Here, if any part of the instantiation changes, the instantiation is again eligible to fire (cf: [Widom and Finkelstein 1990]).

There are two types of capabilities that have been added to the RHS to access set-oriented PVs or CEs. The first is aggregate operations on an entire set such as *set-remove* and *set-modify*. The second is an iterator that executes its body on each subset of the instantiation, having a distinct value for a specified set-oriented variable.

The **foreach** iterator has a designated set-oriented variable and a block of statements. **Foreach** breaks up the instantiation's relation into subrelations (or *subinstantiations*) where each subinstantiation has only one value for the iterator variable. This is similar to performing an SQL *group-by* on that variable. The **foreach** iterator accesses the items in ascending, descending, or default order. By default, the values are considered in the order in which they would have occurred as separate instantiations in the conflict set.

The iterator's statements is executed for each subinstantiation. This process is described further by discussing separately the two cases of the iterator variable, namely, when it is a set-oriented PV or CE.

6.1. The Foreach Operator on Set-Oriented Pattern Variables

Using a set-oriented PV as an iterator partitions the instantiation by value. The iterator executes its block of statements once for each unique value in the PV's domain. During this execution, the instantiation is reduced to tuples where the PV's attribute is equal to the current value. The iterator variable can now be accessed as a regular PV bound to the current value. Note that the relations formed during the iteration also could have been created if the iterator variable had been specified as scalar in the LHS. However, the subinstantiations would have been different instantiations, rather than present during the execution of iterator. Hence, by matching on a set of values and iterating over them, subinstantiations are made accessible in a single rule firing that would otherwise have been formed into separate instantiations requiring multiple rule firings.

```
(p GroupByTeam
    [player ^team <t> ^name <n>]
    -->
    (foreach <t>
        (write <t>)
        (foreach <n>
            (write <n>))))
```

```
Instantiation:
1: player A Jack
2: player A Janice
3: player B Sue
4: player B Jack
5: player B Sue

First outer iteration, <t> = B,
subinstantiation constrained to:
3: player B Sue
4: player B Jack
5: player B Sue
   First inner iteration, <n> = Sue,
subinstantiation constrained to:
       3: player B Sue
       5: player B Sue
   Second inner iteration, <n> = Jack,
subinstantiation constrained to:
       4: player B Jack
Second outer iteration, <t> = A, etc.
```

Figure 4: A Set-Oriented Rule and its Iterations over PV Bindings

When there are nested **foreach** operators, the effect is compositional: Each iterator acts to reduce the size of the subinstantiation further by performing a selection as described above. In other words, when there are nested **foreach** operators, the innermost block is executed once for each

combination of the distinct values of iterator variables, with the subinstantiation reduced to the result of a selection specifying the iterator variables' attributes equal to their current values. This is illustrated in Figure 4, rule **GroupByTeam**, that only uses set-oriented PVs[2]. This rule has a single instantiation. It uses nested **foreach** iterators to display all players on each team. However, since PVs are value-based, the two WMEs with value **Sue** are considered as part of the same subinstantiation. Hence, **Sue** will just be printed once for team B. Note how the current value of the team PV (<t>) constrains the domain of the name PV (<n>) in each iteration.

6.2. The Foreach Operator on Set-Oriented CEs

The operation of **foreach** on set-oriented CEs is similar to that described above for PVs but with two differences. First, the distinct values of the iterator variable now refer to the WMEs themselves. A convenient way to think of this is to imagine iterating over distinct time-tags, since there is a one-to-one mapping between time-tags and WMEs. During the execution of the **foreach** block, the iterator variable is a regular CE variable bound to a single WME. The second difference arises directly from this fact. Since, the iterator variable is bound to a specific WME, all of the set-oriented PVs that were referred to in that CE can only have a single value in their domain. Therefore, during the execution of the **foreach** block on a set-oriented CE, all the PVs referenced within that CE are treated as regular PVs.

Above, an analogy between **foreach** operators and the **:scalar** clause was established for set-oriented PVs. A similar one holds for the set-oriented CEs. The subinstantiations formed by performing a **foreach** on a set-oriented CE are the same as the instantiations that would have been formed if that set-oriented CE had been designated a regular CE. However, those subinstantiations would have been separate instantiations.

7. Expressive Power of Set-Oriented Rules

The above description of the set-oriented rules has described their basic capabilities. However, it is only when seeing the succinctness with which frequently required operations can be expressed that their real utility is revealed. Set-oriented constructs enhance rule-based systems by allowing for the concise expression of processing unknown quantities of data, processing based on second order information, and hierarchical decomposition of data structures.

7.1. Iterating Over Collections of WMEs

Unknown amounts of data stored in working memory are often processed through unbounded iteration in OPS5 programs [Cooper and Wogrin 1988]. When the iteration modifies the WMEs, state must be maintained to assure that the same WMEs are not modified repeatedly (*e.g.* by marking the WMEs as they are processed). Additional complexity is introduced by the requirement of several rules and the state they maintain. Set-oriented rules allow the entire collection of WMEs to be accessed within the execution of a single instantiation. The first set-oriented rule in Figure 5, **SwitchTeams**, updates collections of WMEs. The **set-modify** operation is used to express the conceptual unity of the operation of switching the members of the two teams. Without the set-oriented constructs, multiple instantiations are required and extra state must be maintained to record previously processed WMEs.

Counting WMEs can be accomplished by iteration. This cardinality can then be stored in another WME and used in subsequent LHS matching. However, the value is not automatically updated when the size of the collection changes. These difficulties are resolved by providing the ability to directly match second order information such as cardinality, as shown in Figure 5, **SwitchTeams**.

```
Modify a set of elements
(p SwitchTeams
    { [player ^team A] <ATeam> }
    { [player ^team B] <BTeam> }
    :test ((count <ATeam>) == (count <BTeam>))
    -->
    (set-modify <ATeam> ^team B)
    (set-modify <BTeam> ^team A))

Team A Players Grouped with their Team B
Competitors
(p GroupByA
    [player ^name <n1> ^team A]
    [player ^name <n2> ^team B]
    -->
    (foreach <n1>
        (write <n1>)
        (foreach <n2>
            (write <n2>))))

Remove Duplicate Players
(p RemoveDups
    { [player ^name <n> ^team <t>] <P> }
    :scalar (<n> <t>)
    :test ((count <P>) > 1)
    -->
    (bind <First> true)
    (foreach <P> descending
        (if (<First> == true)
            (bind <First> false)
        else
            (remove <P>))))

Alternative Remove Duplicate Players
(p AlternativeRemoveDups
    { [player ^name <n> ^team <t>] <P> }
    -->
    (foreach <n>
        (foreach <t>
            (bind <First> true)
            (foreach <P> descending
                (if (<First> == true)
                    (bind <First> false)
                else
                    (remove <P>))))))
```

Figure 5: Powerful Set-Oriented Rules

[2]Below each rule, a specific instantiation is identified as the one that will be considered. Then for every iteration of the **foreach** operators, the instantiation is given constrained according to how the iterator will decompose it.

When there is a hierarchical information structure to be processed, several rules are needed and extra state must be maintained as the structure is traversed[3]. Set-oriented constructs allow of the WMEs to be matched in one instantiation and then hierarchically decomposed via the foreach iterator. **GroupByA** in Figure 5 prints out each member of Team A along with all of the members with whom they will have to compete. Certain hierarchical structures require transitive closure functionality in order to match the relevant WMEs. The specification of this functionality has not yet been investigated.

7.2. Removing Duplicate Working Memory Elements

The third rule in Figure 5, **RemoveDups** can only be accomplished in regular OPS with great difficulty: reducing working memory to a set[4]. This rule finds instances of multiple team players with the same name and same team and then deletes all but the most recent one. Notice that there will be one instantiation of this rule for each player-team pair occurring in multiple WMEs. An alternative formulation, **AlternateRemoveDups**, simply matches all player-team pairs and iterates over the values to remove redundant elements. However, this rule cannot discern whether any duplicates exist, thus its instantiation can fire unnecessarily.

8. Using Set-Oriented Constructs in Database Systems

There have been numerous proposals to add rules to database systems. Rules can be used to maintain consistency and views and to perform inference. The designers of rule-based languages have also been eager to merge with databases thereby obtaining concurrency control and persistence as found in database systems. Several systems have proposed using OPS5 as a model from which to create a database system with a flexible rules facility [Sellis *et al.* 1988]. These proposals suffer from the tuple-orientation of OPS5 which inhibits the database from exercising its strengths in performing many concurrent operations on large amounts of data. Set-oriented constructs can provide a notation to flexibly specify the rule-based operations that database systems efficiently perform.

8.1. The DIPS System

The DIPS system [Sellis *et al.* 1989] implements OPS5 rules using the facilities of a relational database system. The partial matches stored in Rete β memories are stored in COND tables. There is a COND table for each class of WMEs that initially contains CEs with matching classes. The table has attributes for:

- the rule identifier
- the ordinal number of the CE (e.g. 3rd)
- the attributes referenced in the CE (an attribute for each)
- a subrelation (RCE) storing the classes of the other CEs and their ordinal numbers
- mark bits for each CE in the rule to indicate whether it has been matched.

When a WME is created it is compared against the tuples in the table for its class. Each tuple that successfully matches causes the tables referenced in its RCE attribute to be updated. The RCE attribute tells what other CEs are constrained by this WME and can now be more fully instantiated.

New copies of these referenced tuples are created that replace shared variables with the constants found in the inserted WME and with a mark bit set to indicate the CE that has been matched. When all tuples in a rule have all their marks set an instantiation exists for that rule. DIPS attempts to execute all satisfied instantiations concurrently, relying on transaction semantics to block inconsistent updates to the working memory. For a fuller account the reader is refered to [Sellis *et al.* 1989].

Unfortunately, this scheme suffers because:

- Instantiations frequently conflict. A special case of this is where multiple instantiations of a single rule invalidate each other (e.g. try to remove the same WME). [Raschid *et al.* 1988]
- Executing instantiations in parallel provides no mechanism for hierarchical access to the data, accessing the data in order, or performing aggregate operations on it.

These problems are directly addressed by set-oriented constructs. Moreover, the addition of set-oriented constructs to DIPS can be easily accomplished.

8.2. Adding Set-Oriented Constructs to DIPS

When adding set-oriented constructs to DIPS we need to change the COND tables. Rather than storing a bit for each relevant CE that has been satisfied, we store a WME identifier that uniquely identifies the WME. This gives the ability to have multi-sets in WM as OPS5 does. Processing this attribute is analogous to the mark attribute, except that when a tuple is inserted into a COND table an identifier is stored rather than a bit being set.

An example of using COND tables to match set-oriented rules is given in Figure 6. This example is a simplification of one in [Sellis *et al.* 1989] The rule identifier is not shown since it is identical for both COND tables. This example shows the tables after they have been updated for the WMEs shown. The integers to the left of the WMEs are their identifiers (or time-tags). Also shown is an SQL query which suffices to select from the COND tables the appropriate SOIs.

---

[3]SOAR uses multiple WMEs to describe a single data structure [Laird *et al.* 1986]. Using OPS5 rules, it is not possible to access a complete SOAR data structure since the number of WMEs it contains is not fixed.

[4]In order to fully appreciate the set-oriented solution, the reader is encouraged to attempt to express this task in regular OPS5.

```
(p rule-1
    (E ^name <x> ^salary <s>)
    [W ^name <x> ^job clerk]
    -->
    ...
```

COND-E

CEN	name	salary	RCE	WME-TAGS
1	<x>	<s>	(W,2)	nil
1	Mike	<s>	(W,2)	1
1	Mike	<s>	(W,2)	3

COND-W

CEN	name	job	RCE	WME-TAGS
2	<x>	clerk	(E,1)	nil
2	Mike	clerk	(E,1)	2
2	Mike	clerk	(E,1)	4

```
WM
1: (W ^name Mike ^job clerk)
2: (E ^name Mike ^salary 20000)
3: (W ^name Mike ^job clerk)
4: (E ^name Mike ^salary 25000)
Query to retrieve SOIs
select COND-E.WME-TAG, COND-W.WME-TAG
    from COND-E, COND-W
    where COND-E.RULE-ID = COND-W.RULE-ID
    and COND-E.WME-TAGs is not NULL
    and COND-W.WME-TAGs is not NULL
    group-by COND-E.WME-TAGS
Relation containing SOIs
```

	COND-E.WME-TAGS	COND-W.WME-TAGS
Group 1	1	2
	1	4
Group 2	3	2
	3	4

Figure 6: Set-Oriented DIPS

All matching instantiations of a set-oriented rule are initially selected. These are then formed into groups based on the WME identifiers of the non-set-oriented CEs and the set-oriented PVs specified in the scalar clause. These two types of scalar values, condition elements and scalar pattern variables, partition the relation into the set-oriented instantiations. In addition, the test expression is evaluated for each SOI. In the example rule **rule-1** the first condition element of class E is non-set-oriented so it is used to partition the relation into two sub-instantiations as shown. The RHS constructs can be similarly implemented as selects on the set-oriented instantiation using the iterator variable in a group-by condition.

For rules with more than two condition elements the attribute storing the WME identifiers contains a list of values. Initially, the list is of NULL values; later as relevant WMEs are created, it is filled in with their identifiers. This list can be normalized by storing it in a separate relation and using an association value to join with it. The DIPS project took this approach in normalizing the RCE attribute which also contains a nested relation.

9. Summary

Set-oriented constructs bring rule-based languages closer to database systems by moving them from tuple-based processing to set-based processing, allowing many commonly performed tasks to be concisely and efficiently specified. This increases the likelihood of OPS-like languages being used as the rule language for relational database systems; such languages can now be set-oriented while still retaining data driven control.

References

1. Cooper T.A. and Wogrin N. (1988) *Rule-based Programming with OPS5*. San Mateo, California: Morgan Kaufman Publishers Inc.

2. Delcambre L.M.L. and Etheredge J.N. (1988) The Relational Production Language: A Production Language for Relational Databases. In Kerschberg L. (ed.), *Expert Database Systems. Proceeding from the Second International Workshop*. Benjamin/Cummings.

3. Forgy C.L. (1981) OPS5 User's Manual. Technical Report, Department of Computer Science, Carnegie-Mellon University.

4. Forgy C.L. (1982) Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence 19(1)*: 17-37.

5. Gordin D.N. and Pasik A.J. (1991) Set-Oriented Constructs for Rule-Based Systems. *Proc. Seventh Conference on Artificial Intelligence Applications*, Miami Beach, FL.

6. Gupta A. (1984) Parallelism in Production Systems: The Sources and Expected Speed-up. Technical Report, Department of Computer Science, Carnegie-Mellon University.

7. Laird J., Rosenbloom P., and Newell A. (1986) *Universal Subgoaling and Chunking*. Boston, Massachusetts: Kluwer Academic Publishers.

8. Miranker D.P. (1986) *TREAT: A New and Efficient Match Algorithm for AI Production Systems*. Ph.D. Thesis, Department of Computer Science, Columbia University.

9. Pasik A.J. (1989) *A Methodology for Programming Production Systems and its Implications on Parallelism*. Ph.D. Thesis, Department of Computer Science, Columbia University.

10. Raschid L., Sellis T., and Lin C. (1988) Exploiting Concurrency in a DBMS Implementation for Production Systems. *Proc. of the Intern. Symp. on Databases in Parallel and Distributed Systems*, Austin, TX.

11. Schor M.I., Daly T.P., Lee H.S., and Tibbitts B.R. (1986) Advances in Rete Pattern Matching. *AAAI-86*, pages 226-232.

12. Sellis T., Lin C., and Raschid L. (1988) Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms. *ACM-SIGMOD International Conference on the Management of Data*, pages 404-412.

13. Sellis T., Lin C., and Raschid, L. (1989) Data Intensive Production Systems: The DIPS Approach. *SIGMOD Record,* Vol 18, Number 3, Nov. 1989, ACM Press.

14. Stonebraker M., Jhingran A., Goh J., and Potamianos S. (1990) On Rules, Procedures, Caching, and Views in Data Base Systems. *ACM-SIGMOD International Conference on the Management of Data*, pages 281-290.

15. Tzvieli A. (1988) On the Coupling of a Production System Shell and a DBMS. *Third International Conference on Data and Knowledge Bases.*

16. van Biema M., Miranker D.P., and Stolfo S.J. (1986) The Do-loop Considered Harmful in Production System Programming. *First International Conference on Expert Database Systems*, pages 88-97.

17. Vesonder G. (1988) Rule-based Programming in the Unix System. *AT&T Technical Journal 67(1)*: 69-80.

18. Widom J. and Finkelstein S.J. (1990) Set-oriented Production Rules in Relational Database Systems. *ACM-SIGMOD International Conference on the Management of Data*, pages 259-270.