A Concurrent Rule Scheduling Algorithm for Active Rules*

Ying Jin
California State University, Sacramento
Department of Computer Science
Sacramento, CA 95819, USA
jiny@ecs.csus.edu

Susan D. Urban and Suzanne W. Dietrich
Arizona State University
Ira A. Fulton School of Engineering
Department of Computer Science and Engineering
Tempe, AZ 85287, USA
s.urban@asu.edu, dietrich@asu.edu

Abstract

The use of rules in a distributed environment creates new challenges for the development of active rule execution models. In particular, since a single event can trigger multiple rules that execute over distributed sources of data, it is important to make use of concurrent rule execution whenever possible. This paper presents the details of the Integration Rule Scheduling (IRS) algorithm. Integration rules are active database rules that are used for component integration in a distributed environment. The IRS algorithm identifies rule conflicts for multiple rules triggered by the same event by analyzing the read and write sets of each rule. A unique aspect of the algorithm is that the conflict analysis includes the effects of nested rule execution that occurs as a result of using an execution model with an immediate coupling mode. The algorithm therefore identifies conflicts that may occur as a result of the concurrent execution of different rule triggering sequences. The rules are then formed into a priority graph, defining the order in which rules triggered by the same event should be processed. Rules with the same priority can be executed concurrently. The IRS algorithm guarantees confluence in the final state of the rule execution. The IRS algorithm is applicable for rule scheduling in both distributed and centralized rule execution environments.

Author Keywords: Active rules, concurrent rule execution, rule scheduling algorithm, confluence analysis

1. Introduction

Active database systems extend traditional databases by supporting mechanisms to automatically monitor and react to events that are taking place either inside or outside of the database system [1, 2]. Active rules form the core of any active database system. An active rule, also known as an Event-Condition-Action (ECA) rule, typically consists of three parts: an event, a condition, and an action. An event describes an

* This research was supported by NSF Grant No. IIS-9978217.

occurrence that causes a rule to be triggered. The condition is a query over data sources that is checked when the rule is triggered, declaring the set of circumstances that must exist for the action of the rule to be processed. The action is executed in response to condition evaluation and can be used to modify data, retrieve data, or perform application procedures. Active rules were originally designed in the context of centralized database environments and have been used for integrity constraint maintenance and view maintenance as well as general monitoring and notification of specific database states and activities.

In addition to the design of architectures for the execution of active rules [1], past research on active rules has focused on the design of rule execution models [1, 3, 4, 5, 6, 7]. A rule execution model determines how a set of rules behaves at runtime. Additional research on active rules has investigated properties of active rule behavior, such as *termination* [8, 9, 10, 11, 12] and *confluence* [8, 9, 10, 11, 13, 14, 15]. The termination property guarantees that a set of rules will not result in infinite, cyclic rule execution [8]. The confluence property guarantees that the final result of rule execution does not depend on the order in which rules are chosen for execution [8].

Active rules currently exist in commercial database systems in the form of triggers [16]. More recently, active rules have proven useful for controlling activities in centralized and distributed workflow systems [17, 18, 19, 20, 21] and for supporting event-based, application integration in distributed environments [22, 23, 24, 25, 26]. Our own research has investigated the use of active rules, known as Integration Rules (IRules), for developing a declarative event-based approach to component integration [27, 28, 29, 30]. Through the support of the distributed rule execution engine that is provided by the IRules environment [31, 32], an application can automatically respond to events from remote components by testing conditions over distributed components and invoking global transactions that execute over distributed sources. The integration rule processor developed as part of the IRules project has been designed for the integration of components with well-defined interfaces based on the Enterprise Java Beans component model [33].

The use of rules in a distributed environment such as that of the IRules project creates new challenges for the development of active rule execution models. In particular, since a single event can

trigger multiple rules that execute over distributed sources of data, it is important to make use of concurrent rule execution whenever possible. Our work has developed the Integration Rule Processing (IRP) algorithm and the Integration Rule Scheduling (IRS) algorithm. The IRP algorithm is based on an algorithm originally presented in [15], using *execution cycles* and *levels within cycles* to control the nested execution of integration rules in a distributed environment. The results of the IRP algorithm are reported in [32, 34]. The IRS algorithm enhances the IRP algorithm with an approach for scheduling the sequential and concurrent execution of multiple rules triggered by the same event. An important aspect of the IRS algorithm is that it guarantees confluence for concurrent rule execution.

This paper presents the details of the IRS algorithm. The algorithm analyzes the read and write sets of each rule in the set of rules triggered by an event to identify rule conflicts. Non-conflicting rules can be executed in parallel. But for any two non-conflicting rules, r_1 and r_2, in a rule set, if r_2 triggers r_3, r_1 can potentially conflict with r_3 if r_1 and r_2 are allowed to execute concurrently. As a result, the algorithm makes use of the triggering graph to include the cascaded rules triggered by this initial rule set in the analysis process, assigning priorities to the analyzed rules. The prioritized rules are formed into a priority graph, defining the order in which rules triggered by the same event should be processed. Rules with the same priority can be executed concurrently. Furthermore, the IRS algorithm guarantees confluence in the final state of the rule execution. Confluence for the concurrent execution of rules has not been addressed in past research on confluence analysis. An added benefit of the IRS algorithm is that it is applicable for rule scheduling in both distributed and centralized rule execution environments.

The rest of this paper is organized as follows. Section 2 presents existing research on confluence analysis and compares the IRS algorithm with existing research. Section 3 presents an overview of the IRS algorithm along with assumptions and terminology. Section 4 describes the data access algorithm for conflict analysis of a rule set and the cascaded rules associated with each rule in the rule set. Section 5 elaborates on the priority graph construction algorithm for adding priorities to a rule set based on the conflict analysis described in Section 4. Section 6 illustrates how to analyze priority graphs during rule execution to schedule the sequential vs. concurrent execution of rules. Section 7 proves the correctness of

the IRS algorithm, illustrating how the algorithm guarantees confluence for rules that are allowed to execute concurrently. The paper concludes in Section 8 with a summary of the contribution of the IRS algorithm and a discussion of future research.

2. Related Work

In [8, 9, 11, 13], confluence analysis is based on the definition of *commutativity* of rule pairs. Two rules r_i and r_j *commute* if, starting from the same initial state, the final state of rule execution is the same regardless of which rule is executed first. Six conditions are defined to determine the commutativity of two rules r_i and r_j. If r_i and r_j do not satisfy any of the six conditions, then r_i and r_j are guaranteed to commute. The six conditions are: 1) r_i triggers r_j, 2) r_i untriggers r_j, 3) r_i activates r_j, 4) r_i deactivates r_j, 5) reverse i and j in conditions 1-4, 6) r_i's action and r_j's action do not commute. According to the definition in [8], a rule is untriggered if it is triggered at some point during rule processing, but when it is chosen for execution, the rule is no longer triggered because all triggering conditions were undone by other rules. Rule r_i activates rule r_j if execution of r_i's action causes new data to satisfy r_j's condition. Rule r_i deactivates rule r_j if r_i's action deletes all data that satisfies r_j's condition. The research in [11] produces a less conservative solution. The confluence analysis is based on a propagation algorithm that uses extended relational algebra to determine the affect of one rule over another. The algorithm determines how a query in a rule condition can be affected by the execution of a data modification operation in the action of another rule.

Whereas the work in [8, 9, 11, 13] addresses the theoretical question of how to determine if a set of rules is confluent, the algorithm presented in this paper is a constructive solution, using the theoretical results of past work to provide a practical implementation algorithm for organizing the multiple rules triggered by one event into an execution plan that guarantees confluence. The work in [9] was developed for Starburst but states that the theoretical concepts were never implemented. The IRS algorithm is different since it addresses confluence for sequential *and* concurrent execution of rules. Past research has

only addressed confluence of a rule set where rules are executed sequentially in different orders. The IRS algorithm shows that concurrent rule execution must demonstrate confluence not only for all rule pairs, but also for the cascaded rule sets of each rule pair.

The research in [10] is similar in nature to the research in [8]. Their work defines *independence*, which is similar to the definition of commutativity. Two rules r_1 and r_2 are independent if the sequential execution of r_1 after r_2 produces the same database state as the execution of r_2 after r_1. Voort has proven that static detection of independence is decidable. Voort has presented the confluence analysis for set and instance-oriented rule execution. According to [10], all qualifying objects are processed at the same time in set-based semantics, while one qualifying object is processed at a time in instance-based semantics. The rule model in [10] is restrictive. The algorithm assumes that the action of a rule cannot modify the data accessed by the condition, which restricts the use of this algorithm.

A different approach for confluence analysis is presented in [14]. Active rules are modeled into *conditional term rewrite rules*. The database state is encoded as a set of object terms. The update requests from users and event occurrence are represented as message terms. By mapping active rules to conditional term rewrite rules, results on confluence of *conditional term rewrite systems* are used to analyze the confluence problem of active database rules. The implementation of the algorithm in [14] is complex for even small rule applications, while the IRS algorithm presented in this paper provides simplicity for implementation.

The research in [15] uses *execution graphs* to analyze confluence. Execution graphs assume the use of confluence tables provided by the work of [12, 14]. An execution graph presents conflict, priority, and triggering relationship between rules. The application designer explicitly specifies the priority relationships among rules. An execution graph allows the designer to visually trace the execution by following the dynamic evolution of the execution graph as nodes enter and leave the graph. Within the execution environment of [15], the application designer make use of execution graphs to interactively control the flow of execution by dynamically manipulating relationships among execution rules. The IRS algorithm assigns priorities based on the analysis of read/write conflicts. Whereas the work in [15] was

designed for use in the interactive resolution of rule confluence, the IRS algorithm was designed for automatic resolution of confluence.

3. Overview of the Integration Rule Scheduling Algorithm (IRS)

Rule scheduling determines the order in which rules will be executed when multiple rules are triggered at the same time [2]. The objective of IRS is to schedule rules to achieve confluence for sequential and concurrent rule execution. IRS is specifically designed for the scheduling of rules with an immediate coupling mode. IRS follows the logic of depth-first (nested) rule execution: if one rule is scheduled to be executed before another rule, then all immediate rules triggered by the first rule will be executed before the execution of the second rule.

The IRS algorithm consists of three sub-algorithms: 1) the *data access* algorithm, 2) the *priority graph construction* algorithm, and 3) the *priority graph analysis graph* algorithm. After the compilation of rules, the data access algorithm executes first. Based on the results of the data access algorithm, the priority graph construction algorithm generates a priority graph. At execution time, the rule manager determines the execution order of rules by following the structure of the priority graph using the analysis graph algorithm. Sections 4, 5, and 6 detail the description of the three sub-algorithms. Before discussing the details of the algorithm, this section first presents motivational examples as well as the definitions and assumptions of IRS.

3.1 Motivational Example

To design the IRS algorithm, we consider the data access property of active rules *and* rule triggering relationships. If two rules try to read the same data, the execution results are the same no matter which rule will read the data first. In contrast, if two rules try to write the same data, the execution results may be different depending on which rule is executed first. For example, suppose the action of rule r triggers two rules, r_x and r_y. Suppose r_x writes database value v to 50, while r_y writes v to 0. During the

concurrent execution of r_x and r_y, if r_x is executed first, the final value of *v* will be 0. If r_y is executed first, the final value of *v* will be 50. So there may be a different final state of the database if rules try to write the same data. Recall that a rule set is confluent if the final result of rule execution does not depend on the order in which a rule is chosen for execution [1]. Therefore, r_x and r_y are not confluent and we should not schedule r_x and r_y to execute concurrently.

In addition to the analysis of data access properties, we need to consider rule triggering properties to design a scheduling algorithm that also supports concurrent rule execution. For example, suppose the action of r triggers r_x and r_y, and r_x triggers r_{x1}, as shown in Figure 1. Suppose r_x and r_y are confluent. Naively, we can schedule r_x and r_y to be executed concurrently. During the execution, however, r_{x1} is triggered by r_x. At that time, if r_y is still executing, then r_{x1} and r_y execute concurrently. Suppose r_{x1} and r_y write the same data. The final execution result is non-deterministic since the result will depend on the order in which r_{x1} and r_y modify the data. Therefore, it is not correct to schedule r_x and r_y to be executed concurrently. To schedule the concurrent execution of two rules, it is necessary to consider triggering relationship among rules.

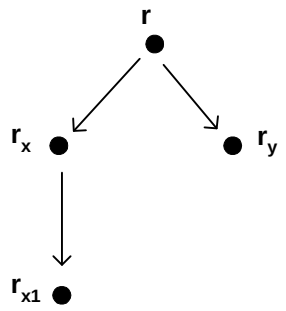


Fig. 1 Example of Rule Triggering

3.2 Definitions and Assumptions

The definition of graph properties of this paper follows the definition in Balakrishnan [35]. A graph G= (V, ED) consists of a finite set *V* of *vertices* and a finite set *ED* of *edges*, where edge *ed* in *ED* is associated with a pair of vertices *v* and *w* in *V*. A *directed graph* or *diagraph* is a structure G= (V, ARC),

where *V* is a finite set of *vertices* and *ARC* is finite set of *arcs* such that each arc *arc* in *ARC* is associated with an ordered pair of vertices *v* and *w*. We use *arc=(v,w)* to represent that *arc* is an arc from *v* to *w*. In a *diagraph*, the *outdegree* of a vertex is the number of arcs incident from the vertex. The *indegree* of a vertex is the number of arcs incident to the vertex. A *path* between two vertices v_1 and v_n is a finite sequence of vertices and edges of the form $v_1, arc_1, v_2, arc_2, \ldots, v_i, arc_i, \ldots, arc_{n-1}, v_n$, where arc_i is an edge between v_i and v_{i+1}. A path between a vertex and itself is a *closed path*. A closed path in which all the edges are distinct and all the vertices are distinct is a *cycle*. A graph with no cycles is an *acyclic graph*.

Definition 1 (Triggering relationship)

A rule r_i triggers a rule r_j, denoted as $r_i \rightarrow r_j$, if and only if r_i's action raises an event e such that e triggers r_j. In this case, r_i and r_j are said to have a triggering relationship.

Since the action of a rule r_i can raise an event that triggers another rule r_j, we use Definition 1 to define the triggering relationship between r_i and r_j, denoted as $r_i \rightarrow r_j$.

Definition 2 (Triggering graph):

Define a *triggering graph* as a diagraph TG (R, E), where R is a finite set of rules and E is finite set of arcs. Each arc e_i in E is associated with an ordered pair of vertices r_i and r_j in R. An arc e_i from r_i to r_j represents the fact that $r_i \rightarrow r_j$.

Definition 2 defines a triggering graph to represent the triggering relationships among all rules in an application, where vertices represent rules and arcs represent triggering relationships. Figure 1 presents an example of a triggering graph, in which rule r triggers r_x and r_y. The IRS algorithm assumes that the triggering graph is an acyclic graph and thus does not exhibit the potential for infinite rule triggering behavior. Research related to removing cycles in a cyclic triggering graph can be found in [3, 12, 13, 36, 37].

Given rule set R, define the *Cascaded Rule Triggering Set* with respect to an event, as shown in Definition 3. When an event triggers a rule, the triggered rule can trigger other rules, thus forming a cascaded set of rules triggered by this event.

Definition 3 (Cascaded Rule Triggering Set):

Recursively define rule set CR(e) = {r| r$\in$R and ((e triggers r) OR ($r_i \rightarrow$ r, where $r_i \in$CR(e)))} as the cascaded rule triggering set of event e.

Given an event e and its Cascaded Rule Triggering Set CR(e), the IRS algorithm described in the following sections schedules rules to guarantee the confluence property of rules in CR(e). The correctness of IRS will be proven in Section 7. The scheduling process assumes that information about the read and write set of a rule can be obtained from the metadata of the environment. The set of data read or written by a rule is the union of the data read or written by the condition and the action of a rule. In the case of the IRules environment, this information is generated from the rule and transaction language compilers and stored in the metadata repository [27, 29, 32]. Rule conditions specify a read set that can be extracted from compilation of the rule condition. Rule actions potentially specify a read set and a write set. In the IRules environment that provided the basis for this research, the rule action triggers one procedure, which is either an application transaction or a method associated with a distributed component. An event can be raised after the execution of each procedure. An application transaction can invoke multiple methods on distributed components, where each method can contain multiple data access and modification statements. The read and write sets of methods can be obtained in one of two ways. In an environment where the method code is accessible, the code can be analyzed to extract the read/write sets. In a black-box environment where the code is not accessible, each distributed component must view the rule processing unit as a trusted component and declare the read/write sets associated with each method.

4. Data Access Algorithm

In the data access algorithm, the data set read or written by a rule includes the data accessed by the rule and its triggered rules. The following definitions and formulae formally present the foundation of the data-access algorithm.

Definition 4 (Initial Read Set):

Define the initial read set RS_i of a rule r_i as a set that contains the data read by r_i regardless of its triggering relationships.

Definition 5 (Initial Write Set):

Define the initial write set WS_i of a rule r_i as a set that contains the data written by r_i regardless of its triggering relationships.

Definitions 4 and 5 define the initial read and write set. For example, if rule r_1 triggers r_2 and r_2 triggers r_3, then r_1 and r_2 have a triggering relationship, and r_2 and r_3 have a triggering relationship. The initial read set RS_2 of r_2 only consists of the data read by r_2. The initial write set WS_2 of r_2 only consists of the data written by r_2. The data accessed by r_1 and r_3 and not by r_2 are not included in RS_2.

Definition 6 (Refined Read Set):

Define the refined read set RD_i of a rule r_i as a set that contains the data read by r_i and the data read by the rules in $CR(e_i)$, where e_i is the event raised by the action of r_i.

Definition 7 (Refined Write Set):

Define the refined write set WD_i of a rule r_i as a set that contains the data written by r_i and the data written by the rules in $CR(e_i)$, where e_i is the event raised by the action of r_i.

Definitions 6 and 7 define the refined read and write sets of a rule. For example, if r_1 triggers r_2, then WD_1 includes the data written by r_1 and the data written by r_2. Formulae 1 and 2 present how to generate the refined read and write sets, where R is the domain of integration rules.

Formula 1 (Refined Read / Write Set Without Triggering Relationships):

$\forall r_1 \in R$, given RS_1 as the initial read set of r_1 and WS_1 as the initial write set of r_1,

if r_1 triggers no rules, then $RD_1 = RS_1$ and $WD_1 = WS_1$, where RD_1 is the refined read set of r_1 and WD_1 is the refined write set of r_1.

Formula 2 (Refined Read / Write Set With Triggering Relationships):

$\forall r_1 \in R, r_2 \in R$, then RD_1 and WD_1 are the refined read and write sets of r_1, and RD_2 and WD_2 are the refined read and write sets of r_2 without considering the triggering relationship between r_1 and r_2. If r_2 triggers r_1, then update the refined read and write set of r_2 as $RD_2' = RD_1 \cup RD_2$ and $WD_2' = WD_1 \cup WD_2$.

Based on Formulae 1 and 2, the data access algorithm specifies how to generate a refined read set and a refined write set. The data access algorithm consists of three steps, which are described in Figure 2. The input to this algorithm is a newly compiled rule. The output of this algorithm is the updated refined read /write sets of rules in the system. If there is a set of rules to be compiled, the rules are input to the algorithm one by one.

When a rule r_x is compiled by the system compiler, the compilation generates a new triggering graph including r_x. It is possible to get all the rules triggered by r_x when querying the rule metadata. The compilation can also generate the initial read and write set RS_x and WS_x of r_x. The initial read and write sets are stored in the metadata.

In step 1, the data access algorithm does not consider any triggering relationships of r_x. So RD_x is equal to RS_x, while WD_x is equal to WS_x.

In step 2, the data access algorithm queries the metadata manager to get all the rules triggered by r_x. For any rule r_j of this set of rules, update RD_x by taking the union of RD_x and RD_j as the new value of RD_x, as well as taking union of WD_x and WD_j as the new value of WD_x, by applying Formula 2. In step 3, the data access algorithm queries the metadata manager to get all of the rules that trigger r_x. For any rule r_i of this set of rules that trigger r_x, update RD_i by taking the union of RD_i and RD_x as the new value of RD_i, as well as taking union of WD_i and WD_x as the new value of WD_i. After we update r_i, we also need to update the refined read and write set of any rule that triggers r_i. This recursive process is captured in the recursive call to updateTriggeringRules() in Step 3. When a refined read or write set of a rule is generated, it is stored in the metadata. So in the execution of this algorithm, initial read and write sets and refined read and write sets of rules can be retrieved dynamically.

Public void dataAccessAlgorithm(Rule r_x)

```
{
        /* Step 1 */
        RDx = RSx;
        WDx = WSx;

        /* Step 2 */
        updateTriggeredRules(rx);

        /*Step 3 */
        updateTriggeringRules(rx);
}

//Step 2:
public void updateTriggeredRules(Rule rx)
{
        T = {rj | rx triggers rj };
        For each rj ∈ T
        {
                RDx =RDx U RDj;
                WDx =WDx U WDj;
        }
}

//Step3:
public void updateTriggeringRules(Rule rx)
{
        T = {ri| ri triggers rx};
        For each ri ∈ T
        {
                RDi =RDi U RDx;
                WDi =WDi U WDx;
                UpdateTriggeringRules(ri);
        }
}
```

Fig. 2. Data Access Algorithm

Figures 3 to 5 present an example that uses these three steps to generate the refined read set of a rule. After rule r_1 is compiled, RS_1 contains element *a* to represent that r_1 reads data *a*. As shown in Figure 3, in the first step, by applying Formula 1, we get $RD_1=RS_1 = \{a\}$.

The second step is in Figure 4. As shown in Figure 4a, two rules r_2 and r_3 are triggered by r_1. First, we need to union RD_1 and RD_2, as shown in Figure 4b, resulting in $RD_1 = \{a,b,c\}$. Second, we union RD_1 and RD_3, resulting in $RD_1 = \{a,b,c\}$, as shown in Figure 4c. The third step to illustrate the data access algorithm is in Figure 5. As shown in Figure 5a, r_4 and r_5 trigger r_1. First, we union RD_1 and RD_4 to update RD_4 to $\{a,b,c\}$, as shown in Figure 5b. Second, we union RD_1 and RD_5 to update RD_5 to $\{a,b,c,g\}$, as shown in Figure 5c. There are no other rules that trigger r_4 and r_5, so the algorithm ends.

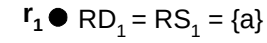
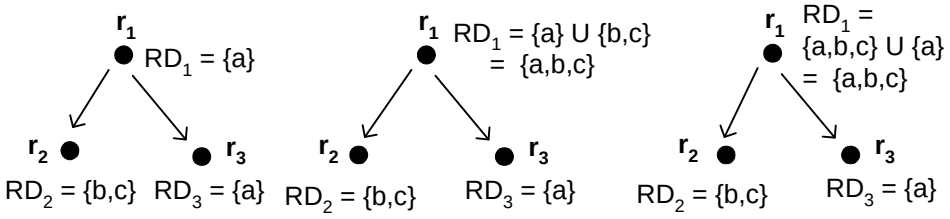
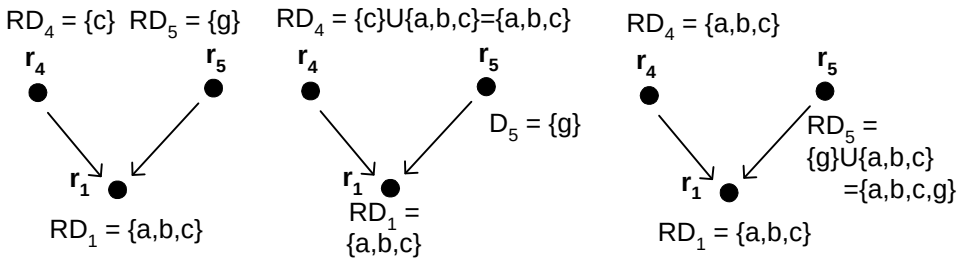


Fig. 3. Step 1 of Data Access Algorithm Example



a. Initial State b. Union RD_1 and RD_2 c. Union RD_1 and RD_3

Fig. 4. Step 2 of Data Access Algorithm Example



a. Initial State b. Union RD_4 and RD_1 c. Union RD_5 and RD_1

Fig. 5. Step 3 of Data Access Algorithm Example

The above example of these three steps for the *data access algorithm* illustrates that a rule can get its refined read set by following step 1 and step 2 in Figure 2. In addition, any rule that triggers this rule should update its read and write sets by following step 3 in Figure 2.

5. Priority Graph

Data access sets are used to form a priority graph for rule execution. A priority graph is used to resolve conflicting relationships between rules. Before constructing a priority graph, it is important to formally define rule conflicts.

Definition 8 (Rule Conflicts):

$\forall r_i \in R$, $r_j \in R$, assume RD_i and WD_i are the refined read and write set of r_i, and RD_j and WD_j are the refined read and write set of r_j. Then r_i and r_j conflict if and only if $RD_i \cap WD_j \neq \Phi$ or $WD_i \cap RD_j \neq \Phi$ or $WD_i \cap WD_j \neq \Phi$.

Definition 8 states that two rules conflict if and only if the intersection of the read-write, write-read, or write-write sets of these two rules is not empty. In IRS, any rules that conflict should be assigned different priorities, while non-conflicting rules can be assigned the same priority. Rules with the same priority can be executed concurrently, while rules with different priorities should be executed sequentially. The IRS algorithm uses a *Priority Graph* to assign priorities to rules.

Definition 9 (Priority Graph):

Define a *Priority Graph* as an acyclic directed graph PG = (V, A), where V is the set of vertices that represent rules triggered by the same event and A is the priority arc set. There is a priority arc $a(r_i, r_j)$ from r_i to r_j if and only if r_i and r_j conflict and r_i has higher priority than r_j.

In the IRS algorithm, conflicting rules are assigned priorities according to the timestamp associated with the compilation time of each rule. Rules compiled earlier have higher priorities than a newly compiled rule. Assume that no rule is compiled with any other rule at exactly the same time. As we know, compilation of rules results in the population of rule metadata. When retrieving a set of rules from the database, we therefore know whether a rule is compiled (and populated into the metadata) earlier than another rule. When we retrieve rules triggered by an event from the rule metadata, we get an array of rules, where the rule with a lower index is compiled earlier than a rule with a higher index. For any pair of conflicting rules in a priority graph, if r_i is compiled earlier than r_j, then r_i has higher priority than r_j. In the priority graph, an arc from r_i to r_j represents that r_i has higher priority than r_j.

Figure 6 presents the algorithm for generating a priority graph. The input is an event e. The algorithm generates a priority graph for all rules triggered by the given event e. The output is the priority graph of those triggered rules with respect to the event. Following the index order of the array, the refined read and write set of each rule is compared with that of another rule that has a higher index value

in the array. If the two rules are conflicting, there is a priority arc from the rule with a lower index to the rule with a higher index.

```
//Generate a priority graph for all the rules triggered by the given event.
Public PriorityGraph GeneratePriorityGraph (Event e)
{
        PriorityGraph pg = new PriorityGraph();
        Array a= e.getAllTriggeredRules(); //get all the rules triggered by this event
        pg.addVertices(a);
        n=a.length;
        For i = 0 to n-2
        {
            ri = a[i];
            For j = i+1 to n-1
            {
                rj= a[j];
                If (RDi ∩ WDj ≠ Φ or WDi ∩ RDj ≠ Φ or WDi ∩ WDj ≠ Φ)
                Then
                {
                        Arc c =priorityArc(ri,rj);
                        pg.addArc(c);
                }
                j=j+1;
            }
            i=i+1;
        }
        return pg;
}
```

Fig. 6. Generating Priority Graph Algorithm

An example of a priority graph is shown in Figure 7. An event triggers three rules r_1, r_2, and r_3. r_1 has a higher priority than r_2 and r_3. r_2 and r_3 are non-conflicting rules. Dashed-line arrows represent priority arcs in the priority graph.

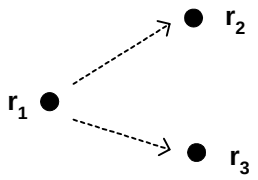


Fig. 7. Example of Priority Graph

```
Public void dataAccessAlgoirthm(Rule rx)
```

```
{
        /* Step 1 */
        RDx = RSx;
        WDx = WSx;

        /* Step 2 */
        updateTriggeredRules(rx);

        /*Step 3 */
        updateTriggeringRules(rx);
}

//Step 2:
public void updateTriggeredRules(Rule rx)
{
        T = {rj | rx triggers rj };
        For each rj ∈ T
        {
                RDx =RDx U RDj;
                WDx =WDx U WDj;
        }
}

//Step3:
public void updateTriggeringRules(Rule rx)
{
        T = {ri|ri triggers rx};
        For each ri ∈ T
        {
                RDi =RDi U RDx;
                WDi =WDi U WDx;
                UpdateTriggeringRules(ri);

                //update the prority graph associated with this rule
                e = ri.getEvent();
                GeneratePriorityGraph(e);  //update the priority graph for e
        }
}
```

Fig. 8. Final Version of Data Access Algorithm

Up to this point, the IRS algorithm can construct priority graphs for rules. One event can trigger a set of rules. This set of rules has one priority graph. The graph not only defines what rules are triggered by this event, but also what priorities exist among those rules. A rule r_i with n incoming arcs, indegree(r_i) = n, in a priority graph presents that at least n rules have higher priority than r_i. So a rule with a zero indegree has the highest priority. For a rule r_j with m outgoing arcs, outdegree(r_j) = m represents that r_j has higher priority than at least m rules.

Recall that in the third step of the Data Access Algorithm in Figure 2, we revised the refined read and write set of existing rules because of its triggering relationship with the new rule. When a refined read or write set of a rule is updated, the priority graph associated with this rule should also be updated since a priority graph is generated based on the refined read and write set. Therefore the third step of the Data Access Algorithm needs to be revised to include the update of the priority graph. As shown in step three of Figure 8, the revised algorithm gets the event e that triggers the current updating rule r_i. Then the algorithm gets all the rules triggered by e and updates the priority graph of this array of rules.

6. Priority Graph Analysis

After a priority graph is generated for each event at rule compilation time, each priority graph is stored in the metadata. At execution time, the rule manager can retrieve the priority graph for a specific event from the metadata manager. The rule manager performs the *priority graph analysis* algorithm to execute rules in the priority graph. The *priority graph analysis* algorithm is shown in Figure 9. The algorithm defines that non-conflicting rules are executed concurrently, while conflicting rules are executed sequentially according to priorities. The input to the algorithm is a priority graph. The logic of the algorithm is: 1) the rule manager executes only those rules with zero *indegree* since those rules have highest priority; 2) two un-connected rules can be executed concurrently; and 3) after rules finish executing, the corresponding vertices and all the outgoing arcs are removed from the graph. The remove operation may cause the update of *indegree* values of other rules. The above steps are repeated until there are no vertices left in the graph.

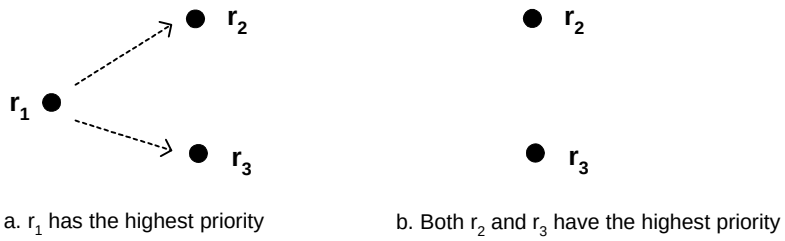
```
Public void analysisRulesAlgorithm(PriorityGraph pg)

{
        Set n= pg.getSetOfRules();
        /*n = set of rules in the priority graph.*/

        While (n ≠ ϕ)
        {
          Set doneSet = executeHighestPriorityRules(n);
          PriorityGraph pg = updatePriorityRuleGraph(doneSet, n, pg);
          n = pg.getSetOfRules();
```

```
        }
}

Public Set executeHighestPriorityRules (Set verticesSet)
{
        /*the rule manager selects rules with highest priority from verticesSet
          for execution, returns the set of rules that have been executed */

        //compute indegree of each vertices in the verticesSet
        for each vi in verticesSet
                compute indegree (vi);

        //construct a new set h that contains all the rules with the highest priority
        h = { vi | indegree(vi) = 0, vi ∈ verticesSet }

        //concurrently execute rules in h
        concurrentExecute(h);

        //return this set of rules that have been executed
        return h;
}

Public PriorityGraph updatePriorityRuleGraph (Set doneVerticesSet, Set verticesSet, PriorityGraph pg)
{
        /* doneVerticesSet contains rules that finished execution*/.

        // remove all related vertices and arcs from the priority graph
        for each vi in doneVerticesSet
        {
                for each vk in verticesSet
                        if arc(vi,vk) exists
                        then remove arc(vi,vk) from pg;
                remove vi from pg;
        }
        return pg;
}
```

Fig. 9. Analysis Graph Algorithm

Figure 10 illustrates how to execute the analysis graph algorithm. The initial set of rules N is shown in Figure 10a. As shown in Figure 10a, the rule manager gets the set of rules H with highest priority, where H= $\{r_1\}$. Then the rule manager calls concurrentExecute(H) to concurrently execute any rules in H. In this case, only r_1 is executed. Then the algorithm calls the updatePriorityRuleGraph method. The updatePriorityRuleGraph method removes r_1, arc(r_1, r_2), and arc(r_1, r_3) from N. As a result, the indegree values of r_2 and r_3 become zero. Now the new set of rules to be executed is N = $\{r_2, r_3\}$. Because both r_2

and r_3 have zero indegree, they are both selected into the highest priority set and executed concurrently. At this point, all the rules have been executed and the algorithm ends.



a. r_1 has the highest priority b. Both r_2 and r_3 have the highest priority

Fig. 10. Analysis Graph Algorithm Example

7. Correctness of the IRS Algorithm

Recall that we have defined the cascaded rule triggering set in Section 2. Given an event e and its cascaded rule triggering set CR(e), the IRS algorithm schedules rules to guarantee the confluence property of rules in CR(e). To prove the correctness of IRS, we need to prove that any two rules in CR(e) that are allowed to execute concurrently are confluent.

This section first presents a lemma. The lemma is then used to prove the correctness of the IRS algorithm. Lemma 1 presents the use of the refined read set and the refined write set to check the confluence of two rules. Recall that RD_j and WD_j are the refined read set and the refined write set of r_j, as defined in Definition 6 and Definition 7. In the proofs in the rest of this section, we use square brackets enclosed in Italic characters to illustrate the reason for a result.

Lemma 1 (Use refined read/write set to check confluence):

Given $r_i \in CR(e)$, $r_j \in CR(e)$, where CR(e) is the Cascaded Rule Triggered Set of a given event e. If $RD_i \cap WD_j = \phi$ AND $WD_i \cap RD_j = \phi$ AND $WD_i \cap WD_j = \phi$, then r_i and r_j are confluent.

Proof:

First consider the case when a rule (say r_j) does not read any data from or write any data to the database: $RD_j \cup WD_j = \phi$. Since $RD_i \cap WD_j = \phi$ and $WD_i \cap RD_j = \phi$ and $WD_i \cap WD_j = \phi$, then the order of execution of r_i and r_j does not effect the database state. So r_i and r_j are confluent.

Now consider the case when a rule can read or write data: $RD_i \cup WD_i \neq \phi$ and $RD_j \cup WD_j \neq \phi$.

Denote the execution of rule *r* changing value x from one state X_u to another state X_v as $X_u \xrightarrow{r} X_v$.

Suppose the initial database state of x is X_I.

Define set ALL to consist of all of the data in the database.

Define set OTHER such that OTHER = ALL – $((RD_i \cup WD_i) \cup (RD_j \cup WD_j))$

$\forall$ x $\in$ ALL, then x $\in RD_i \cup WD_i$ OR x $\in RD_j \cup WD_j$ OR x $\in$ OTHER.

Now let's prove that r_i and r_j are confluent in any of the following three cases: (1) x $\in RD_i \cup WD_i$, (2) x $\in RD_j \cup WD_j$, and (3) x $\in$ OTHER.

(1) In the case of x $\in RD_i \cup WD_i$

Let us check the database state of x with respect to the order of execution of r_i and r_j.

Since $RD_i \cap WD_j = \phi$ and $WD_i \cap WD_j = \phi$,

then $(RD_i \cup WD_i) \cap WD_j = \phi$.

Since x $\in RD_i \cup WD_i$,

then x $\notin WD_j$

Further, there are two cases for r_j: either r_j does not trigger any rule (case 1.1) or r_j triggers rules (case 1.2).

Case (1.1) r_j does not trigger any rule.

So $WD_j = WS_j$ *[by formula 1]*.

By x $\notin WD_j = WS_j$, we know r_j does not write x to the database.

Case (1.2) r_j triggers other rules.

Suppose r_j's action raises event e_{rj} and CR $(e_{rj}) = \{r_{j1}, r_{j2}, \ldots r_{jm}\}$

$\forall$ $r_{jk} \in CR(e_{rj})$

$\exists\ e_{rj} \rightarrow r_{j1} \rightarrow r_{j2} \ldots \rightarrow r_{j(k-1)} \rightarrow r_{jk}$, where $r_{j1}, r_{j2}, \ldots, r_{j(k-1)} \in CR(e_{rj})$, *[by Definition 3]*

then $WD_j \supseteq WD_{j1} \supseteq WD_{j2} \supseteq \ldots \supseteq WD_{j(k-1)} \supseteq WD_{jk}$ *[by Formulae 1 and 2]*

Since $WD_j \supseteq WD_{jk}$ and $x \notin WD_j$, then $x \notin WD_{jk}$.

So $\forall\ r_{jk} \in CR(e)$, r_{jk} does not write x to the database.

In other words, any rule cascaded triggered by r_j does not write x to the database.

By (1.1) and (1.2) we know r_j and all its cascaded triggered rules do not write x to the database.

Therefore the database state of x is the same before and after the execution of r_j.

Suppose $X_I \xrightarrow{ri} X_v$,

If r_i executes before r_j, then

$X_I \xrightarrow{ri} X_v \xrightarrow{rj} X_v$

If rj executes before ri, then

$X_I \xrightarrow{rj} X_I \xrightarrow{ri} X_v$

Since $X_v = X_v$,

then the database state of x is the same regardless of the execution order of r_i and r_j.

Therefore r_i and r_j are confluent in the case of $x \in RD_i \cup WD_i$.

(2) In the case of $x \in RD_j \cup WD_j$

Since i and j are commutative,

then exchange i and j, and this step of the proof is the same as the proof in (1).

Therefore r_i and r_j are confluent in the case of $x \in RD_j \cup WD_j$.

(3) In the case of $x \in$ OTHER

We abbreviate the proof here without repeating the same proof structure and results from Case (1).

Since OTHER = ALL – $((RD_i \cup WD_i) \cup (RD_j \cup WD_j))$,

then $x \notin WD_i$ AND $x \notin RD_i$ AND $x \notin WD_j$ AND $x \notin RD_j$.

Let us check the database state of x with respect to the order of execution of r_i and r_j.

Since $x \notin WD_i$,

Then the database state of x is the same before and after the execution of r_i.

Since $x \notin WD_j$,

then the database state of x is the same before and after the execution of r_j.

As a result, the database state of x is the same regardless of the execution order of r_i and r_j.

Therefore r_i and r_j are confluent in the case of $x \in$ OTHER.

By (1), (2), and (3), r_i and r_j are confluent.

This proves the Lemma.

Next, we can prove the correctness of IRS using Lemma 1. To prove the correctness of IRS, we need to prove that any two rules in CR(e) that are allowed to executed concurrently are confluent, given that an event e triggers rules. Notice that a rule in CR(e) can be triggered directly by e. It is also possible that a rule in CR(e) is not directly triggered by e, but is indirectly triggered by e according to the definition of cascaded rule triggering set. We use Roman numerals to mark intermediate results achieved during the proof. Other parts of the proof will reference these intermediate results.

To Prove: $\forall\ r_1 \in CR(e)$ and $r_2 \in CR(e)$, where CR(e) is the Cascaded Rule Triggering Set of an event e; if r_1 and r_2 are allowed to execute concurrently using IRS, then r_1 and r_2 are confluent.

Proof: There are three possible situations: (1) both r_1 and r_2 are directly triggered by e, (2) either r_1 or r_2 is directly triggered by e (but not both), (3) Neither r_1 nor r_2 is directly triggered by e.

(1) In the case that both r_1 and r_2 are directly triggered by e.

Since r_1 and r_2 are allowed to execute concurrently using IRS,

then r_1 and r_2 have the same priority. *[by Analysis Graph algorithm]*

Therefore r_1 and r_2 do not conflict in the priority graph of e, *[by Definition 9]*

Therefore NOT($RD_1 \cap WD_2 \neq \phi$ OR $WD_1 \cap RD_2 \neq \phi$ OR $WD_1 \cap WD_2 \neq \phi$). *[by Definition 8]*

Therefore $RD_1 \cap WD_2 = \phi$ AND $WD_1 \cap RD_2 = \phi$ AND $WD_1 \cap WD_2 = \phi$. *[by set theory]*

Therefore r_1 and r_2 are confluent. *[by Lemma 1]*

(2) In the case that either r_1 or r_2 is directly triggered by e (but not both).

Suppose e directly triggers r_1

Since $r_2 \in CR(e)$ and

$\exists\ r_x \in CR(e)$, $e \rightarrow r_x$, and $r_x \rightarrow r_i \rightarrow r_i+1 \rightarrow \ldots \rightarrow r_{i+n} \rightarrow r2$, where

$r_i, r_{i+1}, \ldots, r_{i+n} \in CR(e)$, *[by Definition 3]*

(i) then $RD_x \supseteq RD_i \supseteq RD_{i+1} \supseteq \ldots \supseteq RD_{i+n} \supseteq RD_2$ and

$WD_x \supseteq WD_i \supseteq WD_{i+1} \supseteq \ldots \supseteq WD_{i+n} \supseteq WD_2$ *[by Formulae 1 and 2]*

Since r_1 and r_2 are allowed to executed concurrently using IRS,

then r_1 and r_x are allowed to executed concurrently. *[by definition of depth-first execution]*

Therefore r_1 and r_x have the same priority. *[by Analysis Graph algorithm]*

Therefore r_1 and r_x do not conflict in the priority graph of e. *[by Definition 9]*

Therefore $NOT(RD_1 \cap WD_x \neq \phi$ OR $WD_1 \cap RD_x \neq \phi$ OR $WD_1 \cap WD_x \neq \phi)$. *[by Definition 8]*

Therefore $RD_1 \cap WD_x = \phi$ AND $WD_1 \cap RD_x = \phi$ AND $WD_1 \cap WD_x = \phi$. *[by set theory]*

Since $RD_2 \subseteq RD_x$ and $WD_2 \subseteq WD_x$, *[by previous result (i)]*

then $WD_1 \cap RD_2 = \phi$ and $RD_1 \cap WD_2 = \phi$ and $WD_1 \cap WD_2 = \phi$. *[by set theory]*

Therefore r_1 and r_2 confluent. *[by Lemma 1]*

(3) In the case that neither r_1 nor r_2 is directly triggered by e.

Since $r_1 \in CR(e)$,

then $\exists\ r_y \in CR(e)$, e triggers r_y, and $r_y \rightarrow r_j \rightarrow r_{j+1} \rightarrow \ldots \rightarrow r_{j+m} \rightarrow r_1$, where

$r_j, r_{j+1}, \ldots, r_{j+m} \in CR(e)$. *[by Definition 3]*

(ii) Therefore $RD_y \supseteq RD_j \supseteq RD_{j+1} \supseteq \ldots \supseteq RD_{j+m} \supseteq RD_1$ and

$WD_y \supseteq WD_j \supseteq WD_{j+1} \supseteq \ldots \supseteq WD_{j+n} \supseteq WD_1$. *[by Formulae 1 and 2]*

Since $r_2 \in CR(e)$ and

$\exists\ r_x \in CR(e)$, $e \rightarrow r_x$, and $r_x \rightarrow r_i \rightarrow r_{i+1} \rightarrow \ldots \rightarrow r_{i+n} \rightarrow r_2$, where

$r_i, r_{i+1}, \ldots, r_{i+n} \in CR(e)$, *[by Definition 3]*

(iii) then $RD_x \supseteq RD_i \supseteq RD_{i+1} \supseteq \ldots \supseteq RD_{i+n} \supseteq RD_2$ and

$WD_x \supseteq WD_i \supseteq WD_{i+1} \supseteq \ldots \supseteq WD_{i+n} \supseteq WD_2$. *[by Formulae 1 and 2]* Since r_1 and r_2 are allowed to execute concurrently using IRS,

then r_x and r_y are allowed to execute concurrently. *[by definition of depth-first execution]*

Therefore r_x and r_y have the same priority. *[by Analysis Graph algorithm]*

Therefore r_x and r_y do not conflict in the priority graph of e. *[by Definition 9]*

Therefore NOT($RD_y \cap WD_x \neq \phi$ OR $WD_y \cap RD_x \neq \phi$ OR $WD_y \cap WD_x \neq \phi$). *[by Definition 8]*

Therefore $RD_y \cap WD_x = \phi$ AND $WD_y \cap RD_x = \phi$ AND $WD_y \cap WD_x = \phi$. *[by set theory]*

Since $RD_1 \subseteq RD_y$, $WD_1 \subseteq WD_y$, $RD_2 \subseteq RD_x$, $WD_2 \subseteq WD_x$, *[by previous results (ii) and (iii)]*

then $WD_1 \cap RD_2 = \phi$ and $RD_1 \cap WD_2 = \phi$ and $WD_1 \cap WD_2 = \phi$. *[by set theory]*

Therefore r_1 and r_2 confluent. *[by Lemma 1]*

End of the Proof.

This section has presented the correctness of the IRS algorithm. First, we presented a lemma for proving the correctness of using the refined read/write set of a rule to determine the confluence of any two rules in CR(e) with an empty intersection between the read and write sets of the rules. Next, using the lemma, we proved that IRS is correct in that any two rules in CR(e) that are allowed to execute concurrently are confluent. The use of the refined read/write sets in the proof guarantees that cascaded rule triggering will not interfere with the concurrent execution of rules in CR(e).

8. Summary and Future Research

This paper has presented the IRS algorithm for scheduling the sequential and concurrent execution of active rules. The IRS algorithm consists of three sub-algorithms: the data access algorithm, the priority graph construction algorithm, and the priority analysis algorithm. For all of the rules triggered by an event, the three sub-algorithms identify conflicting rules by analyzing the read and write sets of each rule,

including the cascaded set of rules triggered by each rule. The IRS algorithm therefore identifies conflicts that may occur as a result of the concurrent execution of different rule triggering sequences. The IRS algorithm assigns relative priorities to conflicting rules and forces the sequential execution of conflicting rules according to the priorities assigned. Non-conflicting rules are executed concurrently. The IRS algorithm guarantees confluence for concurrent rule execution. The proof of correctness for the IRS algorithm has also been presented.

Although the IRS algorithm was originally developed for the execution of rules in the distributed execution environment of the IRules project [32], the algorithm can also be applied to rule execution in a centralized environment. In either case, the algorithm is a static, compile-time algorithm that requires interacting with the metadata of the environment for analysis of rule triggering sequences and the read/write sets of rules and transactions. For better performance in a distributed environment, it is desirable for the algorithm to execute in a single location with distributed access to the metadata. Furthermore, as a static algorithm, the IRS algorithm inherits the characteristics of compile-time analysis techniques and, as a result, may conservatively identify conflicts between rules that may actually be confluent at run-time. The proof in Section 7, however, guarantees that concurrently executing rules are always confluent.

The IRS algorithm has been tested within a prototype for rule scheduling. The algorithm does not yet address the issue of deleting rules from the system, but the deletion of rules is a straight-forward extension of the current algorithm. Future research is focused on integrating the algorithm into the IRules runtime environment, which is currently being revised for the execution of integration rules over Grid Services. Performance evaluation of the IRules environment has already been conducted for the sequential execution of rules [32]. Additional performance evaluation is needed for concurrent execution of rules scheduled using the IRS algorithm within the IRules runtime environment.

Reference

[1] J. Widom and S. Ceri, *Active Database System.* Morgan Kaufmann publishers, San Francisco, California. 1996.

[2] N. Paton and O. Diaz, Active Database Systems. *ACM Computing Surveys.* 31 1 (1999), pp. 3-27.

[3] J. Widom, The Starburst Rule System: Language Design, Implementation and Application. *IEEE Data Engineering Bulletin.* December 1992, pp. 15-18.

[4] H. Branding, A. P. buchmann, T. Kudrass, and J. Zimmermann, Rules in an Open System: The REACH Rule System. *Rules in Database Systems.* 1993, pp. 111-126.

[5] G. Kappel and W. Retschitzegger, The TriGS Active Object-Oriented Database System - An Overview. *SIGMOD Record.* 27 3 (1998), pp. 36-41.

[6] C. Collet, T. Coupaye, and T. Svense, NAOS: Efficient And Modular Reactive Capabilities in An Object-Oriented Database System, in: Proceeding of 20th International Conference on Very Large Data Bases, Santiageo-Chile, September 12-15, 1994, pp. 32-143.

[7] S. Gatziu and K. R. Dittrich, SAMOS: An Active Object-Oriented Database System. *Data Engineering Bulletin.* December 1992, pp. 23-26.

[8] A. Aiken and J. Widom, Behavior of Database Production Rules: Termination, Confluence, and Observalbe Determinism, in: Proceedings of ACM SIGMOD International Conference on Management of Data, San Diego, United State, 1992, pp. 59-68.

[9] A. Aiken, J. M. Hellerstein, and J. Widom, Static Analysis Techniques for Predicting the Behavior of Active Database Rules. *ACM Transactions on Database Systems*. 25 3 (1995),pp. 269-332.

[10] L. V. D. Voort and A. Siebes, Termination and Confluence of Rule Execution, in: Proceedings of the 2nd International Conference on Information and Knowledge Management, Washington D. C., United State, 1993, pp. 245-255.

[11] E. Baralis and J. Widom, An Algebraic Approach to Static Analysis of Active Database Rules. *ACM Transactions on Database Systems*, 25 3 (2000), pp. 269-332.

[12] A. P. Karadimce, Termination and Confluence Analysis in an Active Object-Oriented Databases. Ph.D. Dissertation, Arizona State University, Department of Computer Science and Engineering, 1997.

[13] E. Baralis and J. Widom, An Algebraic Approach to Rule Analysis in Expert Database System, in: Proceedings of 12th International Conference on Very Large Data Bases, Santiago, Chile, September, 1994, pp. 457-486.

[14] A. P. Karadimce and S. D. Urban, Conditional Term Rewriting as a Formal Basis for Analysis of Active Database Rules, in: Proceedings of the 4th Internal Workshop on Research Issues in Data Engineering, Houston, Texas, 1994, pp. 156-162.

[15] T. Ben Abdellatif, An Architecture for Active Database Systems Supporting Static and Dynamic Analysis of Active Rules Through Evolving Database States. Ph.D. Dissertation, Arizona State University, Department of Computer Science and Engineering, 1999.

[16] SQL Standard. American National Standards Institute, Information technology-Database languages. 1999.

[17] F. Casati, S. Ceri, B. Pernici, and G. Pozzi, Deriving Active Rules for Workflow Enactment, in: Proceedings of Database and Expert Systems Applications, Switzerland, 1996, pp. 94-115.

[18] K. Karlapalem and P. C. K. Hung, *Security Enforcement in Activity Management Systems*, Dogac, etc. Eds. Springer-Verlag Publisher. 1998.

[19] G. Kappel and W. Retschitzegger, The TriGS Active Object-Oriented Database System - An Overview. *SIGMOD Record.* 27 3 (1998), pp.36-41.

[20] S. Ceri, P. Grefen, and G. Sanchez, WIDE - A Distributed Architecture for Workflow Management, in: Proceedings of 7th International Workshop On Research Issues In Data Engineering, 1997, pp. 76-79.

[21] A. Dogac, E. Gokkoca, Design and Implementation of A Distributed Workflow Management System: METUFlow. *Workflow Management Systems and Interoperability,* Dogac, et al. Ed. Springer-Verlag Publisher, 1998, pp.61-91.

[22] S. Chakravarthy and R. Le, ECA Rule Support for Distributed Heterogeneous Environments. In: Proceedings of International Conference on Data Engineering, 1998, pp. 601.

[23] A. Koschel and R. Kramer, Configurable Event Triggered Services for CORBA-based Systems, in: Proceedings of 2nd International Enterprise Distributed Object Computing Workshop, San Diego, California, November 1998, pp. 306-318.

[24] A. Koschel and P. C. Lockemann, Distributed Events in Active Database Systems - Letting the Genie out of the Bottle. *Journal of Data and Knowledge Engineering*. 25 (1998), pp. 11-28.

[25] H. Fritschi, S. Gatziu, and R. Dittrich, FRAMBOISE – an Approach to Framework-Based Active Database Management System Construction, in: Proceedings of the 7th ACM International Conference on Information and Knowledge management, November, 1998, pp. 364-370.

[26] M. Cilia, C. Bornhovd, and A. buchmann, Moving Active Functionality from Centralized to Open Distributed Heterogeneous Environments, in: Proceedings of 9th International Conference on Cooperative Information Systems, Trento, Italy, September 2001, pp. 195-210.

[27] S. W. Dietrich, S. D. Urban, A. Sundermier, Y. Na, Y. Jin, and S. Kambhampati, A Language and Framework for Supporting an Active Approach to Component-Based Software Integration. *Informatica*. 25 4 (2001), pp. 443-454.

[28] S. D. Urban, S. W. Dietrich, Y. Na, Y. Jin, and A. Sundermier, The IRules Project: Using Active Rules for the Integration of Distributed Software Components, in: Proceedings of the 9th IFIP 2.6 Working Conference on Database Semantics: Semantic Issues in E-Commerce Systems, Hong Kong, April 2001, pp. 265-286.

[29] S. D. Urban, S. W. Dietrich, A. Sundermier, Y. Jin, S. Kambhampati, and Y. Na, Distributed Software Component Integration: A Framework for a Rule-Based Approach, in: Handbook of Electronic Commerce in Business and Society, Watson, R., Lowery, P. and Cherrington, J. Ed. 2002. pp. 395-421.

[30] S. D. Urban, S. Kambhampati, S. W. Dietrich, Y. Jin, and A. Sundermier, Event Processing for Rule-Based Integration: Database Technology Applied to Distributed Components, in: Proceedings of 6th International Conference on Enterprise Information Systems, Portugal, April 2004.

[31] Y. Jin, S. D. Urban, S. W. Dietrich, and A. Sundermier, An Execution and Transaction Model for Active, Rule-Based Component Integration Middleware, in: Proceedings of the Engineering and Deployment of Cooperative Systems, Beijing, China, September 2002, pp. 403-417.

[32] Y. Jin, An Architecture and Execution Environment for Component Integration Rules. Ph.D. Dissertation, Arizona State University, Department of Computer Science and Engineering, 2004

[33] EJB 2000, Enterprise Java Beans Specification, Sun Microsystems, version 2.0. http://www.javasoft.com/products/ejb/docs.html

[34] Y. Jin, S. D. Urban, S. W. Dietrich, and A. Sundermier, An Integration Rule Processing Algorithm and Execution Environment for Distributed Component Integration. Submitted for journal publication 2004.

[35] V. K. Balakrishnan, *Introductory Discrete Mathematics*. Prentice Hall Publisher, 1991

[36] S. D. Urban, M. K. Tschudi, S. W. Dietrich, and A. Karadimce, Active Rule Termination Analysis: An Implementation and Evaluation of the Refined Triggering Graph Method. *Journal of Intelligent Information Systems.*12 1 (1999), pp. 27-60.

[37] J. Bailey, L. Crnogorac, K. Ramamohanarao, and H. Sondergaard, Abstract Interpretation of Active Rules and Its Use In Termination Analysis, in: Proceedings of the International Conference on Database Theory, Delphoi, Greece, 1997, pp. 188-202.

[38] R. Peri, Compilation of Integration Rule Language. M.C.S. Report, Arizona State University, Department of Computer Science and Engineering, 2002