Architecture and Algorithms for Distributed Rule Management and Processing

***George Dimitoglou*† *and* *Shmuel Rotenstreich*††,**

†*Department of Computer Science, Hood College, Frederick, MD, USA*
††*Department of Computer Science, The George Washington University, Washington, DC, USA*

Summary
Many distributed computing environments are based on the paradigms of peer-to-peer networks and grid-based computing. These environments tend to be structurally dynamic but with volatile resource and service availability. Such environments have rules, constraints, and guidelines that govern how tasks are performed. The rules describe how resources, components and services should be allocated and resemble business rules in human organizations. Centralized rule processing in such dynamic environments suffers from scalability and reliability and issues.. We present the architecture and processing algorithms of a distributed rule management system. The system allows the management, distribution and execution of rules in distributed environments characterized by volatile recourse and service availability.

Key words:
Distributed computing, rule processing algorithms, rule management system, P2P, grid environments

1 Introduction

Distributed, networked environments have implicit or explicit rules and constraints governing how different computations are performed. These rules describe how environment resources and services "do business" so they resemble *business rules* in human organizations. Business rules are statements that specify and enforce logic for constraints, validations and other actions that constitute and enforce policies.

In computing environments policies can relate to various operating aspects such as quality of service (QoS), security and resource utilization. Grid computing and Peer-to-Peer (P2P) networks are typical environments with explicit and implicit rules. In grid-based computing specific rules are enforced with respect to the accessibility, availability and utilization of resources. In P2P-based environments, implicit but enforceable rules, such as the volume of a peer's file sharing determines the peer's status and role in the network. In both of these examples, using rules enables the consistent and systematic enforcement of constraints.

In this paper we present a general, distributed rule management system, applicable to any distributed environment. Existing rule systems offer centralized architectures, which don't scale well and introduce a single point of failure.

Throughout this work we assume the operating environments consisting of numerous resources, services, loosely coupled objects, groups and hierarchies engaging in different activities. The activities can vary from simple transactions such as resource discovery and file exchanges to complex computations such as resource negotiation and collaborative problem-solving.

The paper is organized in two general thematic parts. The first part describes the rules, their formulation and their processing by rule engines. The second part describes the algorithms and the distribution of the rule engines.

In the next sections we present the related work in the area of rule processing and describe the general problem. Then, we present the proposed rule management system, the rule formulations, types and the core mechanism: the rule engine. The rest of the paper contains the algorithms for rule processing and the distributed architecture along with techniques that mitigate inherent issues to distribution such as conflict resolution and fault tolerance. We conclude by summarizing the work and discussing future work.

2 Related Work

In the past, rules and their enforcement, have been investigated in the areas of artificial intelligence and expert systems [25] under the concept of "production rules" [2, 7, 30]. Grosof et al [12-15, 26] recently extended this work by taking advantage the declarative nature of business rules, expressing them using XML and introduced rule conflict-handling methodologies. Rules have also been examined in databases as "triggers" and "event-condition-action" (ECA) rules. Irrespective of their name, rules played an instrumental role in introducing reactive behavior in applications and enhancing automated validation techniques [9]. Their use has propelled the development of another genre of database systems known as active databases [6, 8, 10, 22]. Numerous prototypes (HiPAC [5, 8], POSTGRES [27, 28] Ariel [17], Starburst [31, 32]) were developed following the active database

Manuscript received August 5, 2008.
Manuscript revised August 20, 2008.

approach and many evolved to full-scale commercial systems. Today, trigger and production rule functionality has been extended from databases and expert systems to business processes, replacing the terms production rules and triggers with the term business rules. As a result, business rule management systems have been developed to "serve" regulations to business processes and activities. Many of these systems have successfully borrowed principles from databases (e.g. triggers), expert systems (e.g. inference algorithms) and workflow systems [3, 4, 23, 29]. Object-oriented (OO) technologies have extended these systems even further. Most new systems no longer use triggers or stored procedures, but store rules as programmable objects translated to executable code that executes when rules are applied [18, 21]. Several commercial OO-based business rule systems exist (e.g. ILOG [20], JRules, JESS, OPS/J, Blaze Advisor [18]).

3 Problem Description

Architecturally, centralized rule management systems often share a common, two-tier structure, comprised of a rule repository and a software tier providing application logic for rule management and enforcement. The majority of repositories are based on a database management system (DBMS) while other implementations have followed different software engineering practices (e.g. functional, procedural, and object-oriented). The ways of expressing rules have been equally diverse, leading to numerous proprietary, incompatible formats.

However, centralization is inflexible, has scalability limitations and introduces a single point of failure. Another deficiency is the inability of such systems to handle dynamic changes in the rules and the environment. Few systems allow rule changes during real-time operation [3, 4] and many require processing halts to perform rule updates.

4 The Rule Management System

To address these problems, we developed a distributed rule management system. In the following sections we present the rule types, distributed rule engines and the relevant algorithms for rule enforcement, task processing and conflict resolution.

4.1 Infrastructure and Environment

In a network environment services and resources (e.g. CPUs, permanent storage, memory) are widely available and accessible. To utilize them a necessary underlying infrastructure must exit to provide basic network services such as addressing, routing, messaging, resource discovery (i.e. functionality similar to the Domain Name System) and task management (i.e. tracking of tasks processed in the environment). To emulate the environment, we use *Middleware++*, a distributed computing platform [10] which provides this functionality.

In the environment, a *task* can be a computation, or a resource or a service request. All tasks are subject to rules. Sequencing and combining tasks can compose more complex activities. The proposed rule processing system is extensible enough to handle complex activities but for simplicity, we omit task composition and focus on rule enforcement to simple, atomic tasks.

4.2 Rule Types

Rule types can be classified in two categories depending on the effects of their enforcement. Rules that affect task parameters belong to either *Stimulus/Response* or *Computation Rules*. Similarly, rules that affect the task execution sequence belong to either *Structure* or *Operational Constraint Rules*.

We assume that $n \geq 1$ and $k \geq 1$ so that

$$P_1 \wedge P_2 \wedge ... \wedge P_n$$

is a conjunction of n predicates and $A_1 \wedge A_2 \wedge ... \wedge A_k$ is the set of conjunctions of k actions. Predicates (P_n) and results (A_k) express values of operands using relational operators from set S, where $S = \{>, <, \neq, =, \leq, \geq\}$.

Definition 1. *Stimulus/Response Rules* define what conditions must be met before an activity can legally take place: R_{SR}: *if* $P_1 \wedge P_2 \wedge ... \wedge P_n$ *then* $A_1 \wedge A_2 \wedge ... \wedge A_k$

If a *"WHEN"* condition is used, the predicate is temporal:

$$R_{SR'}: \textit{when } P_1 \wedge P_2 \wedge ... \wedge P_n \textit{ then } A_1 \wedge A_2 \wedge ... \wedge A_k$$

Definition 2. *Operation Constraint Rules* define constraints that must hold before and/or after an activity;

$$R_{OC}: \textit{if } P_1 \wedge P_2 \wedge ... \wedge P_n \textit{ then } [Q]\, A_1 \wedge A_2 \wedge ... \wedge A_k[S]$$

Where Q is a pre-condition that states the properties that must hold whenever the activity is to be performed and S is a post-condition that states the properties the activity guarantees will hold when it is completed. Isolating the right-hand side of the R_{OC}, we observe a Hoare triple in the form: $[Q]A_k[S]$, expressed as: "if Q is true before an activity A_k is executed, and the execution of A_k terminates, then S is true afterwards". It is implied that the triple does not assert that A_k will terminate.

Definition 3. *Structure Constraint Rules* specify constraints on tasks, which must never be violated. These rules are similar to *class* and *loop invariants* in software engineering. A *class invariant* refers to an assertion describing a property, which holds for all instances of a class. A *loop invariant* refers to an assertion that must be satisfied prior to the first execution of a loop, and preserved at each iteration, so that it will hold on loop termination. In the context of task and rule processing, a rule R expresses an invariance property for task T_i, if R is valid in every state("*must always hold*") of the computation during processing of task T_i. The rules can be described in the general form:

$$R_{SC}: \textit{it must always hold that } P_1 \wedge P_2 \wedge ... \wedge P_n$$

or, with a more the complex construct:

R_{SC}: *it must always hold that*

$$if \quad P_1 \wedge P_2 \wedge ... \wedge P_n \; then \; A_1 \wedge A_2 \wedge ... \wedge A_k$$

Definition 4. *Computational Rules* describe processing algorithms or equations. This is a widely used general case of *Stimulus/Response Rules,* where the predicates are TRUE and the activity is either an algorithm execution or a computation:

$$R_{CR}: \quad y=f(x)$$

4.3 Rule Elements

Each rule is expressed in an XML document, decomposing rule parts into document elements. The only required elements for expressing the rules are listed in Table 1. As there is no limit to the number of additional elements or types, the XML schema (XSD) can be extended or customized accordingly.

Table 1.Elements of an XML rule document

Element	Description
ruleID	Composite value, in the form: *<RuleEngineID>.<Rule Number>.*
rule_type	(*see* section 4.2*)*
rule_task_type	Category of tasks a rule is applicable to.
priority	Rule execution priority. Integer between 1-4 . Value of 1 indicates highest priority.
LHS, RHS	Identifies the conjunction conditions of the *n* predicates (P) and the conjunction of the *k* results (A) respectively.
status	Current rule state. See Fig. 1 for possible states.
creationdate, effect-date, expiry-date, lifetime	Timestamp-elements

Rules may be in any of the following states: formulated (*new*), activated (*active*), enforceable (*enforced*) or deactivated (*dormant*). Formulation occurs when a "higher power" (i.e. user) creates new rules.

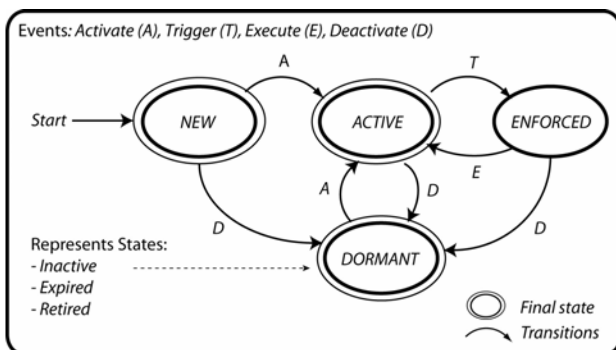

Fig. 1 Rule life cycle state transition diagram.

New rules can then be activated. An activated rule may become enforced, i.e. applied to tasks, or become deactivated (inactive, retired, expired) or *dormant*. A dormant rule can be re-activated at any time. The rule execution life-cycle can be represented (Fig. 1) as a state transition diagram with the four possible states: *new*, *active*, *inactive* and *enforced*. The first three states are final while the last is not. Rule execution is deterministic and can be expressed as a deterministic finite automaton, *R* by the 5-tuple: R= (Q, Σ, q_0, F, δ) where Q is a finite set of states, Σ is a finite set of input events *e* with Σ={*activate*, *trigger*, *execute*, *deactivate*} and q_0 (*new*) being the start state. F is the set of final states and a subset of Q with F={*new*, *active*, *dormant*}; δ is the transition function from Q x Σ to Q so that δ(q, *e*) is a state for each state q and input event *e*.

In human organizations, rules may have different degrees of enforcement ranging from strict to very relaxed or even ignored. Discretionary rule enforcement complicates rule management beyond the scope of this work. We assume all rule enforcement to be strict.

4.4 System Architecture

The basic elements of the system architecture are the rule processing engine and the rule repository.

4.4.1 Rule Engine

The Business Rule Engine (BRE) processes tasks and enforces rules, provides rule maintenance facilities and communicates with other rule engines. Collaboration between rule engines is necessary when different resources (e.g. CPUs) collaborate, each with their own rule engines and rules.

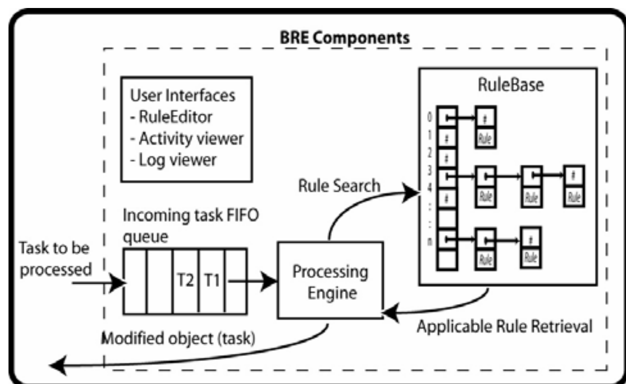

Figure 2: BRE Architecture and Processing Diagram

The core of the BRE is a processing engine that contains the logic, a rule repository for rule storage, a first-in-first-out (FIFO) queue to manage incoming tasks and a set of graphical user interfaces for rule management (editing) and real-time monitoring (Fig. 2).

4.4.2 Rule Repository

The rule repository stores the XML rules. We have experimented with both an external database management system and a data structure (hash table) for the implementation of the repository. In the former

implementation, a record stores a rule with each XML element occupying a separate record field. All rule engine functionality is facilitated via SQL commands. In the data structure approach, rules are stored in a hash table with each rule and the resulting hash structure implemented as a Java object.

During the development and further experimentation of the prototype, we favored the data structure implementation. It has offered us a lightweight and portable, repository without the overhead of a database and relieved the rule engine of the generation of SQL queries. Implementing rules as objects provided the ability to invoke methods that directly access rules and their parameters.

4.5 Rule Computation and Processing

With the rule engines and repositories in place, the environment can process task and enforce rules. In the next sections we describe the task processing algorithms and the conflict resolution algorithms to address resource sharing conflicts that are likely to appear in a distributed environment.

4.5.1 Task Processing

Each rule engine is instantiated object that accepts tasks in XML format and processes them in a FIFO order. When tasks arrive at a rule engine, the task type is identified and applicable rules are retrieved from the rule repository generating a rule *candidate set*.

Applicable rules, are all the rules with a matching *t*ask type. While each rule is designed to constrain or regulate a specific task, each rule also includes a *rule_task_type* element to specify the kind of tasks it is applicable to. Rules in the *candidate set* are examined for possible conflicts. Conflicts are resolved using a conflict resolution mechanism described in the next section. Rules are enforced via method invocation according to rule predicates (conditions) and modifying task parameters according to rule results (actions). Upon completion, the now rule-enforced task is released for execution.

A rule-processing algorithm (Fig. 3) is used to match, retrieve, resolve conflicts and apply business rules to incoming tasks. In method RetrieveRules the initial matching of the *task_type* element of an incoming task is performed against the contents of the rule repository. The resulting collection of all applicable rules constitutes the *candidate set*. The *candidate set* may contain rule conflicts, which have to be resolved before rule enforcement. Method CategorizeRules performs the first step of conflict resolution by identifying and separating the *Structure Constraint* rules from the *candidate* set, forming a subset of *candidate set* rules identified as "*scr*". The ResolveConflicts method receives both the *candidate* and *scr* sets. The method proceeds to apply a pre-defined conflict resolution technique, which in turn produces a conflict-free rule execution sequence. The last step (ApplyRules) of the algorithm applies (enforces) the rules on the task.

```
PROCESSTASK (in: task T;  out: task T')
 begin
   PARSETASK (in: task; out: task_type) {
     whileTask(T)  do
       store task_type info
     end while; }

   RETRIEVERULES (in: task_type; out: candidate set){
     for task_type do
        find rules in repository where rule_type=task_type
        store rules in candidate set;
     endfor;   }

   CATEGORIZERULES (in: candidate set; out: scr set, candidate set) {
     for all rules in candidate set do
       if rule_type=Structure Constraint then
         store in scr set;
         remove from candidate set;
     end for ;  }

   RESOLVECONFLICTS (in: ipnutset[scr, candidate]; out inputset'){
     for all rules in inputset do
       check for conflcits;
     end for;
     if conflicts exist then sort inputset by priority, timestamp;
     else sort inputset by priority; }

   APPLYRULES (in: ipnutset'[scr, candidate], Task T;  out: Task T') {
     for rules in inputset do
       enforceRule();
     end for; }
 end;  }  //ProcessTask
```

Fig. 3: Algorithm and methods for task processing and rule enforcement

During this processing sequence and after the creation of the candidate set all activities are treated as a single atomic operation to preserve the integrity of the operation and the state of the rule engine.

4.5.2 Dynamically Prioritized Conflict Resolution

Each rule engine contains conflict resolution logic capable of ensuring rule enforcement integrity. When the candidate set is generated, it may contain conflicting rules.

Conflict falls under two general categories. In the first category, conflicts affect the sequence of rule execution (Sequence Conflicts). Such conflicts could manifest between two applicable rules, with one accepting a task and the other rejecting it. In the second category, conflicts that result in non-deterministic task parameter modifications are classified as Parameter Conflicts. An example of a Parameter Conflict appears when two or more rules modify the same task parameter but compute its value using different formulas and produce different results.

A dynamically prioritized conflict resolution algorithm resolves conflicts before rule enforcement. The algorithm is dynamic because it is applied over a dynamically generated set of applicable rules for a specific task rather than being indiscriminately applied to all rules residing in a repository. The objective of the algorithm is

to create a rule enforcement sequence and remove any conflicting rules. These objectives are accomplished by using the following sequence of strategies:

(a) **Logical Precedence**. It works by placing rules that should logically be executed first at the top of the rule enforcement sequence. These rules prevail over any conflict with other rules and their role is to evaluate whether a task should be further processed or not. In some cases, by terminating task processing, they provide an additional crude resolution mechanism by preempting any further conflict resolution consideration.

(b) **Priority-based Rule Ordering.** It works by sorting the rules based on their priority. In addition, it eliminates duplicate rules and guarantees that any rule can only be applied once to a specific task, thus preventing repetitive enforcement and loops.

(c) **Temporal Ordering.** It is the last conflict resolution tactic and the least utilized. It resolves conflicts by promoting rules with the latest creation date. Although an arbitrary mechanism, it assumes that more recently defined rules reflect the most up-to-date snapshot of the environment's regulations.

During the execution of these conflict resolution strategies, additional conflicts may emerge. For instance, Logical Precedence may place two conflicting rules at the top of the enforcement sequence. When this occurs, the conflict resolution mechanism attempts to resolve these conflicts by repeatedly applying the entire conflict resolution strategy.

Due to their nature, Structure Constraint Rules (SCR) must be applied first. For instance, a SCR that mandates that "*task T_i must execute after task T_j*" indicates the specific rule must be enforced independently of any other rule type that may be modifying parameters of either task. Therefore, SCR rules are identified and removed from the candidate set, creating a temporary separate rule pool. If a Sequence Conflict exists with one of the conflicting rules not in this SCR pool, the non-SCR is automatically discarded as the SCR always prevails. If both rules are part of the SCR pool, the conflict is resolved by comparing their priority. If the priority criterion also fails to provide a resolution, the rule with the latest creation date value prevails. The rationale behind this arbitrary temporal ordering resolution assumes that older rules may be "stale" and less reflective of the latest set of constraints.

Next, the SCR pool and the rules in the candidate set are examined for Parameter Conflicts. If two rules affect one or more common parameters when enforced, the rules are conflicting. Again, the same conflict resolution sequence as before is used, first applying Priority-Based Rule Ordering and if this criterion also fails, proceed with the most recent creation date.

The same conflict resolution sequence takes place for any of the remaining conflicting rules in the candidate set. When all conflicts are resolved, the rules are sorted in both the SCR pool and the candidate set by descending Priority order. The combination of these two rule-sets represents the enforcement sequence. The conflict resolution mechanism then releases control and the rules are enforced.

5 Distributed Rule Engines

Thus far, the architecture of a standalone rule engine has been described. In distributed environments there may be numerous rule engines, each serving a resource or service hierarchy, or an ad hoc collection of components (e.g. a cluster of CPUs, an array of storage devices). Therefore, applicable rules for a task may be scattered in multiple repositories and collaboration among rule engines becomes necessary.

There is no restriction on the number or location of rule engines. Engines scattered throughout an environment may contain complementary rule sets. With multiple rule engines, issuing a task is followed by a search for all rules applicable to that task. The search starts from a single rule engine and continues by searching peer engines for additional applicable rules. Engine selection is based on either proximity or a Least Utilization Factor (LUF) algorithm. Proximity is determined by topology and relationship. By topology, two nodes residing within the same network partition (i.e. same subnet) are considered "closer" than nodes residing in two different subnets. By relationship, two objects that are members of the same resource or service group unit are "closer" than non-member objects.

The LUF algorithm requires the polling of multiple engines and exchange of messages to retrieve the FIFO queue size from each engine and determine the smallest task-processing load. Unfortunately, the LUF incurs communication costs and may provide unreliable measurements as it retrieves rule engine load information for a specific point in time, which may quickly change and become inaccurate. However, the LUF algorithm is useful and supports the notion of distributing the processing load among various rule engines. When the Proximity technique is used, a congested rule engine may become even more congested simply because it is "closer" to a group of very active objects. Using the LUF technique, the processing load is distributed to other, less busy engines.

5.1 Distributed Rule Processing Algorithm

A typical issue in distributed environments is resource discovery, and in this case, rule engine discovery. Upon instantiation, each rule engine registers its contact information with a discovery service running in the environment [10]. At the same time, each engine retrieves contact information about other engines already registered and stores that information in an internal list.

```
FETCHPEERS (in: ResDiscEngine) {
  begin
    if p2p_enabled=true then
      retrieve my_peers from ResDiscEngine;
      store peer_info in peer_registry;
    end if;
  end;
PROPAGATEQUERY(in: query,TTL; out: q2pr) {
  begin
   if TTL < 3 then
     for each entry in peer_registry do
        forward query (q2pr) to my_peers ;
     end for;
   end if;
  end; }
```

Fig. 4: Algorithm segments for peer discovery and query propagation

The contents of this list are updated at regular intervals, serving as mini-registries to generate queries for peer engines when distributed rule searches are initiated. In turn, contacted peer engines generate new connections according to their own mini-registries. As a result, discovery of other engines is a byproduct of the search process.

Using the mini-registries is advantageous in terms performance and fault tolerance. The performance improvement is realized by avoiding the processing overhead incurred from the interaction with the Resource Discovery Service (e.g. network communication, issuing remote service requests, handling responses). Fault tolerance is attained by not depending on the availability of the Resource Discovery service to complete the distributed searches.

The algorithm in Figure 4 describes the additional algorithm segments for obtaining other peers and propagating queries.

When a task is sent for rule processing (Figure 5), it arrives to a single rule engine (BRE_0), which plays the role of the search initiator and rule collector. BRE_0 first searches within its own rule repository and then checks for the availability of relevant rule types in peer engines (BRE_n). Peer engines forward search queries to their peers and so on, until a time-to-live (TTL) variable is met. The TTL variable ensures query termination by preventing searches of infinite depth. Hoping from one engine to the next, allows peer engines to generate new connections to their peers, making the discovery of further engines a byproduct of the search process.

After peer engines have been queried, results are returned to BRE_0 for rule enforcement. Enforcement follows the PROCESSTASK algorithm described in section 4.5.1. The only difference with the stand-alone engine processing is the contents of the candidate set, which now include applicable rules retrieved from remote engines.

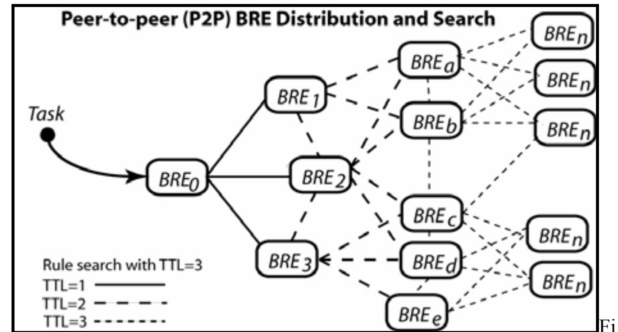

Figure 5: Rule search in a P2P environment with n BREs.

If a rule already exists in the candidate set, the incoming rule is discarded to improve performance by limiting the number of rules to be processed. Once the population of the candidate set is completed the dynamic conflict resolution mechanism goes in effect.

6 Nested Rules & Extraneous Conditions

It is possible that rules may need to be nested, improving on their power of enforcement. While nested rules may provide a better "packaged" solution, they increase processing complexity and the risk of causing non-terminating conditions such as endless loops during enforcement. The proposed system supports nested rules and provides a mechanism to avoid extraneous conditions.

6.1 Nested Rules

During rule formulation, a rule G may be nested, referring to one or more rules. When this occurs the rule types are formulated as follows.

A *Stimulus-Response* rule may be a nested rule and refer to one or more $G_{1..k}$ rules or refer to the conjunction of $G_{1..k}$ rules with A_k other actions, formed as:

$$R_{SR}: \textit{if } P_1 \wedge P_2 \wedge ... \wedge P_n \textit{ then } A_1 \wedge G_1 \wedge ... \wedge G_k \wedge A_k$$

where P_n are the predicates, A_k are the resulting actions and $G_{1..k}$ nested rules.

An *Operation Constraint Rule* may behave as a nested rule if it has $G_{1..k}$ rules both in conjunction with the activities (A_k) and the pre and post-conditions (Q, S):

$$R_{OC}: \textit{if } P_1 \wedge P_2 \wedge ... \wedge P_n \textit{ then}$$
$$[Q\ G_2]\ A_1 \wedge G_1 \wedge A_2 \wedge .. \wedge A_k\ [S\ G_3]$$

Similarly, a *Structure Constraint Rule* may become a nested rule if it has $G_{1..k}$ nested rules in conjunction with the activities (A_k):

$$R_{SC}: \textit{if } P_1 \wedge P_2 \wedge ... \wedge P_n \text{ accept} \mid \text{reject}$$
$$A_1 \wedge G_1 \wedge A_2 \wedge .. \wedge G_k \wedge A_k$$

Finally, *Computation Rules* (R_{CR}: $y=f(x)$) may also be nested if they contain mathematical recursive definitions within the $f(x)$ component. Since Computation Rules are user-defined functions, it is impossible to provide a universal nested rule formalism. Therefore, we accept axiomatically that such rule can be generated.

These nested rule formalisms indicate that if a rule with nested rules is properly stated, then it is valid. To enforce such rule, all the nested rules are fetched in the *candidate set* and processing continues as previously described. A similar case may also appear in distributed searches with multiple engines having local rules referencing rules on remote engines. When nested rules are permitted in a distributed setting, processing termination is a concern since rules may trigger each other indefinitely, causing infinite loops.

In a single rule engine, nested rules are managed with intelligent user interfaces that forbid new rules to be entered if a loop is detected. With multiple rule engines, there is no single interface to detect loops. When rule engines process rules with references to remote engines, the remote engine is queried and matching rules are fetched and processed as if they were local rules. This does not guarantee that remote rules do not contain nested rules (local and remote) causing a cascading effect of loops between rules and remote engines. Unlike the case of single, standalone rule engine repositories where loops may be detected and avoided as early as rule definition, the case of multiple, distributed rule engines is much more complex. There is no accurate up-to-date global view of all rule-sets and attempting to detect potential looping references cross multiple engines is computationally expensive.

Determining in advance whether the rules are guaranteed to terminate is an undecidable problem, although conservative algorithms do exist and generate warnings indicating the possibility of infinite looping. This problem is addressed by enforcing a strict, rule execution depth limit. The depth limit guarantees that if a sequence of rules is repeated, a certain number of times then rule processing is terminated and results up to that point are returned for processing.

6.1.1 Optimistic Fault Tolerance

Grid and P2P environments are comprised of units that often become unavailable for a certain time periods or, they completely disappear. This condition may arise between collaborating rule engines and it is addressed by implementing basic fault tolerance mechanism. After task enforcement, non-local rules are integrated and kept in the rule engine that initiated and enforced the rules. This integration of non-local rules serves as an optimistic fault tolerance mechanism and a performance enhancing technique for environments with static rule-sets. In case of a remote engine failure, access to the local copy of the remote engine's rules provides a degree of fault tolerance. In addition, retrieval performance may increase by treating the local instance of the remote rules as a cached copy. For environments with dynamic rule-sets (rule-sets that continually get updated), rule engines can be configured to ignore any local rule-sets and always attempt to retrieve the latest rules.

Overall, the use and participation in the distributed rule engine environment is optional and can be changed during real-time operations. The ability to decline participation is advantageous for environments that wish to impose strict enforcement of very specific rule-sets. On the other hand, environments with multiple collaborating engines, operating with broader rule-sets and looser rule enforcement requirements, can take advantage of the scalability benefits offered by the P2P-based processing.

7 Conclusion and Future Work

We have presented the architecture, components and functionality of a distributed system for rule representation and computation.

The objective of our work was to provide a comprehensive distributed rule processing system that reconciles the shortcomings of existing monolithic, centralized systems. The described system and implementation provides a viable replacement, using multiple, distributed, lightweight rule engines that can operate concurrently, avoid bottlenecks and single points of failure.

We have presented the architecture, components and functionality of this system and described its implementation. We expressed rules using standard XML constructs with a minimal, but very general, predefined range of syntax and semantics. We proceeded by developing the concept of lightweight rule engines able to accept tasks, apply relevant rules, resolve conflicts, modify tasks and return them for distribution and execution. We extended the use of the single rule engine to a network of multiple loosely connected collaborating engines allowing rule sharing. The presented solution has the usual many advantages of a distributed system over centralized solutions and typical problems to overcome when distributing the system (e.g. rule conflicts) are addressed.

Overall, by borrowing operating principles from P2P networks, we developed a general, distributed rule processing architecture that can be applied to any environment with similar computation and distribution requirements.

For the future, we plan on taking a two-step approach. First, we will use the proposed system to exercise the implementation using a particular case study. Second, we will undertake performance analysis studies to determine the processing efficiency and cost overheads of the typical operation and the conflict resolution mechanisms.

References

[1] A. Aiken, J. Widom, and J. M. Hellerstein, "Behavior of database production rules: Termination, confluence, and observable determinism," in Proc. of the ACM SIGMOD Intl. Conference on Management of Data, June 1992.

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

[3] F. Casati, S. Ceri, B. Pernici, and G. Pozzi, "Deriving active rules for workflow environment," 7th Intl.

Conference on Database and Expert Systems Applications, Lecture Notes in Computer Science: Springer-Verlag, 1996, pp. 94-110.
[4] S. Ceri, P. Grefen, and G. Sanchez, "WIDE-A Distributed Architecture for Workflow Management," in RIDE97-7th Intl. Workshop on Research Issues in Data Engineering, Birmingham, UK, 1997.
[5] S. Chakravarthy, "A research project in active, time constraint database management," Xerox Advanced Information Technology, Cambridge, MA, Technical Report XAIT-89-02, 1989.
[6] S. Chakravarthy, "Rule Management and Evaluation; An Active DBMS Perspective," ACM-SIGMOD Record, vol. 18, pp. 20-28, 1989.
[7] R. Davis, "Interactive Transfer of Expertise: Acquisition of New Inference Rules," Artificial Intelligence, vol. 12, pp. 121-157, 1979.
[8] U. Dayal, B. Blaustein, A. Buchmann, et al, "The HiPAC Project: Combining Active Databases and Timing Constraints," SIGMOD Record, vol. 17(1), pp. 51-70, 1988.
[9] U. Dayal, E. Hanson, N., and J. Widom, "Active Database Systems," in Modern Database Systems: The Object Model, Interoperability, and Beyond, W. Kim, Ed. Reading, Massachusetts: Addison-Wesley, 1994.
[10] Dimitoglou, G., Moore, P., Rotetenstreich, S. (2003) “Middleware for Large Distributed Systems and Organizations,” Proc. of the Intl. Symposium on Information and Communication Technologies, Trinity College, Dublin, Ireland, Sept. 24-26, pp. 553-559.
[11] K. R. Dittrich, S. Gatziu, and A. Geppert, "The Active Database Management System Manifesto: A Rulebase of ADBMS features," in Proc. of the 2nd Intl. Workshop on Rules in Database Systems, vol. 985: Springer, 1995, pp. 3-20.
[12] J. Greer, G. McCalla, V. Kumar, J. Collins, and P. Meagher, "Facilitating Collaborative Learning in Distributed Organizations," in Computer Support for Collaborative Learning 97. Toronto, Ontario, 1997.
[13] B. N. Grosof, "Business Rules for Electronic Commerce: Interoperability and Conflict Handling," IBM, Project Overview Talk Slides 1999.
[14] B. N. Grosof, "Compiling Prioritized Default Rules into Ordinary Logic Programs," IBM, Research Report RC 20836 (92273), 1999.
[15] B. N. Grosof, "Contracts, Policies, and Prioritized Rules in XML Agent Communication," presented at 18th Meeting of FIPA, UMBC Technology Center, Baltimore, MD, 2000.
[16] B. N. Grosof, Y. Lambrou, and H. Y. Chan, "A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML," presented at 1st ACM Conference on Electronic Commerce (EC99), Denver, CO, USA, 1999.
[17] E. N. Hanson, "Rule condition testing and action execution in Ariel," in Proc. of the ACM SIGMOD Intl. Conference on Management of Data, May1992.
[18] HNC Software Inc., "Developing Real World Java Applications with Blaze Advisor," San Jose, CA, USA, Technical White Paper 2001.
[19] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," Communications of the ACM, vol. 12, pp. 576-580, 1969.
[20] ILOG Business Rules Team, "Business Rules: Powering Business and E-Business," Paris, France, White Paper 2001.
[21] D. R. McCarthy and U. Dayal, "The Architecture of an Active Database Management System," in Proc. of the 1989 ACM SIGMOD Intl. Conference on Management of Data, 1089, pp. 215-224.
[22] B. Meyer, Object-oriented software construction, 2nd. ed. Upper Saddle River, N.J.: Prentice Hall PTR, 1997.
[23] J. A. Miller, S. A.P, K. K. J., and W. X., "CORBA-based run-time architectures for workflow management systems.," Journal of Database Management, July 1996.
[24] A. Poetzsch-Heffter, "Deriving Partial Correctness Logics From Evolving Algebras," in 13th World Computer Congress, vol. I: Technology/Foundations, B. P. a. I. Simon, Ed.: Elsevier, Amsterdam, the Netherlands, 1994, pp. 434--439.
[25] D. Roach and H. Berghel, "The physiology of PROLOG expert system inference engine," presented at Proc. of the 1990 ACM SIGSMALL/PC Symposium on Small Systems, Crystal City, VA, 1990.
[26] L. Rouvellou, L. DeGenaro, H. Chan, K. Rasmus, B. N. Grosof, and B. McGee, "Combining Different Business Rules Technologies: A Rationalization," presented at OOPSLA 2000 Workshop on Best Practices in Business Rule Design and Implementation, Minneapolis, MN, USA, 2000.
[27] M. Stonebraker, "The POSTGRES next generation database management system," Communications of the ACM, vol. 34, pp. 78-92, 1991.
[28] M. Stonebraker, A. Jhingaran, J. Goh, and S. Potamianos, "On rules, procedures, caching and views in database systems," in Proc. of the ACM SIGMOD Intl. Conference on Management of Data, May 1990.
[29] X. Wang, "Implementation and evaluation of CORBA-Based Centralized Workflow Schedulers," in Computer Science. Atlanta: University of Georgia, 1995.
[30] D. H. D. Warren and L. M. Pereira, "PROLOG: The Language and its Implementation Compared to LISP," SIGPLAN Notices, vol. 12(8), 1977.
[31] J. Widom, R. J. Cochrane, and B. G. Lindsay, "Implementing set-oriented production rules as an extension to Starbust," in Proc. of the Seventeenth Intl. Conference on Very Large Databases, 1991.
[32] J. Widom and S. J. Finkelstein, "Set-oriented production rules in relational database systems," in Proc. of the ACM SIGMOD Intl. Conference on Management of Data, May 1990.

George Dimitoglou received a Ph.D. in Computer Science with concentration in Parallel and Distributed Computing from the School of Engineering & Applied Science of The George Washington University; Before joining the faculty at Hood College he spent over a decade working in the industry and government with a last post at NASA’s Goddard Space Flight Center. He is a member of the ACM, IEEE and the Mathematical Association of America.

Shmuel Rotenstreich received a Ph.D, and a M.S in Computer Science from the University of California, San Diego and a B.Sc in Mathematics, from Tel Aviv University. He is an associate professor of computer science at the School of Engineering & Applied Science of The George Washington University. In the last several years his work has been focused in the area of computational aspects of large organizations. This includes large P2P systems, military systems and sensor systems. He also works on large scale emergency response problems and better resource discovery methods.