A Rule Warehouse System for Knowledge Sharing and Business Collaboration

UF CISE TR01-006

Youzhong Liu
Charnyote Pluempitiwiriyawej
Yuan Shi
Herman Lam
Stanley Y. W. Su

Database Systems R&D Center
University of Florida
CSE 470, P.O. Box 116125
Gainesville, FL 32611-6125
{yoliu, cp0, yshi, hlam, su }@cise.ufl.edu

H. Chan
T. J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532
hychan@us.ibm.com

Table of Contents

1. Introduction

The emergence of the Internet and World Wide Web technologies is one of the most significant technical achievements in history. The Internet has changed much of our everyday lives. Email has become the most convenient method for personal and business communications. Online news is an inexpensive and prompt method of getting information about what is happening around us. People around the world can overcome geographical constraints and keep in touch with each other easily. More and more people and organizations are using the Internet to share information, to do collaborative work, and to perform business transactions. Online commerce and e-business have become a reality.

Companies worldwide are experiencing tremendous opportunities and growth which has been enabled by the Internet. At the same time, the Internet and the Web have forced companies to re-examine the fundamentals of their business and develop new strategies to approach a Web-based marketplace. E-business is designed to meet this new challenge by automating the business activities of companies that form a supply chain. Rather than using Internet sites as mere addresses for simple interactions, companies are increasingly using Web applications to automate and enrich interactions among customers, employees, and business partners to improve their efficiency [LAC99].

E-business is promising because it is advancing us to transparent and efficient markets and businesses. ***Transparency*** is a knowledge-based concept, which implies that participants have knowledge about the markets around them. Market alternatives become transparent, and, consequently, participants may change their behavior based on some knowledge to achieve additional benefits. The Internet eliminates one of the major limitations to market transparency: geography. If we can model approval procedures, purchasing limits, preferred suppliers, volume purchasing agreements and other aspects of business knowledge, and provide a means to *share the knowledge*, we can enhance business transparency.

According to [PHI00], there are four major phases in the evolution of B2B technology. B2B appeared first in the form of *EDI* (Electrical Data Interchange). *EDI* has had a major impact in reducing errors and shrinking processing times for certain types of transactions. However, EDI technology is brittle and difficult to change in a dynamic marketplace. The point-to-point connections of EDI provide no community or market transparency. *Basic E-commerce* follows *EDI* as the second phase of B2B. In this phase, retailers sell their products through their Web sites. Today, this form of commerce is widely available from many companies, such as Amazon.com and Buy.com. However, most of the applications only support simple sale and purchase operations. Few of them support post-sale tracking and services. Even less Web companies provide automatic procurement support. Almost none of them support pre-sale planning and negotiation. Phase three of B2B is currently unfolding – third party Web destinations that bring trading partners together into a common community. This phase is called *Communities of Commerce*. Communities of enterprises create market transparency. The intersection of buyers and sellers with related interests creates an opportunity to serve a large percentage of those interests. Some of the existing eMarketPlaces provided by CommerceOne and Ariba belong to this category. The next stage, *collaborative e-business*, builds on phase

three by adding support for other business processes before, during, and after the order. *Collaborative e-business* fills in the gaps around e-commerce. It is a more complete reflection of the complex interactions between demand and supply chains.

Collaborative e-business is attractive because of the tremendous benefits it offers. Efficiency is the key to win or stay in contention in the e-Business battle. Business Process Reengineering (BPR) and workflow technology [WFMC] enable the optimization of the internal management of an enterprise to improve efficiency. However, inter-enterprise *collaboration*, one of the most important factors that affect the efficiency of the entire supply chain, is still in its infancy in e-business. Automated collaboration is very limited in current e-businesses. Unlike Business-to-Customer (B2C) operations, most of the Business-to-Business (B2B) operations are still performed manually. Telephone and fax are still the most important business tools. People spend much time and effort in communicating and understanding business partners' business rules and regulations.

In order to automate e-business collaboration, the *business knowledge of different business partners needs to be shared electronically* and be used to *solve business problems collaboratively.* The business knowledge of individual companies is commonly expressed in terms of business events and business rules and managed by some rule processing systems. The event and/or rule representation and the rule processing system of one company may be different from those of others. It is important to have a way of capturing and sharing these heterogeneous events and rules because they are important resources just like data, application systems, hardware systems, etc., that can be contributed by individual companies to the joint business. Similar to the concept of a data warehouse system, a Rule Warehouse System can be developed to import, transform, cleanse, and manage these heterogeneous rules. Businesses may use the Rule Warehouse System to solve complex problems that cannot be solved by the rules of individual companies. In this manner, the Rule Warehouse System can facilitate cooperative problem solving among business partners to provide better B2B solutions. Also, rules managed by the Rule Warehouse System can be exported to legacy rule systems for use by individual companies, thus achieving business knowledge sharing.

One of the main obstacles of developing such a Rule Warehouse System is the problem that rules imported into the Rule Warehouse may contain inconsistency, redundancy, and cyclic conditions. They need to be verified by the Rule Warehouse System to make sure that these rule anomalies do not exist. Rule base verification has been an important area of research in the expert system community. Techniques for verifying expert system rules are available. However, the verification of a rule base containing action-oriented rules is a much more challenging problem because this type of rules contains method or procedure calls which can have side effects. Besides, this type of rules is triggered by different events, and the execution of rules may post events to trigger other rules.

1.1. Research Objectives and Approach

In this paper, we introduce the concept of a business Rule Warehouse System to enable e-business collaboration by supporting business rule sharing, collaborative

problem solving, and collaborative interaction using business events and rules. The results reported in this paper are based on the concepts presented in an earlier EECOMS white paper [LIU00a]. A Rule Warehouse System consists of a Rule Warehouse and a Rule Warehouse Management System (RWMS). The Rule Warehouse provides a persistent repository to integrate and store heterogeneous business rules in a virtual enterprise. The RWMS provides a set of functions to manage the Rule Warehouse and provides a set of services to users and applications to enable e-business collaboration.

After a survey of existing rule types and rule systems in Section 2, an architecture of a Rule Warehouse System is given in Section 3. Also in Section 3, we identify the following set of RWMS services required to enable e-business collaboration:

- Rule Import, Verification, and Export Services
- Collaborative Problem Solving Service
- Warehouse Management Service
- Constraint Satisfaction Service
- Event-Trigger-Rule (ETR) Service

A set of key technologies required to support the RWMS services is also outlined in Section 3. Then, in the remainder of this report, we will provide some details on those technologies that are our focuses in this project. In Section 4, an ***Active Object Model*** (AOM) is presented as a common representation for the Rule Warehouse to represent dissimilar business rules for the purpose of rule verification. AOM is an object-based knowledge model capable of defining objects in terms of attributes and methods (just like traditional object models) as well as events, different types of rules, and triggers. In Section 5, we will describe how a Deductive Rule Engine (DRE) and an Action-oriented Rule Engine (ARE) can be integrated to perform collaborative problem solving, using the knowledge in the Rule Warehouse. In particular, we integrate a Common Rule System (a DRE developed by the IBM Watson group) and an ETR Server (an ARE developed by the University of Florida group) to perform collaborative problem solving in the context of Scenario X used in the EECOMS's December 2000 technical assessment demonstration. In Section 6, we will present the results of our work on rule verification. We will introduce verification methods for detecting inconsistency, redundancy, and non-termination anomalies, which may exist in the knowledge specifications stored in the Rule Warehouse. A novel approach to verify action-oriented rules will be presented. Finally, in Section 7, we will present a summary, some concluding remarks, and future work.

2. Business Rules and Rule Management

High-level specification and management of business rules is more effective than implementing rules in programs using traditional programming languages. Rules are high-level specifications of logic and control which are needed to conduct and manage intra- and inter-enterprise operations [EEC99]. By separating the business logic from the programming logic in an application, business rules can be changed without changing the rest of the application [BLA]. An additional benefit of specifying business rules in a high-level rule specification language is that it is easier for nonprogrammers, such as business executives, to understand.

In recent years, management of business rules has attracted much attention. John Zachman provided a useful context for discussing the architecture of an information system in [ZAC87]. Much of the research work on business rules is based on the "Zachman Framework." In [GOT97, HAY99, PLO99], the authors pointed out the necessity and importance of better management of business rules. [SEI99] proposes a repository-based approach to manage business rules. Ronald Ross wrote a comprehensive book on this subject [ROS97b], and Barbara von Halle published a number of articles in Database Programming and Design [HAL97a, HAL97b], in which the importance and use of business rules is stressed. In this section, we will review different types of rule systems, including the ones used in the EECOMS project, and some definitions of business rules.

2.1. Rules and Rule Management

Rules have long been a common knowledge representation in the artificial intelligence (AI) area and, more recently, in other areas such as active database systems. Four types of rules are commonly recognized and distinguished: logic rules [GON97], production rules [GON97], constraints [HUA00], or action-oriented rules [LEE00].

Logic rules can be processed by different types of logic-based rule engines using different inferencing schemes (e.g., forward chaining vs. backward chaining) to solve various types of problems in AI. Logic rules are expressed in the form, P→ Q, which stands for "If P is true, then Q is true," where P and Q are logical expressions and P is the antecedence and Q is the consequence. Production rule system is one type of logic-based rule system. It is also known as expert system and is designed to solve complex problems by using experts' knowledge captured in rules. In an expert system, data are stated as facts in a fact base. A fact is inserted into the fact base if it is true, and removed from the fact base if it becomes false. An example commercial rule system of this type is the expert-system-based products of the Haley Enterprise, Inc. They are designed to support business rule processing in the eCRM (electrical Customer Relationship Management) application area [HAL99]. Haley Enterprise's products use an extended Rete Algorithm (Rete++) to provide better performance and scalability in rule processing.

Constraints are widely used in database and business areas for specifying many types of data and business constraints. They are used in database systems to enforce security and integrity constraints and by constraint satisfaction processors to verify if some data conditions violate some specified constraints. For example, a data integrity constraint

states that the salary of an employee cannot exceed the salary of his manager. If any operation which changes an employee's salary violates the constraint, the transaction associated with the operation will be rolled back. In business, a product specification (in terms of constraints) of a buyer can be matched against the capabilities (in terms of constraints) of a supplier to determine if the supplier is a qualified one in a supplier selection process. Constraints are often used in constraint satisfaction processing systems for verifying if some input specification violates the constraints managed by these systems. The semantics expressed by logic rules and constraints are very similar. The general form for logic rules, P→ Q, can also be used to express different types of constraints as shown below:

1. True → Delivery_day > 15

2 True → Food_cost + Tax = Total_charge

3. Quantity > 100 → Delivery_day > 10

The first constraint is called an attribute constraint and the last two are inter-attributed constraints. The antecedents of the first two expressions are True, specifying the constraints must be unconditionally enforced.

Since the semantics captured by constraints can be expressed as logic rules, we can treat constraints as logic rules and convert them into the same representation. This is important because when we do rule base verification (to be discussed in Section 6), we need to have a uniform rule representation so that theory proving techniques can be applied on logic rules to perform rule base verification. A set of verified logic rules can be used in a logic-based rule engine for inferencing purposes or in a constraint satisfaction processing system for constraint checking.

Action-oriented rules are commonly used in active database systems [HAA90, HAN93, MCC89, STO88, WID96]. There are two general types of action-oriented rules: triggers and event-condition-action (ECA) rules. Triggers in active databases enforce business knowledge by automatically invoking data operations when a predefined condition is satisfied. In a trigger definition, a condition and some data operations are specified [HAN93]. A database operation such as Delete, Insert or Update would trigger the evaluation of the condition part of a trigger. If the condition is True, then the data operations of the trigger are performed automatically. For example, a business rule specifies that, if a customer's address is modified, then any unshipped orders for that customer should be shipped to his/her new address. To support this business rule, a trigger is defined to update the shipping address of the unshipped orders of that customer in an *Order* table whenever that customer's address in the *Customer* table is modified.

Database triggers only react to storage operations on the data in a database. However, business rules need to be applied to both database and non-database operations. Furthermore, the triggered actions of a business rule are not always database operations. ECA rules [HAA90, MCC89] are generalization of database triggers in that the events that trigger a rule can be any event, and triggered actions can be any type of operations, not just database operations. The semantics of an ECA rule is, "When an event is posted, the condition part is checked. If the condition is true, then perform the action." The semantics of an ECA rule is different from that of a logic rule or constraint in two ways.

One difference is that an ECA rule has an event specification, and the evaluation of the Condition and Action parts is subject to the occurrence of an event; whereas, a logic rule or constraint does not have the concept of event. We note here that it is possible to treat events as data and include them in the expressions of logic rules and constraints. However, the concept of a posted event triggering the evaluation of data condition (i.e., the order of checking) is not captured. The second difference is that an ECA rule explicitly specifies operations that are to be performed in the Action part of the rule, whereas a logic rule or constraint does not contain such a specification. The Condition part of an ECA rule corresponds to a logic rule or constraint specification. They can all include procedure calls.

ECAA rule [LAM98] is an extension of ECA rule by specifying the alternative actions if the condition is evaluated to False. Action-oriented rule systems can be used to detect the occurrence of business events and take proper actions to enforce business rules. An example of commercial rule systems that use this approach is the Blaze Advisor [BLA], which is the main product of Blaze Software, Inc. In this system, business rules and policies are separated from the procedural logic, enabling applications to adapt to changes in business. A natural-language-like language is used to allow business users to define business rules.

2.2. Rule Systems Used in EECOMS

In EECOMS, several rule systems have been used to support supply chain management. We shall briefly describe their use and the collaborations between them.

The University of Florida group (UF) uses a constraint language, which is modeled after OMG's constraint language, to express the requirements and constraints associated with products and services that suppliers/buyers want to sell/purchase. The language and a constraint satisfaction processor are used in UF's automated negotiation server for evaluating the contents of a negotiaton proposal against some pre-registered constraints. An action-oriented rule engine, the ETR Server [LEE00], is used to process decision rules that implement negotiation strategies [SU00, HUA00]. The group at the University of North Carolina at Charlotte (UNCC) uses ILOG as their rule engine [EEC99] for supplier selection and optimization. The group at the University of Maryland at Baltimore County (UMBC) uses a rule language, which is a close derivative of KIF [EEC99], in an agent-based negotiation system. Constraints and action-oriented rules are also used in EECOMS's work on security and virtual situation room [EEC00]. Vitria's Businessware is used to model business processes in a supply chain environment chain [VIT]. The system uses ECA rules to specify conditional transitions among activities in process models. IBM Watson uses Courteous Logic Programs (CLP) [GRO99b] as the rule interlingua for exchanging heterogeneous logic-based rules.

The colloboration between different types of rule engines have been demonstrated in the EECOMS project. For example, the rule engines used by UNCC and UF have been integrated to demonstrate the interaction between supplier selection and negotiation in EECOMS's Scenario 1. The UF group developed a translator for converting UF's constraint language into IBM's CLP to demonstrate the use of the rule interlingua. UF's ETR Server has been integrated with Vitria's Businessware to demonstrate the interaction

between business rules and business processes in EECOMS's Scenario 5.

2.3. Business Rules

In 1993, an IBM User Group, GUIDE, established a project to articulate the requirements and possible approaches to specifying business rules. The group published a final report in 1997 [GUI97]. The definition of a business rule given by the GUIDE project is widely referenced in the literature. It defines a business rule as "… a statement that defines or constrains some aspect of the business. This must be either a term or fact (described as a ***structural assertion***), a constraint (described as an ***action assertion***), or a ***derivation***." A *fact* or *structural assertion* states an item of importance to the business, which either exists as a concept of interest, or exists in relationships to other items of interest. Using EECOMS's Scenario X as our running example, an example of a fact is: "In 1999, the car model Honda Accord was involved in 23,456 accidents nationwide." An *action assertion* is a statement about some dynamic aspect of business. It specifies constraints on the results that actions can produce. An example of a constraint is: "Any change to the inventory of a given model of cars (the action) is subject to the constraint that the number of doors on hand for that model must not fall below some threshold." A *derived fact* is created by a derivation, which is an inference or a mathematical calculation from terms, facts, other derivations, or action assertions. An example derived fact is: "The number of doors to be replaced for a given car model can be obtained by multiplying the number of accidents which involve this particular model of car by the probability that one or more doors will be damaged in an accident."

Recently, the Meta Data Coalition proposed a Business Rule Model (BRM) [MDC99] by adopting GUIDE's business rule definition. Business rules are classified in a similar way as GUIDE into ***term rules***, ***fact rules***, ***action rules***, and ***inference rules***. *Term rules* and *fact rules* are pretty much the same as the facts or structural assertions in GUIDE. In BRM, *action rules* are statements that are concerned with the invocation of actions, such as events, pre-conditions, and post-conditions. BRM is different from GUIDE in that event is considered as a part of the condition. *Inference rules* in BRM describe the inference or derivation of a business rule from other rules or by mathematical calculations. They capture knowledge that is dynamically derived instead of explicitly stored.

Later in Section 4, we will illustrate our support to the above business rule definition and classification. We will use an Active Object Model (AOM) to incorporate business rules in the definition of business object. In AOM, constraint-oriented and action-oriented rules will be used to represent inference rules and action rules, respectively, as defined in BRM.

3. Rule Warehouse System

A ***Rule Warehouse System*** consists of a ***Rule Warehouse*** and a ***Rule Warehouse Management System (RWMS)***, which processes and manages the rules stored in the Rule Warehouse. The Rule Warehouse provides a persistent repository to integrate and store heterogeneous business rules of business companies, which constitute a virtual enterprise, such as a supply chain. The RWMS provides a set of services to users and applications to enable e-business collaboration. In Section 3.1, we describe the RWMS services and outline the key technologies required to support these services. In Section 3.2, an architecture of a Rule Warehouse System is given. In Section 3.3, two architectural alternatives for implementing a Rule Warehouse System are discussed.

3.1. Rule Warehouse Management System Services

An RWMS provides several services to e-business users and applications: These services are important for achieving ***business rule sharing***, ***collaborative problem solving***, and ***collaborative interaction*** among businesses. We categorize RWMS services into five categories, as shown in Figure 3.1.

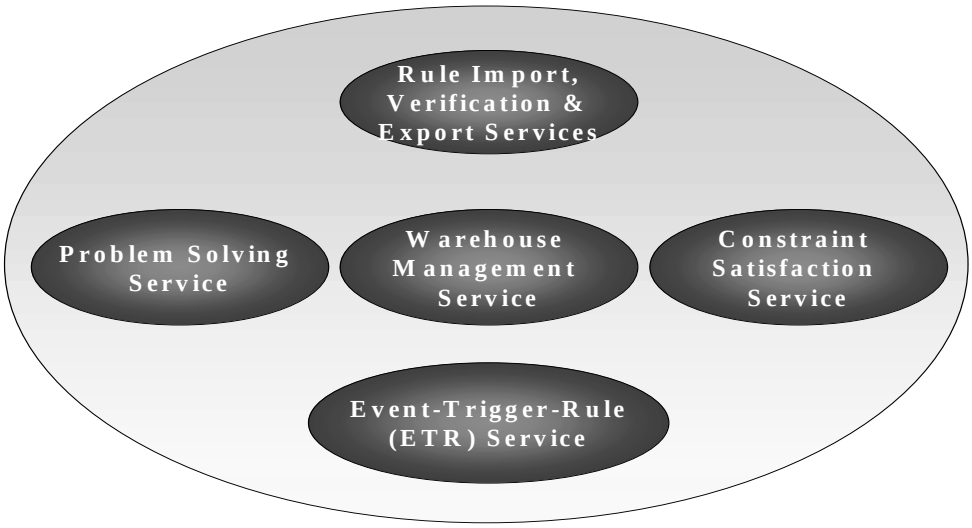

Figure 3.1 Rule Warehouse Service Illustration

Rule Import, Verification, and Export Services. The rule *import* service facilitates the importing of dissimilar rules from heterogeneous sources into the Rule Warehouse. For importing rules into the Rule Warehouse, heterogeneous business rules specified in different "native" rule languages of some existing rule systems are first translated into a rule interlingua. From the interlingua representation, these rules are translated into the common rule representation supported by the Rule Warehouse System.

Newly imported rules need to be combined with the rules that have been imported into the Rule Warehouse and need to be verified together to detect any rule anomalies (i.e., inconsistency, redundancy and non-termination) in the combined rule set.

The rule *export* service facilitates the exporting of rules from the Rule Warehouse to some existing rule systems. The exported rules must be translated first into the rule interlingua. From the interlingua representation, these rules are then translated into the native rule languages of the existing rule systems.

Within EECOMS, the IBM Watson group has done much work on a rule interlingua for exchanging logic-based rules among dissimilar rule systems [GRO99a, GRO99b]. In our work, we have concentrated on a common rule representation in the Rule Warehouse (see Section 4) for rule verification purposes (see Section 6). Along with the rule management service to be described below, the rule import, verification, and export services play a key role in realizing the goal of rule sharing among partners of a virtual enterprise.

Rule Management Service. This service is used to determine what rules are to be shared. It provides browsing and directory functions for browsing the contents of the Rule Warehouse, and for retrieving the events and rules or the meta-data associated with events and rules (including the types of rules, the source of the rules, authorization information, etc.). Additionally, the rule management service provides functions to define new events and rules, and edit and delete existing events and rules. For human users, business events and rules need to be specified at a high-level of abstraction, using an easy-to-understand representation and a user-friendly GUI-based interface.

Collaborative Problem Solving Service. One main benefit of a Rule Warehouse System is to integrate knowledge (in the form of events, rules, and constraints) from different sources and use them to solve problems that cannot be solved by using the knowledge and rule engine of any individual source. Another benefit is to allow a better solution to a problem by using the integrated knowledge of a Rule Warehouse System. For example, a car manufacturer wants to predict how many doors to manufacture for the coming year. It may use its own data and rules to do the prediction. However, the knowledge represented by these data and rules can be quite limited. If the manufacturer can have access to knowledge managed by a Rule Warehouse System, the prediction can be much more accurate. We believe that this rule warehouse service is useful in many other applications in supply chain management, such as procurement planning, cooperative designing and execution, and eCRM (Customer Relationship Management).

The problem-solving service of the Rule Warehouse System allows a user to issue a request (or query), which provides some known information and asks for some information unknown to the user. Upon receiving the request, the Rule Warehouse System will proceed to solve the problem using the rules in the Rule Warehouse. As will be described in the next section, rules in the Rule Warehouse can be constraint-oriented or action-oriented, each type is executed by its respective rule engine available in the Rule Warehouse System. In our work, we have concentrated on the integration of a deductive rule engine with an action-oriented rule engine so that knowledge in different forms (constraint-oriented rules and action-oriented rules) and from different sources can be used to solve problems collaboratively. The integration of these two types of rule

engines will be detailed in Section 5. Also, in Section 5, we will describe the application of the collaborative problem-solving service of the Rule Warehouse in Scenario X of EECOMS's December 2000 technical assessment.

Constraint Satisfaction Service (CSS). Constraints are widely used to represent business requirements. Constraint satisfaction processing is the task of comparing one set of constraints (e.g., constraints in question) against another set of constraints. For example, an incoming negotiation proposal/counter-proposal specified as a set of constraints can be compared with an enterprise's constraints to determine if the proposal is acceptable [SU00]. Another example is the selection of a supplier from a group of suppliers. Each supplier's capability (specified as a set of constraints) can be compared with the selection requirements (another set of constraints) to evaluate a supplier. With respect to the Rule Warehouse System, we define the constraint satisfaction processing service as the following. Given an input set of constraints S_i, the RWMS will check against the constraints S_r stored in the Rule Warehouse. If all the constraints of the input set do not conflict with S_r, the no-conflict result will be reported by RWMS. If any conflict has been found, the specific constraint that causes the conflict will be reported. This last action of the constraint satisfaction service is quite different from most of the traditional constraint satisfaction processors, which can only report the conflict condition but are not able to pinpoint which constraint or constraints cause the conflict condition. In our work on an automated negotiation server, we have made extensive use of a constraint satisfaction processor for evaluating the contents of negotiation proposals [SU00, HUA00].

Event-Trigger-Rule (ETR) Service. The operations of an e-business enterprise are often subject to the influence of business events and rules. Business events can be all things of importance to an enterprise. They can represent data states that have been reached, actions taken by users through a browser, signals from external devices, before (or after) the execution of a method, before (or after) the enactment of a business process, etc. In a business environment, the occurrence of an event may require the invocation of some business rules. These rules can be used to enforce business policies and strategies, security and integrity constraints, or to enact business processes. Automatic notification and activation of business rules in a timely manner is critical to the success of a business enterprise.

For providing the business event and rule service in a distributed environment, we modify the conventional event-condition-action (ECA) rule approach used in the active database system area [HAA90, WID96] and introduce an Event-Trigger-Rule (ETR) specification [LAM98] for defining events, rules and their inter-relationships. Unlike ECA rules, we separate events and rules specifications, which can thus be defined by different people or organizations, and use trigger specifications to tie events with rules or structures of rules. A rule in the ETR specification consists of a condition, an action and an alternative action specification. It specifies a small granule of control and logic and can be combined with other rules for forming a rule structure to specify a larger granule of control and logic. We also distinguish "triggering events" from events that participate in composite event expressions (or event history) in our trigger specifications. An example of a trigger specification is "When E1 or E2 occurs, verify if E3 and E4 have already occurred within a specified time window. If so, activate a structure of rules."

Events allow loosely coupled systems (from different organizations of a virtual enterprise) to inter-operate and collaborate with minimal dependency (i.e., by event publish-subscribe). Rules allow non-trivial interoperation and collaboration at a high level (i.e., by rule specification instead of coding in a programming language). Triggers (along with rules) provide the flexibility required in an e-business environment in linking events with rules or rule structures.

The ETR service of the Rule Warehouse System can be a stand-alone service, which responds to an event posted by an external source (e.g., an application system or a graphical interface). It will perform event history processing, then it triggers appropriate rules to perform some actions. Additionally, it can be used in collaboration with the deductive rule engine to perform collaborative problem solving. In Section 5, we will describe how the ETR service is used to support collaborative problem solving. In particular, we will describe how it is used in Scenario X of the EECOMS's December 2000 technical assessment.

3.2. Rule Warehouse System Architecture

Figure 3.2 shows the architecture of a Rule Warehouse System. There are six major components in the Rule Warehouse System:

- Rule Warehouse (RW) – The Rule Warehouse is a repository of the rules. Additionally, it contains the catalog information about the rules (e.g., types of rules, source of the rules, security information, etc.).
- Rule Warehouse Manager (RWM) – RWM provides functions to support rule management services, such as browsing, directory functions, event/rule definition, modification, and deletion. The other components of the Rule Warehouse System use the RWM services to read from and write to the Rule Warehouse. RWM also assists in the import and export services.
- Rule Warehouse Interface (RWI) – External users and applications interact with the Rule Warehouse System by calling the APIs or using the GUIs provided by RWI. Upon receiving the requests, RWI will dispatch them to the proper components based on the types of requests.
- Rule Verifier (RV) – The Rule Verifier is used to verify if the set of rules in the Rule Warehouse contains rule anomalies, such as inconsistencies, redundancies and non-termination.
- Constraint Satisfaction Processor (CSP) – CSP is used to perform the Constraint Satisfaction Service described in Section 3.1.
- Deductive Rule Engine (DRE) – DRE is a traditional logic-based rule engine, of which IBM Watson's Common Rule Systems (CRS) [GRO99a, GRO99b] is an example. It is used in conjunction with an action-oriented rule engine to provide the Collaborative Problem Solving Service.
- Action-oriented Rule Engine (ARE) – ARE is used to manage business events and process action-oriented business rules. The University of Florida's Event-Trigger-Rule (ETR) Server [LEE00] is an example of this type of rule engine. ARE responds

to an event posted by an external source or by the DRE. It performs event history processing, then triggers appropriate rules to perform some actions.

We will now describe how the components of the Rule Warehouse System interact with each other to provide the five types of Rule Warehouse services described in Section 3.1.

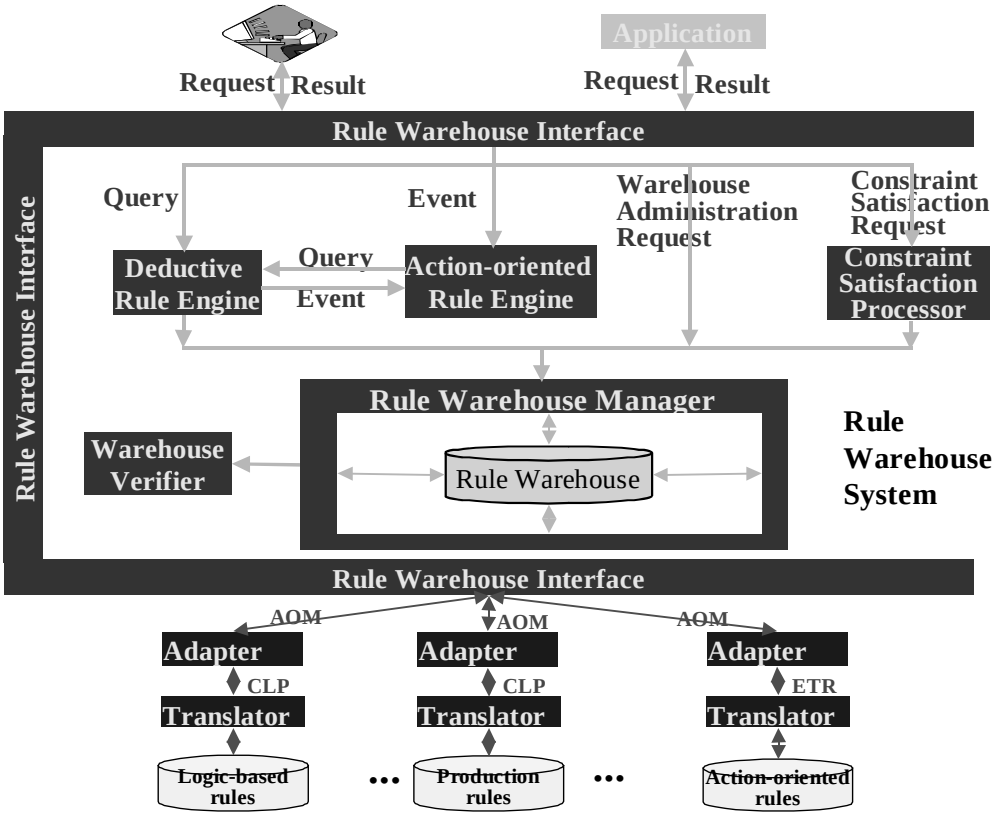

Figure 3.2 Rule Warehouse System Architecture

3.2.1. Rule Import, Verification, and Export Services

For rule importing, each rule source (shown at the bottom of Figure 3.2) makes use of a source-specific translator to convert rules from its native format into a rule interlingua. This is essentially a schema as well as a syntactic translation of rules. The translator needs to have some knowledge about the schema (i.e., object entities, attributes, and their inter-relationships) assumed by these native rules. The interlingua serves as a neutral representation, through which heterogeneous rules are converted and shared. Based on the two general types of rules discussed before, we propose to use the IBM Watson's Courteous Logic Programs (CLP) [GRO99a, GRO99b] as the interlingua for translations between constraint-oriented rules and the University of Florida's Event-Trigger-Rule (ETR) specification [LEE00] as the interlingua for translation between action-oriented rules. Both interlingua representations are parsed and converted by adapters into an ***Active Object Model*** (AOM) representation. AOM is an extended object model capable of representing objects in terms of attributes/properties and methods (just like the

traditional object model) as well as events, rules, constraints, and triggers. We will describe AOM in more detail in the next section.

Upon receiving the import request, the Rule Warehouse Interface (RWI) will forward the import request to the Rule Warehouse Manager (RWM), which will combine the imported rules with the rules already stored in the Rule Warehouse and invoke the Rule Verifier to verify the entire set of rules for inconsistency, redundancy and non-termination. If no anomaly is found, the imported rules are deposited into the Rule Warehouse by RWM. The catalog of the Rule Warehouse is updated accordingly. Also, the verified rules will be loaded into the respective (DRE and ARE) rule engines. If the Rule Verifier finds some anomalies or possible anomalies, proper messages will prompt the user or the Rule Warehouse Administrator via RWI to examine the imported rules. The user or the Rule Warehouse Administrator may then correct the rules to remove the identified anomalies or choose to ignore some possible anomalies.

Rule exporting works in a similar way as rule importing. The rules to be exported to an existing rule system are first converted by an adapter from the AOM representation to an interlingua representation. A target-specific translator is then used to convert these rules from the rule interlingua representation into the native representation of the target rule system. It is the rule system's responsibility to verify if the imported rules (exported by the Rule Warehouse System) conflict with its own rules. It should be noted that not all the rules of the Rule Warehouse can be exported to an existing rule system because of the semantic mismatch between the rule representations of the warehouse and the native rule system.

3.2.2. Rule Management Service

The rule management service provides browsing and directory functions for users to browse the contents of the Rule Warehouse and to retrieve events, rules, and catalog information associated with events and rules (e.g., the types of rules, the sources of the rules, the authorization information, etc.). This service is necessary to support an effective sharing of rules. Before a user can make a proper request to export rules from the Rule Warehouse into his/her rule system, he/she needs to know what rules are available in the Rule Warehouse and how they may meet his/her needs. A browsing facility is important for this purpose. A user may browse the warehouse before issuing a retrieval request for rules through RWI. The retrieval request is forwarded to the Rule Warehouse Manager (RWM), which performs the retrieval operation and presents the result to the user.

Additionally, RWM provides functions to define new events and rules and to edit and delete existing events and rules. For the convenience of human users, business events and rules need to be specified at a high-level of abstraction, using an easy-to-understand representation, and a user-friendly graphical interface. Rule definition and editing can be performed using the GUI facility provided by RWI. After receiving new rules or modified rules, RWI will dispatch them to RWM, which will activate the Rule Verifier to verify if the new or modified rules will introduce rule anomalies to the entire rule base. If no anomaly is found, RWM will deposit these rules into RW.

3.2.3. Collaborative Problem Solving Service

In our work on the automated negotiation server, we have demonstrated the collaboration between a constraint satisfaction processor and an action-oriented rule engine in conducting business negotiations. In this work on a Rule Warehouse System, we focus on the collaboration between a deductive rule engine and an action-oriented rule engine to do collaborative problem solving.

The Deductive Rule Engine (DRE) in the Rule Warehouse System is enhanced to combine the basic deduction capability of a traditional forward/backward chaining rule engine with the ability to raise exceptions during rule execution and to post "helper" events to the action-oriented rule engine. Currently, we recognize two general types of exceptions: rule conflict (i.e., multiple rules derive different values for the same attribute), and an inference process cannot proceed due to some missing rules. In either case, the action-oriented rule engine is consulted to see if there are rules that can resolve the conflict or generate or obtain the needed information for DRE to proceed.

The Action-oriented Rule Engine (ARE) is based on the general event-condition-action paradigm. It handles the execution of action-oriented rules. ARE responds to both asynchronous and synchronous events. If the triggering event is asynchronous, a separate thread is spawn and the control is returned immediately to the program that posted the event. If the event is synchronous, then the program is "blocked". In either case, the triggering event will trigger the processing of a linear, tree, or network structure of rules. The rules are condition-action-alternative-action rules. The semantics is "when triggered, check the condition. If the condition is true, then perform the operations specified as the action. Otherwise, perform the operations specified as the alternative action." If the triggering event is synchronous, control is returned to the waiting program when the rule processing is completed.

For collaborative problem solving, a user's request is sent to the Rule Warehouse Interface in the form of a query, either through its API or GUI. The query is directed to DRE, which starts the problem solving process. There are several possibilities in this process:

1. The query can be answered based on the constraint-oriented rules stored in the Rule Warehouse. In this case, DRE will produce the answer.
2. The query can be answered based on the constraint-oriented rules and DRE is able to produce the answer. However, some additional actions are required during the problem solving process (e.g., to enforce some security or integrity constraints). In this case, DRE would post events to trigger action-oriented rules managed by ARE.
3. The query cannot be answered by DRE due to incomplete data or missing rules. In this case, synchronous events are posted by DRE to trigger rules managed by ARE. Some ARE rules may be able to provide DRE the needed data for it to continue its deduction process. If ARE can, DRE's deduction process will be continued. Otherwise, DRE will ask the user to provide the additional information through RWI. If additional information can be obtained from the user, the DRE will continue with its deduction process. If not, the problem-solving process terminates without producing an answer.

3.2.4. Constraint Satisfaction Processing Service

Any object of interest in business such as a supplier, a retailer, a product, etc., can be described in terms of a set of descriptive attributes, attribute constraints, and inter-attribute constraints. If the constraints of the objects that are relevant to a virtual enterprise have been imported into the Rule Warehouse, a Constraint Satisfaction Processor (CSP) can be developed to provide the constraint satisfaction processing service. This service would compare a given set of constraints against a selected set of constraints stored in the Rule Warehouse to determine if these two sets contain conflicting constraint specifications. The input set may represent the specification of a product and the selected set may represent the capability and constraints of a potential supplier of the product. If conflicts in constraint have been identified by CSP, a report on the types of conflicts, the attributes that conflicted, etc., can be provided by CSP as the output. If no conflict were found in these two sets, CSP would report the case also. Depending on the expressive power of a constraint specification language, if the language allows the use of enumeration or range to specify allowable values for an attribute, multiple combinations of attribute values specified in the selected set may satisfy those of the input set. In that case, these combinations of attribute values can be returned.

A possible application of the CSP service is in automated negotiation demonstrated by UF's negotiation server. In that system, a buyer's negotiation proposal is specified as a set of constraints associated with the attributes of a product. The constraint specification is received by a supplier's negotiation server and processed against the pre-registered constraints of a supplier to determine if the attribute and inter-attribute constraints of the two specifications conflict with one another. Another application is in supplier selection. In this case, a buyer's constraint specification would be processed against the constraint specification of every potential supplier. The results of the comparison can then be ranked according to how well they match with the buyer's specification. In our work, we plan to extend the constraint satisfaction processor used in the negotiation server to provide the CSP service.

3.2.5. Event-Trigger-Rule (ETR) Service

Events allow loosely coupled systems that are contributed by different organizations of a virtual enterprise to inter-operate and collaborate with minimal dependency using the concepts and techniques of event publishing, subscription, and notification. Rules allow non-trivial interoperation and collaboration to be specified at a high level instead of coding the intended control and logic. Triggers provide a very flexible way of linking events with rules to capture the dynamic properties of e-business.

The ETR service can be a stand-alone service, which will respond to an event posted by an ***external*** source. Upon receiving an event notification, the Rule Warehouse Interface (RWI) will forward the request to the Action-oriented Rule Engine (ARE). ARE will first perform event history processing, and then triggers the associated rules to perform some actions. These rules can be either business rules that have been collected from different partners or enterprise-wide business rules. Note that in processing such action-oriented rules, a query may be sent to the Deductive Rule Engine to obtain the answer to a problem that cannot be solved by the action-oriented rules alone (e.g.,

verifying the condition part of an action-oriented rule may call for an interaction with the Deductive Rule Engine).

As described in the previous section, the ETR service can also be used by the Deductive Rule Engine (DRE) in collaborative problem solving. In this case, a user poses a query to the Rule Warehouse System, which cannot be answered by the DRE alone due to incomplete data or missing rules. One or more synchronous events are posted by the DRE to the ARE to trigger rules that perform suitable actions to obtain the required data. In Section 5, we will describe how the ETR service is used to support the collaborative problem solving service of the Rule Warehouse. In particular, how it is used in Scenario X of the EECOMS's December 2000 technical assessment will be described.

3.3. An Alternative Rule Warehouse System Architecture

Shown in Figure 3.3 is a centralized RWS. All the rules are imported, verified, and stored in a centralized RW. The service requester interacts with RWS by calling its APIs or accessing the Web-based interface directly. RWS provides the functions as described in previous sections. Our demo in the EECOMS's December 2000 meeting, to be described in Section 5, is an application of a centralized Rule Warehouse in collaborative problem solving.

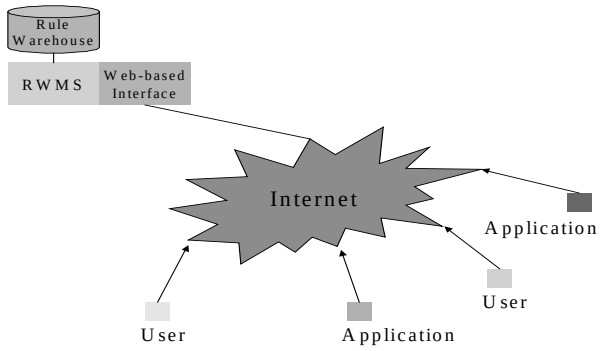

Figure 3.3 Centralized Rule Warehouse System Architecture

In order to achieve scalability and high performance, we envision that a global warehouse system can be constructed using a network of geographically distributed Rule Warehouse Systems. Each "local" Rule Warehouse System is a replication of the Rule Warehouse Management System (RWMS). They are installed at some selected Web sites and cooperate with one another to provide the Rule Warehouse services. Communications among Rule Warehouse Management Systems and between these systems and their users can be through the existing Web infrastructure.

Figure 3.4 shows a Web-based, distributed Rule Warehouse infrastructure. In this figure, a network of geographically distributed Rule Warehouses and their corresponding RWMSs are shown. We assume that the warehouses serve a virtual enterprise (e.g., companies that form a supply chain) and together contain the integrated knowledge of the

entire enterprise. Each company registers its action-oriented rules with a RWMS of its choice (presumably the closest one in the network) and deposits the rules in the corresponding local Rule Warehouse. Action-oriented rules are distributed but not replicated because they can be triggered by events by event notifications over the Internet. However, constraint-oriented rules of the virtual enterprise are replicated in all the local warehouses so that they can be used to support problem solving and constraint satisfaction processing in each RWMS without having to access remote constraint-oriented rules.

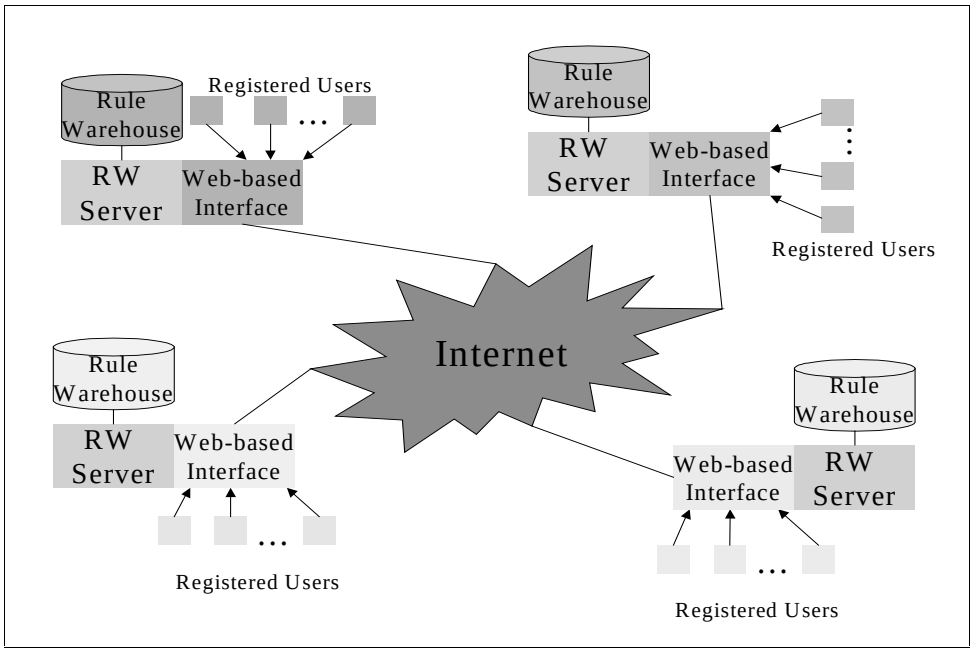

Figure 3.4 Network of geographically distributed Rule Warehouses

The distribution and replication of business rules is transparent to the users of the distributed Rule Warehouse System. From the user's point of view, this architecture is logically centralized but physically distributed. For processing action-oriented rules in this architecture, a user or application would post an event. Using an event service, an event notification would reach all the subscribers of the event on the Internet. The notification would trigger the processing of those action-oriented rules that are associated with the event and stored at different sites. For the Problem Solving Service and the Constraint Satisfaction Processing Service, a request is sent to a local Rule Warehouse System. Constraint-oriented rules that have been replicated and stored locally are sufficient to generate a response.

3.4. Section Summary

In this section, we have described the services provided by a Rule Warehouse System. We presented the design of a Rule Warehouse System and discussed two architectural alternatives for its implementation. We also outlined the key technologies required to support the Rule Warehouse services.

In the next three sections, we will provide some details on those technologies that are the focuses of our project. In Section 4, an ***Active Object Model*** (AOM) is presented. This model is used to provide a common representation for objects, events, and rules imported into the Rule Warehouse. The common representation is needed for the Rule Warehouse Management System to perform rule verification. AOM is an object model capable of modeling objects in terms of not only attributes and methods (just like the traditional object model) but also events, constraints, rules, and triggers. In Section 5, we will describe how an enhanced Deductive Rule Engine (DRE) and an Action-oriented Rule Engine (ARE) in the Rule Warehouse System can be integrated to perform collaborative problem solving. To demonstrate this integrated approach to problem solving, we have integrated the Common Rule System (a DRE developed by the IBM Watson group) and the ETR Server (an ARE developed by the University of Florida group) to perform collaborative problem solving in the context of EECOMS's Scenario X. In Section 6, we will present our work on rule verification. Verification methods and algorithms for detecting inconsistency, redundancy, and non-termination anomalies in the Rule Warehouse will be presented.

4. Active Object Model: A Common Knowledge Representation

The distributed object technology exemplified by OMG's CORBA, Microsoft's COM and DCOM, and Java's RMI is very powerful for modeling all things of interest to companies that form a virtual enterprise as business objects. However, the underlying object model of this technology only captures the structural properties of a business object in terms of attributes/properties and its behavioral properties in terms of methods. It is not able to capture business events and business rules associated with individual business objects nor events and rules associated with the interrelationships among business objects. For a Rule Warehouse System, we extend the traditional object model into an Active Object Model (AOM). Like traditional object models, AOM can be used to define individual business objects in terms of attributes and methods. In addition, events, rules and triggers applicable to a number of business objects can also be defined. Both constraint-oriented and action-oriented rules can be explicitly specified. The model is **active** in the sense that rules, which capture business policies, regulations, constraints, strategies, etc., can be automatically triggered to perform some meaningful operations upon the occurrences of events. The object model provides a common knowledge representation for all business objects imported into the warehouse. Such a common representation is needed to allow the Rule Warehouse Management System to do rule verifications.

As described in Section 2, The Meta Data Coalition's Business Rule Model (BRM) classifies business rules into ***term rule***, ***fact rule***, ***action rule***, and ***inference rule***. This classification is consistent with that of GUIDE. In that section, we also pointed out that different types of constraints can be expressed in the form of logic rules (P→Q). In AOM, we divide rule specifications into two parts: a constraint specification to capture term rules, fact rules and inference rules associated with business objects and an ETR specification to capture action rules.

4.1. Active Object Model

Figure 4.1 gives an overview of the modeling constructs of AOM. The basic building block of AOM is object class. An AOM ***class*** specification consists of the conventional object specification of *attributes* and *methods*. Also included in an AOM class specification are ***class-level knowledge specification (CKS)*** and ***instance-level knowledge specification (IKS)***. Class-level knowledge specification specifies the common knowledge that every instance of that class shares. Instance-level knowledge specification specifies the knowledge that applies to a particular object instance. Knowledge specification is optional and consists of *constraint-oriented* and *action-oriented KS*. Constraints specified in a class can be *attribute constraints* (constraints on individual attributes) or *inter-attribute constraints* (constraints among multiple attributes). Action-oriented KS consists of *events*, *action-oriented rules*, and *triggers* that relate events to rules.

Classes are contained in a schema. An AOM ***schema*** contains a set of *classes*, other (sub-) *schemas*, and ***schema-level knowledge specification (SKS)***. For example, a schema can be used to capture the structural and behavioral properties, and knowledge

specifications of a component in a component architecture. Thus, sub-schemas represent sub-components of a component. AOM allows the nesting of schemas to any number of levels. *Schema-level knowledge specification* consists of *inter-class constraints* or *inter-class action-oriented rules*.

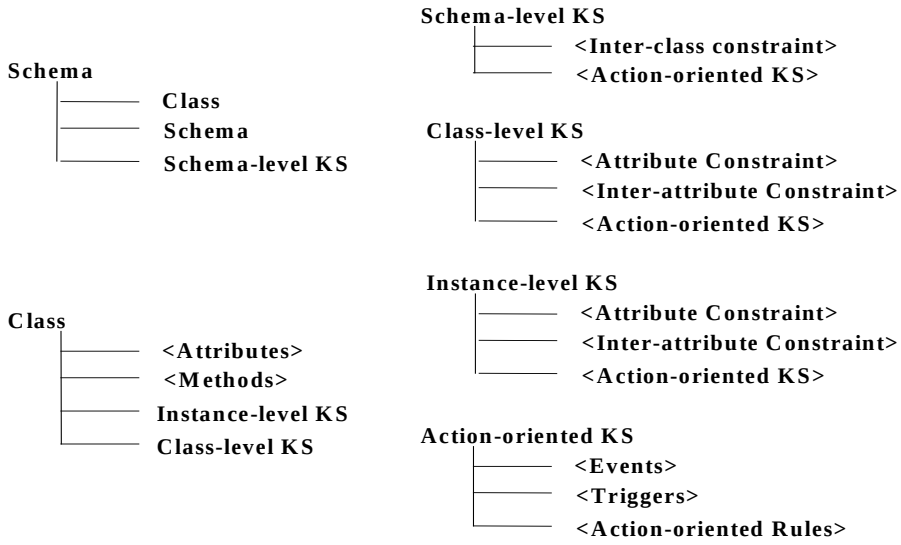

Figure 4.1 Constructs of the Active Object Model

Inheritance in AOM is defined in the same way as inheritance defined in the traditional object model. Thus, in addition to attribute and method inheritance, objects of a subclass can inherit the knowledge specification(s) of its superclass(es). For example, if class *c* is a subclass of *sc*, both belonging to schema *s*, then an object instance *i* of class *c* will have the following knowledge specification.

- The instance-level knowledge specification of instance *i*.
- The class-level knowledge specification of class *c*.
- The schema-level knowledge specification of schema *s* and its upper level schemas if this knowledge specification is related to class *c*.
- The public class-level knowledge specification inherited from the superclass *sc* and *sc*'s superclasses.

More detailed description for AOM can be found in [LEE00]. In the remainder of this section, we will describe in more detail the knowledge specification part of AOM.

4.2. Constraints

Constraints in AOM can be categorized into *Attribute Constraint (AC)*, *Inter-Attribute Constraint (IAC)* and *Inter-Class Constraint (ICC)*. The first two are part of a class-level knowledge specification, whereas constraints of the third type are given in the schema-level knowledge specification.

The syntax for constraints is:

Constraint name: Antecedence(P) → Consequence(Q)

The semantics of a constraint is "if P is true, Q is true," where P and Q are logical expressions. Unlike the logic expressions used in expert systems, P and Q in the AOM may include method invocations. Sometimes constraints can be unconditional, i.e., P is always "true." For an unconditional constraint, the Consequence can be specified without the Antecedence. The syntax is:

Constraint name: Consequence(Q)

4.2.1. Attribute Constraints

An attribute constraint specifies the legitimate values of an attribute of a class or of an object instance. Attribute constraints can have the following three forms:

- A simple predicate. For example, in an *Order* object, if attribute *delivery_day* has to be greater than 2, then the predicate *(delivery_day > 2)* is used to specify the constraint.
- A complex logical expression with AND, OR and NOT operators. For example, if we want to specify the constraint that delivery_day must be an even number and be less than 10, the logical expression ((delivery_day % 2 == 0) && (delivery < 10)) can be used. Attribute constraints are usually unconditional. However, in some cases, they can be conditional. For example, a business rule may say, "if the delivery_day is less than 10, it has to be a multiple of 2." We can specify this rule as "(delivery_day < 10) → (delivery_day % 2 == 0)", which is shown as constraint C2 in Figure 4.2.

```
Class RequestForQuote{
    …
    int price;
    int delivery_day;
    String payment_method;

    Attribute Constraint:
        C1: price = RANGE [300, 400];
        C2: (delivery_day < 10) → ( delivery_day % 2 == 0);
        C3: payment_method = ENUMERATION ["credit card", "cash"];
}
```

Figure 4.2 Attribute Constraints

- Two keywords, ENUMERATION and RANGE, can be used in attribute constraint specifications. ENUMERATION lists all possible values of an attribute and RANGE gives the range of values that an attribute may have. For example, a business rule may specify that the payment method can be either credit card or cash (i.e., payment_method = ENUMERATION ["credit card", "cash"] shown as C3 above).

Another business rule may state that price must be between $300 and $400 (i..e., price=RANGE[300, 400]). In a range specification, '[' or ']' can be used to mean a closed scope and '(' or ')' means an open scope.

4.2.2. Inter-attribute Constraints

An inter-attribute constraint describes the relationship between two or more attributes. An example constraint of this type is "model == Honda_Accord && partname == DOOR → damage_probability = 0.2." It specifies that the damage probability of the door of a Honda Accord is 0.2 (based on some existing statistics), if this type of car is involved in an accident.

The attributes referenced in an inter-attribute constraint specification can be of primitive data type or object reference. Attributes in the above example are of primitive data types. The following example in Figure 4.3 shows an attribute with an object reference.

```
Class Employee{
    …
    int salary;
    Employee manager;
    ….
    Inter-Attribute Constraint:
    C1: salary < manager.salary;
}
```

Figure 4.3 Inter-Attribute Constraints

In the class definition for class Employee, we use "salary < manager.salary" to specify the business rule that every employee's salary has to be lower than his/her manager's salary.

4.2.3. Inter-class Constraints

An inter-class constraint specifies the relationship between the attributes of two or more classes or the existential relationship between objects in some classes. For example, in Figure 4.4, the inter-class constraint "Class1.A1+Class2.B1 > 100 → Class1.A2 + Class2.B2 < 100" specifies that if the sum of attribute A1 of Class1 and attribute B1 of Class2 is greater than 100, then the sum of A2 of Class1 and B2 of Class2 has to be less than 100. As shown in the sample schema given below, inter-class constraints are specified at the schema level, outside of a class definition because multiple classes in a schema are involved.

Existential relationship between objects can also be specified as an inter-class constraint. For example, a business rule specifies, "Whenever there is an *Order* object for a buyer, there must be a *Credit* object to describe the credit of buyer". In AOM, the above constraint will be specified as an inter-class constraint using the keyword *EXIST*:

EXIST Order → (EXIST Credit ^ (Credit.buyerName = Order.buyerName))

```
Schema Sample {
    Class Class1{
        …
        int A1;
        int A2;
        ….
    }
    Class Class2{
        …
        int B1;
        int B2;
        ….
    }
    Inter-class Constraint:
     Class1.A1+Class2.B1 > 100 → Class1.A2 + Class2.B2 < 100;
}
```

Figure 4.4 Inter-Class Constraints

4.3. Action-oriented Knowledge

Action-oriented knowledge can also be specified at the instance level, class level, or schema level. Action-oriented knowledge specification consists of definitions of *events*, *action-oriented rules*, and *triggers.* In this section, we will give an overview of the action-oriented knowledge specification in AOM. Detailed specification can be found in [LEE00].

4.3.1. Events

AOM distinguishes two types of events: events associated with methods and explicitly posted events. A *method-associated event* is related to a method execution. Depending on the "coupling mode," such an event can be posted "before" or "after" a method is executed. Shown in Figure 4.5(a) is a definition of the class *Supplier*. Also shown in Figure 4.5(b) is the action-oriented knowledge specification associated with that class. The event *OrderShipped* is an example of a method-associated event (i.e., type "method") defined in the class *Supplier*, which is in the schema *SCSchema*. Method-associated events are posted synchronously. Thus, trigger and rule are processed and executed in the same "thread" as the method execution. In this example, the event *OrderShipped* is posted right after the method *shipOrder* is executed (i.e., the coupling mode is "after"). After the rules are triggered (if any), the control is returned to the program which made the call to the method *shipOrder*.

An event can have parameters. For method-associated events, the parameters are the parameters defined for the called method. When an event is posted, parameter values will be passed to rules that are triggered by the event. This allows data from the called method to be passed for use in rule processing. Also, an event that is posted

synchronously needs a return type. For a method-associated event, the return type is the return type of the called method.

```
Class Supplier
{
      //Attributes…
          int credit;
          boolean platinum;

          //Methods …
          public void shipOrder(int orderID, …) {
            // …
          }

          public void specialService(string orderID, string type, int flag) {
            //…
          }
}
```

Figure 4.5(a) Action-oriented Knowledge Example: class definition

An *explicitly posted event* can be defined at the class level or schema level. The event *OrderException* in Figure 4.5(b) is an example of an explicitly posted event (i.e., type "explicit") defined in the class *Supplier* of the schema *SCSchema*. The posting of an event would create an instance of the event type. An explicit event can be posted synchronously or asynchronously. If the event is posted asynchronously, a separate thread is created and the posting program will not wait for the return of the control. Also, data is not expected to return and the return type is ignored. If it is posted synchronously, the posting program is "blocked" to wait for the return of the control and data (if any). The return type is defined in the event specification.

An explicitly posted event can also have parameters, which are defined in the event specification. Parameters for this type of events provide a way for a program to pass data to rules for use in rule processing. When an event is posted, parameter values will be passed to rules that are triggered by the event.

4.3.2. Action-oriented Rules

An action-oriented rule in AOM has a condition-action-alternativeAction (CAA) structure. When a CAA rule is triggered, the expression specified in the *CONDITION* clause is evaluated. If it evaluates to true, the operations in the *ACTION* clause are executed. Otherwise, the operations in the *ALTACTION* clause are executed. AOM allows a "guarded expression" to be specified in the *CONDITION* clause. A guarded expression is a sequence of logical expressions as "guards" to a final expression. If any one of the guards is evaluated to False, the rest of the rule is skipped (i.e., not applicable). If all the guards are evaluated to true and the final expression is also true, the *ACTION* clause is executed. Otherwise, the *ALTACTION* clause is executed. In a rule specification, the CONDITION clause can be omitted. In that case, the ACTION clause

must be given and will be unconditionally processed. If the CONDITION clause is given, either the ACTION clause or the ALTACTION clause may be omitted, but not both.

IN	SCSchema::Supplier
EVENT	OrderShipped
DESCRIPTION	The credit of the customer is modified.
TYPE	METHOD
COUPLING MODE	AFTER
OPERATION	shipOrder //Associated method name.
IN	SCSchema::Supplier
EVENT	OrderException (string orderID, string exceptionType)
DESCRIPTION	An exception has occurred for an order.
TYPE	EXPLICIT
IN	SCSchema::Supplier
RULE	DowngradeCustomer
DESCRIPTION	Downgrade a platinum customer.
CONDITION	(platinum == true) && (credit < 0);
ACTION	{platinum = false;}
IN	SCSchema::Supplier
RULE	OrderExceptionRule (string orderID, string type)
DESCRIPTION	Handles order exceptions.
CONDITION	(type == 's') && (platinum == true);
ACTION	{specialService(string orderID, string type,1); }
ALTACTION	{specialService(string orderID, string type,2); }
IN	SCSchema::Supplier
TRIGGER	OrderExceptionTrigger (string orderID, string type)
TRIGGEREVENT	OrderException
EVENTHISTORY	OrderShipped AND OrderException
RULESTRUC	DowngradeCustomer > OrderExceptionRule

CAA rules provide a very general way for specifying conditional enforcement of integrity and security constraints, business policies and regulations that are relevant to the operation of a real or virtual enterprise. Each CAA rule represents a small granule of control and logic. A number of these rules, when executed in a certain order or structure, can represent a large granule of control and logic needed to enforce a composite business rule.

Figure 4.5(b) shows two examples of action-oriented rules, *DowngradeCustomer* and *OrderExceptionRule*. *DowngradeCustomer* specifies that if the customer is a platinum member and its credit is less than 0, he/she will be downgraded to a non-platinum member. *OrderExceptionRule* handles an order exception by calling the specialService method. Depending on the type of exception and the platinum membership, different parameter values are sent to this method by this rule. The ACTION and ALTACTION clauses specify the operations that should be carried out, which can be assignments to

some object attributes and/or method invocations. These clauses may also contain statements that post events to trigger other rules.

The above described the basic capabilities of an AOM rule. For details on some other capabilities (e.g., guarded condition, rule variables, exception handling, rule return type, etc.), interested readers should consult the reference [LEE00].

4.3.3. Triggers

Triggers tie events to rules. A trigger specifies an *event structure* and a *rule structure*. An event structure has two parts: TRIGGEREVENT and EVENTHISTORY. The TRIGGEREVENT part specifies a number of events called triggering events, each of which, when posted, would trigger the evaluation of the event history specified in the EVENTHISTORY part. If the event history is evaluated to true, the structure of rules specified in the RULESTRUC clause is processed. Otherwise, the structure of rules will not be processed. EVENTHISTORY is used to specify the relationship between some events that have already occurred or posted. An event history expression given in this clause is called a "composite event" in the active database literature. It can contain a number of simple predicate expressions with logical or sequence operators. For example, in the trigger *OrderExceptionTrigger* given in Figure 4.5(b), the EVENTHISTORY expression specifies that both events, *OrderShipped* and *OrderException*, must have been posted if the rules specified in RULESTRUC are to be fired. An example of an EVENTHISTORY expression that contains sequence operator is (E3 > E2 > E11), which specifies that these three events should have occurred in the given sequence.

The TRIGGEREVENT part is purposely kept very simple. Only the occurrence of any one of the triggering events would trigger the evaluation of a more complex event history expression. The separation of TRIGGEREVENT and EVENTHISTORY allows more explicit specification of what events would trigger the evaluation of event history and rules. This is different from the event specification of some existing ECA rule systems in which, when a composite event is specified, all the events mentioned in the composite event implicitly become the triggering events. In some applications, one may want to specify that only the posting of E2 should trigger the evaluation of "E1 and E2 but not E3". Note, in Figure 4.5(b), although events *OrderShipped* and *OrderException* are defined in the EVENTHISTORY, the event *OrderShipped* is not a triggering event. The posting of *OrderShipped* will not trigger the evaluation of the event history. Only the posting of *OrderException* would trigger the evaluation. If the TRIGGEREVENT is omitted, the default mode is ORing all of the events in the EVENTHISTORY.

The RULESTRUC clause of a trigger specification allows a structure of CAA rules to be triggered when the event specification is satisfied. It specifies the rule execution order and maps the parameters of the event to the individual rules.

There are two operators for specifying the execution order: the operator '>' is used to specify a sequential order of rule execution, and the operator ',' is used to specify a parallel execution. For example, the following expression means that rules R1, R2, R3, and R4 are to be executed sequentially following the specified order.

R1 > R2 > R3 > R4

The following example shows that rules R1, R2, R3, and R4 are to be executed in parallel:

(R1, R2, R3, R4)

In Figure 4.5, the RULESTRUC of *OrderExceptionTrigger* specifies that rule *DowngradeCustomer* is fired first, followed by the firing of rule *OrderExceptionRule*.

4.4. Section Summary

In this section, we have described the key features of the Action Object Model (AOM). We extended the traditional object model by incorporating a knowledge specification component. Knowledge specification can be given at the schema, class, and instance levels. Knowledge specification is optional and consists of two types: *constraint-oriented knowledge specification* and *action-oriented knowledge specification*. Constraints are used as a common representation for rules imported from logic-based and constraint-based systems. Action-oriented knowledge specification in AOM is used as a common representation for rules imported from sources that use action-oriented rules (e.g., ECA systems). Action-oriented knowledge specification consists of definitions of *events*, *action-oriented rules*, and *triggers.* By incorporating these two types of knowledge specifications in the traditional object model, we are able to have a common knowledge representation in the Rule Warehouse that enables the verification of business knowledge contributed by heterogeneous sources.

5. Collaborative Problem Solving Using the Rule Warehouse System

One main benefit of a Rule Warehouse System is to integrate the knowledge acquired from different sources to solve problems that cannot be solved by using the knowledge and rule engine of an individual source. Another benefit is that a better solution may be obtained if the integrated knowledge of a Rule Warehouse System is used. We believe that the collaborative problem solving service provided by the Rule Warehouse System is very useful for solving complex problems in supply chain management, such as problems associated with resource planning, procurement, shipping, customer relationship management, etc. As we have explained in Section 4, business rules can be broadly categorized into two categories: constraint-oriented rules and action-oriented rules. The first category contains rules that are commonly used in logic-based rule systems and production rule systems. It also contains constraints used in many constraint satisfaction processing systems. The second category contains rules that are used in ECA systems or their variations. In problem solving, inference capability based on constraint-oriented rules is very important. The use of action-oriented rules is also very important to direct a business enterprise to take proper actions upon the occurrences of various business events. Ideally, a single rule processing system can be developed for use in the Rule Warehouse System to do logical inferences as well as to perform event-triggered operations. Unfortunately, it is very difficult, if not impossible, to develop such a rule system because there is a paradigm mismatch between the two categories of rules and between their rule processing mechanisms. For that reason, we have chosen to use two separate rule engines, but to integrate their functionality, to do collaborative problem solving.

In this section, we will describe our work on the integration of IBM Watson's deductive rule engine with UF's action-oriented rule engine, so that knowledge in different forms (constraint-oriented rules and action-oriented rules) and from different sources can be used to solve problems collaboratively. Scenario X of EECOMS's December 2000 Technical Assessment is used as the application context for demonstrating the Rule Warehouse System's collaborative problem solving service. A portion of this section is available as an EECOMS design document [LIU00b].

5.1. Scenario X

The details of Scenario X for the EECOMS's December 2000 Technical Assessment can be found in the EECOMS's design document [EEC00]. It is briefly described here. In Scenario X, an automobile manufacturer is planning to start another production run for cars of a certain model. An automobile part that is required for making the cars must be purchased from its supplier (for simplicity but without losing the generality, one supplier is assumed) and its supplier in turn must obtain the material for making the part from its supplier. In Scenario X, as illustrated in Figure 5.1, the manufacturer's ERP system determines the number of rear doors that are needed to build the cars. The number has to be augmented by the door replacement demand estimated by the insurance company, which has the information about the number of insured cars of that particular model and

the likelihood that these cars will involved in accidents and the probability that rear doors will be damaged. The original demand plus the replacement demand equals the total number of rear doors that should be ordered from the first tier supplier, which in turn orders the sheet metal needed for making the doors from the second tier supplier. The scenario assumes that the manufacturer may also order sheet metals directly from the second tier supplier.

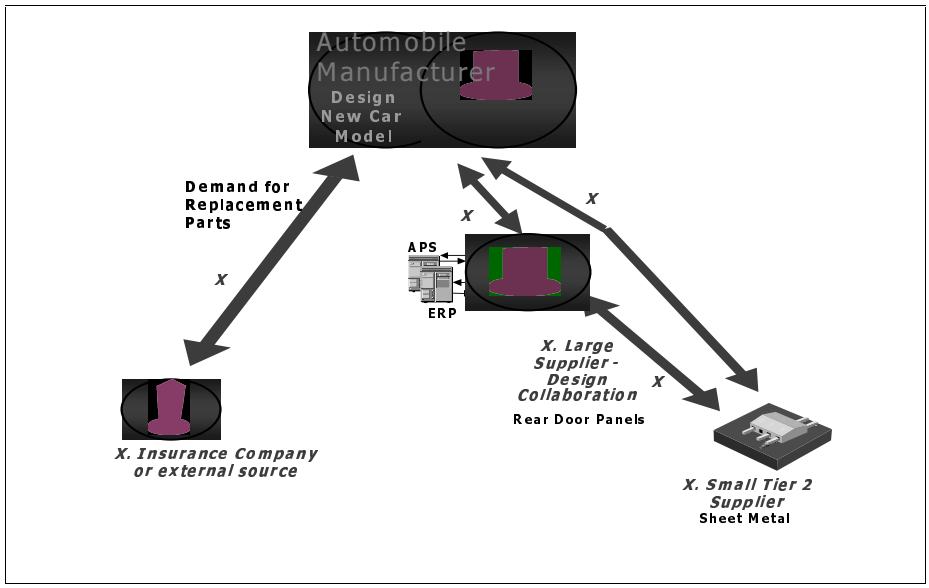

Figure 5.1. Scenario X

5.2. Interaction between a Deductive Rule Engine and an Action-oriented Rule Engine

In order to demonstrate the use of two types of rule engines to solve a problem collaboratively, we define some rules (see Figure 5.2) for demonstration purposes. We assume that these rules have been imported into the Rule Warehouse. Some of them are constraint-oriented rules (Rules 1–6) and two are action-oriented rules (Rule 7-8). The two collaborative rule engines use these rules to calculate the projected number of door replacements. They are designed to demonstrate three problem-solving cases described in the following three subsections. These cases illustrate problems that can not be handled by a single rule engine but can be solved collaboratively by a deductive engine, an action-oriented rule engine, and a requester (user or application system) who requests for information from the Rule Warehouse System.

5.2.1. Conflicting Knowledge

Conflicts may exist between the imported rules since these may represent different knowledge or opinions of different business organizations or different experts. For example, different insurance companies may derive different probability values for the door damage probability of a specific car model based on different statistical information they have gathered. When these derivation rules are imported into the Rule Warehouse,

they would and should produce different values when a requester asks the Rule Warehouse System for the door damage probability. However, in the existing expert systems, different conflict resolution schemes are used to determine which rule is to be fired first. In IBM's Common Rule System, priority is introduced to resolve this kind of conflict by selecting the rule with the highest priority. In both cases, only one rule is fired at a time. Thus, a single value is derived. Ideally, we would like to have a rule engine, which can apply multiple rules and derive alternative values for the same attribute. When this situation occurs, the rule engine would post an event to the action-oriented rule engine to see if some rule exists to resolve the conflict. For example, an action-oriented rule, representing some business policy, may take the average, the minimum or the maximum of the values that are returned. Or, it may ask the requester to resolve the conflict.

Manufacturer's planning department rules

Rule1: num_replacement = num_door * accident_probability * damage_probability
Rule2: num_door = OriginalDemand + existing_door

Insurance company rules

Rule3: if (Model == Honda_Accord && age == 17)
then accident_probability = 0.2
Rule4: if (Model == Honda_Accord && age == 18)
then accident_probability = 0.15
Rule5: if (Model == Honda_Accord && age != 17 && age != 18)
then accident_probability = 0.1
Rule6: if (Model == Honda_Accord && part == DOOR001)
then damage_probability = 0.2

Manufacturer's sales department

Rule7: Event: Query_Existing_Door
Condition: true
Action: call sales system to get the number of existing_doors

Rule 8: Event: Conflicting_Rule (*Vector LHSAttrList, Vector RHSAttrList, Vector RHSValueList*)
Condition: RHSAttrList contains attribute accident_probability
Action: return the average of the given values

Figure 5.2 Rules used for the demonstration

5.2.2. Don't Care Condition

In another situation, the information provided in a query to the Rule Warehouse System only partially matches with the precedence (left side) of all the rules. For example, Rules 3–5 in the Rule Warehouse derive different accident probabilities based on the models of car and different age ranges of the drivers. If the query to the Rule Warehouse System specifies the model of car without providing any age information, none of the rules can be applied in a traditional deductive rule system. No answer would be provided. In this situation, instead of producing no answer, a rule system should report

to the requester that the age information is missing. If the requester is able to provide the missing information, the inference process of the rule system can continue. Also, in some cases, the requester is only interested in the accident probability of a particular model of cars across all age ranges instead of a particular age range. In that case, the requester should be able to let the Rule Warehouse Management System know that the age attribute is a "don't care" condition. The Rule Warehouse System can then ignore the condition and continues its inferencing process by applying the partially matched rules.

5.2.3. Incomplete Knowledge

It is highly possible that a given request cannot be answered based on the rules stored in the Rule Warehouse. For example, Rule 1 states that the number of doors that are to be replaced for a particular model of car can be obtained by multiplying the number of doors per car by the number of accidents for that model and by the door damage probability in an accident. The rule can be applied to determine the number of doors to be replaced. However, if the Rule Warehouse does not have access to the number of accidents or the accident probability (or both), the rule is not applicable. The existing rule engines will fail in this situation without returning any useful state information about the failure. Nor will they produce any partial inference result so that either the query requester or some other rule system (e.g., an action-oriented rule system) can be called upon to provide the needed information. We believe that collaboration between the requester and different types of rule engines to handle different types of rules is important to solve complex problems.

5.3. Demonstration Overview

For the December 2000 technical assessment demonstration, UF's group and IBM Watson's Common Rules group have worked together to demonstrate the use of a simple Rule Warehouse System for rule sharing and collaborative problem solving. EECOMS's scenario X is used as the application context. The focuses of this demonstration are:

1. Integration of two types of rules: constraint-oriented rules and action-oriented rules.
2. Interaction of a Deductive Rule Engine (IBM Watson's Common Rule System, CRS) and an Action-oriented Rule Engine (UF's ETR Server) to collaboratively solve a problem.

By using IBM Watson's CRS as the deductive rule engine, we take advantage of the Courteous Logic Program (CLP) as the interlingua for importing constraint-oriented rules into the Rule Warehouse.

In Scenario X, we assume that the manufacturer and the insurance company have their own business rules to represent their knowledge, as shown in Figure 5.2. The constraint-oriented rules (Rules 1 through 6) have been imported and stored in the CRS. The action-oriented rules (Rules 7 and 8) are stored in the ETR Server. The manufacturer's planning department wants to determine the number of doors that need to be purchased from the supplier for a particular car model, Honda Accord, during the next production run. To determine the total number of doors to be ordered, the manufacturer

wants to take into consideration the number of door replacements that will be estimated as needed by the insurance company. It will post a query to the Rule Warehouse System to ask for the number of door replacements for Honda Accord. This demo is to show how to use dissimilar rules to derive such a value.

As shown in Figure 5.3, the Rule Warehouse Interface (RWI) accepts the user query. It dispatches the query to the Common Rules System (CRS). CRS would proceed to solve the problem using the rules stored in the CRS, and the result is returned to the requester via RWI. If CRS encounters a problem, an event is posted to the ETR Server to see if it can assist in solving the problem. If ETR can provide information that can help CRS, then CRS will continue its processing using the returned value. If ETR cannot provide any additional information, CRS will conclude that the problem cannot be solved. In this case, it will provide RWI with any partial solution it has and some information as to why the problem cannot be solved. RWI can then interact with the requester to obtain some additional information, if possible, to start another round of query processing.

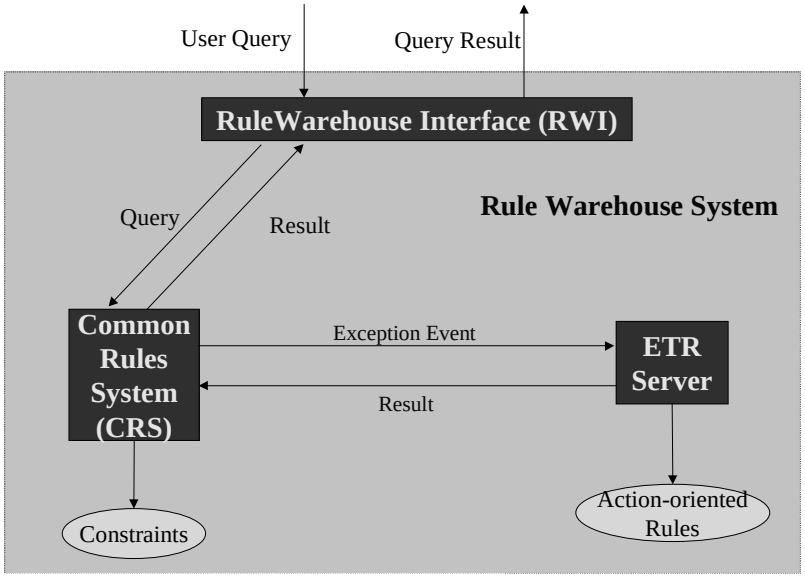

Figure 5.3 Demonstration Architecture

5.4. Interaction between RWI and CRS

Upon receiving a query from RWI, CRS will try to solve the problem. It will return one of the following three possible results.

- The problem can be fully solved and an answer is returned. This can be further classified into two cases. One is that the problem can be solved using only the rules in CRS. The other is that the problem can be solved by CRS with some help from the ETR Server (via events).

- The problem cannot be solved by CRS, even with the help of the ETR Server. This happens when there is no rule in ETR that can provide the information that CRS needs.
- CRS can provide a "partial" solution. In other words, CRS finds some rules that partially match with the data conditions provided by the requester through RWI. It cannot proceed unless it is given some additional information. For example, one or more attributes of the left-hand side expression require values. In this case, CRS needs to inform RWI and ask it to provide some additional information. RWI in turn asks the requester to provide the needed information. The requester can specify the values for the missing attributes or give a "don't care" specification. In the latter case CRS would ignore the attribute(s) in the predicate evaluation. After CRS receives the information for the unknown attribute(s), a new query session is created in CRS.

RWI passes the query as an object to CRS. Basically, a query will contain some known facts in the form of attributes and their values and ask for the values of some attribute(s). CRS will return a QueryResult object to RWI. The class definitions for these objects are given below.

```
Class Query {
      Hashtable attrValuePair;
}

Class QueryResult {
      int state; // state = 1: the query is solved. This is the result.
                        // state = 2: the query can be partial solved. This is the request
                                    //                    for further information.
                        // state = 3: the query can not be solved.
      Hashtable attrValuePair;
}
```

Some comments on the class definition are given below:

- In both classes (Query and QueryResult), the attribute named attrValuePair has Hashtable as its type. Both the input query and the returned query result are specified by a number of attribute-value pairs. The attribute names are used as the hash keys to the hash table. Both the attribute name and the value are of type String.
- In the input query, the known attributes and their attribute values are given. The unknown attributes will have "?" as their value. (The quotes are not part of the value String. We use quotes here for clarity.) If the requester does not care about an attribute being used in an inferencing process, the keyword "?IGNORE" is used as the value of that attribute.
- In QueryResult, if the query can only be partially solved, the contents of the hash table would contain some returned attribute-value pairs and some attributes having the value "?REQUIRED". If an attribute is not known and is not required, it would have the value "?".

In the context of Scenario X, a query given by a user to RWI will be processed in the following steps:

- RWI will call an API to send the query to CRS by passing a Query object with attribute-value pairs: (*model*, "Honda_Accord"), (*partname*, "DOOR001") and *(number_replacement*, "?").
- CRS will return a QueryResult object when it can not proceed because the *age* information is missing. In the returned object, the state is set to 2, the value of attribute *age* is set to "?REQUIRED".
- After RWI receives the QueryResult requesting a value for *age*. RWI will interact with the user and return the information to CRS by calling an API. The parameter is still a Query object; however, this time, the value for attribute *age* is "?IGNORE".
- Finally, in the scenario, CRS will solve the problem (with help from the ETR Server) and return to RWI a QueryResult object with a value for the attribute *number_replacement*.

5.5. Interaction between CRS and ETR

The ETR Server in the Rule Warehouse System is used to manage events and to trigger action-oriented rules. In this demonstration, the ETR server is used to extend CRS's problem solving capability in a flexible way. We demonstrate how to solve the two major problems for rule sharing defined in Section 5.2, namely, incomplete knowledge and conflicting knowledge. For the explanations of these two problems, the reader should refer back to Section 5.2. In the incomplete knowledge case, there is no rule defined for an attribute in CRS. A *missingRule* event is posted to ETR to seek help. In the conflicting knowledge case, there are multiple conflicting rules that are defined in CRS. A *conflictingRule* event is posted to ETR to resolve the conflict. The events posed by CRS to ETR are synchronous events, which are simply external function calls from CRS. The syntax and semantics of these events are explained below:

```
String missingRule (String attrName )
```

The *missingRule* event is posted when there is no rule defined in CRS for an attribute *attrName*. The parameter passed by the *missingRule* event is *attrName*. If there is an ETR rule that can produce a value for that attribute (e.g., invoke an "ERP" system to generate the value), then the value is returned to CRS. If there is no such rule defined in the ETR Server, then a failed status is returned to CRS to indicate that the problem cannot be solved by ETR either.

The rationale for this type of events is as follows. In a conventional rule system, if there is no rule that matches a pattern, then the processing would stop and the problem will not be solved. In this approach, rather than stopping, an event is posted to the ETR to see if it can provide some help. In a sense, the ETR Server produces a (virtual) rule "on demand" for CRS so that the processing can continue.

The second event is named *conflictingRule*. This event is posted when there are multiple rules that produce different values for the same attribute. The attribute is shown

in the right hand side of the rule (RHS) and all the conditions on the left hand side (LHS) are satisfied.

```
Vector conflictingRule( Vector LHSAttrList, Vector RHSAttrList, Vector RHSValueList )
   // Parameter LHSAttrList is the list of the attributes that appear in the LHS.
   // Parameter RHSAttrList is the list of the attributes that appear in the RHS.
   // Parameter RHSValueList is the list of the values for the attributes in the RHS.
   Each // element in the list is a list, which is the value list for that attribute.
   // For example, at the RHS, we have the following rules:
   //      Rule 1: A ^ B => C =3 ^ D = 5
   //      Rule 2: A ^ B => C = 4 ^ D = 6
   // Then LHSAttrList will be (A, B). RHSAttrList will be (C, D),
   // RHSValueList  will be ((3, 4), (5, 6)).
```

The rationale for this type of events is to enhance a rule system like CRS with a flexible, "user-defined" resolution mechanism. For example, in a rule system like Prolog, if there are multiple rules that match a request, then some default resolution scheme is used to select one of the rules to fire. In CRS, there is a priority scheme that can help resolve the conflict. However, if the conflicting rules have the same priority, there is some default conflict resolution scheme. Here, we provide an additional conflict resolution scheme, by posting a *conflictingRule* event. Any user-defined resolution scheme can be specified by one or more ETR rules. For example, for the rules given above, the triggered ETR rule can take the average of the values (3.5) for C and the maximum value (6) for D, or it can select the result of one of the rules, say, Rule 2 (i.e., C = 4 and D = 6).

5.6. Demonstration Interaction Diagram

The above interaction can be illustrated more clearly by using the interaction diagram shown in Figure 5.4:

Step 1: Requester poses the query to the Rule Warehouse Interface (RWI).
Step 2: RWI forwards the query to the Common Rules Systems (CRS).
Step 3: During processing, CRS cannot proceed because no rule exists for the attribute *existing_door*. CRS posts a *missingRule* event to request for help to get the value for attribute *existing_door*.
Step 4: ETR replies to CRS with the value for attribute *existing_door*.
Step 5: CRS continues with the processing. It encounters another problem and posts a *missingRule* event to request for help to get the value for attribute *age*.
Step 6: ETR replies to CRS with no answer for attribute *age*.
Step 7: CRS informs RWI that only a partial solution can be provided. The process cannot continue because the value for attribute *age* can not be determined.
Step 8: RWI prompts user to provide the information for *age*.

Step 9: User informs RWI that age is a *"don't care"*.
Step 10: RWI forwards the *"don't care"* information to CRS.
Step 11: CRS will process the query again. In this case, CRS posts a *conflictingRule* event to ETR because rules 3, 4 and 5 provide "*conflicting*" solutions for *accident_probability* (now that attribute *age* is a "don't care".)
Step 12: ETR replies to CRS with the result of *accident_probability* by applying the resolution scheme defined by an action-oriented rule.
Step 13: With that information, CRS can complete the solution to the query and return the solution to RWI.
Step 14: RWI forwards the final solution to the user/Calling Program.

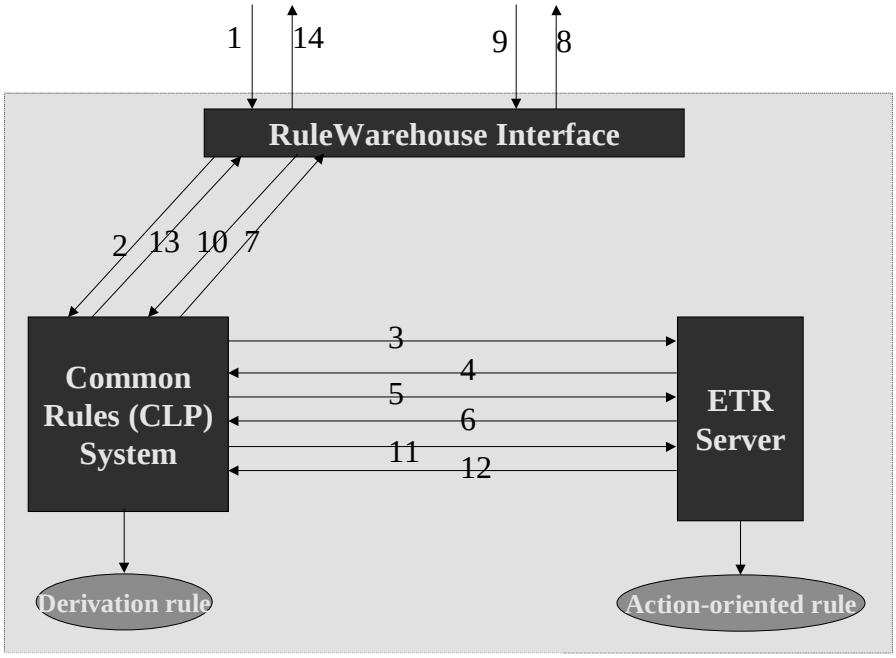

Figure 5.4 Demo Interaction Diagram

5.7. API Definition

5.7.1. API between CRS and RWI

The interactions between CRS and RWI are shown in the following APIs:

```
public class CRS {
    public CRS();
    public CRS(String ruleFile);
    public QueryResult query(Query query);
    public void setRuleFile(String ruleFile) throws RuleFileErrorException;
  }
```

RWI creates an instance of the CRS Class with an input file of rules. CRS then uses the rules from this file for subsequent query evaluation.

```
String ruleFile;
 CRS crs = new CRS(ruleFile);
```

Or

```
CRS crs = new CRS();
crs.setRuleFile(ruleFile);
```

The setRuleFile method is used when a new ruleFile is required to evaluate a query.

RWI executes a query via the following method call:

```
Query query = …;
 QueryResult crs.query(query);
```

The method *query* takes a *Query* object and returns a *QueryResult* object, which contains the result of the query evaluation. The Classes Query and *QueryResult* were defined in Section 5.3.

Should the condition of "*Conflicting rules*" or "*Missng rule*" occurs during query evaluation, CRS creates an instance of the RuleWarehouse.ETR.ETRWrapper Class and posts the corresponding synchronized event with the proper parameters, followed by a CRS PAUSE state until the necessary information is returned. CRS issues the following method call to post an event:

```
ETRWrapper ew = new ETRWrapper().
String etrResult = ew.postMissingRuleEvent(attName);
```

Or

```
Vector etrResult = ew.postConflictingRuleEvent(LHSAttrList,RHSAttrList, HSVAttrlist);
```

5.7.2. API between CRS and ETR

CRS needs to follow the following steps to interact with ETR:

- Create an instance of RuleWarehouse.ETR.ETRWrapper ew.
- Call ew.postMissingRuleEvent() or ew.postConflictingRule() to post the event with proper parameters.

Here is the definition of the class named ETRWrapper.

```
public class ETRWrapper
{
        public String  postMissingRuleEvent ( String attrName );
        public Vector postConflictingRule( Vector LHSAttrList, Vector RHSAttrList,
                                           Vector RHSValueList );
}
```

Section 6. Verification of Action-oriented Rules

In a Rule Warehouse System, rules from heterogeneous sources are imported into the Rule Warehouse to support knowledge sharing and business collaboration. As explained in Section 4, these rules can be broadly classified into two categories: logic (or constraint-oriented) rules and action-oriented rules. In the Rule Warehouse system, the different types of rules will be translated into the common AOM representation so that they can be *verified* along with existing rules in the Rule Warehouse.

Much has been done on verification of logic rules and there has been some work done on the verification of action-oriented rules. A survey of the existing works is given in Section 6.1. In Section 6.2, we will state the assumptions and basic definitions on which our rule verification work is based. Sections 6.3 and 6.4 will present our work on the detection of the three anomalies that may exist in rules stored in a Rule Warehouse: non-termination, inconsistency and redundancy. In Section 6.3, the termination problem for action-oriented rules is defined formally and an algorithm for detecting non-termination is given. The algorithm will make use of Triggering Graphs (TG), Activation Graphs (AG) and Deactivation Graphs (DG). In Section 6.4, the inconsistency and redundancy problems are defined formally and our approach and algorithm for detecting these anomalies are given. In Section 6.5, we will discuss the remaining problems of rule verification for a Rule Warehouse System. We will also present our approach to these problems and some preliminary results.

6.1. Related Work

Existing works on rule verification deal mostly with verification of logic rules. Early works [CRA87, NGU85, SUW82] detected simple manifestations of redundant rules, contradictory rules, and missing rules. The detection is based on rule connectivity and pair-wise checking, with time complexities of $O(n)$ and $O(n^2)$, respectively. Pioneering systems include the RCP [SUW82] and CHECK [NGU85]. More recent works [GIN88, PRE92, ROU88, STA87] extend the definitions of anomalies beyond simple pairs of rules. Redundancies and contradictions arising over chains of rules can be detected, as well as the detection of more subtle cases of missing rules. The time complexity for detection is usually $O(b^d)$, where $\boldsymbol{b}$ is the breadth of the rule base, defined as the average number of literal in-rule antecedents, and $\boldsymbol{d}$ is the depth of the rule base, defined as the average number of rules in an inference chain. There is no correlation between the number of rules of declarations in the system and the cost of verification [PRE94]. Three other rule base verification systems are worth noting: COVER [PRE92], PREPARE [ZHA94] and EHLPN [WU97]. In COVER, only relevant combinations of data items are considered. Smaller environments are tested first, and any larger environments that are subsumed by the smaller ones are not detected. PREPARE uses Petri-net models to study rule verification; however, it assumes that no variables appear in the rule bodies, and it does not deal with relationships involving negative information in rules. EHLPN uses an enhanced high-level Petri-net. It requires the closed-world assumption, conservation of known and unknown facts, and refraction.

Other works perform detection of anomalies based on the semantics of rules instead

of their syntactic representations alone; examples include [ROS97] and [WU93]. In [WU93], Wu and Su formally defined the problem of inconsistency and redundancy of a knowledge base based on rule semantics and related it to the concept of unsatisfiability in formal logic in order to make use of the solid theoretical foundation established in logic for rule verification. A unified framework was developed for both rule verification and rule refinement. Also, a reversed subsumption deletion strategy is used to improve the verification efficiency.

Some works have been done to address the termination and confluence problem of a rule set in active databases; however, these works are limited in that the conditions and actions of the rules are restricted to be database operations only. Termination means that the rule processing is guaranteed to terminate for any user-defined application [BAR98]. Confluence means that the execution of a rule set will reach the same final database states for an application regardless of the order of rule execution [BAR98]. [KAR94] reduces rules into term rewriting systems and applied known algorithms to attack the termination problem; however, the approach is very complicated even for a small rule set. [WEI95] presents techniques for termination analysis of rules with delta relations in the context of OSCAR, an object-oriented active database system. The concept of Triggering Graph is introduced in [BAR93] to detect the termination of a database rule set. The work is extended in [BAR94] to support both termination and confluence by limiting the rule set to Condition-Action rules. [AIK95] tries to determine the termination and confluence of a database production rule set statically for both CA and ECA rules. [BAR95] proposes a technique to deploy the complementary information provided by Triggering Graphs and Activation Graphs to analyze the termination of ECA rule sets; however, techniques for constructing the graphs are assumed. In [BAR00], an extended relational algebra is presented and applied by a "propagation" algorithm to form the graphs. [BAR98] combines the static analysis of a rule set and the detection of endless loops during rule processing at runtime.

Although some important research results have been achieved in the above works on the verification of database rules, all of them require that the conditions and actions of rules be database operations, i.e., insertion, update, deletion and selection. Moreover, most of the works handle CA rules only. Even in those that can handle ECA rules, either the solution is very restrictive (e.g., [BAR95] requires that there is no cycle in a Triggering Graph), or some limitations are put on the ECA rules to reduce them to CA rules (e.g., the Quasi-CA rules introduced in [BAR00]). Finally, none of the work addresses the invocation of methods in the condition and action of a rule.

6.2. Assumptions and Basic Definitions

Several types of anomalies have been identified in the existing works of the verification of logic rules in expert systems: inconsistency, redundancy, subsumption, unnecessary if, dead end, and unreachable rules [GON97]. In this work, we focus on the verification of knowledge specifications in a Rule Warehouse System. Consequently, some of the anomalies defined for expert systems in the literature are not applicable to this work. For example, anomalies such as dead end and unreachable rules can cause problems for expert systems; however, in a Rule Warehouse System, rules can be exported from the Rule Warehouse to a legacy rule system and be applied together with

some other rules. Thus, even if the rules are unreachable inside the Rule Warehouse, they may become reachable in conjunction with other rules in the legacy rule system. Consequently, from the Rule Warehouse System point of view, it is not regarded as an anomaly.

In the context of Rule Warehouse System, we will consider three types of anomalies: namely, non-termination, inconsistency and redundancy. Furthermore, since these anomalies have been extensively studied for logic rules, we will focus on these anomalies in the context of *action-oriented rules*. Before we formally define and present the algorithms for detecting these anomalies, we shall first give our assumptions and provide some basic definitions that will be used for the remainder of this section.

Simplifying assumptions

In order to make the verification of action-oriented rules a tractable problem, we will first make some simplifying assumptions about this type of rules. Then, in the next stage of research, we will relax some of these assumptions.

Recall from Section 4, the action-oriented rules defined in the AOM model are represented in the ETR (event-trigger-rule) constructs. The trigger specification has a set of triggering events, an optional event history specification, and a structure of triggered rules. For the current work on verification, we will assume the following:

- There is no event history; or, if one is provided in a trigger specification, the evaluation of the event history processing will always return true.
- There is a single rule in the rule structure.
- The triggered rules are CA (condition-action) rules, not CAA (condition-action-alternativeAction) rules. Note that a CAA rule can be expressed by two CA rules.

Based on the above assumptions, we have the following definitions for action-oriented rules.

Definitions for action-oriented rules:

Definition 6.2.1: A **rule set** *R* is a set of action-oriented rule specifications under consideration.

Definition 6.2.2: An **action-oriented rule** *r* (from here on, we shall call it **rule** for short) is a "condition-action" (CA) rule.

Definition 6.2.3: A rule *r* is **fired** if it is **activated** and **triggered.**

Definition 6.2.4: A rule *r* is **activated** if the condition part of the rule evaluates to true. If the condition evaluates to false, *r* is **deactivated**.

Definition 6.2.5: A **trigger** *t* specifies a **triggering event** *e* and the rule *r* it triggers. The rule *r* is **triggered** when the event *e* has been posted.

Rule execution model

When an event *e1* is posted, it will cause all the triggers which have *e1* as a triggering event to trigger the corresponding rules. Each triggered rule *r* is a CA rule. If the condition part of *r* is true (i.e., activated), then the action part of the rule is executed (i.e.,

fired). In the execution of the action part of a rule, two types of actions can have impact on rule verification:

- An action may post another event *e2*, initiating another rule triggering cycle.
- An action may change the state of an object, which may affect the condition part of another rule (i.e., activate or de-activate a rule).

Modeling the side-effects of methods

One important issue concerning the action-oriented rule verification is the invocation of methods in both the condition part and the action part of a rule. The execution of a method may have side effects that can cause rule execution anomalies. For example, events may be posted within a method, operations in a method may change the state of an object, or the result of an evaluation of the condition part of a rule may depend on the data read by a method invoked in the condition part. Thus, for rule verification purposes, it is important to capture the side effects of methods that are invoked in a rule:

Definition 6.2.6: The **read set** of *m* is the set of attributes that may be read by *m*, where *m* is a method.

Definition 6.2.7: The **write set** of *m* is the attributes whose values may be changed by *m*.

Definition 6.2.8: The **event set** of *m* is the events that may be posted in *m*.

Definition 6.2.9: A method *m1* can be specified to be **contradictory** to method *m2.* Logically, this is equivalent to *m1* = NOT(*m2*) and *m2* = NOT(*m1*).

Definition 6.2.10: A method *m1* can be specified to be **equivalent** to method *m2*. Logically, this is equivalent to *m1* = *m2*.

```
Example 6.1:
Class Accident_statistics{
    String model;
    int accident_probability DERIVED;
    Rule1 {
        Condition:  model = “Honda Accord”;
        Action:     accident_probability  = 0.2;
      }
    Rule2 {
        Condition:  model = “Honda Accord”;
            Action:         accident_probability  = 0.15;
    }
    Rule3 {
        Condition:  model = “Honda Accord”;
        Action:     accident_probability  = 0.1;
    }
}
```

Definitions for objects and object states:

Definition 6.2.11: The **property set** *P* of an object *O* is a set containing all the attributes defined in the class *c* to which *O* belongs. For example, in Example 6.1 shown below, the property set of the object *Oas* of class *Accident_statistics* is *P={model, accident_probability}*

Definition 6.2.12: An **object state** S_O of object *O* is an instantiation of the property set *P* of object *O*. An example object state of object *Oas* is *{"Honda Accord", 0.01}.*

Definition 6.2.13: An **object execution state** S_E is a pair (S_O, R_A), where S_O is the object state and $R_A \in R$ is a set of activated rules of S_O. An example object execution state is *({"Honda Accord", 0.01}, {R1, R2, R3}),* where *R1, R2* and *R3* are activated rules.

6.3. Non-Termination Detection

Action-oriented rules may interact in complex and sometimes unpredictable ways. In particular, there may be cycles in the rule set in which rules trigger and fire each other indefinitely, causing a non-termination problem [BAR98]. In this section, we will first give the definition of the non-termination anomaly in the context of an action-oriented rule set, followed by an algorithm to detect this type of anomaly.

Definition 6.3.1: An ***externally-generated event*** is an event posted by a source outside of a rule set. For example, an event can be posted by an application program or within the implementation of a method.

Definition 6.3.2: A ***rule-generated event*** is an event posted by rules in a rule set. For example, an event can be posted in the action part of a rule or in a method invoked in the condition or action part of a rule.

Definition 6.3.3: A **rule set *R*** for object *O* is **non-terminating** if there exists a rule $r \in R_A$ such that *r* is fired repeatedly and indefinitely with or without an externally generated event *e* being posted. $R_A \in R$ is the set of activated rules of S_E , and S_E is an object execution state of *O*.

Recall from Definitions 6.2.3 that a rule is fired only if it is triggered and activated (i.e., condition part of the rule is true). Thus, in order for a rule to be fired repeatedly and indefinitely after an event is posted from outside a rule set (e.g., from an application program), the rule must be triggered repeatedly and indefinitely by some action within the rule set. Furthermore, the rule must be activated when it is triggered.

Example 6.2:

```
Class A {
      //attribute and methods definitions…

      Event e1, e2;
      Trigger t1 {TriggeringEvent e1;  RuleStruct: r1;}
      Trigger t2{TriggeringEvent e2;   RuleStruct: r2;}

      Rule r1 {Condition: true; Action: post e2;}
      Rule r2 {Condition: true; Action: post e1;}
}
```

For example, the rule set in *Example 6.2* will not terminate. If *e1* is posted by an application, trigger *t1* will trigger the firing of rule *r1*, which posts event *e2*. Event e2 will cause trigger *t2* to fire rule *r2*, which in turn posts event *e1* to repeat the entire triggering process again. Thus, rules *r1* and *r2* will be fired repeated and indefinitely after *e1* is initially posted. This is also true if e*2* is posted by an application.

6.3.1. Algorithm for Non-Termination Detection

In this section, we will present an algorithm to determine whether a given action-oriented rule set can terminate. The algorithm is based on the detection of ***triggering cycles*** within a rule set. A triggering cycle may or may not cause a termination problem. Upon the detection of a triggering cycle, the algorithm determines under what conditions the trigger cycle may (but not necessarily) cause the non-termination anomaly.

[BAR95] and [BAR98] proposed the use of Triggering Graphs (TG) and Activation Graphs (AG) to detect the non-termination anomaly in active databases. We will extend the idea to support the detection of non-termination for general action-oriented rule sets. The basic idea is as follows. Triggering Graphs will be used to determine triggering cycles in a rule set. Upon the detection of a triggering cycle, we will use Activation Graphs and De-activation Graphs (DG) to determine if the triggering cycle may cause the non-termination problem. In particular, if there exists one rule in the triggering cycle whose condition is false (i.e., de-activated) when triggered, then the triggering cycle will be broken and thus will not cause the non-termination problem. The triggering cycle will cause the non-termination problem only if, for every rule in the cycle, its condition is true (i.e., activated) when triggered.

Definition 6.3.4: Let R be the rule set under consideration, the ***Triggering Graph*** (TG) of R is a directed graph $\{V, E\}$, where each node $v_i \in V$ corresponds to a rule $r_i \in R$. A directed edge $<r_j, r_k> \in E$ means that rule r_j generates an event that triggers rule r_k.

Definition 6.3.5: A **triggering cycle** TC is a sub-graph of TG, which forms a cycle in TG.

Definition 6.3.6: Let R be a rule set, the ***Activation Graph*** (AG) of R is a directed graph $\{V, E\}$, where each node $v_i \in V$ corresponds to a rule $r_i \in R$. A directed edge $<r_j, r_k> \in E$ means that the action of rule r_j makes the condition of rule r_k to be true.

Definition 6.3.7: Let R be a rule set, the ***Deactivation Graph*** (DG) of R is a directed graph $\{V, E\}$, where each node $v_i \in V$ corresponds to a rule $r_i \in R$. A directed edge $<r_j, r_k> \in E$ means that the action of rule r_j makes the condition of rule r_k to be false.

Example 6.3 shows several sample events, triggers, and rules. The corresponding TG, AG, and DG are shown in the Figure 6.1.

Example 6.3

Class Supplier{

```
    //Attributes… Methods ...
    int credit;
        boolean platinum;
        booolean discountable;
}
```

Knowledge Specification:

```
    EVENT                creditPenalty
    TYPE                 EXPLICIT;

    EVENT                untrustableSupplier
    TYPE                 EXPLICIT;

    EVENT                specialService
    TYPE                 EXPLICIT;

    RULE                 R1
    CONDITION            credit > 0
    ACTION               credit -= 10;

    RULE                 R2
        CONDITION        credit < 0
        ACTION           discountable=false; post untrustableSupplier event; …;

    RULE                 R3
          CONDITION      platinum == false;
          ACTION         …; Post specialService; …;

    RULE                 R4
          CONDITION      true
          ACTION         …;    Post creditPenalty; …;

    TRIGGER T1
    TRIGGEREVENT         creditPenalty
    RULESTRUC            R1

    TRIGGER T2
    TRIGGEREVENT         creditPenalty
    RULESTRUC            R2

    TRIGGER T3
    TRIGGEREVENT         untrustableSupplier
    RULESTRUC            R3

    TRIGGER T4
    TRIGGEREVENT         specialService
    RULESTRUC            R4
```

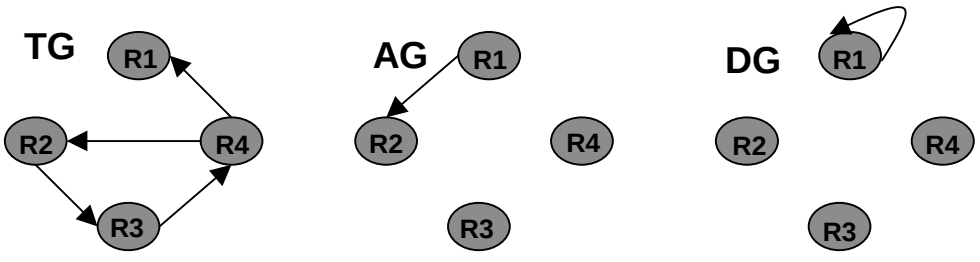

Figure 6.1 Examples of TG, AG and DG

Lemma 6.1: A rule set *R may not terminate* if there is a triggering cycle in the Triggering Graph. If there is no triggering cycle in the Triggering Graph, the rule set *R* will terminate.

[**Proof**] If there *is* a cycle in the TG, then the rule set *will not* terminate if the action part of every rule in the cycle is executed; i.e., each rule in the cycle is activated when it is triggered (and thus fired). If there exists one rule such that the condition is not true (deactivated) when it is triggered, then the triggering cycle will be broken and the rule set *will* terminate. Thus, a rule set *R may* not terminate if there is a triggering cycle in the Triggering Graph.

Next, we will show that if there is no cycle in the TG, then the rule set will terminate. Let us assume, on the contrary, that the rule set will not terminate even without a cycle in the TG. Let us consider a subgraph *SG* that is formed of the rules that are involved in the repeated firing. Note that the in-degree for each node in *SG* must be at least 1. Otherwise, any node with an in-degree 0 will not be fired again without another externally generated event being posted from outside the rule set. Let us start from any node *i* in *SG* and another node *j* such that there is an edge from *i* to *j* in *SG*. We trace backward against the direction of directed edges to a node *h*, which has an edge from *h* to node *i*. The node *h* cannot be node *j* (i.e., the node(s) that has been traversed before)*;* otherwise there is a cycle in *SG*. We repeat this process to reach a new node every time. Since the number of rules in the rule set is finite, the number of nodes in SG is also finite. Eventually, the last node *l* will be reached in this process. If rules in SG are repeatedly fired, there must a directed edge from some node *n to l*, where *n* is a node that we have considered before, thus forming a cycle in *SG*. This contradicts with the assumption that "there is no cycle in the TG."

Definition 6.3.8: A rule *r* ***is in*** a triggering cycle *TC* if it corresponds to a node in *TC*.

Definition 6.3.9: When a rule r_i is triggered before r_j in a triggering cycle, then we denote the order by $r_i < r_j$.

Lemma 6.2: A rule set will terminate if, for each triggering cycle in the TG,

- $\exists\ r_i, r_k$ in DG such that there is an edge from r_i to r_k and
- NOT $\exists\ r_j, r_k$ in AG such that there is an edge from r_j to r_k

where r_i, r_j, and r_k are in the triggering cycle and $r_i < r_j < r_k$.

In other words, a rule set that contains a triggering cycle will terminate if some rule (e.g., r_k) in the triggering cycle is deactivated by the action of another rule (r_i) in the cycle and no other rule (r_j) in the cycle activates it again after the deactivation.

[**Proof**] If there is an edge in DG from r_i to r_k, it means that the action of rule r_i will deactivate rule r_k. In this case, when rule r_k is triggered in the cycle, it will not fire, thus breaking the triggering cycle. However, the cycle will not be broken if r_k is re-activated by another rule r_j, fired after r_i but before r_k is triggered. But, if there is no such r_j in AG, r_k will not be re-activated. Thus, the rule set will terminate.

The example in Figure 6.2 illustrates Lemma 6.2. In the triggering cycle consisting of (R1, R2, R3), rule R2 is deactivated by rule R3, as shown in DG. AG shows that R2 is

activated by R1. Since R1 is triggered and fired after R3 in the cycle, its effect will overwrite the effect of R3. Thus, R2 is activated every time it is triggered. Consequently, the rule set will not terminate.

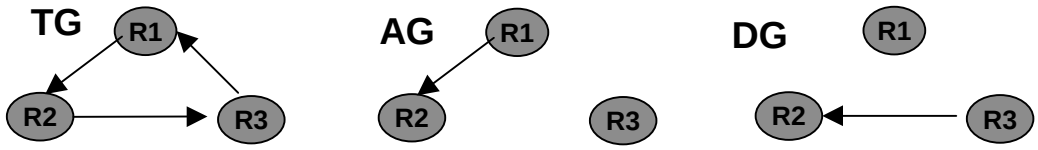

Figure 6.2 Examples for Lemma 6.2

Lemma 6.3: A rule set will terminate if, for each triggering cycle TC in a TG,

(1) ∃ a rule set S, which is a subset of the rules in TC, such that the conjunction of the conditions of the rules in S is unsatisfiable, and

(2) Each of the nodes that correspond to the rules in S has no incoming edge in AG.

[**Proof**] If Condition (1) is true, then at least one of the rules in S will not be activated (i.e., its condition is false) unless some other rule in S activates it (the negation of Condition (2)). For example, if the condition of r_i is "*A*" and the condition of r_j is "*NOT A*", then one of these rules will be deactivated during the triggering cycle if the value of *A* does not change. However, if some rule r_k changes the value of *A* to true before r_i is triggered and another rule r_m changes the value of *A* back to false before r_j is triggered, then the trigger cycle is not broken. Thus, Condition (2) is used to eliminate that situation. Condition (2) says that none of rules in S will be activated (or re-activated) by the action of any of the rules in the triggering cycle.

Theorem 6.1: A rule set is guaranteed to terminate if one of the following three conditions hold:

(1) There is no triggering cycle in TG of the rule set.

(2) If there are cycles in TG, for each triggering cycles in TG,

- ∃ r_i, r_k in DG such that there is an edge from r_i to r_k and
- NOT ∃ rj, rk in AG such that there is an edge from rj to rk

where ri, rj, and rk are in the triggering cycle and ri < rj < rk.

(3) If there are cycles in TG, for each triggering cycle TC in TG,

- ∃ a rule set S, which is a subset of the rules in TC, such that the conjunction of the conditions of the rules in S is unsatisfiable, and
- Each of the nodes that correspond to the rules in S has no incoming edge in AG.

[Proof]

Condition (1):We have shown in Lemma 6.1 that if there is no cycle in TG, then the rule set will terminate.

Condition (2):We have shown in Lemma 6.2 that under the conditions stated in Condition (2), the triggering cycle will be broken and the rule set will terminate.

Condition (3): We have shown in Lemma 6.3 that under the conditions stated in

Condition (3), the triggering cycle will be broken and the rule set will terminate.

Based on Theorem 6.1, an algorithm for non-termination detection in a rule set is given in Algorithm 6.1. In Step 1, the Triggering Graph for the rule set is formed based on the event posting relationships among the rules in the rule set under verification. The event set of the side effect description (see Definition 6.2.8) of each method invoked in the rules will also be used to determine the triggering relationships. In Step 2, all the triggering cycles in the TG are determined. If there is no cycle in the TG, the rule set will terminate and the algorithm will stop. Otherwise, it is possible that the rule set is non-terminating. The algorithm continues to Step 3 to form the AG and DG for the rule set. The read and write sets of the side effect description of each method invoked in the rules will also be used to determine the AG and DG. Step 4 tests termination conditions (2) and (3) presented in Theorem 6.1 to determine whether a particular triggering cycle will cause a problem. If neither condition can be satisfied, the algorithm will report that the rule set may not terminate and identify the specific triggering cycle. If all the triggering cycles can satisfy either condition, the algorithm will report that the rule set will terminate.

If the event posting relationship is known, the time complexity of forming TG in Step 1 is O(|u|+|v|), where |u| is the number of rules in the rule set and |v| is the number of edges in TG. In Step 2, the time complexity for the triggering cycle detection is O(|u|+|v|). If there are cycles in the rule set, we need to identify all the cycles. The time complexity for identifying cycles is $O(|u|^2)$[1]. The time complexity for both Step 3 and Step 4 is O(|u|+|v'|), where |v'| is the number of edges in AG or DG. Thus, the total complexity of this algorithm is O(|u|+|v|) if there is no cycle in TG, otherwise, $O(|u|^2)$.

Algorithm 6.1:

1. Form the TG based on the event posting relationships among the rules in the rule set under verification. The side-effect descriptions of the methods invoked within the rules will also be used to determine the relationships.
2. Determine all the triggering cycles in the TG. If there is no cycle, report that the rule set will terminate and exit. Otherwise, continue.
3. Form the AG and DG based on the relationships among the conditions and actions of the rules. Again, the side-effect descriptions of the methods invoked in the rules will also be used to determine the relationships.
4. For each cycle in the TG,
   a. Determine whether condition (2) or condition (3) in Theorem 6.1 is satisfied. If neither condition is satisfied, report that the rule set may not terminate because of this triggering cycle and quit.
   b. Continue to examine the next cycle.
5. Report that the rule set will terminate.

Let us apply the algorithm to the Example 6.3. From the TG, AG and DG shown in

[1] The time complexity of the identification is $O(|u|^2)$ because one rule may be involved in multiple cycles. The justification is straightforward.

Figure 6.1, we can see that there is a cycle (R2, R3, R4) in the TG. In this case, Condition (2) in Theorem 6.1 is not satisfied because there is no edge between any of the three nodes (R2, R3, R4) in DG. Let us assume that the conjunction of the conditions of the rules (R2, R3, R4) is not unsatisfiable, then Condition (3) is not satisfied. Thus, the rule set may not terminate.

Let us change the rules in the example by letting the condition of rule R4 to be "discountable==true". The corresponding TG, AG, and DG are shown in Figure 6.3. The change resulted in an edge from node *R2* to node *R4* in the DG. This means that the execution of R2 will deactivate R4. In this case, Condition (2) in Theorem 6.1 will be satisfied. Thus, the rule set will terminate.

If we change the condition of rule R4 to "platinum==true", the TG, AG and DG will still be the same as the original example, as shown in Figure 6.1. However, R2 and R4 have contradictory conditions (i.e., the conjunction of the conditions of R2 and R4 is unsatisfiable) and there is no action in the cycle to activate the rules explicitly (i.e., no incoming edge in AG for R2 or R4). Thus, Condition (3) in Theorem 6.1 is satisfied. Consequently, the rule set will terminate.

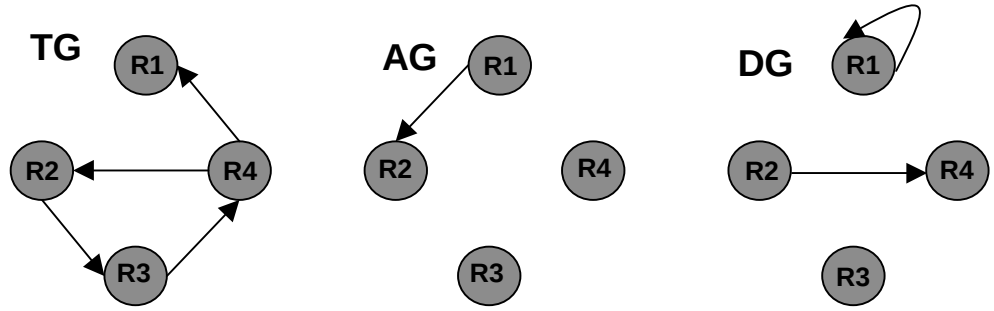

Figure 6.3 Revised Examples of TG, AG and DG

6.3.2. Completeness and Soundness of the Algorithm

As we have shown in Theorem 6.1, the TG, AG and DG can be used to detect the termination of a rule set. In this section, we will discuss some issues concerning the construction of these graphs. To construct these graphs, we need to analyze the rule set under verification to capture the triggering, activation, and deactivation information. In addition, since our work aims to verify general action-oriented rules with conditions and actions that can include method invocations, the side effects of the methods need to be taken into consideration in the construction of these graphs. However, in general, the triggering, activation, and deactivation information cannot be captured deterministically. For example, a rule *r1* may *conditionally* post an event *e1*, which is the triggering event of rule *r2*. Since the triggering relationship is conditional, we cannot construct the TG without making some assumptions. This is true for AG and DG as well. For example, some action in a rule *r1* may conditionally change the state of an object, which in turn activates or deactivates the condition of rule *r2*. Again, because it is conditional, we cannot construct AG or DG without making some assumptions.

There are two possible approaches in making these assumptions. We will classify

one as an *optimistic* approach and the other as a *pessimistic* approach. In the ***optimistic*** approach, we assume the best case and choose the assumptions that will bias the verification toward termination. In other words, we will ignore all *conditional* event postings when we form TG. In forming DG, if the action of one rule conditionally deactivates another rule, we will assume it will. In forming AG, we ignore all conditional activation.

On the other hand, for the ***pessimistic*** approach, we assume the worst case and choose the assumptions that will bias the verification toward non-termination. In forming TG, we will assume that all the *conditional* event postings will indeed post the events. In forming DG, the conditional deactivations are ignored. In forming AG, we assume all conditional activation.

Definition 6.3.10: The algorithm for non-termination detection is **complete** if it detects ***all*** the non-termination anomalies in a rule set.

Definition 6.3.11: The algorithm for non-termination detection is **sound** if every reported non-termination anomaly turns out to be an anomaly.

The *optimistic* approach is *sound*, but not *complete*. In other words, since the optimistic approach is a "best case" analysis, we will miss some non-termination anomalies. However, there will not be any "false positives" in which non-anomalies are reported as anomalies. The *pessimistic* approach is *complete*, but not *sound.* Since the pessimistic approach is a "worst case" analysis, we will detect all the non-termination anomalies; however, the algorithm may return some "false positives".

In conclusion, since the triggering, activation and deactivation relationships may be conditional, it is not possible to have a sound *and* complete detection algorithm. Even if we model all the conditions in the graphs to determine whether the action of one rule will activate or deactivate the condition of another rule, there is still a satisfiability problem; a well-known undecidable problem.

The practical result of this discussion is that we can bias Algorithm 6.1 to take the optimistic or pessimistic approach. This allows a Rule Warehouse Administrator (RWA) to select a proper approach based on the requirement of a specific application. In Algorithm 6.1, the decision of the RWA will determine what assumptions are made in forming the TG in Step 1 and the AG and DG in Step 3.

6.4. Inconsistency and Redundancy

6.4.1. Definition for Inconsistency

Informally, a rule base is ***inconsistent*** if there is an object state such that different activated rules intend to set the object to different states. Inconsistency can be either ***conflicting*** or ***contradictory***. Inconsistent rules will set an object to different object states; whereas, contradictory rules will set an object to contradictory object states. Rules are conflicting when they are inconsistent, but not contradictory. Rules 1, 2 and 3 in *Example 6.1* are conflicting rules because they share the same condition; however, their actions set the object to different (but not contradictory) object states. Rules 1 and 2 in *Example 6.4* are contradictory rules. Most of the current works do not distinguish

conflicting and contradictory rules; however, it is useful for us to distinguish whether two rules are conflicting or contradictory. In the context of using the Rule Warehouse System for collaborative problem solving, conflicting rules represent different experts' opinions and all conflicting rules should be stored in the Rule Warehouse. If we can define proper resolution rules to resolve the conflict, we can realize a resolution scheme "on demand" to take all the opinions into consideration. This concept was illustrated in Section 5 to demonstrate collaborative problem solving in Scenario X. On the other hand, it is not possible to resolve contradictory rules. Thus contradictory rules must be reconciled at verification time. We aim to detect both conflicting and contradictory rules. Before we consider the algorithms to detect them, let us first define conflicting and contradictory formally in this section.

Example 6.4:

```
Class Accident_statistics
{
    String model;
    int accident_probability DERIVED;
    Rule1{
        Condition:   model = M1;
        Action:      accident_probability  = 0.2;
    }
    Rule2{
        Condition:   model = M1;
        Action:      accident_probability  != 0.2;
    }
}
```

Definitions of inconsistency:

Definition 6.4.1: A **rule set *R*** of an object *O* is **inconsistent** if and only if $\exists\ S_E$, such that the action *a1* of *r1* and the action *a2* of *r2* set the object to different states S_{O1} and S_{O2}, where $r1 \in R_A$ and $r2 \in R_A$; S_E is an object execution state; $R_A \in R$ is the set of activated rules of S_E; S_{O1} and S_{O2} are object states of *O*.

Definitions of contradiction:

Definition 6.4.2: The **domain** *D* of attribute $A \in P$ is all the legitimate values of attribute *A*.

Definition 6.4.3: A **value set** *V* of attribute $A \in P$ is a subset of *D*, where *D* is the domain of attribute *A*.

Definition 6.4.4: A **value set** *V1* is **contradictory** to value set *V2* if there is no intersection between *V1* and *V2* and *V2* has all the values in Domain D that is not in *V1*. For example, if *V1* is all the integers that are larger than 2 and *V2* is all the integers that are less or equal to 2, *V1* and *V2* are contradictory. For a Boolean attribute, if *V1* is true and *V2* is false, then *V1* and *V2* are contradictory. For an enumerated attribute, if *V1* is "male" and *V2* is "female," then *V1* and *V2* are contradictory.

Definition 6.4.5: Two **object states** S_{O1} and S_{O2} of an object O are **contradictory** if $\exists$ $a \in P$, such that $V1$ and $V2$ are contradictory. P is the property set of O, $V1$ and $V2$ are the value sets of attribute a in object states S_{O1} and $S_{O2,}$ respectively. For example, for an object of class *Employee*, if an object state S_{O1} says that the gender of an employee is male and S_{O2} says that the gender is female, then the two states are contradictory.

Definition 6.4.6: A **rule set** ***R*** of an object O is **contradictory** if and only if $\exists$ S_E, such that the action *a1* of *r1* and the action *a2* of *r2* set the object to contradictory states S_{O1} and S_{O2}, where $r1 \in R_A$ and $r2 \in R_{A.;}$ S_E is an object execution state; $R_A \in R$ is the set of activated rules of $S_{E;}$ S_{O1} and S_{O2} are object states of O.

Definition of conflict:

Definition 6.4.7: A **rule set** ***R*** of an object O is said to be in **conflict** if it is inconsistent but not contradictory.

6.4.2. Definition for Redundancy

There are many types of redundancy anomalies defined in the existing works for logic rules [WU93], including subsumed rules, unnecessary-if, syntax redundancy, semantics redundancy, etc.

For example, the *unnecessary IF* anomaly can be defined as follows [NGU87]:

Definition 6.4.8: Two rules contain *unnecessary IF* conditions [NGU87] if (1) the rules have the same conclusion, (2) one of the IF conditions in one rule contradicts with an IF condition of another rule, and (3) all the other IF conditions in the two rules are equivalent.

In Example 6.5, the IF condition "*B*" in *R1* and "*¬B"* in *R2* are unnecessary, and in a sense, redundant.

Example 6.5:

R1:	Condition:	A ^ B
	Action:	C
R2:	Condition:	A ^ ¬B
	Action:	C

The following definition of the subsumption anomaly is adopted from [WU93].

Definition 6.4.9: If two rules have the same consequence but one contains more restrictive conditions to fire the rule, then it is said that the rule with more restrictive conditions is *subsumed* by the rule with less restrictive conditions.

In Example 6.6, rule *R1* is subsumed by rule *R2*. Intuitively, if a more restrictive rule succeeds, the less restrictive rule must also succeed, but not vice versa. The subsumed rule is not necessary and is redundant.

Although there are many types of redundancy anomalies defined in the existing work, it is argued in [WU93] that a more general semantics-based definition is necessary in

order to detect a variety of redundancies. [WU93] gave a definition of redundancy based on the *semantic properties* of logic rules. We will adopt and extend the definition to incorporate action-oriented rules.

Example 6.6:

R1: Condition: *A ^ B*
Action: *C*
R2: Condition: *B*
Action: *C*

Definition 6.4.10: The ***execution behavior*** of a set of consistent rules is the transformation of an object from its initial object state S_{oi} *to* its final object state S_{of} by applying these rules. Two sets of rules have the same execution behavior if, given the same initial object state, they reach the same final object state.

Definition 6.4.11: A ***rule set R*** is ***redundant*** if the removal of a rule and/or part of a rule from the set will not change its execution behavior.

Rule set redundancies can be generally divided into two categories. The first category of redundancies can be removed by reducing the number of rules in the rule set without changing its execution behavior. In Example 6.6, if we remove *R1*, the execution behavior of the rule set will not be changed. In Example 6.5, we can rewrite the two rules into one rule (*Condition: A, Action: C*) without changing the execution behavior. The second category of redundancies can be eliminated by removing a part of a rule in the rule set without changing its execution behavior. In Example 6.7, it is easy to show that if we refine *R1* to be (*Condition: A, Action: C*), the execution behavior of the resulting rule set will not be changed.

Example 6.7:

R1: Condition: *A ^ B*
Action: *C*
R2: Condition: *¬B*
Action: *C*

6.4.3. Algorithm for Inconsistency and Redundancy Detection

In this section, we will discuss our objective for detecting rule set inconsistency and redundancy for action-oriented rules in a Rule Warehouse System and present some results of the ongoing investigation. In Section 6.5, we will describe our plans for future work in this area.

Our objective for detecting inconsistency (or redundancy) is to answer the following three questions concerning a rule set: (1) Is the rule set inconsistent (or redundant)? (2)

If the rule set is inconsistent (or redundant), which portion of the rule set is causing the problem? (3) If the rule set is inconsistent (or redundant), under what situation (i.e., what object state), will the rule set cause problem? The answer to Question (1) will inform the Rule Warehouse Administrator (RWA) whether such anomalies exist in the Rule Warehouse. If so, the answer to Question (2) will help the RWA locate and eliminate the problem. If the problem cannot be eliminated, the answer to Question (3) will give the RWA the information concerning what object state can potentially lead to the anomaly.

Our approach to detect both inconsistency and redundancy anomalies is the same. Thus, we will describe them together. There are many existing works (i.e., good algorithms) for the detection of inconsistency and redundancy in a set of ***logic*** rules [ROS97, WU93]. We observed that both inconsistency and redundancy problems for ***action-oriented rules*** could be reduced to their corresponding problems for logic rules. Thus, the existing results obtained for logic rules can be applied with modifications to account for the added features of action-oriented rules: i.e., the existence of events and the invocation of methods in the condition and action parts of a rule.

Example 6.8:

```
Class Accident_statistics{
    String model;
    int accident_probability;
    Event e1, e2;
    Trigger t1 {
        TRIGGERINGEVENT e1;
        RULESTRUCT: Rule 1;
    }

    Trigger t2 {
        TRIGGERINGEVENT e2;
        RULESTRUCT: Rule2;
    }
    Rule1 {
        Condition:   model = M1;
        Action:      accident_probability  = 0.2;
    }
    Rule2 {
        Condition:   model = M1;
        Action:      accident_probability  != 0.2;
    }
}
```

Recall from Section 6.2 the execution semantics of an action-oriented rule. If an event *e1* is posted, and if *e1* is one of the triggering events of a rule *r1*, then the condition of the rule *r1* will be evaluated. If the condition evaluates to true, the action of *r1* will be performed. Note that only those rules that may be triggered by the same event (either *directly* or *indirectly*) need to be verified for inconsistency and redundancy. In Example 6.8, *Rule1* and *Rule2* are contradictory if they are put in the context of logic rule

verification. However, in the context of action-oriented rule verification, *Rule1* can only be triggered by *e1* whereas *Rule2* only by *e2*. In this case it is not possible for *Rule 1* and *Rule 2* to be triggered together. Thus, they are not inconsistent in the context of action-oriented rule verification.

Since only those rules that can be triggered by the same event need to be verified for inconsistency and redundancy, we can partition the rule set based on the triggering events for the purpose of verification. Each partition contains rules, which can be triggered by the same event either *directly* or *indirectly*. Also, each rule in a partition is a CA (condition-action) rule, which can be represented by the logic expression C→A (reads "C implies A"). Both C and A may contain method invocations. Most of the existing logic rule verification algorithms cannot handle methods in C and A. They need to be extended to deal with the side-effects of methods. This is one of the tasks in our future work. We shall revisit this issue in Section 6.6.

Definition 6.4.12: If a rule *r1* is triggered by an event *es*, we say r1 is triggered by *es* **directly**. We say *r2* is triggered by *es* **indirectly** if *r2* is triggered by an event *e*, which is posted by a rule triggered directly or indirectly by *es*.

Algorithm 6.2 (Reduction algorithm):

1. *Get the list of triggering events in the rule set.*
2. *Partition the rules into groups based on the triggering events. If one rule has multiple triggering event, it will be put into multiple groups.*
3. *Rewrite the conditions and actions of the rules to include the side effect of methods.*
4. *Combine the groups g1 and g2 if the action of a rule in g1 may post the event required for group g2. This is to take care of the indirect triggerings.*
5. *Verify the CA rule sets in each group using the available logic-based inconsistency detection algorithm Γ.*

Algorithm 6.2 performs two main functions: (1) uses triggering events (direct and indirect) to partition the rule set so that rules associated with a triggering event are verified together, and (2) reduces an inconsistency or redundancy problem for ***action-oriented rules*** into the corresponding problem for logic rules so that some existing algorithm Γ can be used. Step 1 gets a list of all the triggering events in the rule set. Step 2 partitions the rules based on triggering events (direct triggering). Step 3 considers the side-effects of methods if methods are involved. Step 4 combines the rules that may be indirectly triggered into the same group. Step 5 applies the existing algorithm for logic rules to finish the detection of inconsistency/contradiction and redundancy. After the partition and reduction process, the rules in each group are verified for the appropriate type of anomalies using Γ. Note that Γ is some existing algorithm for detecting inconsistency or redundancy anomalies for logic rules. As described in the next section, in our future work, we plan to extend these algorithms and tailor them to meet the needs

of Rule Warehouse applications.

6.5. Future Work

Action-oriented rule verification is a fertile research area. Although we have made some substantial progress, there is much work left to do. We aim to continue our research in three main areas:

- For the verification of inconsistency and redundancy anomalies, we will enhance the existing algorithms to meet the needs of a Rule Warehouse System.
- The simplifying assumptions stated in Section 6.2 will be relaxed so that more complex verification problems can be solved.
- Integrate the verification of logic rules and action-oriented rules in the Rule Warehouse.

We elaborate on them in the following subsections.

6.5.1. Enhance Existing Verification Algorithm

As mentioned in the previous section, Γ in Algorithm 6.2 is an existing algorithm for detecting inconsistency or redundancy anomalies for logic rules such as the one presented in [WU93]. The algorithm detects inconsistencies and redundancies in a set of logic rules by using a mechanical theorem-proving technique. A level-saturation resolution is performed if two predicates are contradictory. In our future work, we would like to extend this algorithm so that it will distinguish conflicting rules from contradictory rules and will accommodate the side effects of method invocations. We will change the condition used to perform a resolution so that the algorithm will deal with both inconsistency rules and contradictory rules. To handle method invocation, we will define the read set and write set of a rule. In a traditional logic rule, the LHS only checks for data conditions, i.e., only perform READ operations. The RHS asserts or retracts facts, i.e., only performs WRITE operations. If methods are involved, both LHS and RHS can have read and write operations. We will define the read set and write set for every rule and determine the conditions used to perform a resolution based on the intersection of the read sets and write sets of different rules.

6.5.2. Relaxing Simplifying Assumptions

In order to make action-oriented rule verification a tractable problem, we have made some simplifying assumptions given in Section 6.2. In the next stage of our research, we will relax some of the assumptions so that more complex problems can be solved. First, we will include the event history part of a trigger specification in the verification process. If event history is involved in the non-termination detection, a more advanced cycle detection algorithm has to be developed to detect the triggering cycles in the TG. Currently, a rule is considered to be triggered when there is an incoming edge from another rule in the TG. If event history is considered, a complex logic expression has to be satisfied for a rule to be triggered. A cycle *c* can be considered as a triggering cycle only if the event history of each node in *c* can be satisfied by the events posted by the other rules in *c*. The triggering cycle detection algorithm must enforce this constraint.

For the inconsistency and redundancy detection, currently, we partition the rules based on triggering events. In a more general case, the partition should base on the information specified in the event history expression.

Secondly, our current work assumes that there is a single rule in the rule structure of a trigger. We will relax that assumption to include more complex rule structures. In general, the rules in a rule structure may be executed in parallel or sequentially or a combination of both. Current works on rule verification assume that rules are executed in parallel. We shall also explore the semantics of a sequential execution of rules and study the impact it has on action-oriented rule verification.

Finally, we will extend our work to consider CAA (condition-action-alternativeAction) rules instead of just CA (condition-action) rules. However, we do not see that this will have a major impact on the verification techniques we have developed for CA rules since CAA rules can always be re-written as CA rules.

6.5.3. Integrated Verification

In a Rule Warehouse System, we have both logic (or constraint-oriented) and action-oriented rules. These two types of rules may be used independently in some applications. However, for collaborative problem solving, they have to be used together. Thus, there is a need for integrated verification of logic and action-oriented rules.

Currently, as shown in Figure 6.4, we are considering the following approach to address the problem. Logic rules are first rewritten as action-oriented rules. By doing so, all rules are in a common form. At this stage, we can apply the non-termination detection algorithm described in Section 3, which has been developed for verifying action-oriented rules. For detecting other anomalies such as inconsistency and redundancy, we can apply the reduction algorithm (Algorithm 6.2) to partition the integrated rule set and to reduce the problem into a logic rule verification problem so that the existing algorithm for logic rule verification can be applied.

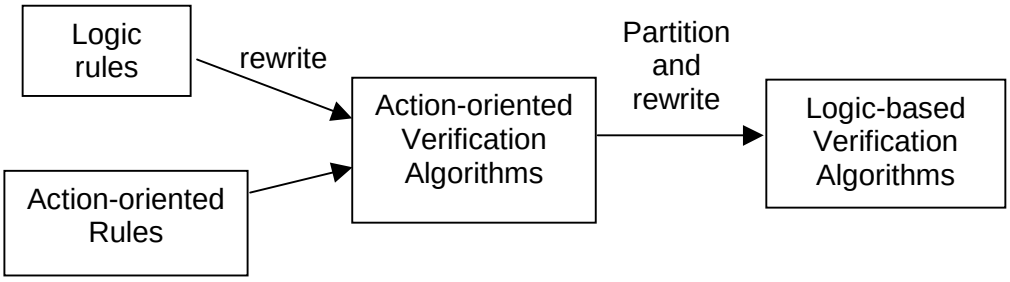

Figure 6.4 Integrated Verification

7. Summary and Future Work

In this paper, we have proposed the use of a business Rule Warehouse System to support the next phase in the evolution of B2B technology: *collaborative e-business*. We have presented an architecture of a Rule Warehouse System and identified a number of services required to enable e-business collaboration through ***business rule sharing***, ***collaborative problem-solving***, and ***collaborative interaction, using business events and rules.*** These services include:

- Rule Import, Verification, and Export Services
- Collaborative Problem Solving Service
- Warehouse Management Service
- Constraint Satisfaction Service
- Event-Trigger-Rule (ETR) Service

Our focus in this project has been on the research and development of technologies to support *business rule sharing* (rule import, verification, and export services) and *collaborative problem solving*.

One main barrier for rule sharing among business partners is that the rules imported into the Rule Warehouse may contain rule anomalies. The imported rules need to be verified to detect these rule anomalies and to resolve them, if possible. Rule base verification has been an important area of research in the expert system community. Techniques for verifying expert system rules are available. However, the verification of rules in a Rule Warehouse is a much more challenging problem because a Rule Warehouse may contain both constraint-oriented rules and action-oriented rules.

In this work, we introduced an Active Object Model (AOM) as a common knowledge representation for a Rule Warehouse to enable the verification of heterogeneous business rules. In AOM, we extended the traditional object model by incorporating a knowledge specification component to specify constraint-oriented knowledge and action-oriented knowledge. Based on this uniformed knowledge representation, we focused on the detection of three types of anomalies that can exist in *action-oriented* rules: *inconsistency, redundancy, and non-termination*. This work complements the existing research on the verification of logic (constraint-based) rules. Unlike the verification of logic rules, the effects of *events* and *triggers* have to be considered. Another important issue concerning action-oriented rule verification is the side effects of *method* invocations. In this paper, each of the rule anomalies was formally defined. Algorithms for detecting the anomalies were developed.

An important benefit of a Rule Warehouse System is to integrate the knowledge acquired from different sources to solve problems that cannot be solved by using the knowledge and rule engine of an individual source. In this paper, we describe our work on the integration of a deductive rule engine (DRE) with an action-oriented rule engine (ARE) to demonstrate collaborative problem solving. IBM's Common Rule System, a DRE, was enhanced to handle the following three problems (with the help of an ARE and the user) that cannot handled by traditional DRE's:

- Conflicting knowledge

- Incomplete knowledge
- Don't care condition.

UF's ETR Server (an ARE) is used to extend the Common Rule System's (CRS's) problem-solving capability in a flexible way. In the incomplete knowledge case, if there is no rule defined for an attribute in CRS, a *missingRule* event is posted to ETR to seek help. In a sense, the ETR Server produces a "virtual rule on demand" for CRS so that the processing can continue. In the conflicting knowledge case, there are multiple conflicting rules that are defined in CRS. A *conflictingRule* event is posted to ETR to resolve the conflict. The rationale for this type of events is to enhance a rule system like CRS with a flexible, "user-defined" resolution mechanism. In case the information provided in a query to the Rule Warehouse System only partially matches with the precedence (left side) of some rules, CRS can interact with the user thorough the Rule Warehouse Interface to obtain additional information. If the user responds with a "don't care" situation, then CRS can ignore the unmatched condition and continue its inferencing process by applying the partially matched rules. It is our belief that the collaboration between the requester and different types of rule engines to handle different types of rules in a Rule Warehouse is important to solve complex problems.

We plan to continue our research in the following three areas:

- For the verification of inconsistency and redundancy anomalies, we will enhance the existing algorithms (developed to verify logic rules) and tailor them to meet the needs of a Rule Warehouse System. In particular, we want to refine the inconsistency detection algorithm to distinguish between conflicting rules and contradictory rules and to take care of the side effects of method invocations.
- The simplifying assumptions used in our work will be relaxed so that more complex verification problems can be solved. We shall work on the verification of rules which have more complex triggers (i.e., triggers that contain event history and rule structure specifications). We will extend our work to consider CAA (condition-action-alternativeAction) rules instead of just CA (condition-action) rules.
- In a Rule Warehouse System, we have both logic and action-oriented rules. These two types of rules can be used independently to provide different types of services. However, for collaborative problem solving, they have to be used together. Thus, we will study techniques for the integrated verification of logic rules and action-oriented rules.

Our long-term goal for this project is to *integrate* several key technologies developed at UF under the support of EECOMS and other NIST, NSF, and DARPA funded projects to support ***collaborative e-business,*** which we believe is the next phase in the evolution of e-business. The developed technologies include:

A Rule Warehouse System

An Automated Negotiation Server.

- A business process modeling and processing system for modeling and concurrent processing of business processes.
- An Event-Trigger-Rule (ETR) Server to provide “active” and collaborative interaction among enterprises (e.g., exception handling).
- Constraint Satisfaction Processor (e.g., to support automated negotiation and supplier selection)
- IKNET, a scaleable Internet-based knowledge network for sharing knowledge contributed by Internet users and business organizations.

References

[AIK95] Aiken, A., Hellerstein, J., and Widom , J., Static Analysis Techniques for Predicting the Behavior of Database Production Rules, In ACM TODS, March 1995.

[BAR00] Baralis, E. and Widom, J., Better Static Rule Analysis for Active Database Systems, http://dbpubs.stanford.edu/pub/2000-17.

[BAR93] Baralis, E., Ceri, S., and Widom, J., Better termination analysis for active databases. In Proceedings of the First International Workshop on Rules in Database Systems, Edinburgh, Scotland, August 1993, pp. 163-179.

[BAR94] Baralis, E. and Widom, J., An algebraic approach to rule analysis in expert database systems. In Proceedings of the Twentieth International Conference on Very Large Data Bases, Santiago, Chile, September 1994, pp. 475-486.

[BAR95] Baralis, E., Ceri, S., and Paraboschi, S., Improved rule analysis by means of triggering and activation graphs. In Proceedings of the Second International Workshop on Rules in Databases Systems, Athens, Greece, September 1995, pp. 165-181.

[BAR98] Baralis, E., Ceri, S., and Paraboschi, S., Compile-Time and Runtime Analysis of Active Behaviors, IEEE Transactions on Knowledge and Data Engineering, Vol. 10, No. 3, May/June 1998, pp.353-370.

[BLA] Blaze Software, “Blaze Advisor Technical White Paper”, http://www.mlsoft.com/products/docrequest.html.

[CRA87] Cragun, B.J. and Steudel, H.J., A decision-table-tased processor for checking completeness and consistency in rule-based expert systems, Int. J. Man-Mach. Stud., Vol. 26, 1987, pp. 633-648.

[EEC99] EECOMS Rules SIG, EECOMS Rules SIG Design Specification, Version 3, Revision 8, http://ciimplex.org, Sep. 1999.

[EEC00] EECOMS Rules SIG, EECOMS Rules SIG Design Specification, Version 4, Revision 2, http://ciimplex.org, Jun. 2000.

[GIN88] Ginsberg, A., Knowledge-base reduction: A new approach to checking knowledge bases for inconsistency and redundancy. 7th National Conference on Artificial Intelligence (AAAI 88), St Paul, MN, Vol(2), 1988, pp. 585-589.

[GON97] Gonzalez, A. and Dankel, D.D., The Engineering of Knowledge-based Systems Theory and Practice, Prentice Hall, 1997.

[GOT97] Gottesdiener, E., Business Rules Show Power, Promise, Issue of Application Development Trends, vol. 4, no. 3, March 1997.

[GRO99a] Grosof, B.N., Labrou, Y., and Chan, H.Y., A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML, the Proceedings of the 1st ACM Conference on Electrical Commerce (EC-99), Denver, Colorado, USA, Nov. 3-5, 1999.

[GRO99b] Grosof, B.N. and Chan, H.Y., IBM CommonRules Version 1.0 Readme, http://www.research.ibm.com/rules/, Aug. 09, 1999.

[GUI97] GUIDE Business Rule Project final report, 1997.

[HAA90] Haas, L.M., Chang, W., Lohman, G.M., et al., Starburst mid-flight: As the dust clears. IEEE Transaction on Knowledge and Data Engineering, 2(1), March 1990, pp.143-160.

[HAL97a] Halle, B. and Plotkin, D., Relaxing Your Data Focus Can Do Wonders for Your Business Rules: The Art of Letting Go, Database Programming & Design, February, 1997.

[HAL97b] Halle, B., Are Business Rules a Silver Bullet? Database Programming & Design, October, 1997.

[HAL99] Hally Enterprise, http://www.haley.com/1198502104597522/THE.html.

[HAN93] Hanson, E.N., Rule Condition Testing and Action Execution in Ariel, Proceedings of the ACM SIGMOD Conference, June 1992, pp. 49-58.

[HAY99] Hay, D., Managing Business by the Rules, http://www.essentialstrategies.com/publications/businessrules/bruleseco.htm

[HUA00] Huang, C., A Web-based Negotiation Server for Supporting Electronic Commerce, Ph.D. Dissertation, Department of Computer and Information Science and Engineering, University of Florida, May 2000. Also, http://www.ciimplex.org, T5 document, members only access, April, 2000.

[KAR94] Karadimce, A.P. and Urban, S.D., Conditional term rewriting as a formal basis for analysis of active database rules. In Fourth International Workshop on Research Issues in Data Engineering (RIDE-ADS' 94), Houston, Texas, February 1994, pp. 156-162.

[LAC99] Lacerra, S., Benson, R. and Wong, K., b~eCommerce Services, Business Services for the New Economy, eBusiness & eCommerce Services Industry Report, Jefferies & Company, Inc., Fall, 1999.

[LAM98] Lam, H. and Su, S.Y.W., Component Interoperability in a Virtual Enterprise Using Events/Triggers/Rules, In Proc. of OOPSLA '98 Workshop on Objects, Components, and Virtual Enterprise, Vancouver, BC, Canada, Oct. 18-22, 1998.

[LEE00] Lee, M., "Event and Rule Services for Achieving a Web-based Knowledge Network," Ph.D. Dissertation, Department of Computer and Information Science and Engineering, University of Florida, April, 2000.

[LIU00a] Y. Liu, C. Pluempitiwiriyawej, H. Lam, J. Hammer, and S. Y. W. Su, Rule Warehousing research and application in the context of Scenario X: version 1, EECOMS research white paper, T5 Rule Management Technology, http://ciimplex.org/ciimplex/TeamDeliverables.

[LIU00b] Y. Liu, H. Lam, and S. Y. W. Su, Rule Warehouse Application in Scenario X EECOMS Design Document, EECOMS research white paper, T5 Rule Management Technology, http://ciimplex.org/ciimplex/TeamDeliverables.

[MDC99] Meda Data Coalition, Business Engineering Models: Business Rules Review Draft, http://www.mdcinfo.com/OIM/models/BRM.pdf, July 15, 1999.

[MCC89] McCarthy, D.R. and Dayal, U., The architecture of an active database management system, In Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 1989, pp. 215-224.

[NGU85] Nguyen, T.A., Perkins, W.A., Laffey, T.J., and Pecora, D., Checking an Expert System Knowledge Base for Consistency and Completeness, in Proc. 9th International Joint Conference on Artificial Intelligence (IJCAI 85), Los Angeles, volume 1, 1985, pp. 375-378.

[NGU87] Nguyen, T.A., P Perkins, W.A., Laffey, T.J., and Pecora, D., Knowledge base verification, AI Magzine, 8(2), summer 1987, pp. 69-75.

[PHI00] Phillips, C. and Meeker, M., The B2B Internet Report Collaborative Commerce, Morgan Stanley Dean Witter, April, 2000.

[PLO99] Plotkin, D., Business Rules Everywhere, Intelligent Enterprise, vol.2, no. 10, 13 July 1999, p. 42-4, 46, 48.

[PRE92] Preece, A., Shinghal, R., and Batarekh, A., Verifying Expert Systems: A logical Framework and A Practical Tool, Expert Systems with Applications, vol. 5, nos. 2-3, 1992, pp. 421-436.

[PRE94] Preece, A., Shinghal, R., Foundation and application of knowledge base verification. International Journal of Intelligent Systems, 9(8), 1994, pp. 683-702.

[ROS97a] Rosenwald, G.W. and Liu, C.C., Rule-based system validation through automatic identification of equivalence classes, IEEE Transactions on Knowledge and Data Engineering, Vol(9)1, Jan./Feb., 1997, pp. 24-31.

[ROS97b] Ross, R., "The Business Rule Book: Classifying, Defining and Modeling Rules", Second Edition, Business Rule Solutions, LLC, 1997.

[ROU88] Rousset, M.C., On the consistency of knowledge bases: The COVADIS system, Computant. Intell. 4, 1988, pp. 166-170.

[SEI99] Seiler, H., "Managed business rules: a repository-based approach", PC AI, vol. 13, no.4, July-Aug. 1999, pp. 16-19.

[STA87] Stachowitz, R.A., Combs, J.B., and Chang, C.L., Validation of knowledge-based system, in Proc. 2nd AIAA/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program, Arlington, VA (Report No. AIAA-87-1685), 1987, pp. 1-10.

[STO88] Stonebraker, M., Hanson, E.N., Potamianos, S., The POSTGRES rule manager, IEEE Transaction on Software Engineering, 14(7), July 1988, pp. 897-907.

[SU00] Su, S. Y. W., Huang, C., and Hammer, J., "A Replicable Web-based Negotiation Server for E-Commerce," *Proc. Of the Hawaii International Conference on Systems Sciences, Minitack on "Evolutin of Business-to-business* Electronic Commerce," Maui, Hawaii, Jan. 4-7, 2000.

[SUW82] Suwa, M., Scott, A.C., and Shortliffe, E.H., An approach to verifying Completeness and Consistency in A Rule-Based Expert System, AI Magazine, 3(4), 1982, pp. 16-21.

[VIT] Vitria Businessware, http://www.vitria.com/products/businessware.html.

[WEI95] Weik, T. and Heuer, A., An algorithm for the analysis of termination of larger trigger sets in an OODBMS, In Proceedings of the International Workshop on Active and Real-Time Database Systems, Skovde, Sweden, June 1995.

[WFMC] Workflow Management Coalition, The Workflow Reference Model (WFMC-TC00-1003), Jan. 1995.

[WID96] Widom, J. and Ceri, S., Active Database Systems, Triggers and Rules for Advanced Database Processing, Morgan Kaufmann Publishers, Inc., 1996.

[WU93] Wu, P. and Su, S., Rule Validation Based on Logical Deduction, Proceedings of ACM 2nd International Conference on Information and Knowledge Management, Arlington, VA, 1993, pp. 164-173.

[WU97] Wu, C. and Lee, S., Knowledge Verification with an Enhanced High-level Petri-Net Model, IEEE Expert, September/October, 1997, pp. 73-80.

[ZAC87] Zachman, J. A., "A framework for information systems architecture", IBM Systems Journal, Vol26, 1987.

[ZHA94] Zhang, D. and Nguyen, D., PREPARE: A tool for knowledge Base Verification, IEEE Transaction on Knowledge and Data Engineering, vol. 6, no. 6, 1994, pp. 983-989.