A WEB-BASED EXPERT SYSTEM FOR VEHICLE REGISTRATION

Alan T. Demmin and Du Zhang

Department of Computer Science
California State University
Sacramento, CA 95819-6021
atdemmin@innercite.com, zhangd@ecs.csus.edu

Abstract. In order to satisfy an increasing need to respond rapidly to changing business requirements and knowledge, and to reduce implementation costs, the mainstream business world is progressively relying on expert systems technology. In addition, recent e-business initiatives are requiring companies to provide “web-enabled” systems in order to maintain a competitive edge in their marketplace. This sets the stage for our study. In this paper, we describe our experience in capturing the business logic and implementing a web-based expert system for vehicle registration fee computation at California Department of Motor Vehicles. The system has been implemented using the Blaze Advisor rules engine and has been deployed to Sun’s J2EE Reference Implementation of a Java application server. Our work has shown that using expert system technology to implement a web-based, business-driven solution is certainly a viable and promising option. Deployment of the rule service as an Enterprise JavaBean exposes the rule-base to various types of business clients, and provides a consistent and adaptable implementation of the business domain.

Keywords: business rules, web-based expert systems, Java technology, vehicle registration.

1. INTRODUCTION

Business rules are declarative statements that define or constrain a business domain and are often implicitly defined within the manual and automated processes of a company. Business rules are directives intended to guide or influence business behavior [20, 21]. The business rules for a company may or may not be formally documented and may only be available as learned or perceived knowledge of the business personnel. Over the last decade, companies across all industries have shown an expanding interest in formally defining and documenting their business rules and establishing them as official artifacts for the company. In fact, definition and documentation of business rules have proven to help clarify business objectives, streamline business processes, and even improve a company’s definition of new or changing business requirements. This helps when justifying business decisions, training new business personnel, and can even reduce the time necessary to deliver software modifications to the production environment.

While there are distinct advantages in simply documenting business rules, the eventual expectation leads to an implementation of the business rules into an automated solution and to making them available to all business processes. In particular, recent e-business initiatives have shown that companies need to rapidly implement changes in order to maintain a competitive edge within the marketplace [12]. This is setting the stage for the integration of expert system technology [11] and intelligent agents [23] within existing business infrastructures.
Though its early applications have been in the financial institutions, insurance companies, and government agencies [4], today the business rule approach finds its way into regulatory institutions and companies large and small that have business units, customers, competitors and overall market conditions. There are many success stories [4]. Using the business rule technology, (a) Orange Denmark achieved a productivity increase of 67% and a 50% increase of the service availability for skill-based routing of service requests. (b) Travelocity.com is able to launch major changes at least once per month and its time-to-market for new products around the world has been reduced in half. And (c) The Canadian National Railway is able to shorten the development time by a factor of 3 (down to four months) of an internet-based system that provides customers with the ability to compare transportation rates, track their shipments, and manage their booking, payment and invoicing online.

The purpose of this study is to investigate the feasibility of utilizing an inference capability to administer the business logic for an enterprise within a web-based architecture. We choose, as the focus of our study, a subset of the business domain for vehicle registration fee computation at the state of California Department of Motor Vehicles (DMV). In particular, the fee computation logic for vessel vehicle types is selected for the prototype system implementation. The business rules for this domain were gathered, organized, and then codified into the rule syntax of Blaze Advisor rules engine [8]. The service of the rules was then deployed to a Java 2 Enterprise Edition (J2EE) web application server and a web browser interface was developed to allow users to inquire about registration fees that are assigned for various transactions.

There are several reasons behind our choice of the DMV project. The first author works as a consultant at DMV and has intimate knowledge of the problem domain. The current fee processing systems at California DMV were first developed in the 1970’s and have been modified many times over the past three decades. The online system used at the local field offices was implemented using the Event Driven Language (EDL), which is the assembly language of the IBM Series/1 machines. The headquarters batch system, which processed the daily transactions and updated the main database during the nightly batch cycle, was implemented in the COBOL for IBM mainframe running the Multiple Virtual Storage (MVS) operating system. Because separate teams at the DMV maintain the two systems, a single change in legislation or business policy can result in different interpretations. Over the years, the two systems have diverged significantly and can sometimes produce inconsistent results. In addition to the potential inconsistency in the two fee computation systems, there is also a significant need for a more modern web-based solution. The current systems were developed using legacy programming infrastructures that are no longer supported and difficult to maintain. A simple change in legislation that should require only weeks for implementation can take months of development because of the complexity involved with the assembly programming language. In fact, the memory requirements of the fee
computation system are beginning to exceed the limits imposed by the Series/1 machine. Together, the business and technical challenges are producing a significant need for system re-engineering and the need for a more strategic production environment.

The widespread acceptance of expert systems technology as an option for business-driven solutions is going to rely heavily on the adoption of formal business rule methodologies. In order to design systems that can facilitate a rapid response to change, it is necessary to minimize the translation time between the business documentation and the technical implementation. A business rule methodology is used to design and develop a business rule system, and includes steps for knowledge acquisition, knowledge representation, and knowledge verification and validation. The reason that businesses are looking to utilize expert systems technology is to simplify the amount of work required in order to respond to changing business requirements. Separating the control logic from the business rules will lead to faster delivery of system modifications, lower information technology costs, and a competitive edge in the marketplace. In fact, the business personnel will also gain a much deeper understanding of the business by simply following a business rule methodology to organize and document their rules. Implementing these rules into an expert system is just another step towards component-based development, where the business rules become an explicit, modifiable component within the automated system.

Now, more than ever, there is a significant need to distinguish the differences between business and technical knowledge, and to organize the business knowledge into a common representation of the business. At the heart of this challenge is the difficult process of extracting, defining, and organizing a company’s rule-base, which will become as important a business artifact as the information within a company’s database.

The results of this study indicate that using expert systems technology to implement a web-based, business-driven solution is certainly a viable and promising option. The ease of this implementation relies heavily on the supporting tools that are supplied with the development environment. The prototype system we developed allows multiple platform deployment, distributed object access, and graphical or model-driven maintenance of its rule base. Our contributions to research in the expert system field consist in a modularized design approach to the rule base that is positioned for future legislation changes, a rule mining method from legacy codes, and a versatile deployment of rules service.

The remainder of this paper is organized as follows. Section 2 briefly describes some background in the area of business rule methodology and expert systems technology. Section 3 focuses on the rule-base engineering process. Section 4 discusses some issues in the prototype system. Section 5 contains performance evaluation of the prototype system through some common use cases for vehicle registration. Comparison with an alternative approach (JESS, Java Expert System Shell [24]) is provided in Section 6. Finally, in Section 7 we conclude the paper with remarks on possible future work.
2. BACKGROUND

The business rule approach utilizes the expert system technology in business domains. It is geared toward identifying the rules required to run a business, documenting these rules, reasoning with them, making them operational in a consistent way, adapting them systematically to changing market conditions and environments, and automating the knowledge management process.

Currently, there are some proposed methodologies. The concept of a fact-model has been introduced within the proposed business rule methodology called Proteus™ [6]. A fact model is a way of modeling the business terms and their relationships and is similar to the entity-relationship diagramming that is used for relational database design. A formal methodology called the STEP™ principles is proposed in [28]. STEP stands for Separating (separating business rules from components not relevant to the business knowledge), Tracing (traceability of business rules in terms of their sources and usages), Externalizing (making business rules explicit), and Positioning (positioning rules for changes). A Rules Maturity Model™ is described in [17].

A number of tools are available presently. Blaze Advisor is a decision building system that employs rules engine technology to deliver automated business solutions [8]. It is Java-based and offers deployment capabilities to most platforms available today. It includes many interfacing capabilities to allow integration with modern and legacy systems and employs the use of XML. It has a graphical User Interface (Advisor Builder), the deployment runtime (Advisor Server), the deployment option (Enterprise JavaBean, or EJB), and the J2EE application server. It also has tools for analyzing a rule-base to locate inconsistencies, rule conflicts, and even circular logic. We use the Blaze Advisor rules engine to develop our prototype system.

Other tools include: (a) ILOG JRules is a Java-based rules engine, for business-driven solutions [15]. Similar to Blaze Advisor, JRules is a product that includes graphical user interfaces for building and modifying rule bases. It also includes capabilities to deploy the automated solution to various platforms and with differing interfaces. (b) JESS is a scientific Java-based expert system shell that is an extension to the CLIPS system developed by NASA in 1984 [24, 10]. Since JESS makes the CLIPS system Java-compatible, it has opened the door to providing many of the multiple platform and multiple interface capabilities of the business driven solutions. (c) IBM CommonRules. This tool stemmed from an earlier IBM project called Business Rules for Electronic Commerce, which included an investigation into rule based technology for business processes involving both business-to-consumer (B2C) and business-to-business (B2B) e-Commerce. This research was particularly concerned with the creation of a core rules technology that could be reused with various heterogeneous applications to share common rules information. The resulting technology is known as IBM CommonRules [13], which is a Java library of
classes to be used as a development platform for rule-based applications and includes a data interchange format defined as an XML schema called Business Rules Markup Language (BRML) [12].

The popularity of the Internet has forced business to pursue web-enabled solutions in order to maintain a competitive edge in their respective industries. This is true not only of B2C relationships but also with B2B communication. The success of e-Commerce and the inter-communication of distributed heterogeneous systems rely heavily on open systems architecture. In turn, the success of open systems architecture relies heavily on the definition and industry acceptance of formal specifications that define the requirements for each new technology. The widespread use of rules engine technology in a web-enabled environment will also rely on a firm definition of a meta-model for rules and a common ontology or specification of concepts for rules information exchange. Listed below are some of the current efforts in defining and developing formal technical specifications.

**Business Rules Group.** The Business Rules Group [5] is an organization of practitioners in the field of systems and business analysis dedicated to defining and supporting the specifications on business rules and how they are related to enterprise organization and systems integration. This group was initially a project within GUIDE (Guidance of the Users of Integrated Data-processing Equipment) International [14] and now exists as its own organization.

**OMG Business Rules Working Group (OMG BRWG).** The Object Management Group (OMG) [19] is an organization established in 1989, dedicated to the establishment of formal specifications for the software industry that are vendor independent and are directed at the standardization and interoperability of object oriented technologies. The Business Rules Working Group (or Business Rules Special Interest Group) is a sub-committee of the OMG Architecture Board, working to establish formal specifications on business rules and/or rules engine technology.

**Java Community Process - JSR94.** The Java Community Process (JCP) is an open organization of Java developers and licensees whose goal is to expand and enhance the Java technology platform and its associated specifications [16]. JSR094 is a specification request to define a formal interface for a Java Rule Engine API to be used as a standard interface for both a Java 2 Standard Edition application and within a J2EE application server. This specification is a first step in standardizing the interface to a rules engine within the Java platform and includes high-level classes within the javax.rules package structure and defines classes for a Rule Service Provider, a Rule Runtime, and a Rule Session, among others.

**Rule Markup Initiative.** The Rule Markup Initiative as another open network of individuals from both the business and academic world, dedicated to establishing a formal XML schema for a Rule Markup Language shared among rules engine vendors [22]. This will produce a standard set of XML tags to be used for rules representation, translation, and interaction between heterogeneous rule processing systems.
As the formal specifications from the OMG and JCP work to provide standards for object management and the Java platform specifically, a formal rule markup will provide a standard way for businesses to share rules and for vendors to provide a consistent representation for rules interchange [26, 27].

*Java 2 Enterprise Edition (J2EE).* It is also important to mention the deployment environment that we chose for our project. The J2EE specification [25] provides for a standard platform that supports component architectures, with consistent application programming interfaces used to interact with the various application services. The J2EE specification is becoming the industry-leading model for application servers based on the Java platform and will most likely be the industry standard for all object oriented enterprise applications in the future.

### 3. RULE-BASE ENGINEERING AND DEPLOYMENT

The engineering process of our rule based solution for the DMV application involves defining all terms, objects, facts and functions, acquiring and organizing rules that comprised the knowledge base, and deploying the rule service in a web-enabled environment. We identify the following high-level requirements for the fee computation system.

1. The system must be well organized and make use of modern industry standards to develop a business model-driven architecture.
2. The system must be well positioned for future modifications in order to provide rapid responses to changing business requirements.
3. The system must be a single implementation of fee computation and must be accessible using various means of communication.
4. The system must be web-enabled, including the ability for applications to interact with the rule-base over the Internet and user interaction through a web browser.

After a brief discussion on the object modeling and term definitions, we will focus on our main contributions in the work: a modularized design approach to the rule base that is positioned for future legislation changes, a rule mining method from legacy codes, and a versatile deployment of rules service.

#### 3.1. Object Modeling and Term Definition

This system has been designed using the principles of object-oriented programming and relies on the use of model-driven architectures. UML class diagrams were developed for the input/output object model and the high-level interface design. This object model was then used for the term definitions within the rule-base.

One of the first design considerations of any software system is the identification of data elements that will be used as input to the system, and similarly, the data elements that will be returned or updated
by the system. When designing rule-based systems, these data elements are the terms and facts that are part of the knowledge base. The first step in designing a rule-base is to identify all of the data elements or terms within the knowledge and assign appropriate term names. A term name can make a significant difference in the comprehension of the rules that utilize these terms.

It was important to first produce a list of terms and object attributes that would be used to develop the initial fact list for the rule-base. After interviews with the business users and analysis of existing programs, the input data elements were grouped into the following objects:

1. FeeTransaction - the type of transaction being requested,
2. FeeVehicle - the vehicle that is involved in the transaction,
3. FeeOwner – the owner of the vehicle that is involved in the transaction.

The information would be processed through the fee computation rules to determine a list of fees for the transaction being requested. In addition, some related fields are also returned as part of the solution. This information was grouped into the following object:

4. FeeResult – the resulting fee list and related information for the transaction.

After gathering all of the necessary information for each object type and gaining formal acceptance on all attribute and term names, a UML class diagram was developed to represent the object model (Figure 1). Since the fee computation system was being developed as a request/response system, the FeeRequest and FeeResponse classes were defined to encapsulate all of the functionality necessary to translate data between a serialized and de-serialized format, to match requests with responses, and to designate which rule service will be accessed. FeeRequest and FeeResponse are composite objects that contain all of the sub-objects used as input and output data within the rule base.

**3.2. Modularized Business Rule-base Design**

Once the terms and object model were defined, a formal methodology was employed to define the business rules. We chose the STEP™ process [28] to separate, trace, externalize, and position rules for changing business requirements. We began by interviewing business users and business analysts to gather facts and rules about vessel registration fee computation. The process also included the use of rule mining techniques to extract business rules from the existing legacy programs.

This methodology led to a better understanding and formal definition of the fee computation business process. When a business process can be formally defined with a series of tasks and sub-processes, this can be represented within a rule-base as a rule-flow. A rule-flow governs the business process and organizes sections of the knowledge base into logical partitions or processing steps. When working with the Advisor Builder, the rule-flow is created and maintained using a graphical editor, much like creating a data flow diagram or the structure chart of a computer program.
Two main aspects of the business were carefully considered in order to provide a well organized system when designing the rule-flow for the business process:

1. logical separation of vehicle types, and
2. logical separation of fee types.

Vehicle types are separated within the rule-flow in order to provide a clear organization of the business and to allow the segregation of rules according to the way legislation is normally passed by the state. When a vehicle type obtains new or changed laws, the affected area of the rule-base can be easily identified and changed. The diagram in Figure 2 is a representation of the main rule-flow for the fee computation system. All vehicles are included in the rule-flow, though only Vessel vehicles have been implemented to date.

The beginning and ending points of a rule-flow can be thought of as the call and return of a program function. The input object model will first be mapped into the knowledge base. Execution then proceeds according to the decision points, tasks, and sub-processes. Each task contains either a function or a rule-set invocation. Within a rule-set, the individual rules are evaluated and placed on the agenda when they

---

**Figure 1.** UML Class Diagram for the Input/Output Object Model.

Vehicle types are separated within the rule-flow in order to provide a clear organization of the business and to allow the segregation of rules according to the way legislation is normally passed by the state. When a vehicle type obtains new or changed laws, the affected area of the rule-base can be easily identified and changed. The diagram in Figure 2 is a representation of the main rule-flow for the fee computation system. All vehicles are included in the rule-flow, though only Vessel vehicles have been implemented to date.

The beginning and ending points of a rule-flow can be thought of as the call and return of a program function. The input object model will first be mapped into the knowledge base. Execution then proceeds according to the decision points, tasks, and sub-processes. Each task contains either a function or a rule-set invocation. Within a rule-set, the individual rules are evaluated and placed on the agenda when they
are enabled. When the agenda is empty, execution returns from the task and the rule-flow is followed along the next path. When the entire rule-flow has been processed, the output object model is returned to the calling agent.

Figure 2. Fee Computation Main Rule-flow.

This Fee Computation Main Rule-flow begins with a function that will initialize variables and tables for the knowledge base. The FeeRequest data that is listed in Subsection 3.1 above is passed into the rule-base as facts. Additional facts are then asserted according to the vehicle’s license type, the transaction type being requested, and some additional vehicle facts in order to begin fee computation. The vehicle type is used to direct the flow to the appropriate sub-process (sub-rule-flow) that will assign all of the fees for the vehicle. After processing through the vehicle specific rule-flow, fees are assigned for any specially requested actions and the output data will be returned in the FeeResponse object.

After the separation by vehicle type was clearly defined, the separation by fee type was next to be developed. Through careful analysis of the fee computation business requirements, three fee types are identified: transaction fees, registration fees and request fees.

1. Transaction Fees are designated for different types of vehicle transactions being performed, like registering new vehicles (original), transferring to a new owner, or declaring a planned non-operation status. Service fees usually include rules for time constraints by which the fee must be paid; else a penalty is assessed for the delinquency.

2. Registration Fees include a registration schedule (generally annual) and get applied to the vehicle for each renewal period. Registration fees normally include penalties for delinquency and can also be collected for prior years up to a statute of limitations.
3. Request Fees are generated for special types of requests like substituting plates and stickers, duplicate registration cards, and when requesting expeditious processing. These fees may also be different according to vehicle type.

Fee types are separated within each vehicle specific rule-flow in order to provide a clear organization of the business and will also allow the segregation of rules according to the way legislation is normally passed by the state. When new fees are adopted into law, the type of fee will first be identified, after which the affected area of the rule-base can be easily identified and changed.

During the course of design, the understanding of the fee computation process was improved, in particular, the logic for registration renewal. After many discussions and brainstorming sessions, it was realized that the aforementioned fee request-response object model needs to be augmented with additional internal concepts to support the logic for registration renewal. In addition, there were some very important terms that needed to be exposed to the rule-base and were actually mentioned within the vehicle code and registration manual, but were never part of the legacy system. All this resulted in the concept of a registration schedule and some important terms that are used as part of the knowledge base (Registration Due Date, Registration Liability Date, Registration Period, Pre-Registration Period, Statute Of Limitations, and Registration Lien Date). The challenge is to define a process that not only generates the appropriate due dates for the entire registration schedule, but also incorporates enough flexibility to accommodate future modifications to the process. The concept of a registration schedule can be broken up into contiguous registration periods, which accommodate the supporting information on registration durations, alternate year mandates, prorating policies, and any other rules that may be applicable, for fee computation. After the registration schedule has been developed, a selection process is performed to determine how many registration periods are due, according to the statute of limitations. For vehicles with annual registration periods, this can potentially include the current registration plus 3 prior years. This design allows for flexible rule modifications to the date registration fees are due, the registration liability date, the amount of time for the pre-registration period, the amount of time for each registration period, the statute of limitations, the triggering of current and prior year fees, and the derivation of a new expiration date.

Once the object model and business process were defined, the rules within the process tasks were created. As knowledge of the fee computation process grew, rule-flows were enhanced and refined logically, term names were improved to represent their true business meaning, and the object model was modified for clarity and efficiency.

3.3. Knowledge Acquisition through Rule Mining
For the knowledge acquisition process, two main pieces of business documentation specific to fee computation include:
1. State of California Vehicle Code,
2. California Department of Motor Vehicles Registration Manual.

Knowledge acquisition also involved the use of a technique called rule mining [18]. Rule mining is a process by which an existing procedural program is analyzed and the business rules are discovered, extracted, and documented. An effective method is to identify program output data and perform searches within the program for statements that modify this data. In turn, any interim data elements that are used in the modification of the output data are also researched. Eventually, the statements are traced back to the input object model. The resulting list of statements contains all of the program code that is used to develop a single output field. This process is called “program slicing” and was used to help start the process of rule gathering. After program slicing was performed, the business users are often able to help determine the true business meaning that supports the implementation in the corresponding program code.

An example of the rule mining is presented here for the extraction of all rules that influence the assignment of the smog code. The smog code is used to designate the necessity of a vehicle smog check, which is charged during vehicle registration. Following are some sections of the existing Cobol program for fee computation. The bold-faced lines are related to the smog code.

**DRIVER MODULE:**

```
IF SPECIAL-EQUIPMENT
  PERFORM 2800-RATE-SPEC-EQUIP
  MOVE L-5 TO SMOG-CODE
  GO TO 0200-EXIT-PROGRAM.

IF VESSEL
  PERFORM 2000-RATE-VESSEL THRU 2099-EXIT
  MOVE L-5 TO SMOG-CODE
ELSE
  IF OFF-HIGHWAY-VEHICLE
    PERFORM 3000-RATE-OHV THRU 3099-EXIT
    MOVE L-5 TO SMOG-CODE
  ELSE
    END-IF.
END-IF.
```

**SUB-MODULE:**

```
IF RP032C-LK-GR-POLLUTER-E79
  MOVE 01 TO RP032C-RET-SMOG-CODE
  GO TO 10400-SMOG-EXIT
END-IF.

IF RP032C-LK-SMOG-TV
  CONTINUE
ELSE
  MOVE 05 TO RP032C-RET-SMOG-CODE
  GO TO 10400-SMOG-EXIT
ENDIF
```

```
MOVE 05 TO RP032C-RET-SMOG-CODE
GO TO 10400-SMOG-EXIT
END-IF

SUB-MODULE:
IF RP032C-LK-EXP-DATE > 19900226
  IF RP032C-LK-NON-SMOG-MP
    MOVE 05 TO RP032C-RET-SMOG-CODE
    GO TO 10400-SMOG-EXIT
  END-IF
ELSE
  IF RP032C-LK-NON-SMOG-MP-OLD
    MOVE 05 TO RP032C-RET-SMOG-CODE
    GO TO 10400-SMOG-EXIT
  END-IF
END-IF
IF RP032C-LK-EXP-DATE > 19900226
  CONTINUE
ELSE
  IF RP032C-LK-COML
    IF RP032C-LK-WGT > '06500'
      MOVE 05 TO RP032C-RET-SMOG-CODE
      GO TO 10400-SMOG-EXIT
    END-IF
  END-IF
END-IF

The technical statements that pertain to the smog code can then be organized into a list of potential rules and propositions. It is important to maintain the proper truth value when dividing up the statements. The connectives of AND, OR, and NOT can be used to combine statements from the program. The following list contains the potential rules for derivation of the smog code.

POTENTIAL RULES:

IF SPECIAL-EQUIPMENT
  MOVE L-5 TO SMOG-CODE
IF VESSEL
  MOVE L-5 TO SMOG-CODE
IF OFF-HIGHWAY-VEHICLE
  MOVE L-5 TO SMOG-CODE
IF RP032C-LK-GR-POLLUTER-E79
  MOVE 01 TO RP032C-RET-SMOG-CODE
IF (NOT) RP032C-LK-SMOG-TV
  MOVE 05 TO RP032C-RET-SMOG-CODE
IF CCT-EFFCT-SMOG-DATE (WA-SMOG-CNTY-CD) = '99999999'
  MOVE 05 TO RP032C-RET-SMOG-CODE
IF RP032C-LK-EXP-DATE > 19900226
  (AND)
  RP032C-LK-NON-SMOG-MP
  MOVE 05 TO RP032C-RET-SMOG-CODE

IF RP032C-LK-EXP-DATE <= 19900226
  (AND)
  RP032C-LK-NON-SMOG-MP-OLD
  MOVE 05 TO RP032C-RET-SMOG-CODE

IF RP032C-LK-EXP-DATE > 19900226
  (AND)
  RP032C-LK-COML
  (AND)
  RP032C-LK-WGT > '06500'
  MOVE 05 TO RP032C-RET-SMOG-CODE

From the list of potential rules, the appropriate business terms are matched to the corresponding program variables. These can be replaced into the above logic to begin development of formal business rules. Discussion with the business experts is often necessary to ensure that the terms are being interpreted appropriately. In addition, the constant values for the smog code are defined as business terms. The following list contains the matches between program variables and their corresponding business terms.

<table>
<thead>
<tr>
<th>Program Variable</th>
<th>Business Term or Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMOG-CODE</td>
<td>Smog code</td>
</tr>
<tr>
<td>RP032C-RET-SMOG-CODE</td>
<td>Smog code</td>
</tr>
<tr>
<td>RP032C-LK-GR-POLLUTER-E79</td>
<td>Vehicle has been designated as a gross polluter</td>
</tr>
<tr>
<td>RP032C-LK-SMOG-TV</td>
<td>Vehicle's type does not require smog check</td>
</tr>
<tr>
<td>RP032C-LK-EXP-DATE</td>
<td>Expiration date</td>
</tr>
<tr>
<td>RP032C-LK-NON-SMOG-MP-OLD</td>
<td>Vehicle's motive power does not require smog check (prior to 1990)</td>
</tr>
<tr>
<td>RP032C-LK-NON-SMOG-MP</td>
<td>Vehicle’s motive power does not require smog check (after 1990)</td>
</tr>
<tr>
<td>RP032C-LK-COML</td>
<td>Commercial vehicle</td>
</tr>
<tr>
<td>RP032C-LK-WGT</td>
<td>Vehicle weight</td>
</tr>
</tbody>
</table>

Also, the smog code itself can be further defined by its constant value. A smog code of ‘1’ means that the vehicle requires a smog check, while a value of ‘5’ designates that the vehicle is exempted from a smog check. Finally, the information above allows the business rules for the smog code to be stated as follows.

**BUSINESS RULES:**

if the vehicle is special equipment
  then the vehicle is exempt from smog check

if the vehicle is a vessel
  then the vehicle is exempt from smog check

if the vehicle is designated as off-highway
then the vehicle is exempt from smog check

if the vehicle’s type is not listed to require smog check
then the vehicle is exempt from smog check

if the vehicle’s registration expiration date contains all 9’s
then the vehicle is exempt from smog check

if the vehicle’s registration expires after February 26, 1990
and the vehicle’s motive power does not require a smog check
then the vehicle is exempt from smog check

if the vehicle’s registration expires on or before February 26, 1990
and the vehicle’s motive power does not require a smog check
then the vehicle is exempt from smog check

if the vehicle’s registration expires after February 26, 1990
and it is a commercial vehicle with gross weight over 6500 pounds
then the vehicle is exempt from smog check

if the vehicle is designated as a gross polluter
then the vehicle requires a smog check

The deliverables from this stage of the design are business rule design documents that include the rule-flow and design diagrams along with the rules organized into rule-sets and documented in natural business language. Table 1 offers a run down on different business rule-sets for vessel fee computation.

Table 1. Business rule-sets for vessel fee computation.

<table>
<thead>
<tr>
<th>Rule set</th>
<th>Number of business rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fee Computation Facts</td>
<td>25</td>
</tr>
<tr>
<td>Return Facts</td>
<td>3</td>
</tr>
<tr>
<td>Vessel Facts</td>
<td>6</td>
</tr>
<tr>
<td>Vessel Service Fees</td>
<td>7</td>
</tr>
<tr>
<td>Vessel Use Tax</td>
<td>15</td>
</tr>
<tr>
<td>Vessel Registration Dates</td>
<td>7</td>
</tr>
<tr>
<td>Vessel Registration Period</td>
<td>2</td>
</tr>
<tr>
<td>Vessel Registration Schedule</td>
<td>5</td>
</tr>
<tr>
<td>Vessel Statute of Limitations</td>
<td>5</td>
</tr>
<tr>
<td>Vessel Current Penalty</td>
<td>5</td>
</tr>
<tr>
<td>Vessel Registration Fees</td>
<td>8</td>
</tr>
<tr>
<td>Vessel Annual Exempt Fees</td>
<td>2</td>
</tr>
<tr>
<td>Vessel Request Fees</td>
<td>5</td>
</tr>
<tr>
<td>Vessel Rounding</td>
<td>3</td>
</tr>
</tbody>
</table>

3.4. Deployment of Rule Service

The runtime application of our prototype system has been designed as an Enterprise Java Bean (EJB) on the server side of a multi-tier environment, and is to be offered as a service within a J2EE application
server [25]. This allows the business logic to be accessible by both the web clients and the application clients of J2EE.

Since Blaze Advisor comes pre-packaged with deployment capabilities that turn any rule service into an EJB service, we choose Blaze Advisor’s deployment model for the project. Figure 3 is a high-level diagram for the main components of the fee computation web application.

In this web application, a client can invoke the rule service through a simple method call to an EJB, passing the input object (FeeRequest) and receiving back the output object (FeeResponse). The actual implementation of the service is hidden from the client. This allows the rule-base to change without the knowledge of the clients, and in fact, a completely different implementation of the business logic can be deployed, as long as the contract for the EJB method call remains constant.

A web client can invoke the EJB rule service through a server-side standard component called a Java Servlet. The servlet acts as a controller in the Model-View-Controller (MVC) design pattern and is named FeeController for this purpose. When the FeeController servlet is initialized, during the first invocation of the doPost() method, it will connect to the rule service EJB by first obtaining the EJB Home object, which will subsequently create a local instance (an EJB Object) of the Fee Computation Bean. The FeeController will then accept an HTTP request object, containing the data necessary to build a FeeRequest object, and performs a method call to the EJB rule service. Upon return, the FeeController
will format the FeeResponse object back into an HTTP response object and post the response back to the web browser client, which then displays the information to the user.

On the other hand, an application client can invoke the EBJ rule service by (a) connecting to the J2EE server and obtaining an EJB Home object to create its local instance of the Fee Computation Bean, and (b) building a FeeRequest object and performing a remote method call to the EJB rule service. Upon return, the application client can process the FeeResponse object in the manner desired.

The Blaze Advisor deployment positions the rule-base for changing business requirements by allowing an update of the rule-base to occur in a separate repository within its own component of the application. In fact, an update to the rule-base can occur without any changes to the client application, servlet, or EJB. The rule-base is stored in a repository file that gets loaded when the EJB is initialized. Advisor Builder and Advisor Innovator are two tools that allow direct update and deployment of the rule-base.

Finally, XML has been used in both the web clients (to communicate the response information returned from the FeeController servlet back to the web browser) and the application clients (to translate the request and response information). Blaze Advisor provides the support to manage rule-bases in an XML format compatible with its rule language. A formal XML schema has been developed for the FeeRequest and FeeResponse objects. These schemas will be used when the data is serialized into XML documents. The documents are then easily processed by Java components using a Java XML parser.

4. PROTOTYPE SYSTEM

In this section, we describe several issues in our prototype system development. Because the tool we used is the Blaze Advisor (Advisor Builder for rule-base development and testing, Advisor Server for the deployment runtime, and Enterprise JavaBean, or EJB for the deployment option) [8], the experience here only reflects its perspective.

4.1. Rule-Base in Terms of Rule-Flows and Rule-Set

A rule-base is implemented within the Advisor Builder in terms of the concepts of rule-flow and rule-set. A rule-flow consists of sub-rule-flows and tasks, and causal relationships among them. Tasks can be defined as either rule-sets or functions. A rule-set is a group of related rules for a particular task. Thus, a rule-flow captures not only the static aspect of a rule-base, i.e., a collection of rules, but also the dynamic structure in terms of the causal relationships among rules.

To begin the development of the fee computation rule-base, an Advisor project was created in Advisor Builder with no rules or other entities. From this point forward, a project can be manipulated using the graphical interface to add and edit classes, objects, rule-flows, rule-sets, rules, functions, variables, and patterns. Figure 4 illustrates sections of the vehicle registration fee computation rule-flow
(the boxes represent tasks and sub-rule-flows of the process and the icons within them represent their types: a rule-flow, a function or a rule-set). Figure 5 is the vessel rule-flow, an expanded view of the sixth box from the left at the bottom of Figure 4. The two shaded boxes in Figure 5 are sub-rule-flows. Working with rule-flows within Advisor Builder is much like working with a diagramming or flowcharting tool. Tasks are dragged-and-dropped onto the rule-flow graph and an entire business process can be generated quickly.

![Vehicle Registration Fee Rule-flow](image)

Figure 4. Vehicle Registration Fee Rule-flow.

The Blaze Advisor rules engine development environment uses its own proprietary programming language called Structured Rule Language (SRL) to define the rule-base and automatically generates the SRL code for the rule-base (see an example in Subsection 6.1.). Knowledge engineer can either interact with Advisor Builder’s graphical interface or work directly with the SRL source code.

When a test of the rule-base is performed (see Section 5), execution begins at the start of the rule-flow. As tasks in the rule-flow are encountered, the implementation of a task is performed. If the task implementation is a function, it is executed in the same manner as a procedural function or sub-routine. If the task implementation is a rule-set, the working memory and pattern network for the Rete algorithm [9] is created with the elements and patterns that exist within the rule-set. The evaluation stage is performed and rules are added to the agenda. After evaluation is complete, a selected rule is fired and the working memory is updated. This process continues until the agenda is clear for the current rule-set. Processing
then continues with the next task in the rule-flow. When the entire rule-flow has been processed, the results can be displayed to the user or returned to the calling module.

Figure 5. Vessel Rule-Flow.

Figure 6 shows the FeeComputationFacts rule-set within the Advisor project. The first three rules of the rule-set are expanded to reveal the English-like syntax of SRL and how syntax highlighting is used to easily recognize keywords within the rule construction. Changing or updating a rule-set is very easy in the graphical interface. The panel in the upper left corner shows how rule-flows and rule-sets are added to the project list as they are defined.

4.2. Object Model Creation

A JavaBean is a class that follows a specific set of implementation constraints in order to provide consistent functionality as a Java component that can then be easily manipulated by other systems. Specifically, a JavaBean has the following properties [1-3]: (a) It is a public class, (b) It contains a public constructor with no arguments (self-initialization), and (c) It contains public access methods (getXXX and setXXX) for each property.

The Blaze Advisor rules engine utilizes JavaBeans to interact with data that will be manipulated outside of the rules engine. For instance, all objects that are passed from a Java application into the rules
engine, are passed as JavaBeans, so that the rules engine can manipulate the data using the public accessor methods. Similarly, all output data that will be returned back to the calling application, will also be passed back as JavaBeans. In order to create the object model for the application, a Java package structure was first defined as follows.

- `edu.csus.ejb.feeserv` - fee computation server (Enterprise JavaBean),
- `edu.csus.feecomp` - fee computation objects (JavaBeans),
- `edu.csus.feetest` - fee computation test clients,
- `edu.csus.servlet` - fee controller servlet,
- `edu.csus.util` - utility classes.

![Image of a software development environment](image)

**Figure 6. Fee Computation Rule-Set Example.**

Since the fee computation system is designed as a request/response system, the FeeRequest and FeeResponse classes are defined as composite objects that will hold all of the data objects for input and output respectively as shown below. The data objects are implemented as JavaBeans and can be manipulated within the rule-base.

- `edu.csus.feecomp.FeeRequest` (composite input class)
  - `edu.csus.feecomp.FeeTransaction`
  - `edu.csus.feecomp.FeeVehicle`
  - `edu.csus.feecomp.FeeOwner`

- `edu.csus.feecomp.FeeResponse` (composite output class)
  - `edu.csus.feecomp.FeeResult`
  - `edu.csus.feecomp.Fee` (list of fees)
Java classes are imported into the Advisor project using the Advisor Builder graphical user interface. In addition to the imported classes, it was necessary to define internal classes for use within the rule-base.

4.3. Rule-Base Deployment

To integrate the rules engine as part of an enterprise solution, the rule-base must be deployed into a Java application environment. There are several ways to develop a runtime version of the rule-base and here we choose to deploy the fee computation rule-base as an EJB in a J2EE Application Server. This is done through a package called Advisor Server in the Blaze Advisor suite of tools, which is a deployment runtime system to expose the rule-base to Java application clients and has support for multiple threading and concurrency control through the concept of a “Rule Agent”. Following are steps for the J2EE deployment of our fee computation application.

1. EJB Deployment Server Configuration.
2. HTML and Servlet Creation.
3. Java Classes Compilation.
5. Enterprise Application Archive.

Due to space limitations, we will not elaborate on those six steps. For details, readers may refer to [7].

5. PERFORMANCE EVALUATION

Three test transactions have been executed each using a different environment (in the Advisor Testing environment, as a web client, and as a Java application client). The performance evaluation of each test transaction is described in the subsections below.

5.1. Advisor Builder Testing

The Advisor Builder development platform not only provides a graphical user interface for developing a rule-base, but also includes the ability to run test transactions, and even a full suite of debugging capabilities that are common with most programming language development tools. This includes defining input test objects, stepping through rules execution, watching term values, setting breakpoints, and printing output to the screen. Advisor Builder can generate many performance reports. Figure 7 shows one such report on execution paths. It captures the execution performance of a test case for the vessel fee computation and indicates an inference chain of 9 levels with the execution time at each level.

---

1 In addition to the server configuration file, there are a host of other files that are generated during this deployment process. These include: EJB classes and interfaces, compilation and build scripts, and test execution scripts.
The Advisor Builder development platform also provides some reporting and monitoring tools to help determine the correctness, completeness, and performance of the rule-base. Information is generated for every rule-flow, rule-set, function, object, and variable within the project and how they relate to each other. A conflict analysis is also generated to show potential errors within the rule-base. The performance reports will display statistical information for the following static and dynamic categories: number of rules, number of entities considered in the rules, maximum depth of rule-flows, total number of properties tested in the rules, total number of rules fired, and total number of predicates evaluated.

5.2. Web Client

The web-based deployment of the fee computation rule-base was the main goal of this project and it has been successfully implemented within Sun Microsystem’s J2EE Reference Implementation, a Java application server. The J2EE server was installed to a desktop machine running Microsoft Windows XP. The fee application was installed into the server and a URL was created for public access through the web server. In order to execute the web transaction, the following steps are performed:

Open a standard web browser (like Microsoft Internet Explorer) and go to the service providing URL. The main page includes a choice of Automobile, Motorcycle, or Vessel fee computation. As of this date, only the Vessel rule-base has been implemented. Selecting the Vessel link will open a form to enter transaction, vehicle, and owner information. When the appropriate information has been entered, click the...
“Calculate Fees” button at the bottom of the form. Once the server machine receives the HTTP request from the client, the J2EE server will process the incoming request using the FeeController servlet. The FeeController servlet extracts the form data, populates a FeeRequest, and invokes the FeeComputationBean, which is the EJB implementation of the fee computation rule-base. The FeeComputationBean builds the initial Rete search network and starts the inference engine. Upon completion, the FeeResponse object will be populated. The FeeController accepts the FeeResponse object and generates the serialized XML representation of this object. This is returned to the client’s browser. Upon receiving the XML response, the xsl-stylesheet reference downloads the XSL document for transformation processing. The web browser’s built-in XSLT processor transforms the XML FeeResponse into HTML and displays the results to the user.

Figure 8 shows the first of the input forms for the same test case as was executed with the Advisor Builder environment in the Subsection 5.1. The data are manually entered in the web browser form and Calculate Fees button is clicked to send the entered data to the server. Figure 9 displays the results.

The performance of the web application has a response time within one second. The rules engine itself consumes two hundred to three hundred milliseconds, and the network traffic can affect the final response time.

There is a startup cost associated with the rules engine in the form of loading external and internal table information and setting up the Rete network. This generally takes about fifteen seconds during the first invocation of the EJB. Once the EJB is initialized, it remains in memory for the EJB container to process future requests.

5.3. Java Application Client

The Java application client was an extra implementation component for this project that shows the advantages of deploying an application as an EJB. The web application provides a user interface to the rules service, while the Java application provides a batch or program interface to the same rules service.

This allows the deployment of a business solution to be accessible and consistent through many different means of communication. To demonstrate this type of client, a Java class was written that would read the request data from an external file, invoke the rules application, and write the response data back to an output file. In order to execute the Java transaction, the following steps are performed:

The input data that is required needs to be formatted into an XML representation of a FeeRequest object and saved to an external file. The batch script entitled FeeTester.bat is executed. The script runs the FeeTester Java client, which parses the input XML file into a FeeRequest object. The Java client invokes the FeeComputationBean, which is the EJB implementation of the fee computation rule-base. The FeeComputationBean will build the initial Rete search network and start the inference engine. Upon
completion, the FeeResponse object is populated. The Java client accepts the FeeResponse object and generates the same serialized XML representation of this object. This is written to an output file.

Figure 8. Web Application Input Form (page 1).

Figure 9. Web Application Output Display.
The application client can achieve a better performance than the web client, since it performs a direct access to the EJB and bypasses the FeeController servlet. Again, this application client could be used to interface with other areas of an existing system and provide access to the same rule-base that is used over the Internet.

6. COMPARISON TO JESS IMPLEMENTATION

There are many tools available in the field, each having its set of features. To see how tools impact on the system development, we choose, as part of our study, to use JESS to implement the same vessel fee computation rule-base for comparison purpose. The reason we select JESS is its availability. In this section, we compare the Blaze Advisor rules engine implementation with the JESS implementation in the following three aspects: rule-base development, rule-base deployment, and testing. Some useful conclusions have been obtained as a result of the comparison.

6.1. Rule-Base Development

Since both Blaze Advisor and JESS are Java-based implementations of the Rete Algorithm, the interfaces and deployment possibilities are similar. However, the rule syntax for the Advisor is based on their own proprietary language (SRL), while JESS accepts the syntax of the CLIPS expert system shell. These two languages are actually very different in their approach to representing declarative rules and asserting/retracting facts as the inference engine executes its processes.

One of the issues to address in the translation from the Advisor to JESS was to determine how external Java objects would be integrated or interfaced with the expert system. Java classes that are defined outside of the rules are imported into the Advisor Builder using features within the tool. Once imported, their public members may then be referenced directly within the rules in the same fashion that internal classes and objects are referenced.

For JESS, the process for linking external classes into the expert system is called “binding”. A local or global variable is bound to a class instance and all class members are available using specially defined constructs for JESS. However, in order to use these objects within the rule-base in the same manner as other objects, variables, and atoms, a special construct called “definstance” must be used, which will generate a template for a fact within the knowledge base that links the external object to the fact. This is the only way to gain the ability to use members of an object in the antecedent of a rule.

Following are two entry point functions from the Advisor rule-base and the JESS rule-base, respectively. Each function will first obtain the FeeTransaction, FeeVehicle, and FeeOwner objects from the composite object FeeRequest, by calling its “get” accessor method. Within the Advisor, simply accessing the member using the dot notation, implies that the “get” accessor is called. There is an additional step within JESS to define an instance fact for each object being bound to the rule-base. The
rules are then executed and after completion, a new FeeResponse object is created and the FeeResult object, which is populated during rules execution, becomes its primary member. The FeeResponse is then returned to the calling module.

**Advisor (SRL) Entry Point:**

```plaintext
function VRFees for {request: a FeeRequest} returning a FeeResponse is
{
feeTransaction = request.feeTransaction.
feeVehicle = request.feeVehicle.
feeOwner = request.feeOwner.
executeAgent().
response is a FeeResponse.
response.feeResult = feeResult.
return response.
}
```

**JESS (CLIPS) Entry Point:**

```plaintext
(deffunction VRFees (?request)

(bind ?*feeTransaction* (call ?request getFeeTransaction))
(definstance feeTransaction ?*feeTransaction* static)

(bind ?*feeVehicle* (call ?request getFeeVehicle))
(definstance feeVehicle ?*feeVehicle* static)

(bind ?*feeOwner* (call ?request getFeeOwner))
(definstance feeOwner ?*feeOwner* static)

(run)

(bind ?response (new edu.csus.FeeResponse))
(call ?response setFeeResult ?*feeResult*)

(return ?response)
```

Another important feature of the Advisor rule-base that needed to be converted to its JESS equivalent was the creation of internal objects that assist with fee computation. In particular, the FeeTable is an object that stores fee amounts, descriptions, and effective dates for the specific fee codes used by vehicle registration. During the process of fee computation, if a certain fee qualifies for assignment, the proper amount must be determined from the FeeTable.

For the Advisor implementation, this table was easily set up as an association table, which is similar to a Java hash table. When a fee code qualifies for assignment, a function is used to examine the FeeTable object and return the appropriate fee. For the JESS implementation, it was easier to define the entire table as a set of rules that are pattern matched by fee code. The appropriate FeeTable entry is found by using the inference engine, instead of a function call.
Once the method was defined for importing and binding Java objects, the next step was to develop the business process and organization of rules within JESS. This requires similar functionality that is represented by rule-flows and rule-sets within the Advisor. After careful consideration of various solutions, one alternative is to use the JESS constructs for module and focus. “A module defines a namespace for templates and rules” [10]. Using the “defmodule” construct, the rules can be grouped into their corresponding rule-sets from the Advisor. Following is the contrast between an Advisor rule-set and a JESS rule-set.

**Advisor (SRL) Ruleset:**

```plaintext
ruleset FeeComputationFacts is
{
  rule NewTypeLicenseVessel is
  if feeVehicle.typeLicense is ("V1" or "V2" or "V3" or "V4"
                               or "V5" or "V6" or "V7")
  then vessel = true.

  rule NewTypeLicenseVesselAnnualExempt is
  if feeVehicle.typeLicense is ("V6" or "V7")
  then annualExempt = true.

  rule PreviousTypeLicenseCategories is
  if feeVehicle.typeLicensePrevious is ("V6" or "V7")
  then priorAnnualExempt = true.
}
```

**Jess (CLIPS) Module:**

```plaintext
(defmodule FeeComputationFacts)

(defrule NewTypeLicenseVessel
  (feeVehicle (typeLicense "V1" | "V2" | "V3" | "V4"
                   | "V5" | "V6" | "V7"))
  =>
  (assert (vessel)))

(defrule NewTypeLicenseVesselAnnualExempt
  (feeVehicle (typeLicense "V6" | "V7"))
  =>
  (assert (annualExempt)))

(defrule PreviousTypeLicenseCategories
  (feeVehicle (typeLicensePrevious "V6" | "V7"))
  =>
  (assert (priorAnnualExempt)))
```
JESS maintains a focus stack, which is a list of modules that can be executed. The top of the stack represents the current focus and the order of execution can be maintained through pushing and popping modules with regard to the stack.

The rule-flow and rule-set organization has been much easier to define within the Advisor by using a graphical representation of rule-flow tasks and associating the appropriate rule-sets for the implementation of each task. In JESS, on the other hand, these features currently require a manual construction using a collection of JESS constructs.

Some of the other challenges encountered with the JESS rule-base development included date and currency manipulation. However, these difficulties arose from using features within the Advisor that are not part of the JESS shell.

6.2. Rule-Base Deployment

Once again, with both the Advisor and JESS being Java-based implementations of the Rete Algorithm, the interfaces to the rules are very similar, and in fact, identical in places. For the Advisor implementation, two application interfaces were developed as mentioned in Section 5. For the JESS implementation, the HTML pages, the Java Servlet, and the Java client use the same source code. Only the business method of the EJB requires modification for the JESS rule-base execution.

Advisor Builder supports the ability to generate much of the deployment code necessary for installation of the rule-base as an EJB within a J2EE application server. This includes the following files:

- Java Home Interface
- Java Remote Interface
- Java Bean Implementation
- XML Server Configuration

The server configuration file specifies different options and features that control aspects of the rule server, for instance how to handle multiple concurrency, the policy for object instantiation and persistence, the defined entry points to the rule service, and other factors that are necessary for Advisor execution. Also within this configuration is the location and access method for the rules themselves. The Advisor allows the deployment of the text-based rules files that are used during rules development or a compiled binary version of the entire rule-base. For our system, the rules were compiled into a single binary file.

For the JESS implementation, it was desired that the rules be compiled into a single deployment file, similar to the Advisor deployment. JESS provides a mechanism for saving and loading the entire state of a rule-base as binary files by using the “bsave” and “bload” constructs. This allows a single binary rules file to be loaded upon startup of the rule service. Following is the comparison of the Advisor and JESS EJB deployment descriptors.
Advisor Deployment Descriptor:

```xml
<jboss-jar>
  <display-name>FeeComputationEJB</display-name>
  <enterprise-beans>
    <session>
      <display-name>FeeComputationBean</display-name>
      <ejb-name>FeeComputationBean</ejb-name>
      <home>edu.csus.ejb.feeserv.FeeComputationHome</home>
      <remote>edu.csus.ejb.feeserv.FeeComputation</remote>
      <ejb-class>edu.csus.ejb.feeserv.FeeComputationBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
        <env-entry-name>ndconfig.server</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>
          http://localhost:8000/advisor/FeeComputation.server
        </env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
</jboss-jar>
```

Jess Deployment Descriptor:

```xml
<jboss-jar>
  <display-name>JessComputationEJB</display-name>
  <enterprise-beans>
    <session>
      <display-name>JessComputationBean</display-name>
      <ejb-name>JessComputationBean</ejb-name>
      <home>edu.csus.ejb.jesserv.JessComputationHome</home>
      <remote>edu.csus.ejb.jesserv.JessComputation</remote>
      <ejb-class>edu.csus.ejb.jesserv.JessComputationBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <security-identity>
        <description></description>
        <use-caller-identity></use-caller-identity>
      </security-identity>
    </session>
  </enterprise-beans>
</jboss-jar>
```

The comparison of the Advisor and JESS EJB business methods is given below.

Advisor EJB Business Method:

```java
public FeeResponse invokeVRFees(FeeRequest request) throws EJBEException {
  try {
```
Object[] applicationArgs = new Object[1];
applicationArgs[0] = request;

// Invoke the entry point
FeeResponse response = (FeeResponse)
    invokeService("FeeComputation", "invokeVRFees",
                 null, applicationArgs);

    return response;
}
catch (Exception e) {
    e.printStackTrace();
    throw new EJBException(e);
}
}

Jess EJB Business Method:

public FeeResponse invokeVRFees(FeeRequest request) throws EJBException {
    try {
        Rete engine = new Rete();
        engine.bload(new
                URL("http://localhost:8000/jess/JessComputation.clb")
                .openStream());

        engine.definstance("?*feeRequest*", request, false);

        // Invoke the entry point and return the result
        return (FeeResponse)
                engine.executeCommand("(VRFees ?*feeRequest*)")
                .externalAddressValue(engine.getGlobalContext());
    }
    catch (JessException je) {
        je.printStackTrace();
        throw new EJBException(je);
    }
    catch (Exception e) {
        e.printStackTrace();
        throw new EJBException(e);
    }
}

The loading of the Advisor rules file is specified within the deployment descriptor and is
automatically loaded through the environment variable ndconfig.server. However, this
functionality needed to be explicitly created for JESS, by loading the rules files during initialization of the
bean using the “bload” construct. Both EJBs implement the following steps in different ways: loading
the rules, passing the input objects, executing the rules, and returning the output objects.

Since the Rete object for the JESS rules engine cannot be persisted inside a Stateless Session bean, it
must be created with each invocation of the business method. This may introduce a performance issue and
can be improved upon in future implementations by creating a pool of pre-loaded rule services that can be
accessed by the EJB. This is the underlying design of the Advisor implementation that is hidden from the rule application developers.

6.3. Testing
Some final conclusions on the comparison between the Advisor and JESS involve the ease of testing and deployment using the Advisor development environment versus the manual effort needed to perform the same functions using JESS.

Recalling from subsection 5.1, it is relatively easy to set up test cases and execute unit tests within the Advisor Builder platform. These features do not exist with the JESS environment. In JESS, the only way to perform debugging of rules is to place print statements within the rules in order to provide a runtime display of the firing order.

There has been some work within the JESS user community to define certain add-on features, such as JSP taglibs, XML integration, syntax files for test editors, and even a Graphical User Interface for working with JESS, called JessWin. However, this interface is rudimentary and does not include any debugging facilities nor does it provide for graphical interaction with the rule-base. While these features may seem extraneous to the base concept of an expert system, the lack of this functionality will always hinder the acceptance of JESS as a viable solution for business driven solutions.

Finally, the automated deployment generation that offers many options for the Advisor could also be implemented for JESS, however none of it exists today. Businesses require the ability to deploy their solutions to varying platforms and tier structures and with different interface requirements. Some additional work could be pursued to provide these different deployment options as a standard set of library classes within the JESS package. This would allow for a quicker implementation of tested methods for loading rule-base files, interacting with XML data, creation of Stateful and Stateless session beans, and integration with standard messaging API’s. In fact, all of the standard J2EE interfaces could be offered along with the newly proposed JSR-94 specification for a standard rules engine API.

7. CONCLUSIONS AND FUTURE WORK
In this paper, we describe our experience in capturing the business logic and implementing a web-based expert system for vehicle registration fee computation at California Department of Motor Vehicles. A prototype system has been implemented using the Blaze Advisor rules engine and has been deployed to Sun’s J2EE Reference Implementation of a Java application server. Our results have shown that using expert system technology to implement a web-based, business-driven solution is certainly a viable and promising option. Deployment of the rule service as an EJB exposes the rule-base to various types of business clients, and provides a consistent and adaptable implementation of the business domain.
There are several lessons learned. First, the methodology of discovering, defining, and organizing the business rules became extremely important. By not applying a business rule approach, it is possible to implement a rule-base that is not easily adaptable to a changing business world and that may not accurately represent the true business domain. By following a formal business-driven methodology, the rule-base organization was easier to develop and the rules were easily traced to existing business documentation. Reducing the time spent translating the legislation and policies that govern a business into the automated system that runs the business, is a key benefit of creating an effective rule-based system. This will allow a company to rapidly respond to changing business requirements and maintain a competitive edge in the marketplace.

Lesson two is the need of tools and development environments that are geared toward multiple communication or interface strategies and for legacy systems integration. Tools need to be graphical or model-driven, have complete testing and debugging environments and conflict and performance analysis facilities, support automated deployment, and have the ability to generate rule maintenance applications that can be exposed to business users over the Internet.

Lesson three is the creation of a process for drafting, defining, and accepting formal standards and specifications. The success of Open Systems Architectures (OSA) relies heavily on the participation of industry leading experts to propose and refine standards and to ensure that the vendor population will provide compliant solutions.

Possible future work includes: a system for complete vehicle registration. This project focused on a sub-section (vessels) of the complete business application for generating the appropriate fees for vehicle registration. A natural extension would be to include developing the rule-base for all vehicle types (automobiles, motorcycles, commercial vehicles, trailers, off highway vehicles, and so forth).

Some more general issues for future work include: development of business rule maintenance capability that is accessible through the Internet, a standard XML schema for business rule representation, and a web-based “rule service” that receives both a rule-base definition and a rule-base query, and returns the results of the service back to the user, over the Internet.

REFERENCES


