A business rule explanation system for web services

Keping, J.

*A Business Rule Explanation System for Web Services **

Keping, J.

May 2003

* published in MCS Thesis of University of New Brunswick. 96 pages. Fredericton, New Brunswick, Canada. May 26, 2003. NRC 46527.

A Business Rule Explanation System for Web Services

by

Keping Jia

MEng – Jilin University of Technology

A Thesis Submitted in Partial Fulfilment of
the Requirements for the Degree of

Master of Computer Science

in the

Graduate Academic Unit of Computer Science

THE UNIVERSITY OF NEW BRUNSWICK

April, 2003

Abstract

Electronic commerce has grown rapidly and brought about enormous change in business firms, markets and consumer behavior. To meet the increased demand, more and more business activities are moving to the World Wide Web because business activities can be reached everywhere and be performed more efficiently.

Business rules technology is one of the most active research areas in e-commerce. It deals with representing and processing regulations and policies regarding how an enterprise conducts its business so that business activities can be carried out electronically. Abstracting business logic from the application procedures, business rule technology enables the fast development of applications that can be rapidly modified. Potentially e-business system based on business rules can explain themselves. The business rule explanation system of this thesis is a prototype system that provides a user with the justifications of the conclusions derived by a business rule system. The justifications are given in the form of the "explanation tree" that provides "how" dialogues to answer how a conclusion is derived, "why not" dialogues to identify the missing criteria for achieving a goal and "what if" dialogues to find out a complete set of preconditions that will be necessary to lead to the grant of a request.

For e-business systems that deliver services over the Internet, a distributed architecture is required because a business activity sometimes needs to involve different partners under different contexts. The prototype is thus built upon the emerging Web Services standards. The next part of the thesis describes how business rules and Web Services technology work together to deliver a loosely coupled, distributed business rule system.

Acknowledgements

I would like to express my gratitude and sincere thanks to my supervisor Dr. Bruce Spencer who has been guiding me through the whole course of the research and spending valuable time in reviewing and editing this thesis. I am also truly grateful to the other member of my supervisory committee, Dr. Ali A. Ghorbani, for his valuable suggestions and constructive criticism.

I also wish to extend my thanks to the members of the Examining Board. They are Dr. Harold Boley, Kirby Ward and Dr. Gregory J. Fleet. Their comments greatly improved the final version of this thesis in many ways.

Thanks also to Scott Buffet, Sandy Liu and Fang Wang who have always been helpful and encouraging, and to National Research Council for the financial assistance to this research work.

Finally, I will give my deepest love and thanks to my wife Qun. I will not accomplish anything without her love and support.

Table of Contents

Table of Figures

1 Introduction

In past few years, electronic commerce has grown rapidly and brought about enormous change in business firms, markets and consumer behavior. The rapid movement towards an e-commerce economy and society is being led by both established business firms and new entrepreneurial firms. More and more B2B (Business to Business) and B2C (Business to Customer) systems are established to facilitate the online businesses and transactions.

Generally speaking, a business system is a software system that is used to help organize a business's data and manage a business's daily activities. Hence, the function of a business system must comply with the business knowledge of the area in which the system is used. For example, before being used by a car dealership company, the business system needs to be first customized to adopt that company's conventions of doing business (e.g. An elite customer can get 5% discount on buying a car), regulatory policies of the industry to which that company is belonged (e.g. A 5 year warranty is mandatory for every new car) and other policies that may change with the marketplace (e.g. policies about financial assistance and auto loans).

In a traditional business system, the knowledge about the business behaviors and policies are embedded in software codes, DBMS (Data Base Management Systems) triggers, stored procedures and database objects. These different approaches inevitably scatter the business knowledge throughout the program and mix the processing of the business policies with other programming functions. As changing business policies and environment cause applications to evolve, it becomes harder and harder to modify these applications. Moreover, as these applications increase in size, the time needed to modify and maintain increases disproportionately [21].

The business rule technology is a new approach of embedding business knowledge into business systems that attracts the attention of current researchers. The main characteristic of the business rule technology is to have the business knowledge formally represented as rules and independently maintained in a knowledge base so that the business rules and the remainder of application functionality are completely separated. In this approach, the rule engine will carry out all the functions by choosing and executing the right rules at the right time in the right order. Abstracting the policies and knowledge of the business from the application procedures, business rules can be maintained and validated without affecting the rest of the application code. This makes business rule technology a perfect solution for fast development of applications that can be rapidly modified. It provides companies with a powerful competitive advantage, as they can be more responsive to changes within the company (internal policy changes), within the marketplace (products and pricing can be changed in response to customer demand and competitive pressures), and the industry (external changes in regulatory policies) [22].

Despite of the numerous research activities in this area [6, 9, 22, 29], none of them have given enough emphasis on the explanation part of the business rule system. However, the demand from the user for the explanation of the conclusions derived from the system is high in online businesses. The system needs to tell a client whether he/she satisfies all the relevant conditions to obtain a service. If a service is denied, it also needs to tell the user why it is denied by showing the user the rule(s) he/she failed to satisfy and how they could be satisfied. Developed from the business rules technology’s unique way of managing and processing policies, the business rule explanation system greatly enhances the maintenance and validation of the business rules. By providing both conclusion and justification, this explanation system also forms the basis of several features that are valuable in the e-business world. For example, a business system can guide prospective partners through interactions that allow them to establish credentials and meet eligibility criteria set by other partners [45] or provide interactions for a user to negotiate exchanges of information for service eligibility [23].

As part of this thesis, an explanation prototype system is implemented which is built on the technologies of the business rules and knowledge-based systems. The interactions between the user and the explanation system take the form of questions. The user first initiates the interaction by submitting a request to the system asking for a service that the system provides. Then, a series of questions will be arranged in order to evaluate the user’s request. The system picks questions based on the following three criteria:

1. The question is relevant to the user’s request.
2. The question belongs to the user’s domain of knowledge.
3. The question is relevant to the current context. In particular, the arrangement of the questions is also based on how the previous questions are answered.

The sequence in which the questions are asked is chosen according to their impact on the solution or reducing the search space of the task so that the interactions of this phase are kept to the minimum. Finally a conclusion of whether the request will be granted or not will be given. From this point, the user can ask for justifications of the conclusion through “how”, “why not” and “what if” questions. As to the “why not” question on a negative conclusion (the request is not granted), an explanation tree will be returned showing the preconditions on which the request can be granted. Some of these preconditions might have been met while the others were not. These preconditions (also called goals) form the nodes of the explanation tree. The user can ask “how” questions on the satisfied nodes to see what goals were achieved through achieving certain subgoals. The user can repeatedly ask how these subgoals were achieved. The user can also ask “why not” questions on any unsatisfied goals to find out what prevents such a goal from being derived. This interactive and repetitive process offers the user all the relevant situations and explanations on demand. During the interaction, a “what if” question could be asked at any time to see how other outcomes are affected if some unsatisfied conditions are assumed to be satisfied.

For example, suppose the user is interacting with a car dealership where the sales policy is governed by rules like

1. An elite customer can get 5% discount on decent or better cars if his payment type is “silver”.
2. “Silver” payment type is pay by 2 year installments with financial assistance of less than $10000.
3. “Silver” payment type is pay by 3 year installments with financial assistance of less than $7000.
4. The preferred customer who is insurance affiliated automatically becomes an elite customer.
5. The customer who buys car insurance in First Rate Co. is insurance affiliated.

If a user is not offered the discount she asks for, she can ask why not, and be given the entire list of rules that could, in principle, allow her to get this discount and be shown the conditions she does not meet. She could ask what would occur if some of these conditions were met, and so could interactively determine the complete list of what she needs to do to achieve her goal. A full interaction demo of this example will be given in Chapter 5.

The architectural aspect of the prototype design deals with problems of how to integrate the business rules technology into the surrounding e-commerce systems. Most of the current software practice for business rule system inherits the idea of component-oriented software design (e.g. COM, DCOM, J2EE) under a 3-tier or N-tier architecture. Because any system built on these protocols heavily depends on a particular language specification or a single vendor's implementation, it is not a perfect solution for business rule systems that may be deployed in a distributed environment and be shared by heterogeneous business applications. In this thesis, a service-oriented architecture – Web Services is discussed. Built on technologies and protocols such as HTTP, SOAP, UDDI and WSDL, Web Services architecture promises more flexibility and interoperability among distributed software components over the web.

In order to build a highly reusable and service-oriented prototype of the business rule explanation system, the architecture design will focus on the following aspects:

1. Design an architectural model to incorporate the business rule explanation system into Web Services architecture. This part deals with how to design the business rule explanation system under the Web Services framework and deploy

it as services that can be shared. In particular, the dual services design and the workload control will be discussed in detail.

2. Design an architectural model for the integration of rule-based Web Services. In some applications, a business rule system needs to be partitioned into several separate parts each of which is an autonomous service that maintains and processes its local rules. In this case, a universal mechanism is needed to link several rule services (when necessary) together for the purpose of query processing. In this thesis, we use predefined “configuration” rules to indicate the linkage between a particular goal and its processing rule service.

The next chapter is a background study of business rules, Expert Systems and Web Services, followed by Chapter 3, which discusses some concerns in designing the interaction modes between the user and the system. Chapter 4 emphasizes the architectural aspect of the prototype system. Chapter 5 presents an application example to illustrate how the explanation system works with the business rule system to provide explanations about the conclusions derived from the system. Conclusions and future work are discussed in Chapter 6.

2 Background

2.1 Business rules

Business rules are "statements that define or constrain some aspect of the business. They are intended to assert business structure or to control or influence the behaviour of the business" [21]. Business rules cover most of business activities that are regulated or guided by business strategy, tradition, culture, policy, and pragmatic experience. For a corporate computer system, business rules are the expression of business knowledge in a formal, accurate as well as computer- and human-understandable format. In traditional software practices, business rules are “buried in application program code, embedded in database structures, and coded as DBMS triggers and stored procedures” [21]. This type of approach entails a lot of problems in the software development and maintenance for two reasons. First, the system developers usually do not have expertise in the management and administration of a particular business. A huge amount of communication is needed between the business people, administrators and the system developers so that the architectural design can be carried out; many changes are also expected during the developing phase. Second, this approach causes costly maintenance later. It is hard to make any changes in the business rules if they are hard coded into the programs and interweaved with controls for other purposes like interface, data flow, data backup and error control. The maintenance problem of the traditional approach becomes more serious because businesses need to change their policy more frequently to cater for the ever-changing demand of customers.

Business rule technology provides a formal and general approach that separates business rules management from the overall control system and makes it possible for business people who have no technical expertise to create, maintain and modify business rules independently.

Business rule languages and rule management are two key elements of business rule technology. A business rule language should be concise and very readable so that statements are understandable, precise, unambiguous and can be managed systematically. The language should be expressive enough so that business rules can be easily described by it. Rule management deals with problems of how to manage and process business rules in a uniform way as well as to integrate them into core business systems.

A good business rule language and management system could greatly simplify the development of the overall system, shorten marketing time, and facilitate instant modification and rule sharing across enterprises.

2.2 Theoretical foundation of business rules

The fields of business rules and Expert Systems do overlap. Because “[r]ules have been used extensively as a way to represent knowledge” [20], the technology underlying Expert Systems is widely employed to automate business rules. These are two kinds of applications of logical reasoning systems, which are built on the theories of representation and reasoning [17].

2.2.1 Representation of knowledge

The object of knowledge representation is to express knowledge in a computer-processable form [37]. A representation of knowledge is defined by two aspects: a language and a semantics for this language.

Taking the diversity of the human’s knowledge into consideration, it is natural to think of making the representation language as expressive as natural languages (e.g. English). But natural languages will not work for a machine proving system. The reason is that a natural language is so rich that it is impossible to be described in a formal way, which means that natural language is not “machine understandable”. So, the syntax of the representation language has to meet the following criteria:

1. It must be expressive enough so that human knowledge can be represented with reasonable ease.
2. It must be unambiguous and context independent so that the knowledge is guaranteed to be exactly the same knowledge at anytime.
3. It must be concise in the sense that there should be an effective inference procedure that can respond within a reasonable time.

Provided the syntax and semantics are defined precisely enough that an inference scheme can be given, the language, together with its inference scheme, are called logic. Among the various existing logics, first order logic is most widely used in practical reasoning systems because of its expressiveness, completeness, consistency and the elegance of its inference procedures. The prototype system of this thesis uses first-order logic as the representation language.

2.2.2 First order logic

First order logic is one of the most important representation languages for automatic reasoning systems. It forms the basis of most representation schemes in Artificial Intelligence (AI). Just as any other language, first order logic is specified by its syntax and its semantics.

- Syntax

The syntax specifies the grammatical structure of the first order logic. It is composed of basic symbols and well-formed expressions.

The basic symbols are:

1. Constants: first order logic uses constants to represent a specific element in the domain of representation. In this thesis, it is represented by names beginning with lower case letters.
2. Variables: a variable potentially represents any element from the domain of representation. In this thesis, it is represented by names beginning with upper case letters.
3. connectives: connectives are predefined logical symbols that are used to construct complex sentences. Connectives used in first order logic are "$\neg$", "$\Rightarrow$", "$\wedge$", "$\vee$", "$\Leftrightarrow$".
4. Quantifiers: "$\forall$" and "$\exists$" are the only quantifiers defined in first order logic. While "$\forall$" denotes the entire body of objects in the universe, "$\exists$" denotes a non-empty subset of it.

5. Predicate: the predicate is used to denote a relation on the domain. It could be proposition letters (nullary relations), properties (unary relations) or n-ary relations.
6. Function: a function is a special kind of relation that relates or maps a tuple of elements to exactly one element in the domain. A function has one or more arguments.
7. punctuation symbols: "(", ")", ",".

Well-formed expressions consist of terms, atoms and formulae defined here as follows:

1. terms: variable and constant are terms; if *f* is a *n*-place function symbol and $t_1 \ldots t_n$ are terms, then $f(t_1, \ldots t_n)$ is a term.
2. Atomic formulae (Atoms): proposition letters are atomic formulae; if *R* is a *n*-place relation symbol and $t_1 \ldots t_n$ are terms, then $R(t_1, \ldots t_n)$ is an atomic formula.
3. Formulae: All atomic formulae are formulae; if *A* and *B* are formulae, then so are $\neg A$, $A \Rightarrow B$, $A \wedge B$, $A \vee B$, $A \Leftrightarrow B$, $\forall x\ A$ *and* $\exists x\ A$. A closed formula (also called sentence) is a formula that contains no free variables[1].

Using Backus-Naur Form, the syntax of first-order logic is shown in Figure 2-1.

[1] An occurrence of a variable in a formula is said to be free if it is neither in the scope of a quantifier with the same variable name, nor is the variable name of a quantifier [12].

```
Formula →AtomicFormula
             |  Formula Connective Formula
             |  Quantifier Variable Formula
             |  ¬ Formula
             |  (Formula)
AtomicFormula → Predicate(Term, …) | Term = Term
Term →Function(Term, …)
          | Constant
          | Variable

Connective →  ⇒ | ∧ | ∨ | ⇔
Quantifier  →  ∀  | ∃
Constant    →  a   | b | john | …
Variable    →  A | V | S | …
Predicate   →before | hasColor | raining| …
Function    →motherOf | leftLegOf | …
```

Figure 2-1 The Syntax Of The First-Order Logic [37]

Substitution is the process of the binding of terms to variables. A substitution usually takes the forms of a finite set of ordered pairs:

$$\{ v_1{:=}t_1,\ v_2{:=}t_2, \ldots\ v_n{:=}t_n\}$$

where $t_1,t_2,\ldots t_n$ is a list of terms and $v_1,v_2,\ldots v_n$ is a list of variables. The process of substitution will replace v_i with t_i in a formula or term in which v_i appears. The result formula/term of a substitution is called an instance of the original formula/term.

For most automatic reasoning systems, formulas are usually first converted into some kind of "normal form" so that they are easier to process mechanically. One of the normal forms of formulas is clause form. A clause takes the form of a disjunction of a finite set of atoms (positive literals) or negation of atoms (negative literals). The Horn clause format and the definite clause format are two very useful clause formats. A Horn clause is clause that has at most one positive literal. A definite clause is clause with exactly one positive literal. The Horn clause and the definite clause are widely used in the automatic reasoning systems because very effective proof methods can be applied.

- Semantics

The semantics of a logic gives meaning to any syntactically valid formula and relates the formulas to the domain. In order to describe the semantics of a formula, the notion of interpretation is introduced to denote the assignment of meanings to the symbols occurring in a formula. Every interpretation includes a non-empty set, *D*, which is called the domain of the interpretation. The elements in the domain are called values. An interpretation *I* of a language *L* is a mapping that makes following associations:

1. Associate a value of *D* with each constant symbol $c \in L$.

2. Associate a function $f_I : D^n \rightarrow D$ with each n-ary function symbol $f \in L$.

3. Associate a predicate $P_I \subseteq D^n$ with each predicate symbol $P \in L$

Based on above notions, the semantics of terms under an interpretation *I* can be defined as follows:

1. constant symbol: the semantics of a constant symbol *c* is the value from D assigned to *c*, which is presented as $I(c)=c_I$.

2. variable: the semantics of a free variable *v* is different for different assignments[2] of values of *D* to it. We denote the semantics of *v* under assignment A as $v^{[A]}$.

3. function: for the function that takes the form of $f(t_1,t_2,\ldots t_k)$ where $t_1,t_2,\ldots t_k$ are terms, the semantics of the function $I(f(t_1,t_2,\ldots t_k)^{[A]})= f_I(I(t_1^{[A]}),I(t_2^{[A]}),\ldots I(t_k^{[A]}))$

[2] An assignment is a mapping from the variables of a formula to a set of the values in D.

Similarly, the semantics of formulae under an interpretation *I* is defined as follows:

4. Atoms: for proposition letter *P, I(P)* is true iff P_I is true; for the n-ary (n>0) predicate $P(t_1,t_2,\dots t_k)$, $I(P(t_1,t_2,\dots t_k)^{[A]})$ is true iff $P_I(I(t_1^{[A]}), I(t_2^{[A]}),\dots I(t_k^{[A]}))$ is true.

5. Formulae: If ∂ and β are formulae, then

   $I((\neg\partial)^{[A]}) = (I(\partial^{[A]}))^c;$

   $I((\partial\Rightarrow B)^{[A]}) = I(\partial^{[A]}) \subseteq I(\beta^{[A]});$

   $I((\partial\wedge B)^{[A]}) = I(\partial^{[A]}) \cap I(\beta^{[A]});$

   $I((\partial\vee B)^{[A]}) = I(\partial^{[A]}) \cup I(\beta^{[A]});$

   $I((\partial\Leftrightarrow B)^{[A]}) = I(\partial^{[A]}) \equiv I(\beta^{[A]});$

   $I((\forall x\partial)^{[A]})$ is true iff $I(\partial^{[A]})$ is true for every assignment of the values from *D* to the free variable *x*.

   $I((\exists x\partial)^{[A]})$ is true iff $I(\partial^{[A]})$ is true for at least one assignment of the values from *D* to the free variable *x*.

If formula ∂ is true in the interpretation *I* for every assignment of the values from *D* to the free variables of ∂, then ∂ is said to be valid in *I* and *I* is said to be a model of ∂. For a set of formulae $S(\partial_1, \partial_2,\dots \partial_n)$ and a formula β, β is called a logical consequence of *S* if β is valid in every model of *S*. It is written $S \models \partial$.

It is worth noting that semantics of a language is not a necessary part for the inference procedure. In fact, an inference procedure can derive valid conclusions without knowing the interpretation of the sentences. But only with the semantics can the derived conclusions make meaningful claims about the intended world. [37]

2.2.3 Reasoning

Reasoning or inference is the mechanical process operating on the syntax level by which conclusions are reached. This process is carried out by a computer to implement the entailment relation between sentences. The most essential part of a reasoning process is the inference rules that decide how an inference procedure can be carried out mechanically.

For example, the inference rule *unit resolution* "$\partial \vee \beta, \neg\beta \vdash \partial$" allows the inference system to derive ∂ as long as "either ∂ or β is true" and "β is false" are held to be true.

An inference procedure derives new formulae from a given set of formulae according to inference rules. Let ∂ be the derived formula, S be the given formulae set and r be the inference rules, this could be written $S \vdash_r \partial$. If all the formulae that are derived from the inference procedure are logical consequence of the original formulae set, the inference rules will be called sound. An inference rule is complete if *for all* ∂ *that* $S \models \partial$, $S \vdash_r \partial$.

In 1930 and 1931, Kurt GÖdel proved for first order logic (defined in section 2.2.2) that we can find inference rules that allow a complete proof procedure [37,12]. In 1965, Robinson published the first single inference rule that by itself is complete – resolution, which gave the hope of building practical inference procedures [37]. There is very likely no polynomial-time complete inference procedure for first-order logic even when constrained to variable-free formulas. To create practical systems, people usually add some extra constraints on the formulas in order to achieve better inference speed. Horn clauses form a useful class of formulae for which a linear inference procedure exists when constrained to variable-free formula.

Another constituent element of an inference procedure is proof strategy. There are several aspects to this. For instance, for a particular query on the clausal formulae, an inference procedure may choose forward chaining, backward chaining or a combination of both. Other strategies like unit preference, set of support and input resolution are used to reduce the search space of the resolution system.

2.3 Expert Systems

Expert Systems (or Knowledge-based systems) are computer programs that are concerned with the concepts and methods of symbolic inference, or reasoning, by a computer, and how the knowledge used to make those inferences will be represented inside the machine.

2.3.1 The building blocks of Expert Systems

In most knowledge-based engineering practices, Expert Systems are composed of five functional components:

- **Knowledge base:** A store of declarative representation of the expertise. One or more knowledge representation schemes are provided for expressing knowledge about the application domains.

- **Reasoning engine:** Implementation of logical inference mechanisms. It manipulates the symbolic information and knowledge in the knowledge base and applies them to the deduction rules[3] so that conclusions can be reached.
- **Knowledge editing and debugging subsystem:** This subsystem helps experts to build knowledge bases. It provides a way to input a user's knowledge to the system as well as debug and modify them.
- **Explanation subsystem:** A subsystem that explains the system's actions. The explanation can range from how the final or intermediate solutions were arrived at to justifying the need for additional data.
- **User interface:** The means of communication with the user. The user interface is generally not a part of the Expert System technology, and was not given much attention in the past. However, it is now widely accepted that the user interface can make a critical difference in the perceived utility of a system regardless of the system's performance [29].

2.3.2 Interaction with Expert Systems

The design of interactions between the user and the Expert System (rule-based system) is important. A rule-based system design tends to provide a general interaction mode to apply to any arbitrary rule base that is used by the system. Typically, the interaction is in the form of queries that are asked by the system and answered by the user or asked by the user and answered by the system.

[3] Deduction rules are a set of rules for deducing the logical consequences of a set of sentences [37].

2.3.2.1 Questions asked by the system

A typical rule base only holds the general knowledge of a domain and the logical relations between each element of the knowledge. This general information and logical structure serves well to give the system a clear direction and criteria to make any decisions for a given task. Yet it lacks the knowledge that is particular to any given task. For example, a bank business rule system may know almost everything about the mortgage policies such as the kinds of mortgages available, the interest rate on each kind of mortgage and their terms, but it has no idea about the customer’s choice (preference) and financial information. Information of this kind is usually provided by the user. Because users are not expected to have any expert knowledge or know anything about how the system works, they can only provide whatever information the system needs in a reactive way. So it is important that the system is able to ask user relevant questions at appropriate times.

2.3.2.2 Questions asked by the user

It is often claimed that an important aspect of Expert Systems is their ability to explain themselves [1]. This means the user can ask the system for justification of conclusions or questions at any point in a consultation with an Expert System. On the other hand, by looking at explanations, knowledge engineers can see how the system is behaving, and how the rules and data are interacting. They serve as the “logical traces for knowledge bases just like program tracing for conventional programs” [37]. Given that the system knows which rules were used during the inference process, it is possible for the system to provide those rules to the user as a means for explaining the results [29]. In fact, most of the existing Expert Systems, e.g. Mycin [7], follow this way in the implementation of explanation and debugging systems [28].

2.4 Introduction to jDREW

jDREW (Java Deductive Reasoning Engine for the Web) is a highly configurable reasoning engine written in Java for definite clause reasoning. It is a composition of several components each of which carries out one aspect of the reasoning functionality. Each component provides its own application programming interface(API) that could be directly used by the programmers or used by other components. Users have the freedom to choose the upper layer component APIs to get a standard and easily controlled engine, or the lower layer component APIs to apply different strategies and build more customized reasoning procedures. Particularly, jDREW provides two packages for the purpose of building the bottom up and the top down inference engines. Through its diverse APIs, jDREW can be easily embedded into a larger Java system, which makes it good for building systems that need an embedded reasoner [40]. The prototype system of this thesis will use the jDREW top down inference engine for the business rule processing.

2.5 Web Services

Compared with other rule-based systems, business rule systems for e-business have their special concerns.

First, the clients may use the system from anywhere across the Internet. There are special concerns on how to provide services across this global network and make full use of existing network infrastructure. The designer of the system has to answer questions like “what underlying protocol should the service be running on?”, “Could a user access the service behind a proxy or firewall?”.

Second, the clients use different platforms and have their own preference for programming languages. How to provide highly compatible rule systems that could be easily used by businesses of different backgrounds is an important aspect of design.

Third, there is no prediction on how a client could use the system. Just taking into consideration the fact that too many enterprises have made a considerable investment in developing their own business or management system and in training employee of using these systems, it is almost impossible for an enterprise to totally abandon the old system and jump into a new one. Therefore, being able to integrate the rule system into legacy systems is of high priority [47].

Fourth, as volume and scale of e-business increases and any entity becomes more and more dependent on others to maintain high competitiveness, it is inevitable that an e-business system will finally incorporate technology with open architecture for integration and interoperation among back-end systems from different business partners [35].

Fortunately, all these questions are just what Web Services are trying to answer.

2.5.1 What are Web Services?

“On the surface, a Web Service is simply an application that exposes a Web-accessible API. That means you can invoke this application over the Web” [47]

The core of Web Services is a set of technologies combined together under an overall architecture that enables services (APIs) to be uniformly accessed across the web. The goal of Web Services is to provide services that:

- are accessible over Internet through high-level standard protocols.
- can be invoked without knowing their programming languages and running environments.
- have their interface described in a format that is both human-readable and machine-readable.
- are discoverable across Internet by a computer.

Web Service technologies mainly aim at the high-level architectures of decentralized system over global network. It is designed to work harmoniously with other existing distributed computing technologies like J2EE, DCOM. Web Service protocols do not restrict the implementation techniques for any individual Web Service.

2.5.2 Web Service architecture

Web Services architecture is a message-based, service-oriented architecture that is based on the notion that everything is a service. Two important components constitute the main infrastructure of Web Services: provider and broker. Together with the service requestor, these three distinct actors compose the lifecycle of a Web Service.

- Service provider: In one aspect, the service provider is the implementer of the Web Service. As a technical term, service provider also denotes Web Service itself or the hosting environment that the Web Service is running on.

- Service broker: The service broker is itself a service provider. Service broker usually has a logically centralized directory of services (UDDI registry) and provides relevant services like intelligent search and business classification or taxonomy.
- Service requester: The service requester is the consumer of the Web Services. Service requester could be a client side program or another Web Service.

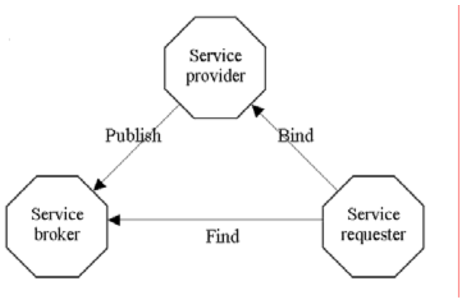


Figure 2-2 Web Services Components

The life cycle of particular Web Service starts from the service provider. There are several ways that a service provider can establish a Web Service. Service providers can build their own Web Services by first developing the core functionality of the service, then extract from the functionality the interface that the service provider want to expose to the outside world and wrap the interface so that it is accessible through the SOAP protocol. Next step, the service provider needs to build a XML based Web Service interface description (WSDL) that includes all the necessary information for invoking a service, e.g. the signature of the service; which communication protocol is used and where to locate the Web Service. In addition to deploying the service on a machine that can be accessed through Internet, the service provider still needs to publish the service interface description to a UDDI directory or broker with necessary taxonomy information so that the Web Service would be easily found by the service requester.

Also, a service provider has the freedom of providing its own implementation of a published service interface or wrapping an existing software application or program into a Web Service.

What a service requester needs to do is to find the required Web Services and invoke them. The UDDI registry provides flexible mechanisms for service discovery. For example, a service requester can find a particular Web Service interface through one or more taxonomies. If a requester knows the name of the service or the company that provides this service, it can directly use this information to get the service. Further more, if a service requester gets the interface of a Web Service, it can easily get all the Web Service implementations for this interface. The interface description is all what a user needs in order to establish client side programs that need to integrate the Web Service functionalities.

The Web Service broker or UDDI registry acts as an intermediary between Web Service providers and Web Service requesters. To carry out this role, a service broker must be publicly known to both service providers and requesters. For Web Service providers, the UDDI registry provides rich standard taxonomies and flexible mechanisms so that service providers can easily publish their services in a most discoverable way or establish their own information hierarchies. For Web Service requesters, it provides diverse searching services so that the data stored in the UDDI registry can be accessed easily and in an organizable way. In fact, a service broker is itself a service provider.

2.5.3 Enabling technologies of Web Services

The goal of Web Services is to achieve high web based interoperability among software applications regardless of their implementing languages and running environments. Web Services adopt and define several technologies and standards that complete each other and work together to achieve this goal.

- HTTP and XML

  HTTP and XML are two technologies of extreme importance for Internet. They are also two fundamental underlying techniques for Web Services. The adoption of HTTP into Web Service aims at making full use of the existing Internet infrastructure. First, Web Service can directly use a web server as the deployment environment, which automatically connects the implementation of Web Services to numerous technologies like COM (Component Object Model), DCOM (Distributed Component Object Model), J2EE (a Java environment for developing and deploying distributed Java software components) and CGI (Common Gateway Interface). Second, by choosing HTTP as a transportation protocol, Web Services are accessible throughout the whole Internet infrastructure without worrying about technical details like the compatibility with the existing proxies and firewalls.

  XML is a kind of markup language that gained rapid acceptance recent years. Because XML message or file is composed of plain text, it is readable by human; Also, because the user can use new tags to define the structure of the message, XML is a very important technology that is adopted by the Web Services architecture to achieve universal interoperability.

- SOAP

  As an XML-based protocol for exchanging information between computers, SOAP can be delivered via a variety of transport protocols (e.g., HTTP, FTP, SMTP.). Written in XML, a SOAP message is entirely language and platform independent. SOAP is the communication protocol among Web Services and between Web Service and client side software. Therefore, a Web Service can be easily glued together with other Web Services and software systems without worrying about the implementation differences between them.

- WSDL

  Web Services use WSDL (Web Services Description Language) to make a complete description about the interfaces exposed by Web Services. The Web Service interface description includes signature of methods and messages accepted by the Web Service; data type information which is either predefined primitive type or user defined object; binding information that tells which transport protocol is used by the Web Service, and endpoint information that specify the location and listening port of a specified service. WSDL uses XML grammar and is designed to be machine-readable so that, to some extent, the process of integrating Web Services into other software systems can be automated.

- UDDI

  UDDI (Universal Description, Discovery and Integration) mainly deals with problems about the discoverability of Web Services. UDDI is a technical specification that can be used to define how to publish (for service provider) and find (for service requester) a Web Service and all information that is necessary for consuming the Web Service.

To ensure maximum interoperability, UDDI is usually implemented as a Web Service that exposes a set of operations according to the UDDI specifications. Hence, UDDI can not only be accessed at design time by developers, but also be accessed at runtime by a program.

3 Interactions with Business Rules

Business rules can be applied to control various aspects of behaviors of business activities. As an example, the rules in a business rule system could encapsulate requirements of doing business with the system owner and help the potential business partner to determine if he meets all these requirements. In this case, the interactions between the system and the user could be composed of two phases. First, the system will apply policy rules to the user specific information to decide the user's eligibility to access a business's service. In this phase, the user will be guided to provide information as input to some particular activities. In the second phase, the user may choose to interact with the explanation system to find out how the system derived a conclusion or what requirements are missing. The interaction models and concerns that apply to the business rule systems are similar to those that apply to Expert Systems.

3.1 Question asked by the system

When a business rule system initiates a question, it is important that the question is relevant to the overall task under the current context. "What are relevant questions" and "what is the appropriate time to ask" are two main concerns about the interaction between a rule-based system and an end-user.

A relevant question can be defined from two aspects:

1. The question is relevant to the task.
2. The information asked by the question should belong to the user's domain of knowledge.

For a rule-based system, the first aspect of the definition of relevancy can be achieved through some steps of inference. But the knowledge base and the reasoning mechanism do not embody any information telling whether the question also belongs to the user's domain of knowledge. Usually a rule-based system needs extra information to tell if a question is "askable". For instance, in a goal-directed top-down inference procedure on definite clauses, each unsatisfied subgoal is an atom of first-order logic which is crucial for the system to get the conclusion denoted by the head of the current rule. Questions about these subgoals are relevant to the process that will lead to the solutions of the task. Yet the system has no way to know if this unsatisfied subgoal is a hypothesis that is proved to be false according to the knowledge base or denotes a demand of information input from user.

A user-friendly rule-based system also needs strategies to prevent the user from being overwhelmed by questions. Of course, a carefully built knowledge base plays an important role on it. Besides the optimization of the knowledge base, other methods apply to reduce the number of questions a system will ask. For instance, before asking a question, the system could first make a more rigorous check to decide if a question is constructive to the final solution of the task. Also, the system needs to keep track of all the questions asked before and directly apply previous answers to any future reoccurrences. While the ordering of questions in human-to-human interactions is important, the questions asked by an Expert System may switch focus frequently.

In this thesis, we focus on variable free questions that can be answered by "yes" or "no". Two predefined predicates are introduced for the purpose of marking variable free questions. One is "$question" that is used in form of "$question(X)" to denote whether or not X is a question that could be answered by the user (belongs to the user’s domain of knowledge). The other one is “\$askable” that tells the system whether it resorts to the user for an answer when an unsatisfied goal is encountered. For example, “\$askable(retired(X)).” means if the knowledge base does not have any fact to say that a person is retired, the system should ask the user instead of simply presuming that the person is not retired. For more flexibility, we do not say directly if something is “askable”. Instead, we tell the system if something belongs to the user’s domain of knowledge and let the system to decide if it is “askable” under a particular context. This could be done by adding rule

$$\$askable(retired(X)) \Leftarrow \$question(retired(X)) \wedge context\_dependent\_conditions\ldots \qquad (1)$$

to the rule base. The “\$askable” predicate could be stored together with any other knowledge or be stored separately in a meta-knowledge base. Stating the askability of a goal as a fact or a rule in the knowledge base has two advantages over simply adding an askability attribute on the predicate.

1. It gives more subtle controls on the askability of a predicate.
2. The system tends to be more consistent by treating askability as a kind of special knowledge.

For example, we could add some restrictions on the askability of the retired(X) in form of rules. Together with rule (1), rule “\$question(retired(X)) $\Leftarrow$ senior(X).” means only a senior person needs to answer this question.

For the question (e.g. retired(Peter)?) asked by the system, the user may have “yes” or “no” answers. For “yes” answer, “retired(Peter)” will be added as a fact to the knowledge base and will be used automatically on following inferences. Hence, the same question need not be asked again. However, “no” answers need to be recorded in a slightly different way. For example, we can not simply add “notRetired(Peter).” to the knowledge base because the system does not know “notRetired(Peter).” means that “retired(Peter).” is a false statement. Here, we introduce another predefined predict “\$not” to indicate that the question has been answered “no”. The “\$not” predicate can affect the askability of a question through the following rule:

$$\$askable(X) \Leftarrow \$question(X) \wedge \sim\$not(X)^{4} \qquad (2)$$

Of course, the flexibilities brought by using rules for askability control do not come without price. One is that this approach takes more computing time. But this overhead is modest if we take into consideration that, for most practical systems, only very simple logic relations are involved in the askability control. In addition, this approach also introduces more complexity to the system. Take the following rule base as an example:

$\$askable(X) \Leftarrow \$question(X) \wedge \sim\$not(X).$

$\$question(a(X)) \Leftarrow b(X).$

$\$question(b(X))$.

In this case, when an unsatisfied subgoal of a(j) is encountered, the system will check if \$askable(a(j)) is true in order decide if a question should be prompted to the user. This check will cause question “?b(j)” be asked first. Question “?a(j)” may or may

4 ~ means negation by failure. If the system failed to prove an atom *a*, the system will take ~*a* as true.

not be asked depending on the user's answer to the first question. So, a single query may fire multiple levels of inference.

The other two methods used to reduce the questions asked by the prototype system of this thesis are inference procedure dependent. jDREW uses SLD-resolution as the inference rule.

The "SLD-resolution" is the abbreviation for linear resolution with selector function for definite clauses. Informally, the SLD-resolution rule is a refinement of resolution that, in each inference step, chooses some subgoal from the current goal and unifies this with the head of a definite clause to produce a resolvent (with the literals ordered in a specific way), which then becomes the new goal [12]. The top-down inference procedure of jDREW using the SLD-resolution rule looks like this:

```
function BACKCHAIN- SLD(knowledgebase KB, proofTree PT, Vector
MatchingResult)

    local variables:  PT', a proofTree variable
    while true do
        if  there is no goal marked "open" in the PT then
            add rootOf(PT)[5] to MatchingResult and return;
        openGoal ← first leaf of PT that isn't marked "closed".
        if  the openGoal is askable then
            ask the question of the openGoal.
            if  the answer is "yes" then
                mark the openGoal as "closed".
            else
                return;
        else
```

[5] rootOf(*PT*) is the goal (a predicate of first order logic) that resides on the root of the proof tree *PT.*

```
            break;
    end
    for each fact f in KB such that $\exists\theta$ SUBST[6] ($\theta$,f ) = SUBST($\theta$,openGoal) do
        PT'= apply $\theta$ to PT and mark the openGoal "closed"
        BACKCHAIN- SLD( KB, PT', MatchingResult)
    end
    for each rule r in KB such that $\exists\theta$ SUBST($\theta$,headof( r)[7]) = SUBST($\theta$,openGoal) do
        PT'= apply $\theta$ to PT and mark the openGoal "closed"
        Attach the bodyOf(r) [8] ($\theta$ already applied) to PT' as the children of
        openGoal and mark them "open".
        BACKCHAIN- SLD(KB, PT', MatchingResult)
    end
```

Figure 3-1 Back-Chaining Using SLD-resolution

The initial state of the proof tree *PT* is a tree that only has a root of the target query that is marked "open".

We first give an example to show how this inference procedure works. For a given query "a(g1,g2)" and a rule base below,

1. $a(X,Y) \Leftarrow b(X) \wedge c(Y) \wedge d(g)$.
2. $a(X,Y) \Leftarrow e(X) \wedge f(X)$.
3. $b(X) \Leftarrow j(X) \wedge k(X)$.
4. $b(X) \Leftarrow j(X) \wedge i(X)$.

[6] SUBST(θ,α) denotes the result of applying the substitution θ to the sentence α. For example: SUBST({x/Sam,y/Pam},Likes(x,y)) = Likes(Sam,Pam) [37].

[7] *headOf*(r) denotes the positive atom of the definite clause r. For example headOf("$d(X) \Leftarrow g(X) \wedge l(X)$") = d(x).

[8] *bodyOf*(r) denotes the negative atoms of the definite clause r. For example BodyOf("$d(X) \Leftarrow g(X) \wedge l(X)$") = g(X) , l(X).

5. $d(X) \Leftarrow g(X) \wedge l(X)$.

6. $d(X) \Leftarrow g(X) \wedge l(fun(X))$.

7. \$askable(c(X)).

8. \$askable(k(X)).

9. \$askable(i(X)).

10. j(g1).

11. g(g).

the above top-down inference procedure of jDREW will go through the following steps as shown in Figure 3-2. In Figure 3-2, the "●" means the goal is supported by the rule base as a fact; the "✖" means the goal is not a fact in the rule base and no rule can lead to it; the "☑" means the question of an askable goal is answered "yes" by the user. We also assume that the depth-first search strategy is from left to right, the rules are scanned according to their sequence in the rule base and all the questions are answered "yes" by the user.

First, a "proofTree" data structure is initiated with a(g1,g2) as its root. The rule (1) is used as the first attempt to prove a(g1,g2). After applying unifer {X/g1, Y/g2} on rule (1), the proofTree changed to Figure 3-2(a). Consequently, rule (3) will be applied for goal "b(g1)", which is shown in Figure 3-2 (b); rule (10) for goal "j(g1)"; \$askable(k(X)) for goal "k(g1), which will cause question "?k(g1)" being asked; \$askable(c(X)) for goal "c(g2), which will cause question "?c(g2)" being asked; rule(5) for goal d(g), which is shown in Figure 3-2 (c); rule(11) for goal "g(g)"; No rule can be applied for goal "l(g)", so the inference procedure traces back one step and uses rule(6) to apply to goal “d(g)”; rule(11) for goal “g(g)”. No rule can be applied for goal “l(fun(g))”, so the inference procedure traces back one step and uses rule(2) to apply to goal “a(g1,g2)” as shown in Figure 3-2 (e); no rule can be applied for goal “e(g1)” and no traceback operation can apply, so the goal “a(g1,g2)” fails.

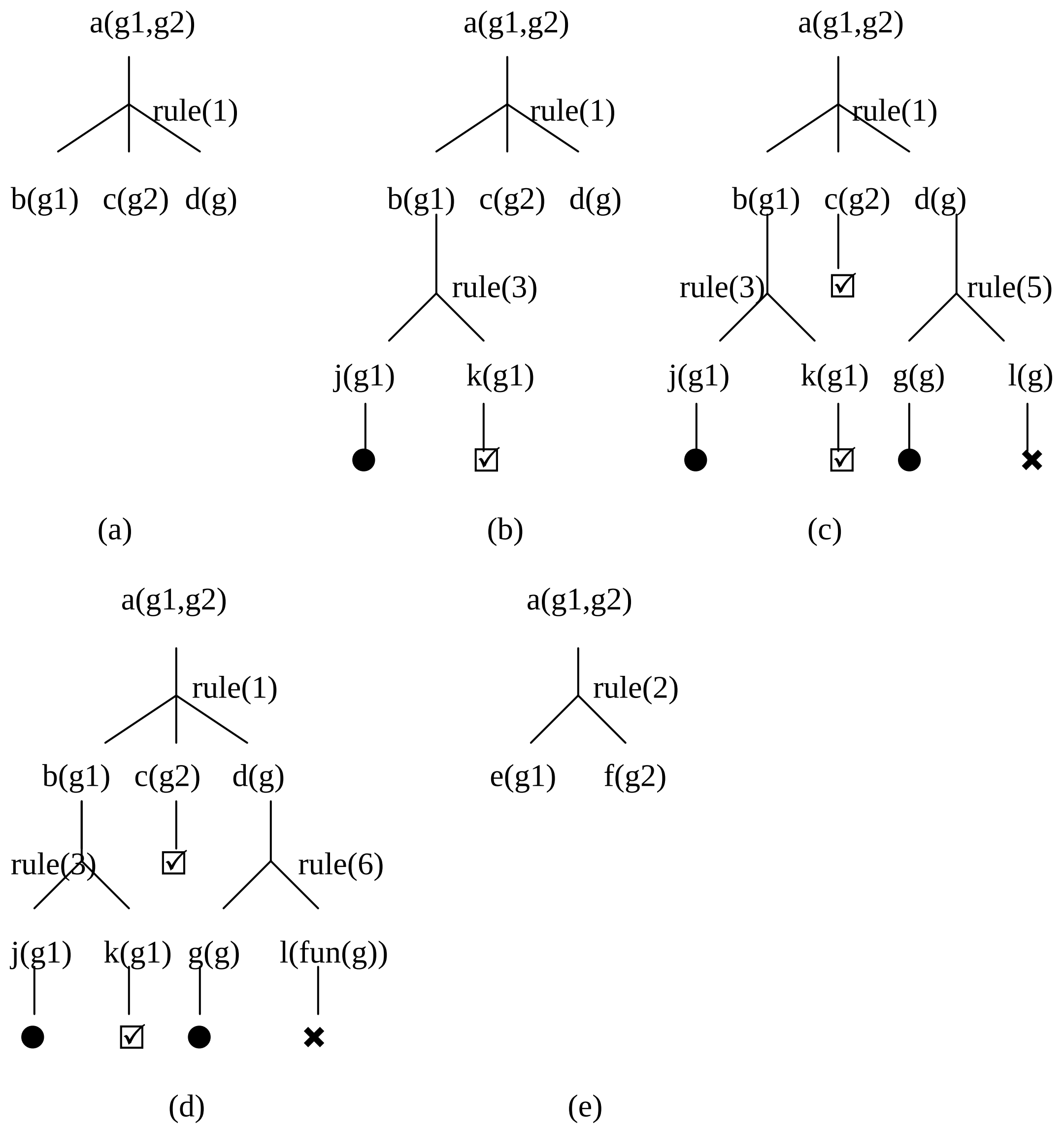


Figure 3-2 Partial Proofs Generated By The Proof Procedure

On the other hand, we can combine all these steps together into a search tree that is composed by interleaving “or” layers and “and” layers. In “and” layer, a node is true if any of its children is true while in “or” layer, a node is true only if all its children are true.

Figure 3-3 is a search tree of the above proof procedure. The question mark denotes that the system will ask the user when the node cannot be proved by the inference procedure.

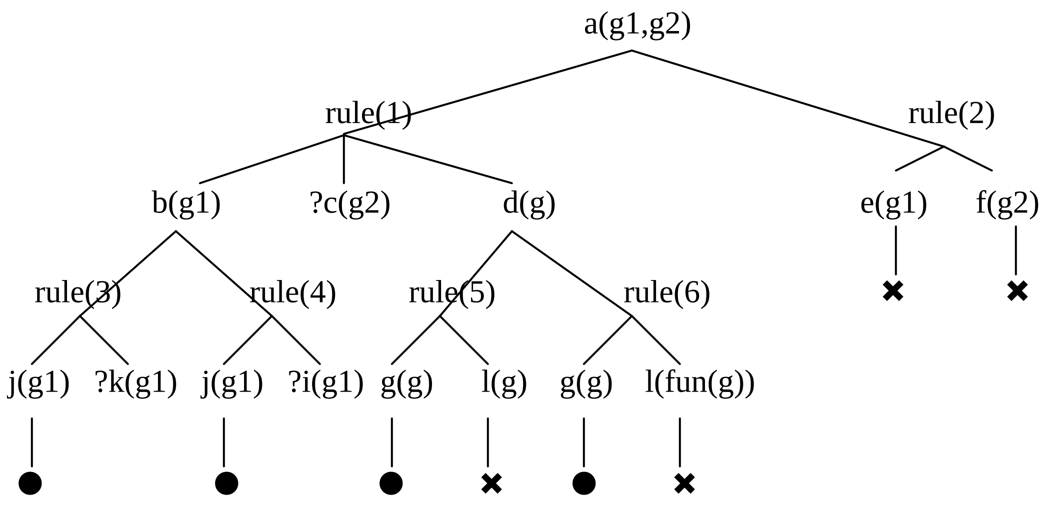


Figure 3-3 The Search Tree

From Figure 3-3 we can see that goal “a(g1,g2)” will fail on rule(1) no matter what answer the user will give on question ?k(g1), ?i(g1) and ?c(g2) because there is no way to satisfy d(g).

- Optimization 1

Optimization could be done to prevent the system from asking the user any question that will not help the final resolution of the task. Based on the observation that the failure of a node in the “and” layer alone can cause the failure of its parent node, the system could check out such nodes and avoid asking any question that appeared in the subtrees that are rooted on their siblings.

- Optimization 2

In addition, the sequence of questions that are asked by the system also affects the number of questions that a user has to answer. Take the following proof tree as an example:

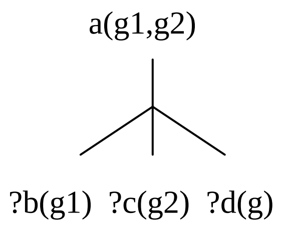


Figure 3-4 Example 1

Assume the user will give ?b(g1) and ?c(g2) a “yes” answer and ?d(g) a “no” answer. The user will have to answer all three questions if the questions are asked on the same sequence as they are encountered by the inference procedure. On the other hand only one question needs to be asked if ?d(g) is asked first. Even though the system has no way to know what question is more likely to be answered “no” by the user, it still can make some heuristic optimizations. By observing the position of the questions in Figure 3-3, we can find out that “no” answer on question ?c(g2) will have more impact on the total number of questions that need to be asked than “no” answers on “?k(g1)” or “?i(g1)”. The reason is that any “no” answer on a higher layer question is more likely to trim off bigger substrees than “no” answer on a lower layer question does. So, for a coarse optimization, we could simply choose the question of higher layer to begin with. Yet the above optimization only works well on a special subset of questions in which the askable nodes form a “tight group” that is defined below.

Definition 1: A set of "and" layer nodes forms a "tight group" if the first common ancestor of all the nodes is the father of at least one node of that set.

Figure 3-5 shows a more general case. In figure 3-5, each number represents a predicate of first order logic. The predicate that is preceded by a question mark means this predicate is askable. In this case, "?4" and "?6" reside on a higher level than "?10". Yet a "no" answer to "?10" will trim off the whole search tree while "no" answer to either "?4" or "?6" can only trim off a subtree that is rooted on the or-node father of either "?4" or "?6". So, in this case, it is better to ask "?10" first.

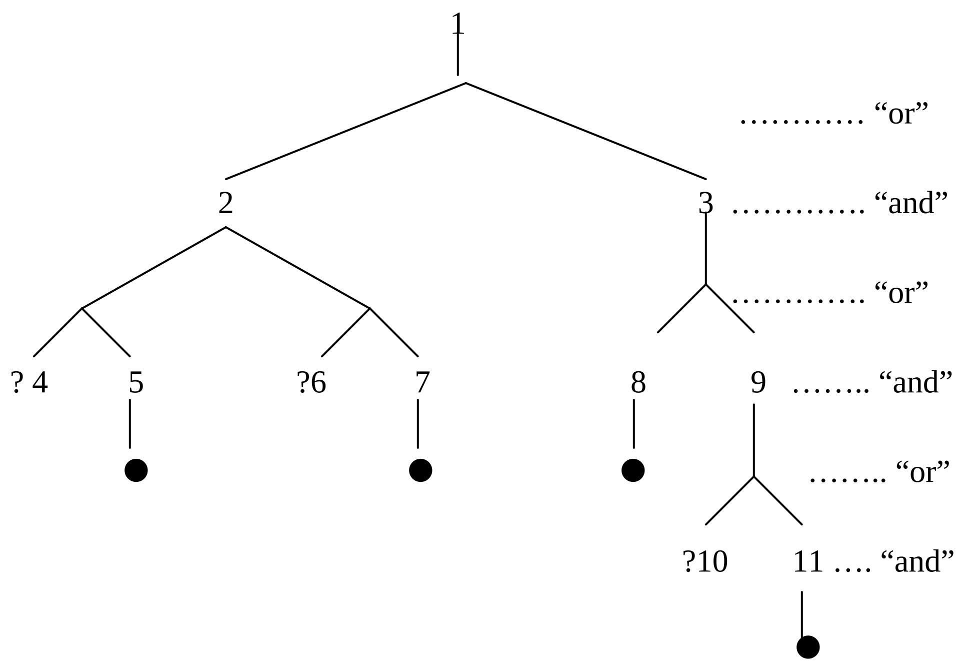


Figure 3-5 Example 1 Of a General Case

For an even more general case as shown in Figure 3-6, it is not easy to tell which question should be asked first without quantifying the impact of each question.

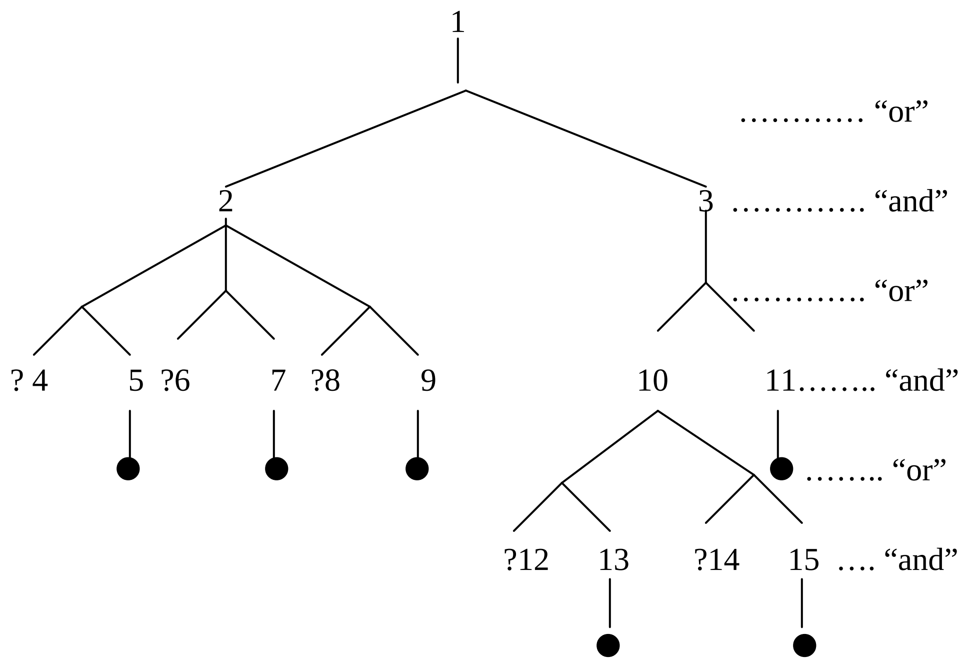


Figure 3-6 Example 2 Of a General Case

In order to quantify each question, we add a factor value to every internal node and question node of the search tree. (Note that each node on the "and" layer is a predicate and each node on the "or" layer is a rule)

The factor value of each internal node is defined as follows:

1. For each internal "and" layer node, the factor value is the sum of the factor values of its children.

2. For each "or" layer node, the factor value is 1.

In Figure 3-7, the number in brackets shows the factor value for each internal node.

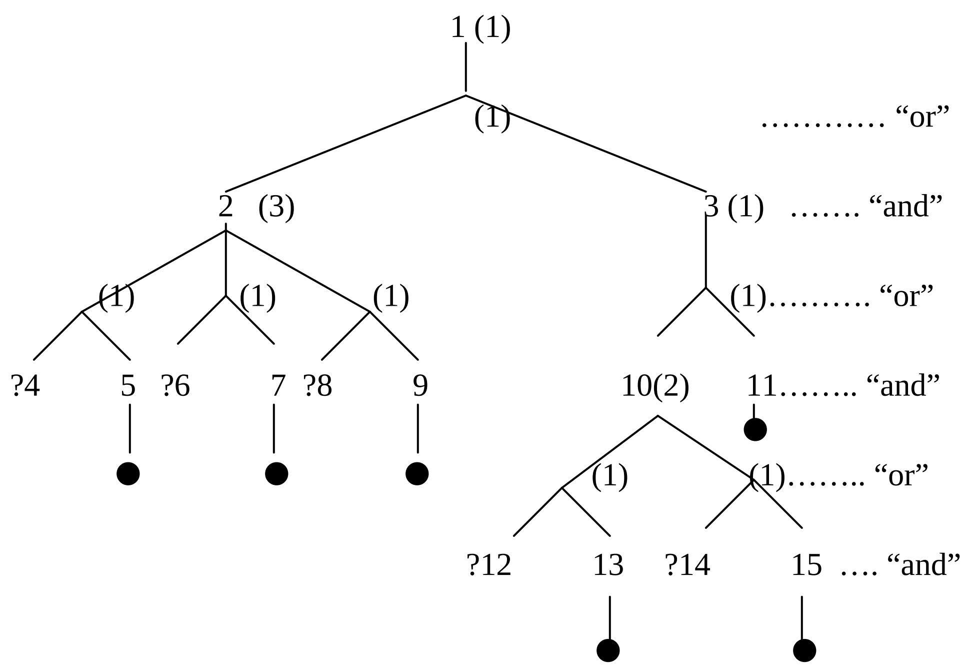


Figure 3-7 The Factor Value of Figure 3-6

The impact of a question is represented by the factor value that is calculated as follows:

Assume along the path from the parent of the question node to the root of the search tree, there are n “and” layers nodes $a_1, a_2, ...a_n$. The factor value v of the question node is:

$$v = \prod_{i=1}^{n} 1/F_{a_i}\ ; \qquad (1)$$

where F_{a_i} denotes the factor value of the node a_i.

According to formula (1), the factor value of “?4”, “?6” and “?8” is 1/3 and the factor value of “?12” and “?14” is 1/2.

In order to calculate the factor value of a question that appears more that once in the search tree, we introduce a variable I_n to denote the impact of a question on a node *n*.

We define I_n as follows:

1. the impact of a question on the question node of itself is 1; the impact of a question on the other leaf nodes is 0;

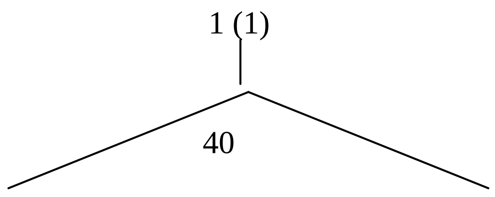
2. for an internal "or" layer node *o*, $I_o = \max_{i=1}^{n} I_{c_i}$ where *n* is the number of children of *o*, c_i is the *i*-th child of *o*.

3. for an internal "and" layer node *a*, $I_a = \sum_{i=1}^{n} I_{c_i} / F_a$ where *n* is the number of children of *a*, c_i is the *i*-th child of *a*.

And the factor value of a question is equal to the impact of the question on the root of the search tree:

$$v = I_{root}; \tag{2}$$

According to formula (2), the factor value of "?4" in Figure 3-8 is "1".



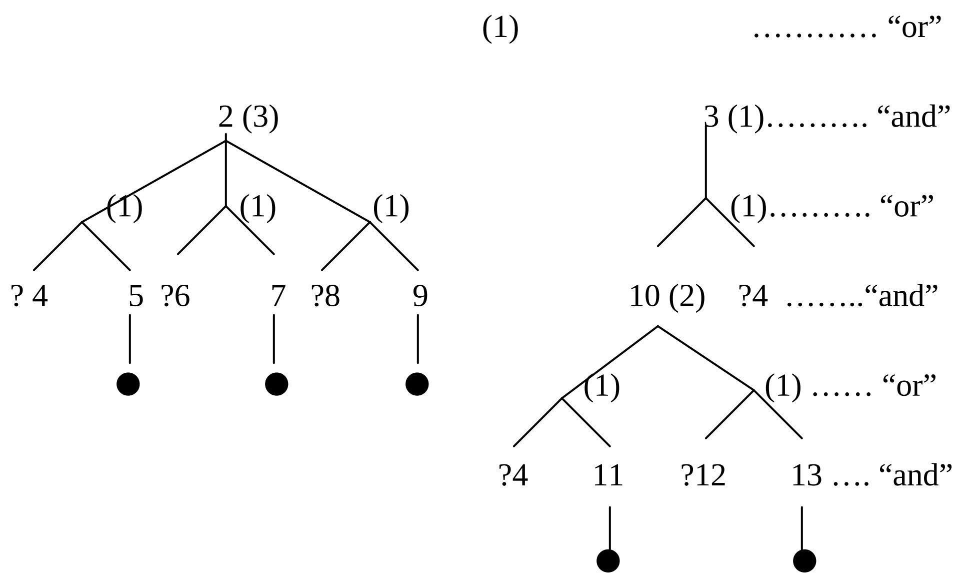


Figure 3-8 An Example Of The Factor Value

We can see that formula (1) is just a special case of formula (2).

By this, we can choose the sequence of the questions according to their factor values. That is the question of the highest factor value will be asked first.

Choosing the question set is another important aspect of this optimization. There are several strategies to choose from.

- Question selection strategy 1

We could choose a question set that includes all the questions that appear in the search tree. The whole optimization process turns out to be finding out a fastest way to trim off the search tree until the whole tree is trimmed off (the goal fails) or, on some steps during the trimming, the goal succeeds. Because the scope of the optimization is the whole tree, this choice of question set makes full use of the trimming capacity of the optimization method. So, it works best when a goal fails, yet it does not perform as well when a goal succeeds.

- Question selection strategy 2

Another proof-oriented strategy is to choose a question set that holds just enough questions for each proof. Whenever a question in the question set is answered “no”, the system will discard the remaining questions and turn to another question set. The following is an example of this approach that chooses the next question set in a depth-first, left to right sequence.

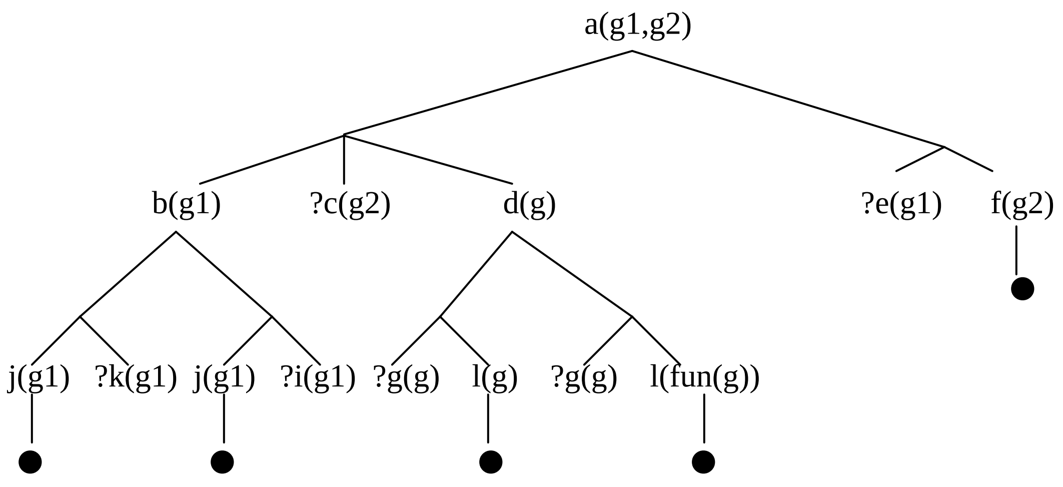


Figure 3-9 Search Tree Illustrating The Question Selection Strategy 2

By using the search tree of Figure 3-9, the first question set will include {?c(g2),?g(g),?k(g1)} (the questions are written in the sequence in which they would be asked). If only the answer of “?k(g1)” is “no”, the system will choose the next question set, that is {?i(g1)}. The last question set is {?e(g1)}. The idea behind this strategy is to focus on the questions that may lead to one valid proof. This seems to work more harmoniously with the goal-oriented top-down proof procedure. Yet, it has a disadvantage -- it limits the optimization locally to a small subset of questions.

We can also apply the “smallest question set” principle to this strategy for better performance. Based on the assumption that each question has equal possibility to be answered “yes” or “no”, a smaller set of questions are more likely to get all “yes” answers than bigger sets. Also useing the search tree of Figure 3-9 as an example, question set {?e(g1)} will be asked first under this principle.

3.2 Questions asked by the user

It is often claimed that an important aspect of rule-based systems is their ability to explain themselves [29]. This means the user can ask the system for justification of conclusions or questions at any point in a consultation with an rule-based system. By looking at explanations, knowledge engineers can see how the system is behaving, and how the rules and data are interacting. They serve as the “logical traces for knowledge bases just like program tracing for conventional programs” [37]. Given that the system knows which rules were used during the inference process, it is possible for the system to provide those rules to the user as a means for explaining the results.

There is a high demand from the user for the explanation of the conclusions derived from the business rule systems. The system needs to tell a client if he/she satisfies all the relevant conditions to obtain a service. If a service is denied, it also needs to tell the user why it is denied by showing the user the rule(s) he/she failed to satisfy and how they could be satisfied. Usually the demand for the business rule systems to explain themselves can be satisfied through three types of questions: “how”, “why not” and “what if”. These questions are usually initiated by the user and answered by the system.

The “how” question is asked by the user to see the proof of some conclusion the system has reached. In effect, it asks “how do you justify that conclusion”. “How” questions can be asked repeatedly until the justification is a fact given to the system and stored in the knowledge base. For example, assume that a customer was offered a 5% discount toward the purchase of a Honda car. By asking “how?”, he will get the answer that he got 5% discount because (1) he is a premium customer and (2) Honda is categorized as regular car. By continuing asking “how?” on (1), he will get answer that he is a premium customer because he spent more than $5000 at this car dealership last year. Asking “how?” on this answer could result in a list of purchases made by this user, showing the total. No further “how” answer could be offered by the system because this is basic information (a fact).

In order to show the user how a goal is achieved, the system will generate a proof tree [31] with the derived goal as the root. Each internal node of the proof tree is itself a goal with a sub-proof tree rooted at it. Each node and its siblings, together with their parent node, will compose an instance of the rule that is used in the proof procedure.

The “why not” question may be asked when the system fails to derive a goal. Repetitively asking the “why not” question will lead the user through the rules that cause the goal to fail. The customer of the previous example could ask “why not a 7.5% discount on the Honda”. The answer may be that in order to get 7.5% discount, (1) he must be a premium customer and (2) Honda must be categorized as luxurious goods. If the client decides to trace down this rule, he may, for example, be told by the system that (1) is satisfied but (2) failed because “Honda is a luxurious car” is not a fact in database and no other rules can lead to this conclusion. In this case, the user may be asked to consider buying an Acura.

A “why not” question needs a slightly different approach because there is no proof tree when a goal fails. However, the proof tree idea is still useful because a “why not” question is concerned with “why a proof tree cannot be built”. By keeping track of the whole proof procedure and marking down all the failure points that prevent a complete proof tree from being built, we have gathered enough relevant information to construct partially completed proof tree with gaps.

A “what if” question is usually asked together with the “why not” question. “What if” question gives the user a chance to know the consequence of assuming a condition is true. For example, a client may find out through “why not” questions that he did not get the discount because his purchase value is less than $100. Then he could use a “what if” question to check if this is the only condition that prevents him from getting the discount. If the discount is granted after asking “what if the purchase is more than $100”, the client then has the choice to really purchase more than $100 to get the discount. But if after asking that “what if” question, the discount is still not granted because other conditions still need to be satisfied (for example, the user should also be a golden card holder), then the client could again ask “why not” questions to find out other conditions to satisfy.

“What if” questions may be asked only on unsatisfied nodes. The response is an explanation tree in which the node is assumedly satisfied. An explanation tree is a tree-like structure containing all the updated information the user has been given so far by asking above three questions. A “how” explanation tree is the same as a proof tree and a “why not” explanation tree is a partial proof tree with “gaps” that lead to the failure of the goal. Assuming to satisfy a goal or undoing such an assumption will make dynamic (sometimes temporary) changes to the rule base. In certain situations, these changes will affect the other branches of the explanation tree. So, propagating this effect all over the explanation tree is needed to keep the tree consistent with the rule base.

3.3 Summary

This chapter discusses in detail the interaction design that will be later used in the prototype system. The design focuses on minimizing the questions that a user has to answer before the system can come up with a conclusion. This is done by optimizing the sequence in which the questions will be asked based on the "ranking" of each question.

The system obtains the "ranking" knowledge of a question on the fly by observing the positions and recurrences of that question node in the search tree. The "ranking" of a question is measured by its ability to trim the search tree when it is answered "no". If a question is answered "yes", we use the smallest question set principle to measure the question's contribution towards the proof of a goal.

For the user initiated interaction, the system can understand three kinds of questions: "how?", "why not?" and "what if?". By combining the use of these three questions, the user will finally be able to trace down the proof tree to find out how a request is granted, why a request is denied and what difference will it make if a condition is satisfied.

4 A prototype of business rule explanation system for Web Services

As part of this thesis, a prototype of a business rule explanation system for Web Services has been designed and implemented. In addition to the core functions that are described in Chapter 3, some architectural aspects of the system will be discussed in this chapter.

4.1 The general design principles of the prototype system

In order to meet the stringent demands of today's e-business, it is necessary to establish an infrastructure that can handle the future extension and the merging of the existing systems. The design of the prototype system follows the following three principles in order to achieve high level of the extendibility and the reusability.

1. Distributed systems

Increasingly the global market requires the construction of an enterprise based on the optimal utilization of both human resources and material resources and fast market delivery, which are almost always geographically separated. Even in a geographically centralized enterprise, administrative domains always divide the whole organization into virtual parts that are relatively independent from each other [34]. This trend implies the high demand of workload distribution, management harmonization, data sharing and decision making that needs the participation and collaboration of several distributed systems. In particular, a distributed business rule system is a key demand of large-scale management that uses rules for the decision-making and action triggering.

Traditional practices of distributed systems treat relatively independent software modules as components that provide particular functionalities. Component-oriented architecture introduced the concept of “interface” to uniquely define the only appearance of a software component seen from the rest of the software. Hence, the interface becomes the only way that identifies a software component and the only way that connects a software component to the rest of the system. Under the component-oriented architecture, interoperability of software components is based on the interface agreements that are at the binary level or virtual machine code level. The immediate consequence of this low-level interface agreement is that the resulting system is tightly coupled by components that must be running on the same definition of language and/or hosting environment.

The continuing trend towards the loosely coupled systems with high levels of interoperability and flexibility has led to the development of a service-oriented architecture under which software systems are connected through the services they provide. When we talk about services, we are in fact talking about the interfaces in a higher and more abstract level. This also addresses an important difference between the software component and the service -- the user consumes a service in a way that is absolutely independent from its implementation details.

2. Interface agreement

Standardizing the interface is required for making a distributed system extendable. Under the Web Service architecture, a service interface definition resides in a higher layer that is independent from the binding protocols and the real implementation of the

Web Service. This serves as an ideal way to define general-purpose interfaces for the interoperation among a category of applications.

3. Autonomy

For the proposed prototype, each participating rule service could be itself a fully functioning business rule system. Each rule service maintains its own rule base and has its own deduction engine that works on these rules. By allowing some distributed parts of the system to be autonomous, we have gained enough flexibility to build peer-to-peer service networks. This, in turn, serves as a way to achieve loosely coupled system integration. Another advantage of introducing the peer-to-peer services is the multi-accessibility of the whole system. Since more than one service node has the capacity to handle the user request, the user can choose any one that is optimal under a particular context, e.g. accessibility, speed etc.

Under the proposed architecture, the whole system could be partitioned according to function, data or both. For the function-based partition, all the services have the same accessibility to the rule base, yet carry on different functionalities. This partition is used when diverse processing is needed on centrally located data. For the data-based partition, the rule base is maintained locally and shared globally through standard service interfaces. This partition is usually used when the rule base is dispersed geographically or logically and maintained separately by different organizations or departments. Autonomy also implies the collaboration of heterogeneous rule systems to provide an overall functionality. For example, we may merge two rule systems of different rule formats and different deduction procedures for the purpose of providing a more complete service.

4.2 Dynamic binding of business rule explanation systems

Binding at runtime is an important mechanism to achieve high scalability of the overall system and maintain the stable quality of the Web Services (QoS). Run-time binding is useful in several scenarios.

1. The client side developers have only the interface of a Web Service.
2. They want to postpone the consideration about which service provider to choose from.
3. The end user demands the capability of changing service providers during the use of a client software.
4. The client software needs a back-up service in case the main service breaks down.

There are many scenarios that need a rule-based Web Service to bind dynamically to other Web Services according to the policies. For example, an enterprise may be composed of several departments each of which maintains its own policy. It is reasonable for each department to provide its own policy service for the purpose of policy checking. For a transaction that needs the signature (needs to comply with the policy) of more than one department, the whole checking procedure will need the participation of several policy services that interact with one another. Ideally this would be done in a way that is transparent to the user so that only one policy service seems to serve the whole process.

In order to realize the rule-based dynamic binding, first we need to choose a general-purpose usage model that can widely fit into the real-word scenarios. In the following example, two rule-based systems want to interoperate to share each other's resources. A log wholesale company may want to connect its sale system to the log provider's sale system so that it can update its selling policy according to the log provider's selling policy. For example, the wholesale company may base its discount policy on the users purchasing history, the purchasing amount of the current transaction and the log provider's current price quote which is in turn based on the wholesaler's purchasing history and the purchasing amount according to the log provider's policy. We can generalize the problem of this category into outsourcing the proof of one or more subgoals to external systems. In the prototype system, we introduce a predefined predicate $outsource(query, bindingKey) to denote where to find a service to process the target query. Here, the "query" takes the form of an atomic sentence of first order logic and the "bindingKey" is an auto-assigned UUID that uniquely represents a binding to a registered Web Service on some UDDI registry.

We can connect rule-based Web Services by assigning $outsource facts or rules to the rule base. For instance, we could add fact $outsource(creditChecking(Person), 'uuid:51890f8b-eac5-45fe-8aaa-59ca745f0fc3') to denote that the credit information checking will be carried out by a Web Service with binding key of "uuid:51890f8b-eac5-45fe-8aaa-59ca745f0fc3" which is assumed to be a service provided by a credit service company. We can also use rules for the control of the conditional outsourcings. For example, the following two rules realize that food product storage queries go to one Web Service while furniture storage queries go to another one.

1. $outsource(inStore(X, Amount), ‘uuid:51890f5b-ehc3-4ffe-8aeba-59ca634f0fc3’) ⇐food(X).

2. $outsource(inStore(X, Amount), ‘uuid:59875f8b-eac6-41fa-8aea-5945745f0fa5’) ⇐furniture(X).

The $outsource predicate acts as a bridge that connects the inference procedures to the universal discovery mechanism of the Web Services--UDDI. UDDI provides full support for static and dynamic bindings to a Web Service. From the WSDL definition of a Web Service, the client side program (or Web Service) knows what to expect from that Web Service. From the endpoint information, the client side program (or Web Service) knows where the Web Service is deployed. UDDI provides several standard methods that fall into two categories – inquiry API and publishing API. Here, we use the ***get_bindingDetail*** function for the purpose of searching a Web Service programmatically at run time.

```
get_bindingDetail
Syntax:
<get_bindingDetail  generic = “1.0”  xmlns = “urn:uddi-org:api”>
         <bindingKey/> ……
<get_bindingDetail>
```

Figure 4-1 The Syntax Of Get_bindingDetail

This function will return the bindingTemplate information that includes binding port information of the target service. The recommended approach is to cache the bindingTemplate information locally and use the cached information for the repeated calls to the same Web Service. In case of the Web Service invocation failure, the get_bindingDetail function needs to be called again to refresh the binding information.

By using the “bindingkey” instead of binding information itself, the system obtains the capacity of tracking Web Services that might relocate over time.

The operational relationship among the inference engine, UDDI and the Web Service is shown in Figure 4-2.

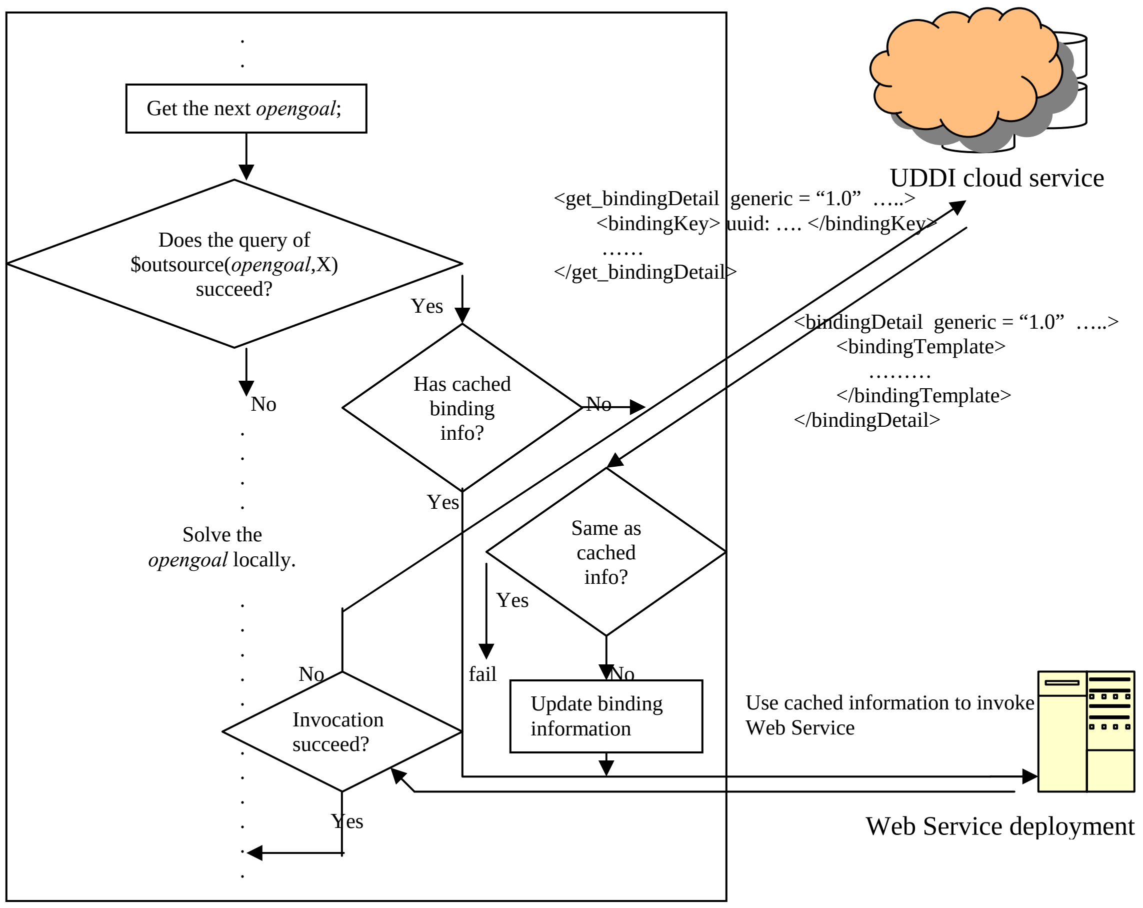

Get the next *opengoal*;
Does the query of $outsource(*opengoal*,X) succeed?
Yes
No
Has cached binding info?
No
Yes
Solve the *opengoal* locally.
Same as cached info?
Yes
No
fail
No
Invocation succeed?
Yes
Update binding information
UDDI cloud service
<get_bindingDetail generic = “1.0” …..>
<bindingKey> uuid: …. </bindingKey>
……
</get_bindingDetail>
<bindingDetail generic = “1.0” …..>
<bindingTemplate>
………
</bindingTemplate>
</bindingDetail>
Use cached information to invoke Web Service
Web Service deployment

Figure 4-2 The Chart Flow Of Outsourcing

Based on the above architecture, all the participating Web Services are organized together in a hierarchical way according to their positions in the whole search tree. Each Web Service is involved into the induction procedure for the proof of a subgoal and exits when the proof of the subgoal is finished. Also, each Web Service is autonomous itself and takes the full responsibility to work alone or involve other services into the local proof procedure. In addition, we can have more than one candidate service for one query, which is very useful in some scenarios. For example, a business entity may have two subsidiaries each of which keeps and maintains its own human resource information. Assume that each subsidiary provides its own Web Service for queries on the employee information. In this case, we need to try both Web Services in order to query information about an employee of this business entity if we do not know which subsidiary he/she is in. For this kind of query, we need to relate several Web Services to a subgoal and to involve them one at a time during the proof. This invocation mode can be achieved by adding multiple $outSource facts for a query, each of which maps to a candidate Web Service. Of course, this solution needs the collaboration of the inference engine so that all the matching instances are returned instead of just the first one. The following pseudocode illustrates how inference procedure of Figure 3-1 works on this case.

```
function BACKCHAIN-SLD(knowledgebase KB, proofTree PT, Vector MatchingResult)
    local variables:  PT', a proofTree variable
                      MA, RMR,  two Vector variables, initially empty

    while true do
        if there is no goal marked "open" in the PT then
            add rootOf(PT)[9] to MatchingResult and return;
        openGoal ← first leaf of PT that isn't marked "closed".
        if the openGoal is askable then
            ask the question of the openGoal.
            if the answer is "yes" then
                mark the openGoal as "closed".
```

[9] rootOf(*PT*) is the goal (a predicate of first order logic) that resides on the root of the proof tree *PT.*

```
                else
                    return;
            else
                break;
        end
        BACKCHAIN-SLD (KB, PT'($outSource(openGoal,X)) , MA)
        if MA is not empty then
            for each binding of X do
                call remote BACKCHAIN-SLD (rmoteKB, PT'(openGoal) [10],
RMR)
            end
            for each element e in RMR do
                Let $\theta$ be the substitution that SUBST($\theta$,e ) = SUBST($\theta$,openGoal)
                PT'= apply $\theta$ to PT and mark the openGoal "closed"
                BACKCHAIN-SLD( KB, PT', MatchingResult)
            end
        else
            for each fact f in KB such that $\exists\theta$ SUBST($\theta$,f )=SUBST($\theta$,openGoal) do
                PT'= apply $\theta$ to PT and mark the openGoal "closed"
                BACKCHAIN-SLD ( KB, PT', MatchingResult)
            end
            for each rule r in KB such that $\exists\theta$ SUBST($\theta$,headof( r)[11]) =
            SUBST($\theta$,openGoal) do
                PT'= apply $\theta$ on PT and mark the openGoal "closed"
                Attach the bodyOf(r) [12] ($\theta$ already applied) to PT' as the children
                of openGoal and mark them "open".
                BACKCHAIN-SLD(KB, PT', MatchingResult)
            end
```

Figure 4-3 Back-Chaining With Outsource

[10] *PT'(openGoal)* means a initial proof tree *PT'* with only one node (root) of *" openGoal"*.

[11] *headOf*(r) denotes the positive atom of the definite clause r. For example headOf("d(X) $\Leftarrow$ g(X) $\wedge$ l(X)") = d(x).

[12] *bodyOf*(r) denotes the negative atoms of the definite clause r. For example BodyOf("d(X)$\Leftarrow$g(X) $\wedge$ l(X)") = g(X) , l(X).

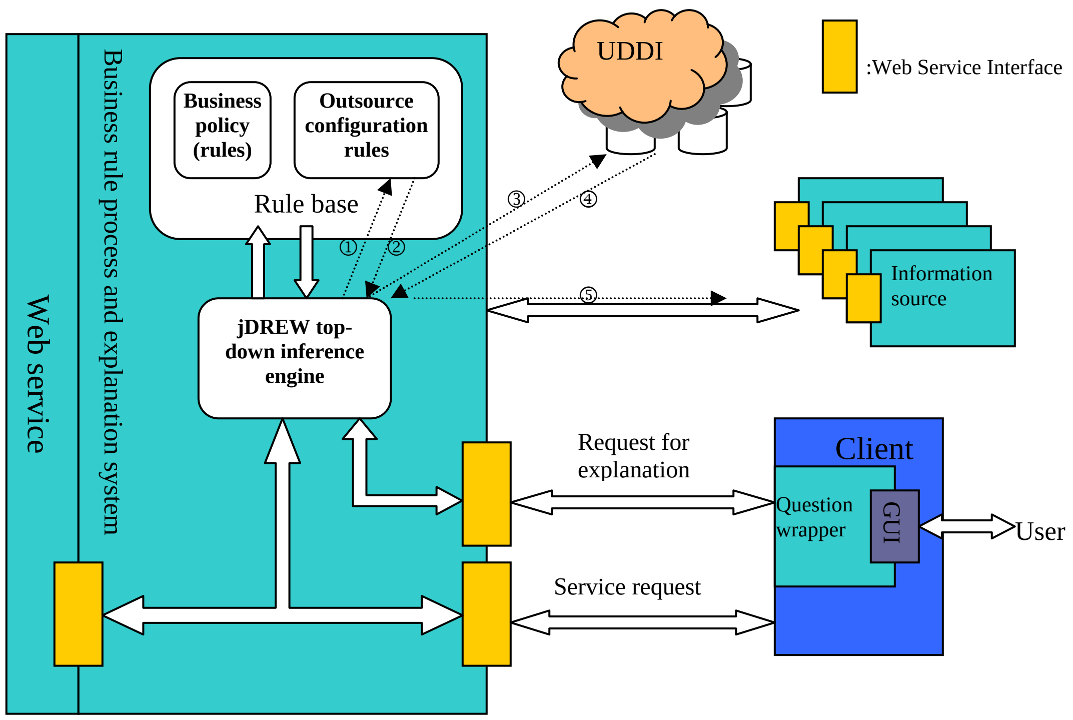


Figure 4-4 The Architecture Overview

Figure 4-4 shows the overall structure of the prototype system. The business rule process and explanation system (BPES) acts as an added layer to the core services of a business system. The core service delegates its service interface to the BPES so that BPES can function as a value-adding intermediary between the core service and the user. Basically, the BPES is a typical rule system that provides the explanation interface to the client. Being deployed as a web service, the BPES uses the configuration rules to locate the other BPES services and connects them through standard Web Services protocols. The arrows from ① to ⑤ illustrate the steps that are involved into each single connection.

4.3 Backup service and workload control

In distributed systems that are more prone to component failure than monolithic ones, the QoS a system can provide is an important concern for the system design. It is even more important in service-oriented distributed systems.

In component-based distributed systems, even though software components are distributed, they conceptually belong to the whole system. The same team or teams that collaborate intimately develop all the software components of the system. Each software component knows which software component will invoke it and in which way this will be done. The whole system also knows every thing about its components, such as whether a software component is computational intensive or communication intensive. So, the QoS of such tightly coupled systems can be measured at design time and usually will not change over time after deployment.

In contrast, the constitutional parts of a service-oriented distributed system are independent service providers that may be shared arbitrarily by many systems. For a public service, it is difficult to predict how many users it will serve at design time and the workload tends to change over time. Hence, a flexible and extendable mechanism is needed in order to provide stable QoS in a changing environment. Here, we propose to use the private UDDI registry and a portal service to realize both backup service and workload control – two important aspects about the QoS.

The only difference between a public UDDI registry and a private one is that the private UDDI registry limits its visibility within an intranet. With a private UDDI, an organization can have the full power and flexibility provided by the UDDI specification to manage its own Web Services while keeping them invisible to the outsiders. The portal service acts as a proxy and exposes itself to the outside world on behalf of the actual Web Service. The Web Service and its portal service may or may not have the same signature as long as the portal service knows how to redirect the request in a understandable format to the actual service.

There are several choices for a service provider to organize its Web Services, portal services and private UDDI registry. The best practice is that they are all deployed on the same intranet. In this case, the intranet usually does not connect to the Internet directly so that no outsiders can bypass the portal service to gain access to the actual services. The Web Services on the intranet register themselves to the private UDDI and the portal service uses the private UDDI to find the Web Service it represents. Also, the portal service registers itself on a public UDDI registry so that it is discoverable.

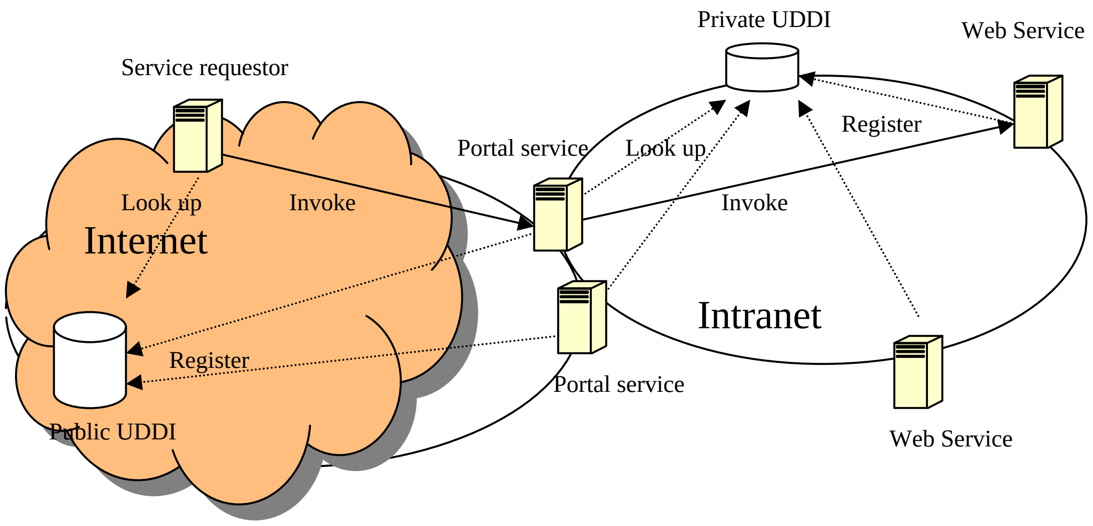


Figure 4-5 The Back Up Web Services

Dividing the service requestor and the Web Service with the portal service allows the deployment details of the Web Service to be hidden from the service requestor. For the purpose of the dual service, we can deploy the same Web Service on two different machines and register them as different services (different serviceKey). Also, a portal service is defined to uniquely represent them. All we need to do is to map the portal service to one of the Web Services (main service) and if that Web Service breaks down, map to the other one (backup service). In the following pages, we will talk about how to use the UDDI function to facilitate the mapping from the portal service to the actual service.

Information registered with UDDI is represented by XML documents containing structures described in UDDI's XML Schema. Figure 4-6 shows the data structure of the registered service.

```
<!-- businessService structure -->
<element name="businessService" type="uddi:businessService"/>
<complexType name="businessService">
   <sequence>
     <element ref="uddi:name" maxOccurs="unbounded"/>
     <element ref="uddi:description" minOccurs="0"
                                    maxOccurs="unbounded"/>
     <element ref="uddi:bindingTemplates"/>
     <element ref="uddi:categoryBag" minOccurs="0"/>
   </sequence>
   <attribute name="serviceKey"
                  use="required" type="uddi:serviceKey"/>
   <attribute name="businessKey"
                  use="optional" type="uddi:businessKey"/>
</complexType>
```

Figure 4-6 The Syntax Of BusinessService element

The businessService element includes information of a single service, including its name, description and an optional list of bindingTemplates. Each businessService is uniquely identified by a UUID (Universally Unique Identifier) called a serviceKey. The serviceKey is assigned to the businessService by UDDI registry when it is published. Among the attributes of the businessService, we pay more attention on the bindingTemplates element that includes information about how and where to access a specific Web Services. Figure 4-7 shows the XML schema of the bindingTemplates data structure.

```
<!-- bindingTemplate structure -->
<element name="bindingTemplate" type="uddi:bindingTemplate"/>
<complexType name="bindingTemplate">
   <sequence>
     <element ref="uddi:description" minOccurs="0"
                                     maxOccurs="unbounded"/>
     <choice>
       <element ref="uddi:accessPoint" minOccurs="0"/>
       <element ref="uddi:hostingRedirector" minOccurs="0"/>
     </choice>
     <element ref="uddi:tModelInstanceDetails"/>
   </sequence>
   <attribute name="serviceKey" use="optional"
                             type="uddi:serviceKey"/>
   <attribute name="bindingKey" use="required"
                             type="uddi:bindingKey"/>
</complexType>
```

Figure 4-7 The Syntax Of BindingTemplate

We are most interested in the tModelInstanceDetails attribute of a given bindingTemplates, the data structure of which is show on Figure 4-8.

```
<!-- tModel structure -->
<element name="tModel" type="uddi:tModel"/>
<complexType name="tModel">
   <sequence>
     <element ref="uddi:name"/>
     <element ref="uddi:description" minOccurs="0"
                                     maxOccurs="unbounded"/>
     <element ref="uddi:overviewDoc" minOccurs="0"/>
     <element ref="uddi:identifierBag" minOccurs="0"/>
     <element ref="uddi:categoryBag" minOccurs="0"/>
   </sequence>
   <attribute name="tModelKey" use="required" type="uddi:tModelKey"/>
   <attribute name="operator" use="optional" type="string"/>
   <attribute name="authorizedName" use="optional" type="string"/>
</complexType>
```

Figure 4-8 The Syntax Of tModel Element

For a SOAP based Web Service, its tModel is usually used to provide information about how to interface the Web Service. The interface that a Web Service supports is uniquely identified by the attribute tModelKey -- an auto-assigned UUID. Based on the tModel data structure, UDDI specification also defines a find_service inquiry according to a Web Service's tModel. As shown in Figure 4-9, the find_service by tModel query needs the businessKey and the tModelKey as the input information. This query will return all the services that belong to the business entity identified by the businessKey and support the interface defined by the tModelKey.

```
Find_service by tModel
Syntax
<find_service generic="1.0" xmlns="urn:uddi-org:api"
                        businessKey=……………………>
        <tModelBag>
                <tModelKey>……………………</tModelKey>
        </tModelBag>
</find_service>
```

Figure 4-9 The Syntax Of Find_service By tModel

For a portal service, it is reasonable to assume that it has the businessKey and tModelKey information about the service it is going to represent. The next step it needs to do is to find out the binding information of the main service and the backup service (in case the main service breaks), which could be done by the find_bindings and get_bindindDetail query. Usually we do not need extra information to denote which is the main service and which is the backup service. Yet we do need a way to decide if a service is running well or not. This could be done, for example, by setting the longest response time or by sending safe messages. Here, we use err_detect() to denote such procedures. The following is a demo mapping procedure for stateless portal services:

```
function SERVICE-MAPPING( ) returns the binding of the service

    Global constants: BK, businessKey information of the service provider
                      TK, tModelKey of the service interface
                      UDDI, address of private UDDI
    Global variables: CB, cashed binding information
    Local variables:  VB, a vector to store binding information of matched
                      services, initially empty.
                      FVB, a vector to store binding information of mal-functioning
                      services, initially empty.

    if  CB is not empty and err_detect(CB) is good
    then return CB
    else add CB to FVB
    VB ← find_service (UDDI, BK, TK)
    While VB is not empty do
        e←removeFirstElementOf ( VB)
        if e has not appeared in FVB and  err_detect(e) is good
        then assign e to CB and return e
        else add e to the FVB if e was not in FVB before
    end
    return null
```

Figure 4-10 The Service Mapping

From the SERVICE-MAPPING function, we can also see that, by using a private UDDI, we need not necessarily limit to one backup service. More important, we can add and remove backup services at any time without worrying about changing or reconfiguring the portal service, which is very important for the runtime restructuring the service deployment. Usually, there is no way for client-transparent backup of the portal service in the architectural level. When necessary, client side backup is needed in case the portal service fails.

Similar mechanisms can be used for the workload control. Here, “workload control” means the distribution of the computational intensive services into several computers in a way that is transparent to the service invokers. For an extendable system, it is essential to be able to add on extra computing power without interrupting the system’s current work. The solution based on the infrastructure illustrated on Figure 4-5 is to have all the participating computers deployed on the same intranet, each of which has a copy of Web Service deployed on it and registered to the private UDDI. The portal service is in charge of redirecting service requests to the idle service or the service with the shortest waiting list.

In fact, for a service that has more than one computer to share the workload under the above structure, each computer is a backup for the others. In this case, instead of setting up one or more separate backup services, we combine these two functions together into the portal service.

Figure 4-11 illustrates a demo working procedure for a portal service that has several mappings.

The portal service maintains a thread (thread marked “*” in Figure 4-11) and a task queue for each copy of the Web Services deployed on the intranet. Each thread locally maintains the serviceKey information for the purpose of redirecting service requests to the corresponding service. In most cases, cached binding information is used instead of serviceKey for each service invocation. Also, the err_detect() function is carried out by the thread. Each of the tasks in the task queue includes the complete information for each service request such as SOAP message and connection (e.g. socket scripter).

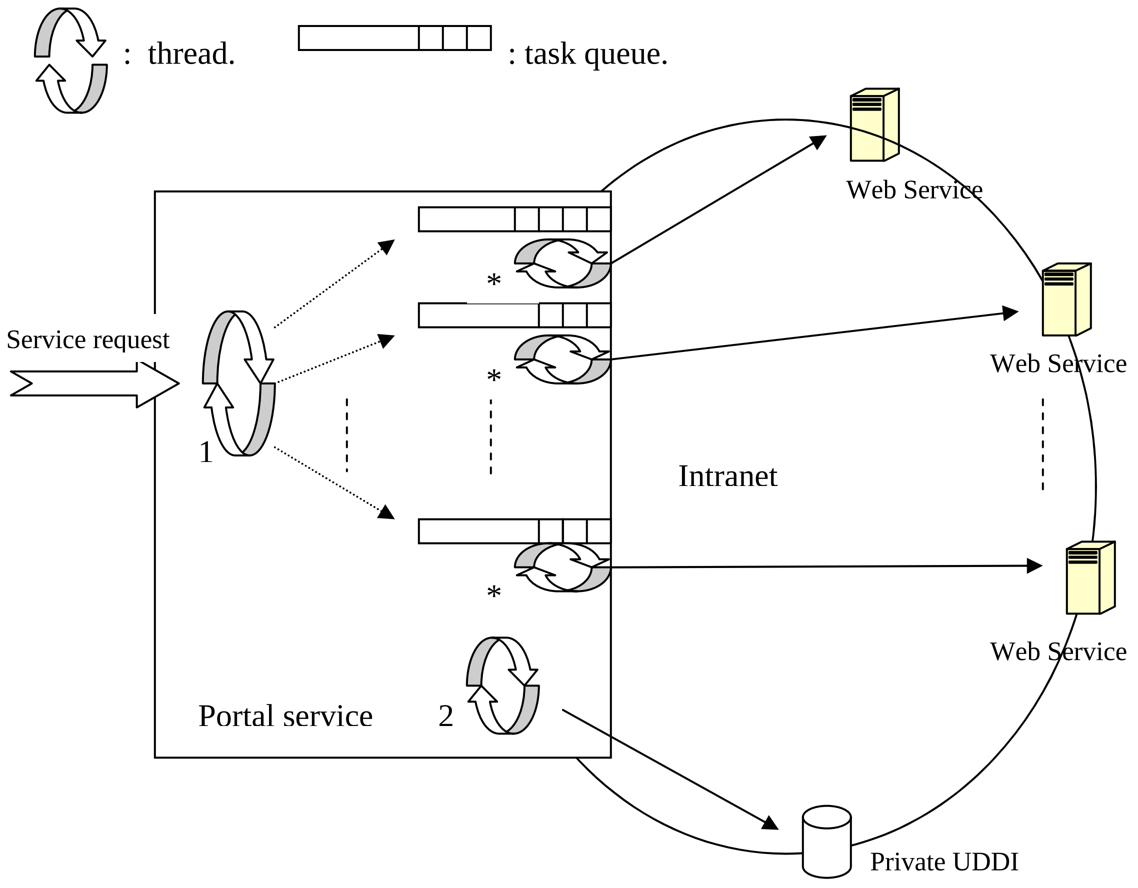


Figure 4-11 The Workload Control

A separate thread (thread 1 in Figure 4-11) is used to receive service requests and dispatch them, together with other necessary information, to one of the task queues. Another thread (thread 2 in Figure 4-11) updates the service information on a regular base.

The following pseudocode describes how these threads work together.

Data type: ServUnit, a data structure represents a Web Service stub. It includes a serviceKey, several status signs, a TaskQueue and a thread scripter.

Global variables: SERVVEC**,** a vector stores all the ServUnits maintained by the portal service

Global constants: *BK,* businessKey information of the service provider

```
                    TK,  tModelKey of the service interface
                    UDDI, address of private UDDI
Thread_1 ( )
    While true do
        receive service request
        apply distribution strategy
        dispatch request to one of the task queues
    end

Thread_2 ( )
    Local variables:  VB, a vector to store serviceKey information of  matched
                      services, initially empty.
    while true do
        VB ← find_service (UDDI, BK, TK)
        mark all the ServUnit in SERVVEC as “obsolete”
        While VB is not empty do
            e←removeFirstElementOf ( VB)
            if  there is a ServUnit  s in SERVVEC that has the same
            serviceKey as e
            then mark s as “updated”
            else begin
                new a ServUnit ns
                ns.serviceKey=e
                ns.status= “updated”
                add ns to SERVVEC
              end
        end
        for each “obsolete” or “broken” ServUnit os in SERVVEC do
            if  os.taskqueue is not empty
            then  redispatch tasks in the queue
            remove os from SERVVEC
        end
        sleep ( time_interval)
    end
Thread_* ( )
    while true do
        tk←GetFirstTaskFrom(TaskQueue)
        if err_check ( ) is good
        then begin
            process tk
            remove tk from TaskQueue
          end
        else mark itself as "broken" and return
    end
```

4.4 Summary

This chapter illustrates the architectural design of the prototype system. The intention of the design is to establish a framework that can handle the future extension and the merging of the existing systems.

The prototype is built upon the emerging standard Web Services architecture. We envision that the prevailing partition of a business rule system is based on data and, as a result, the distributed parts of the system are composed of autonomous, fully functioning services. The prototype uses the predefined predicates to direct the solution of each non-local goal to its particular services, and returns information to the local inference procedure. By doing this, the distributed system can be configured by rules and be connected to the other services at run time by the inference engine.

In distributed systems that are more prone to component failure than monolithic ones, the QoS is an important concern for the system design. The second part of this chapter proposes an architectural framework for the backup service and workload control – two important aspects that affect the QoS.

5 Application example

The previous chapters discuss in general some concerns about the interaction and architectural design of a business rule system. In this chapter, we build an application example to illustrate how to apply the technologies, principles and prototype discussed in chapter 3 and chapter 4. Even though this application example only involves two Web Services and a client (service requester), the design of the prototype system allows it to be expanded to involve an arbitrary number of Web Services hierarchically.

5.1 System overview -- a car dealership business rule system example

This car dealership business rule system example is about the decision-making on what percent of discount a particular user can get for the purchase of a particular kind of car. The user usually initiates the interaction by asking a query (e.g. Can I get a 5% discount on a Toyota?). Then the system tries to satisfy the users query through all possible ways according to the policies, which will lead to questions initiated by the system when it needs the user's input for a particular decision-making. At this phase, the system takes most of the initiatives and the user usually answers questions without knowing the context as to in which way this question relates to his/her query? or which answer (yes or no) will help him get the desired discount? When this phase finished, the user will get a conclusion about whether his request can be met or not based on the policies and his input of information. At this time, the user can ask the system for the explanations on how the decision is made. In this phase, the user can also change his choice for a better chance of getting the discount.

5.2 Policies used by the system

The policy is a key element to control the behaviors of a business rule system. The policies are stored in form of rules. In this demo system, the car dealership arranges the cars, customers and payments under categories. For cars, we have the regular car, the decent car and the luxury car. For customers, we have the super_elite customer, the elite customer and the preferred customer. For payments, we have the golden payment, the silver payment and the bronze payment. Each category has its definition or criteria defined by rules.

5.2.1 Discount policies

The discount policies used by the example system are listed below:

1. Policy: Buying a decent or better car can get 3% discount.

   Rule: discount(V0,V1,'3%') ⟵decentCarOrAbove(V1).

2. Policy: A super_elite customer can get 3% discount on regular cars.

   Rule: discount(V0,V1,'3%') ⟵super_eliteCustomer(V0), regularCarOrAbove(V1).

3. Policy: A super_elite customer can get 5% discount on decent or better cars.

   Rule: discount(V0,V1,'5%') ⟵super_eliteCustomer(V0), decentCarOrAbove(V1).

4. Policy: A super_elite customer can get 5% discount on regular or better cars if he is willing to wait a month for delivery.

   Rule: discount(V0,V1,'5%') ⟵super_eliteCustomer(V0), regularCarOrAbove(V1), delivery_a_month_after_puchase(V0,V1).

5. Policy: An elite customer can get 5% discount on decent or better cars if his payment type is "silver".

   Rule: discount(V0,V1,'5%') ⟵eliteCustomer(V0), decentCarOrAbove(V1),paymentType(silver).

6. Policy: An elite customer can get 5% discount on luxury cars.

   Rule: discount(V0,V1,'5%') ⟵eliteCustomer(V0), luxuryCar(V1).

7. Policy: A preferred customer can get 5% discount on decent or better cars if his payment type is "golden".

   Rule: discount(V0,V1,'5%') ⟵preferredCustomer (V0), decentCarOrAbove(V1), paymentType(golden).

8. Policy: A preferred customer can get 5% discount on decent or better cars if he is willing to wait a month for delivery.

   Rule: discount(V0,V1,'5%') ⟵preferredCustomer (V0), luxuryCar(V1), delivery_a_month_after_puchase(V0,V1).

9. Policy: A super_elite customer can get 7% discount on decent or better cars if his payment type is "silver".

   Rule: discount(V0,V1,'7%') ⟵super_eliteCustomer(V0), decentCarOrAbove(V1), paymentType(silver).

11. Policy: A super_elite customer can get 7% discount on regular or better cars if his payment type is "golden".

    Rule: discount(V0,V1,'7%') ⟵super_eliteCustomer(V0), regularCarOrAbove(V1), paymentType(golden).

12. Policy: A super_elite customer can get 7% discount on luxury cars.

    Rule: discount(V0,V1,'7%') ⟵super_eliteCustomer(V0), luxuryCar(V1).

13. Policy: An elite customer can get 7% discount on decent or better cars if his payment type is "golden".

Rule: discount(V0,V1,'7%') ⟵eliteCustomer(V0), decentCarOrAbove(V1), paymentType(golden).

14. Policy: An elite customer can get 7% discount on luxury cars.

    Rule: discount(V0,V1,'7%') ⟵eliteCustomer(V0), luxuryCar(V1).

15. Policy: A preferred customer can get 7% discount on luxury cars if his payment type is “silver”.

    Rule: discount(V0,V1,'7%') ⟵preferredCustomer (V0), luxuryCar(V1),paymentType(silver).

16. Policy: A preferred customer can get 7% discount on decent or better cars if his payment type is “golden”.

    Rule: discount(V0,V1,'7%') ⟵preferredCustomer (V0), decentCarOrAbove(V1), paymentType(golden).

5.2.2 Customer type policies

1. Policy: A customer is a super_elite customer if he/she bought a luxury car from this car dealership in the past five years.

   Rule: super_eliteCustomer(V0) ⟵purchased(V0, V1, V2), luxuryCar(V1), withtinLast5Years(V2).

2. Policy: A customer is an elite customer if he/she bought a decent car from this car dealership in the past five years.

   Rule: eliteCustomer(V0) ⟵purchased(V0, V1, V2), decentCar(V1), withinLast5Years(V2).

3. Policy: A senior preferred customer will be automatically upgraded to elite customer.

   Rule: eliteCustomer(V0) ⟵preferredCustomer (V0),senior(V0).

4. Policy: A customer is a preferred customer if he/she bought a regular car from this car dealership in the past five years.

   Rule: preferredCustomer (V0) ⟵purchased(V0, V1, V2),regularCar(V1),withinLast5Years(V2).

5. Policy: The senior customer is the customer who is older than 60.

   Rule: senior(V0) ⟵moreThan60YearsOld(V0).

5.2.3 Payment type policies

1. Policy: “Golden” payment type is pay by cash.

   Rule: paymentType(golden) ⟵payBy(cash).

2. Policy: “Silver” payment type is pay by 2 year installment with financial assistance of less than $10000.

   Rule: paymentType(silver) ⟵payBy(2yearInstallment),financialAssistance(lessThan$10000).

3. Policy: “Silver” payment type is pay by 3 year installment with financial assistance of less than $7000.

   Rule: paymentType(silver) ⟵payBy(3yearInstallment),financialAssistance(lessThan$7000).

4. Policy: “Silver” payment type is pay by 5 year installment without financial assistance.

   Rule: paymentType(silver) ⟵payBy(5yearInstallment),financialAssistance($0).

6. Policy: “Bronze” payment type is pay by 4 year installment with financial assistance of less than $15000.

   Rule: paymentType(bronze) ⟵payBy(4yearInstallment),financialAssistance(lessThan$15000).

7. Policy: “Bronze” payment type is pay by 5 year installment with financial assistance of less than $10000.

   Rule: paymentType(bronze) ⟵payBy(5yearInstallment),financialAssistance(lessThan$10000).

8. Policy: “Bronze” payment type is pay by 7 year installment without financial assistance.

Rule: paymentType(bronze) ←payBy(7yearInstallment),financialAssistance($0).

5.2.4 Facts

The following are the facts stored in the rule base

regularCar('Toyota-corrolar').

regularCar('Honda-SL').

decentCar('Volvo-XL').

decentCar('Acura-M').

luxuryCar('BMD-UI').

luxuryCar('lincoln-XG').

purchased('Peter', 'Honda-SL', 1997).   %Peter bought a Honda-SL in 1997 from this car dealership.

5.2.5 Logical relationships between predicates

regularCarOrAbove(V0) ←regularCar(V0).

decentCarOrAbove(V0) ←decentCar(V0).

decentCarOrAbove(V0) ←luxuryCar(V0).

egularCarOrAbove(V0) ←decentCarOrAbove(V0).

5.2.6 Askability control of the predicates

$askable(X) ←$question(X),~$not(X).

$question(moreThan60YearsOld(V0)).

$question(delivery_a_month_after_puchase(V0,V1)).

$question(payBy(X)).

$question(financialAssistance(X)).

5.2.7 Configuring rules

As to this application example, we assume that the discount policy, customer type policy and payment type policy are stored and maintained by one department and facts are stored and maintained by another department. Each department provides its own Web Service for the policy checking and query. Hence, we deploy two Web Services for this purpose – service1 for policies and service2 for facts. Also, there is a client side program that consumes the Web Services and provides a graphical interface to the user. For the client side program, it need not know the deployment details of the Web Services beforehand. All it needs to know is an entry point that is service1 in this example. Service2 will be automatically involved in the proof procedure by the rules for the purpose of the dynamic binding.

We need to add four configuring rules to the rule base of service1. These configuring rules will tell service1 which service is available for the proof of a particular goal that is beyond its scope. They are

1. $outSource(regularCar(X), 'uuid:51890f8b-eac5-45fe-8aaa-59ca745f0fc3').
2. $outSource(decentCar(X), 'uuid:51890f8b-eac5-45fe-8aaa-59ca745f0fc3').
3. $outSource(luxuryCar(X), 'uuid:51890f8b-eac5-45fe-8aaa-59ca745f0fc3').
4. $outSource(purchased(V0,V1,V2), 'uuid:51890f8b-eac5-45fe-8aaa-59ca745f0fc3').

where 'uuid:51890f8b-eac5-45fe-8aaa-59ca745f0fc3' is the bindingKey of the service2.

5.3 Interaction demo

5.3.1 User interface and fundamental operations.

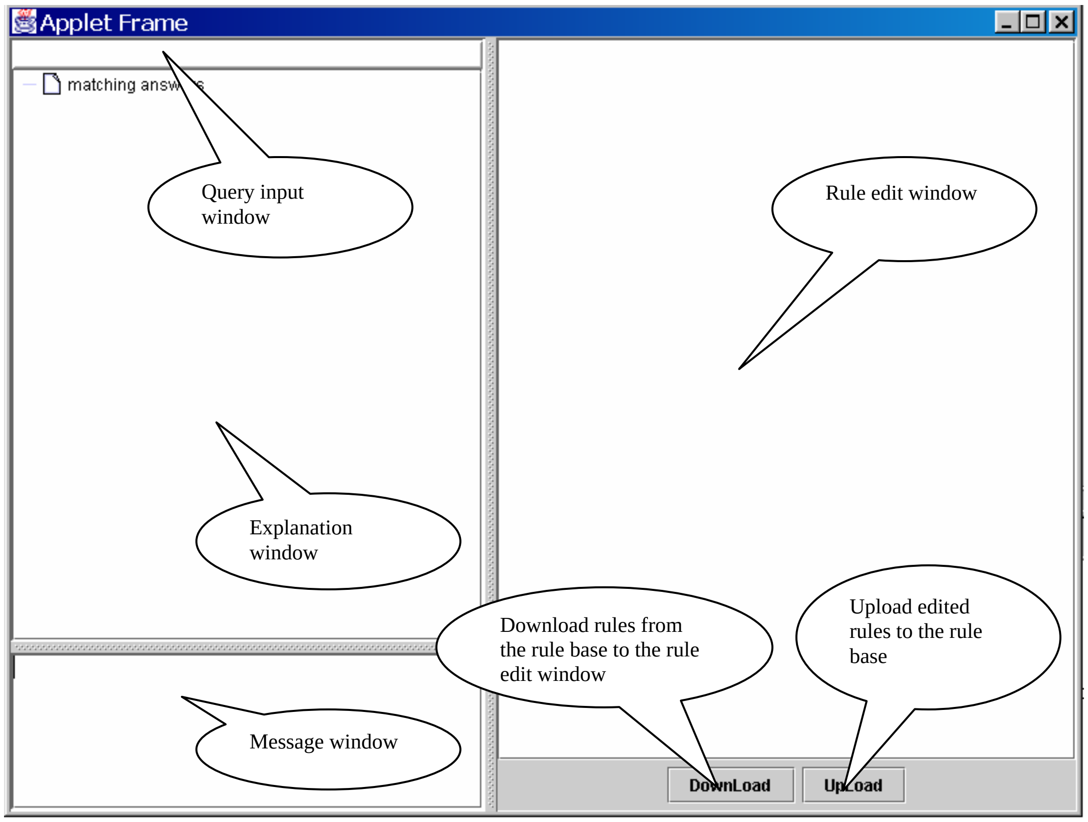


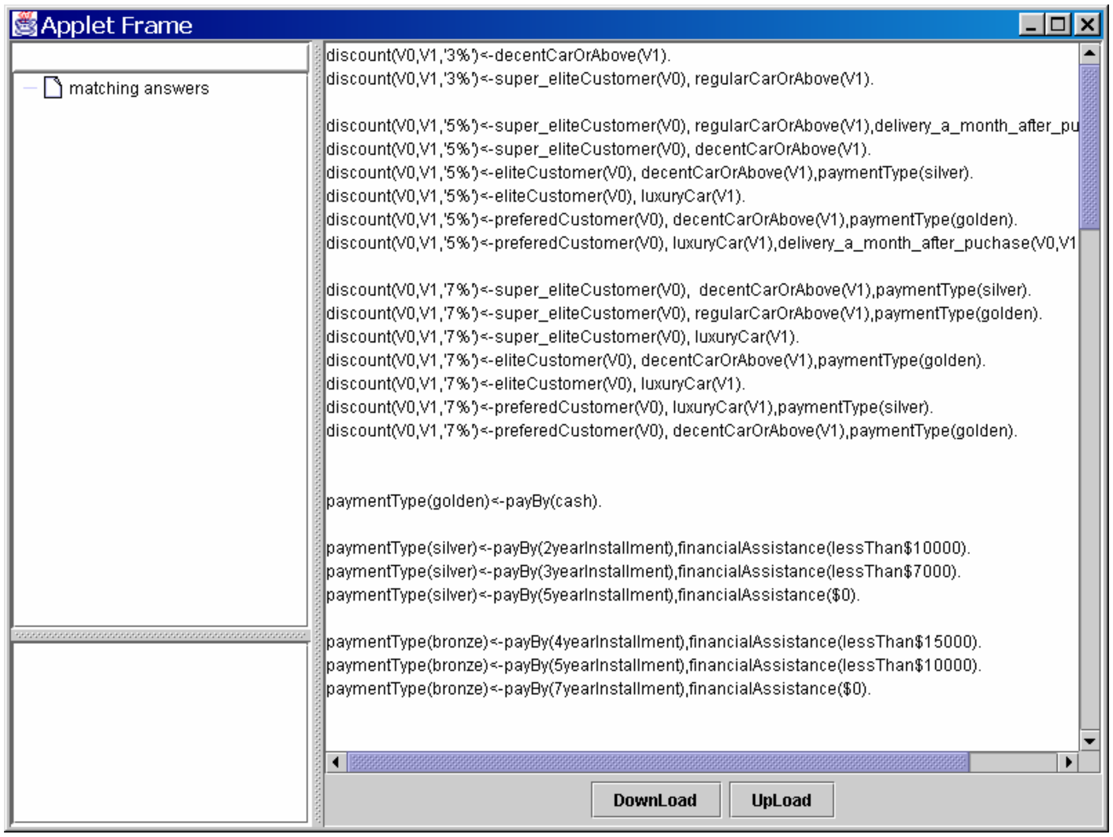
Figure 5-1 GUI Component

As shown in Figure 5-1, the user interface is composed of four windows. They are:

1. Query input window: The user can input query through this window.
2. Explanation window: The user uses this window to interact with the system for the explanations of the query.
3. Message window: Messages (e.g. error messages, debugging messages) will be shown on this window.

4. Rule edit window: Through this window, businessmen can edit (change, add, remove) and maintain rules.

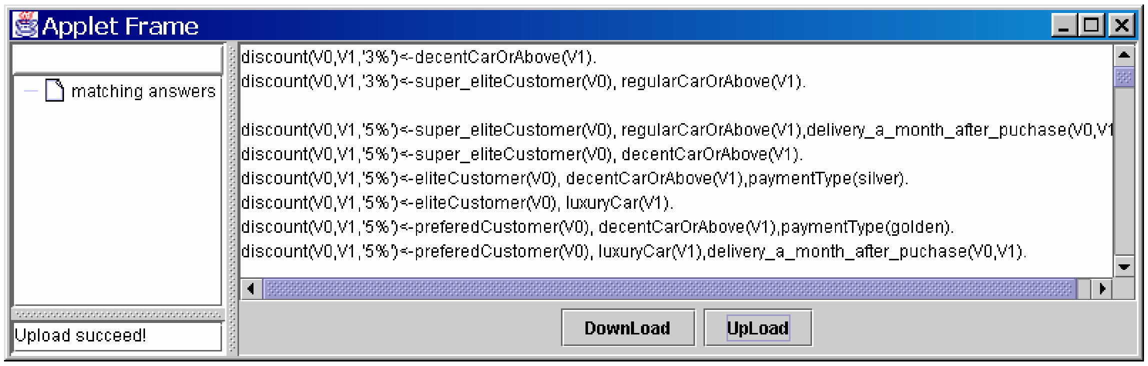
Figure 5-2 shows the downloaded rules from the rule base after “DownLoad” button was pressed.



```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),delivery_a_month_after_pu
discount(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
discount(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentType(silver).
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),paymentType(golden).
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_month_after_puchase(V0,V1

discount(V0,V1,'7%')<-super_eliteCustomer(V0),  decentCarOrAbove(V1),paymentType(silver).
discount(V0,V1,'7%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),paymentType(golden).
discount(V0,V1,'7%')<-super_eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'7%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentType(golden).
discount(V0,V1,'7%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'7%')<-preferedCustomer(V0), luxuryCar(V1),paymentType(silver).
discount(V0,V1,'7%')<-preferedCustomer(V0), decentCarOrAbove(V1),paymentType(golden).


paymentType(golden)<-payBy(cash).

paymentType(silver)<-payBy(2yearInstallment),financialAssistance(lessThan$10000).
paymentType(silver)<-payBy(3yearInstallment),financialAssistance(lessThan$7000).
paymentType(silver)<-payBy(5yearInstallment),financialAssistance($0).

paymentType(bronze)<-payBy(4yearInstallment),financialAssistance(lessThan$15000).
paymentType(bronze)<-payBy(5yearInstallment),financialAssistance(lessThan$10000).
paymentType(bronze)<-payBy(7yearInstallment),financialAssistance($0).
```

Figure 5-2 The Rule Edit Window

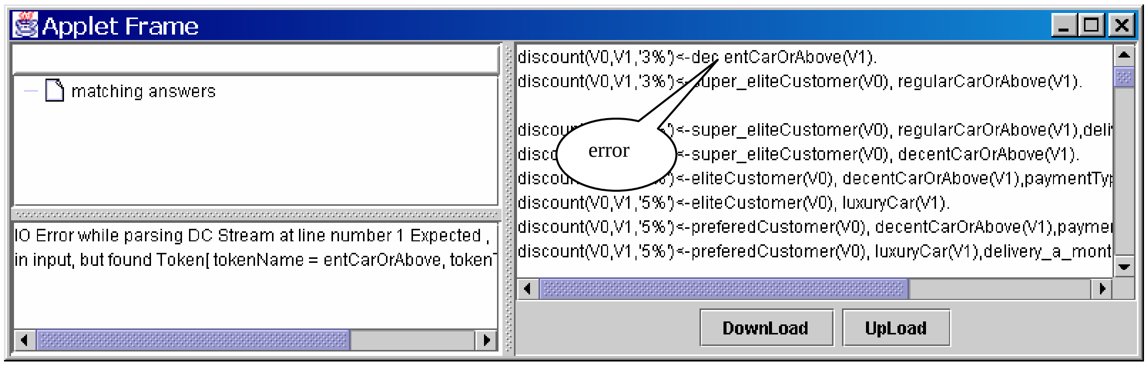
Pressing the “UpLoad” button will trigger the parsing module of jDREW [40] for the purpose of grammar checking. The message window will show the “update succeed.” message if no grammar error in the rules or according error messages if there are some grammar errors. Figure 5-3 and Figure 5-4 show an example of each of these two cases.



```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),delivery_a_month_after_puchase(V0,V1
discount(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
discount(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentType(silver).
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),paymentType(golden).
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_month_after_puchase(V0,V1).
```

```
Upload succeed!
```

Figure 5-3 Successful Upload



```
discount(V0,V1,'3%')<-dec entCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discou          )<-super_eliteCustomer(V0), regularCarOrAbove(V1),deli
disc            <-super_eliteCustomer(V0), decentCarOrAbove(V1).
discou          )<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTy
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_mont
```

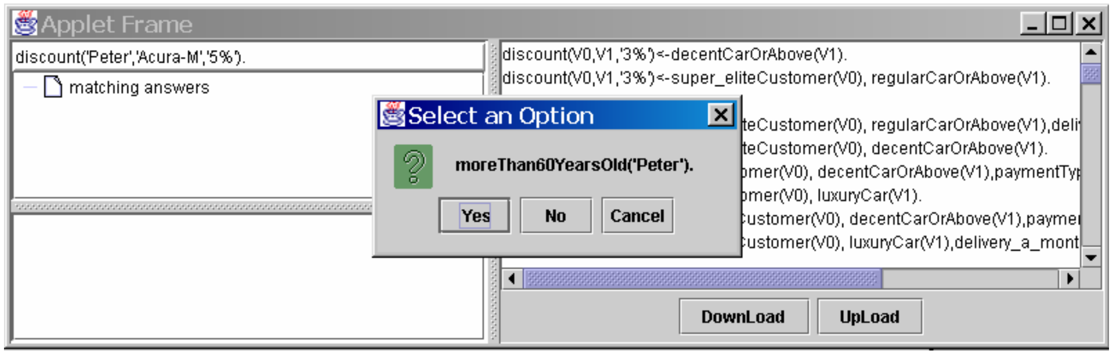
```
IO Error while parsing DC Stream at line number 1 Expected ,
in input, but found Token[ tokenName = entCarOrAbove, token
```

Figure 5-4 Error Detection

5.3.2 User query demo.

In this section, we will show the interactions between the user and the system for the query about “If Peter can get 5% discount for Acura-M”. The rule format of this query is “discount(‘Peter’, ‘Acura-M’ ,'5%').”.

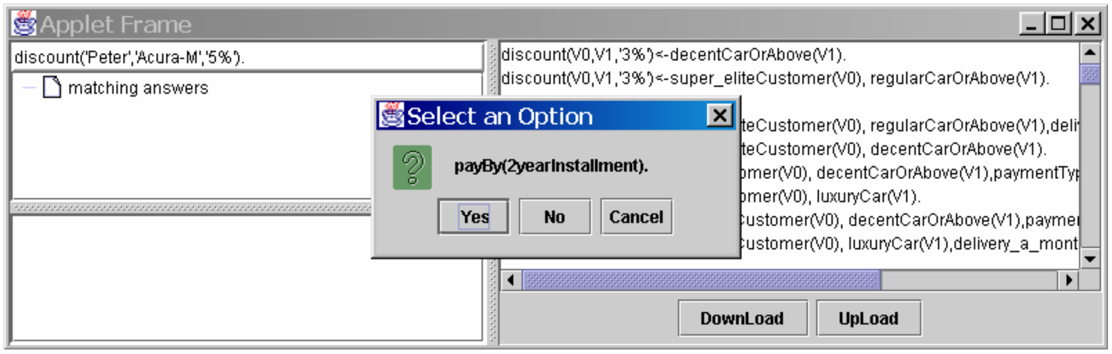
After the user input the query, the system will ask the user if he is older than 60.



```
discount('Peter','Acura-M','5%').
matching answers

discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).
teCustomer(V0), regularCarOrAbove(V1),deli
teCustomer(V0), decentCarOrAbove(V1).
omer(V0), decentCarOrAbove(V1),paymentTy
omer(V0), luxuryCar(V1).
ustomer(V0), decentCarOrAbove(V1),payme
ustomer(V0), luxuryCar(V1),delivery_a_mont

Select an Option
moreThan60YearsOld('Peter').
```

Figure 5-5 The Question (1) Initiated By The System

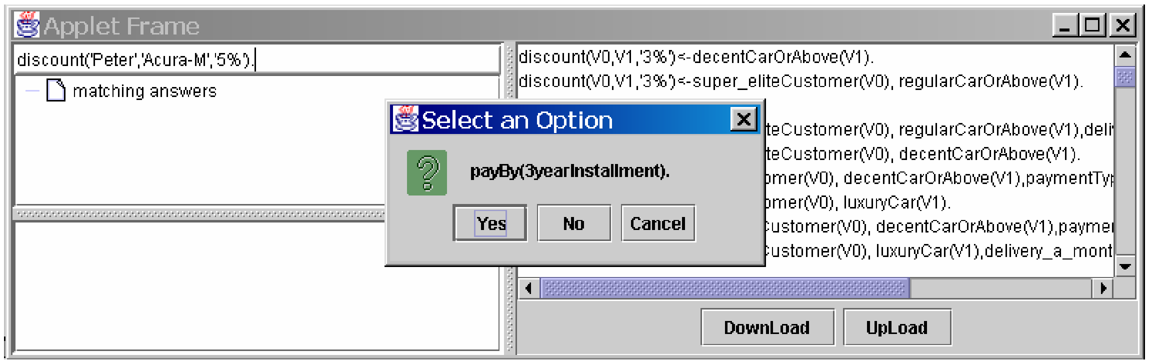
If Peter chooses "yes" for this question, the system will pop up another question, which is shown in Figure 5-6.



```
discount('Peter','Acura-M','5%').
matching answers

discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).
teCustomer(V0), regularCarOrAbove(V1),deli
teCustomer(V0), decentCarOrAbove(V1).
omer(V0), decentCarOrAbove(V1),paymentTy
omer(V0), luxuryCar(V1).
ustomer(V0), decentCarOrAbove(V1),payme
ustomer(V0), luxuryCar(V1),delivery_a_mont

Select an Option
payBy(2yearInstallment).
```

Figure 5-6 The Question (2) Initiated By The System

Assume that Peter does not want to pay by 2 years installment. Subsequently, the system will initiate a series of questions until the query is satisfied or all the possible ways are tried. Figures 5-7, 5-8 and 5-9 are examples of the questions initiated by the system in the case that the user answered "no" to the question of Figure 5-7 and "yes" to the questions of Figure 5-8, 5-9. After this, the system gets a positive conclusion of the query and the query is shown in the explanation window. The check mark in front of the query denote that the query is proved.

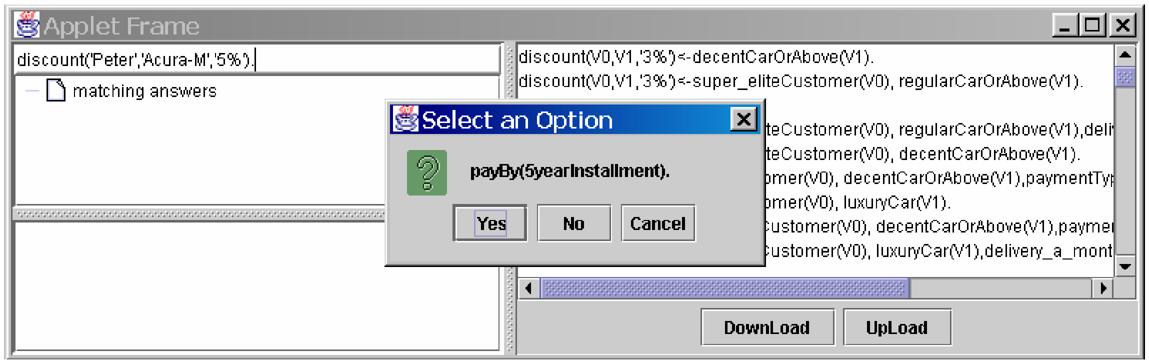


```
discount('Peter','Acura-M','5%').
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

teCustomer(V0), regularCarOrAbove(V1),deli
teCustomer(V0), decentCarOrAbove(V1).
omer(V0), decentCarOrAbove(V1),paymentTyp
omer(V0), luxuryCar(V1).
ustomer(V0), decentCarOrAbove(V1),payme
ustomer(V0), luxuryCar(V1),delivery_a_mont
```

payBy(3yearInstallment).

Figure 5-7 The Question (3) Initiated By The System

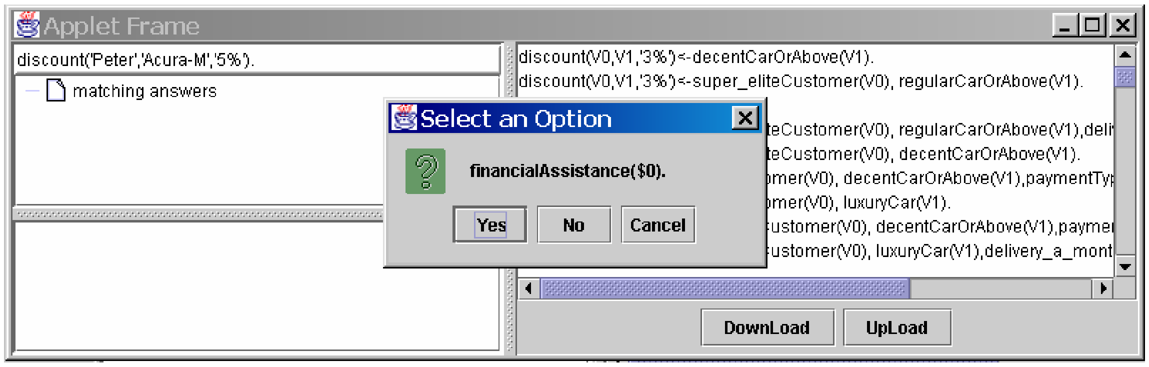


```
discount('Peter','Acura-M','5%').
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

teCustomer(V0), regularCarOrAbove(V1),deli
teCustomer(V0), decentCarOrAbove(V1).
omer(V0), decentCarOrAbove(V1),paymentTyp
omer(V0), luxuryCar(V1).
ustomer(V0), decentCarOrAbove(V1),payme
ustomer(V0), luxuryCar(V1),delivery_a_mont
```

payBy(5yearInstallment).

Figure 5-8 The Question (4) Initiated By The System

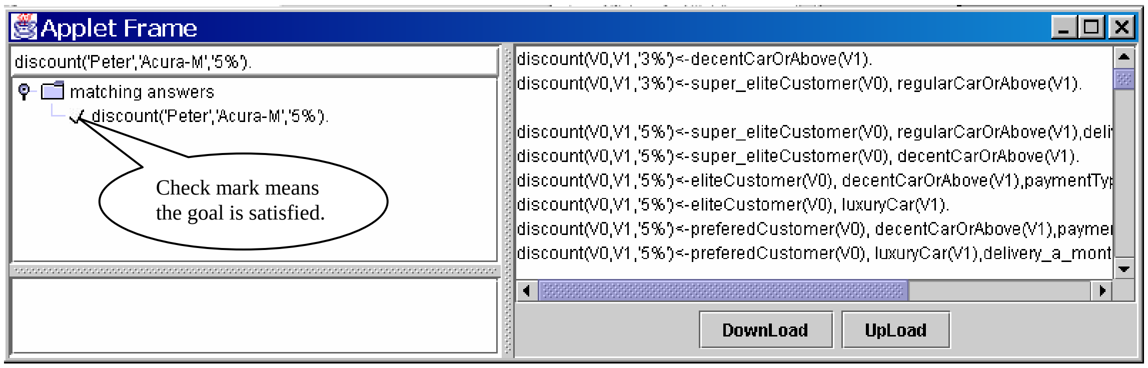


```
discount('Peter','Acura-M','5%').
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

teCustomer(V0), regularCarOrAbove(V1),deli
teCustomer(V0), decentCarOrAbove(V1).
omer(V0), decentCarOrAbove(V1),paymentTyp
omer(V0), luxuryCar(V1).
ustomer(V0), decentCarOrAbove(V1),payme
ustomer(V0), luxuryCar(V1),delivery_a_mont
```

financialAssistance($0).

Figure 5-9 The Question (5) Initiated By The System



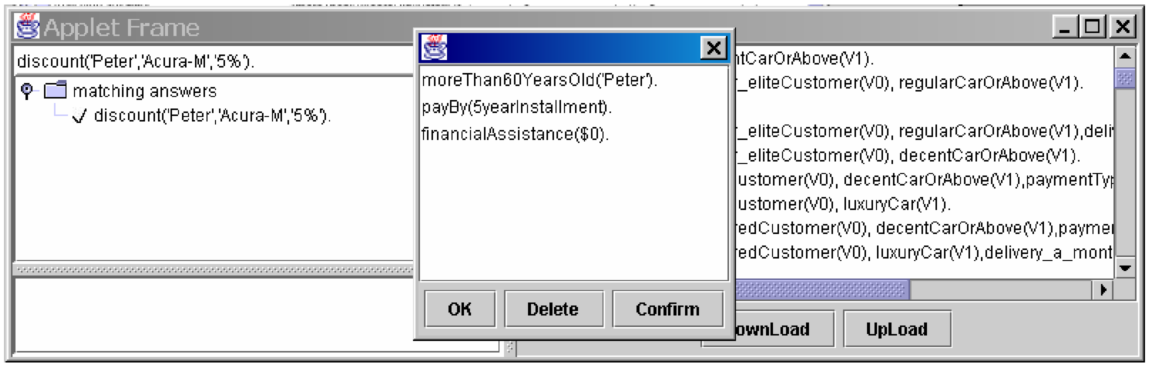
```
discount('Peter','Acura-M','5%').
```

```
matching answers
  ✓ discount('Peter','Acura-M','5%').
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),deli
discount(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
discount(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_mont
```

Figure 5-10 The Result Of The Query

By clicking the right mouse button over the explanation window, the user can check again about the pre-conditions on which the query is proved. This is shown on Figure 5-11.



```
discount('Peter','Acura-M','5%').
matching answers
  discount('Peter','Acura-M','5%').
```

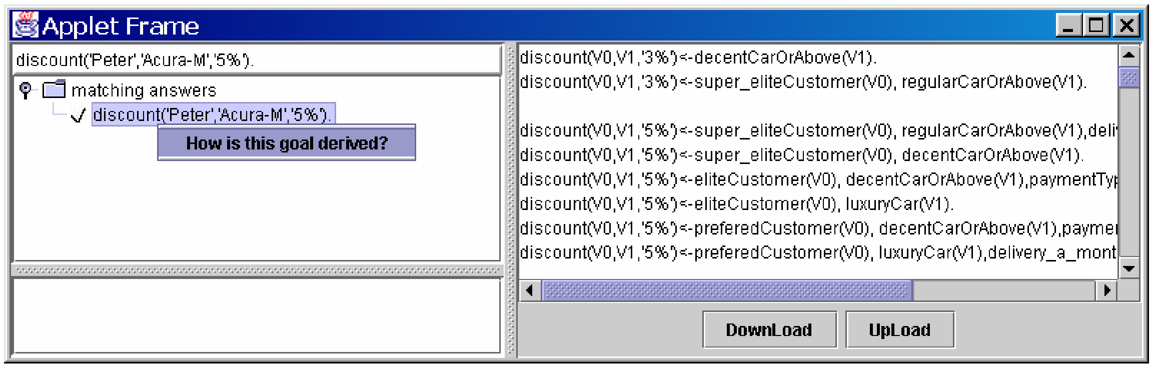
```
moreThan60YearsOld('Peter').
payBy(5yearInstallment).
financialAssistance($0).
```

```
ntCarOrAbove(V1).
_eliteCustomer(V0), regularCarOrAbove(V1).

_eliteCustomer(V0), regularCarOrAbove(V1),deli
_eliteCustomer(V0), decentCarOrAbove(V1).
ustomer(V0), decentCarOrAbove(V1),paymentTy
ustomer(V0), luxuryCar(V1).
edCustomer(V0), decentCarOrAbove(V1),payme
edCustomer(V0), luxuryCar(V1),delivery_a_mont
```

Figure 5-11 The Pre-conditions Window

Consequently, the user can choose to ask “how?” question on the query to see how the query is proved. As shown in Figure 5-13, each child of the tree structure in the explanation window is a precondition for the proof of its father. So, the hierarchical structure of Figure 5-13 could be read as:

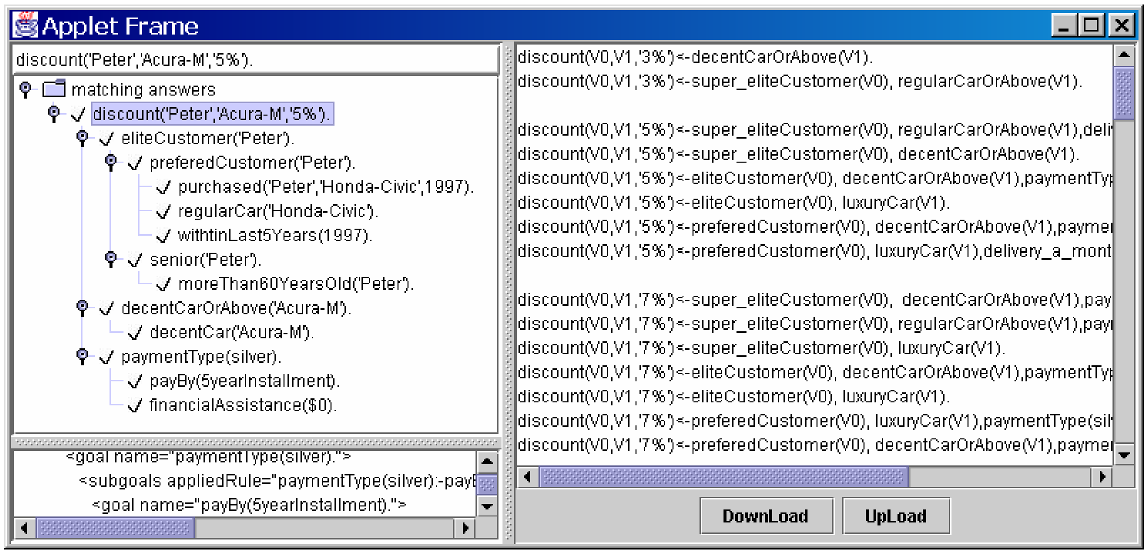
- Peter can get 5% discount on Acura-M because 1. Peter is an elite customer; 2. Acura-M is a decent car; 3. Peter chooses silver payment type.
- Peter is a elite customer because Peter is a senior preferred customer .
- Peter is a preferred customer because Peter bought a regular car ‘Honda-Civic’ in the last five years.
- Peter is a senior person because Peter is older than 60.
- Peter chooses the 5 years installment payment without financial assistance, which is a silver payment.



```
discount('Peter','Acura-M','5%').
matching answers
  discount('Peter','Acura-M','5%').
        How is this goal derived?
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),deli
discount(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
discount(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_mont
```

Figure 5-12 The "How" Quesiton

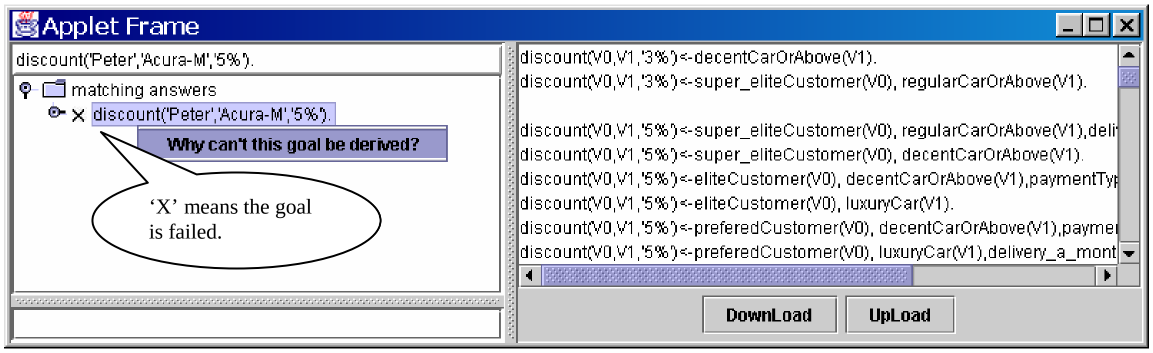


```
discount('Peter','Acura-M','5%').
matching answers
  ✓ discount('Peter','Acura-M','5%').
    ✓ eliteCustomer('Peter').
      ✓ preferedCustomer('Peter').
        ✓ purchased('Peter','Honda-Civic',1997).
        ✓ regularCar('Honda-Civic').
        ✓ withtinLast5Years(1997).
      ✓ senior('Peter').
        ✓ moreThan60YearsOld('Peter').
    ✓ decentCarOrAbove('Acura-M').
      ✓ decentCar('Acura-M').
    ✓ paymentType(silver).
      ✓ payBy(5yearInstallment).
      ✓ financialAssistance($0).

<goal name="paymentType(silver).">
  <subgoals appliedRule="paymentType(silver):-payb
    <goal name="payBy(5yearInstallment).">
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),deli
discount(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
discount(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_mont

discount(V0,V1,'7%')<-super_eliteCustomer(V0),  decentCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'7%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'7%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'7%')<-preferedCustomer(V0), luxuryCar(V1),paymentType(sil
discount(V0,V1,'7%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
```

Figure 5-13 The System Response On the "How" Question.

As described before, Peter answers the questions under the circumstance that Peter does not know how much the answer will help him towards or hinder him from obtaining the intended discount level. The explanation system serves as a way for the user to know how the decision is made and also give the user a chance to change his choice. For example, assuming Peter only gives the question "moreThan60Years(Peter)?" the "yes" answer, the system will deny Peter's query,


Applet Frame
matching answers
Why can't this goal be derived?
'X' means the goal is failed.
DownLoad
UpLoad

```
discount('Peter','Acura-M','5%').
```

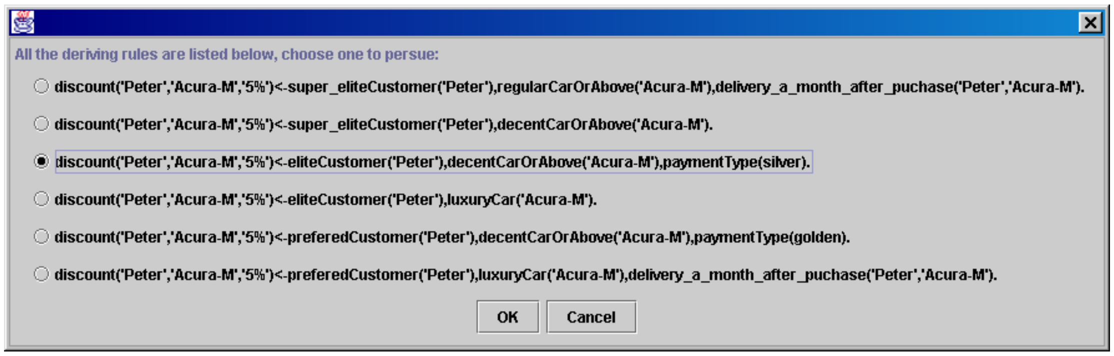
```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),deli
discount(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
discount(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_mont
```

Figure 5-14 The "Why not" Question.

which is shown in Figure 5-14. The 'X' in front of the query means the query is denied. In this case, Peter can ask "why not" question on the query for explanations.

As a response to this "why not" question, the system will show all the possible ways through which Peter can get the 5% discount. What Peter needs to do is to choose one way to continue the query.

Figure 5-15 shows Peter chooses the third option to continue.

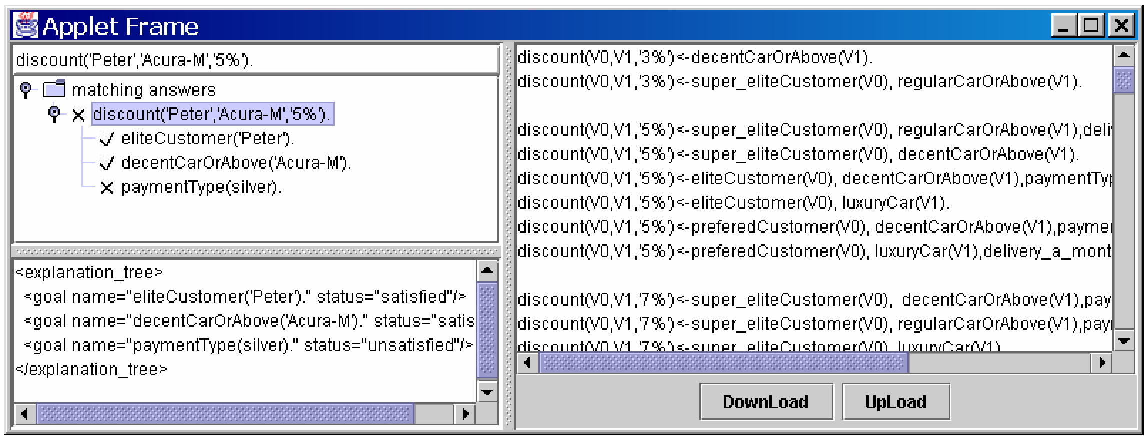


All the deriving rules are listed below, choose one to persue:

- ○ discount('Peter','Acura-M','5%')<-super_eliteCustomer('Peter'),regularCarOrAbove('Acura-M'),delivery_a_month_after_puchase('Peter','Acura-M').
- ○ discount('Peter','Acura-M','5%')<-super_eliteCustomer('Peter'),decentCarOrAbove('Acura-M').
- ● discount('Peter','Acura-M','5%')<-eliteCustomer('Peter'),decentCarOrAbove('Acura-M'),paymentType(silver).
- ○ discount('Peter','Acura-M','5%')<-eliteCustomer('Peter'),luxuryCar('Acura-M').
- ○ discount('Peter','Acura-M','5%')<-preferedCustomer('Peter'),decentCarOrAbove('Acura-M'),paymentType(golden).
- ○ discount('Peter','Acura-M','5%')<-preferedCustomer('Peter'),luxuryCar('Acura-M'),delivery_a_month_after_puchase('Peter','Acura-M').

Figure 5-15 The Choice Dialog

As a response, the system will show further explanations under this way. Figure 5-16 shows that Peter satisfied the first two preconditions but failed the third. Peter can continue to ask "how" questions on the satisfied preconditions and "why not" questions on the unsatisfied conditions. Figures 5-17 ~ Figure 5-22 show the following interactions that might happen between Peter and the system. The Question mark in front of a node of the explanation tree (on Figures 5-20, 5-21 and 5-22) means that the node is askable and can be satisfied by answering “yes”. The information exchange between the client program and the Web Service is shown in the message window in the XML format.

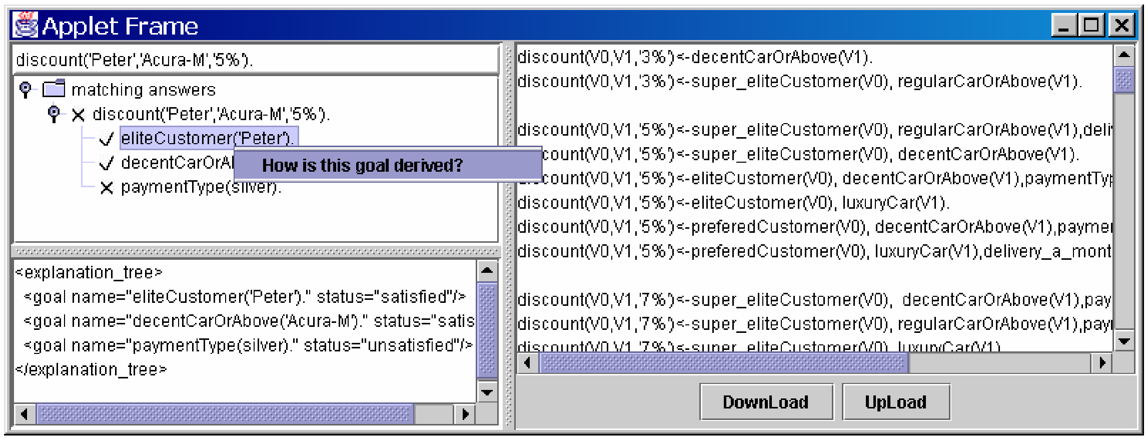


```
discount('Peter','Acura-M','5%').
matching answers
  ✗ discount('Peter','Acura-M','5%').
      ✓ eliteCustomer('Peter').
      ✓ decentCarOrAbove('Acura-M').
      ✗ paymentType(silver).
```

```
<explanation_tree>
 <goal name="eliteCustomer('Peter')." status="satisfied"/>
 <goal name="decentCarOrAbove('Acura-M')." status="satis
 <goal name="paymentType(silver)." status="unsatisfied"/>
</explanation_tree>
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),deli
discount(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
discount(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTy
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_mont

discount(V0,V1,'7%')<-super_eliteCustomer(V0),  decentCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), luxuryCar(V1)
```

Figure 5-16 The System Response On the “Why not” Question.

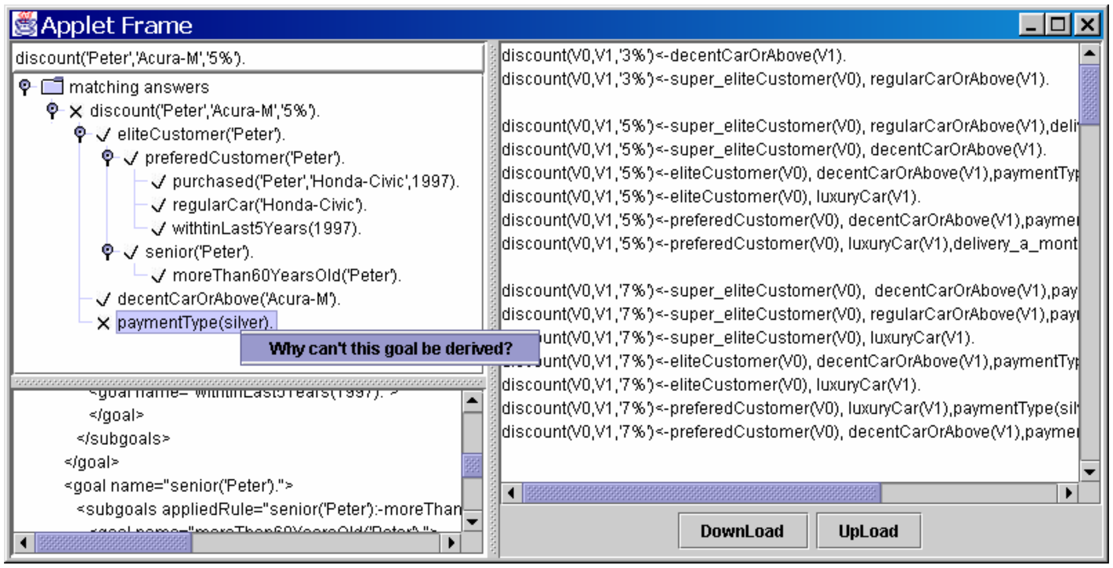


```
discount('Peter','Acura-M','5%').
matching answers
  ✗ discount('Peter','Acura-M','5%').
      ✓ eliteCustomer('Peter').
      ✓ decentCarOrAl
      ✗ paymentType(silver).
```

```
<explanation_tree>
 <goal name="eliteCustomer('Peter')." status="satisfied"/>
 <goal name="decentCarOrAbove('Acura-M')." status="satis
 <goal name="paymentType(silver)." status="unsatisfied"/>
</explanation_tree>
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),deli
  count(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
  count(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTy
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_mont

discount(V0,V1,'7%')<-super_eliteCustomer(V0),  decentCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), luxuryCar(V1)
```

Figure 5-17 The “How” Question.

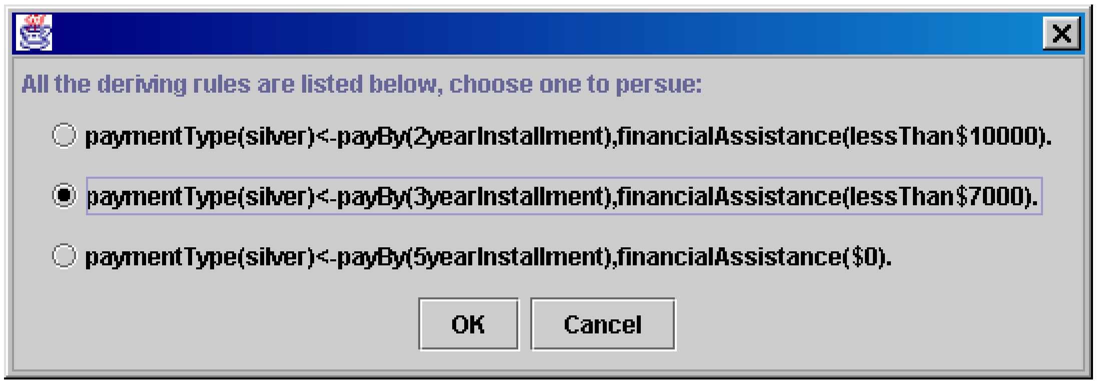


```
discount('Peter','Acura-M','5%').

matching answers
  X discount('Peter','Acura-M','5%').
    ✓ eliteCustomer('Peter').
      ✓ preferedCustomer('Peter').
        ✓ purchased('Peter','Honda-Civic',1997).
        ✓ regularCar('Honda-Civic').
        ✓ withtinLast5Years(1997).
      ✓ senior('Peter').
        ✓ moreThan60YearsOld('Peter').
    ✓ decentCarOrAbove('Acura-M').
    X paymentType(silver).
```

```
          </goal>
        </subgoals>
      </goal>
      <goal name="senior('Peter')."> 
        <subgoals appliedRule="senior('Peter'):-moreThan
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),deli
discount(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
discount(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_mont

discount(V0,V1,'7%')<-super_eliteCustomer(V0),  decentCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),pay
    unt(V0,V1,'7%')<-super_eliteCustomer(V0), luxuryCar(V1).
    unt(V0,V1,'7%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'7%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'7%')<-preferedCustomer(V0), luxuryCar(V1),paymentType(sil
discount(V0,V1,'7%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
```

Figure 5-18 The “Why not” Question.



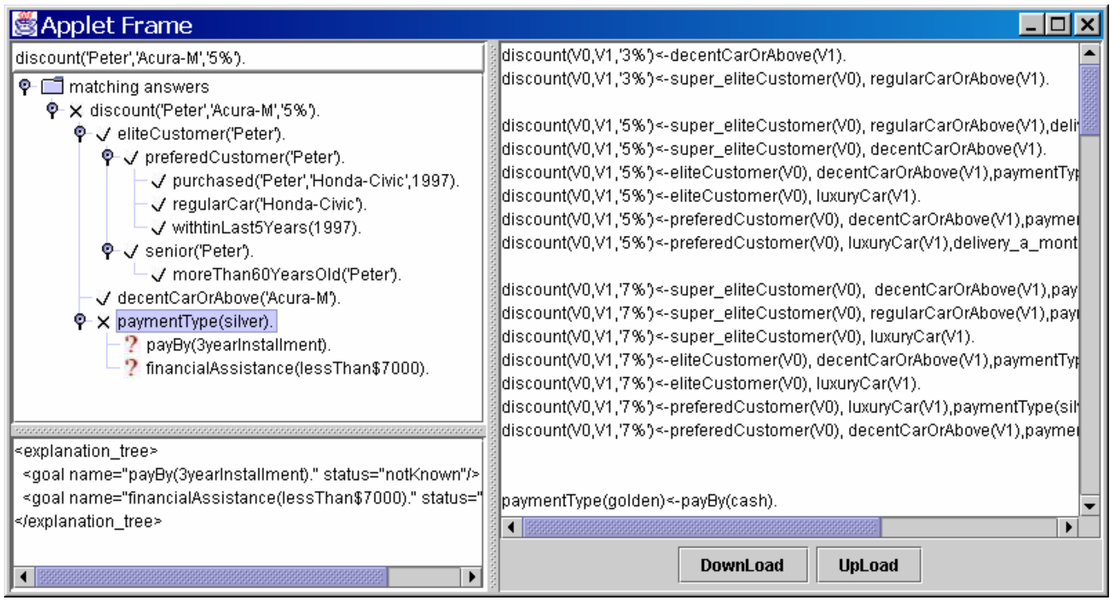
All the deriving rules are listed below, choose one to persue:

( ) paymentType(silver)<-payBy(2yearInstallment),financialAssistance(lessThan$10000).

(●) paymentType(silver)<-payBy(3yearInstallment),financialAssistance(lessThan$7000).

( ) paymentType(silver)<-payBy(5yearInstallment),financialAssistance($0).

Figure 5-19 The Choice Dialog

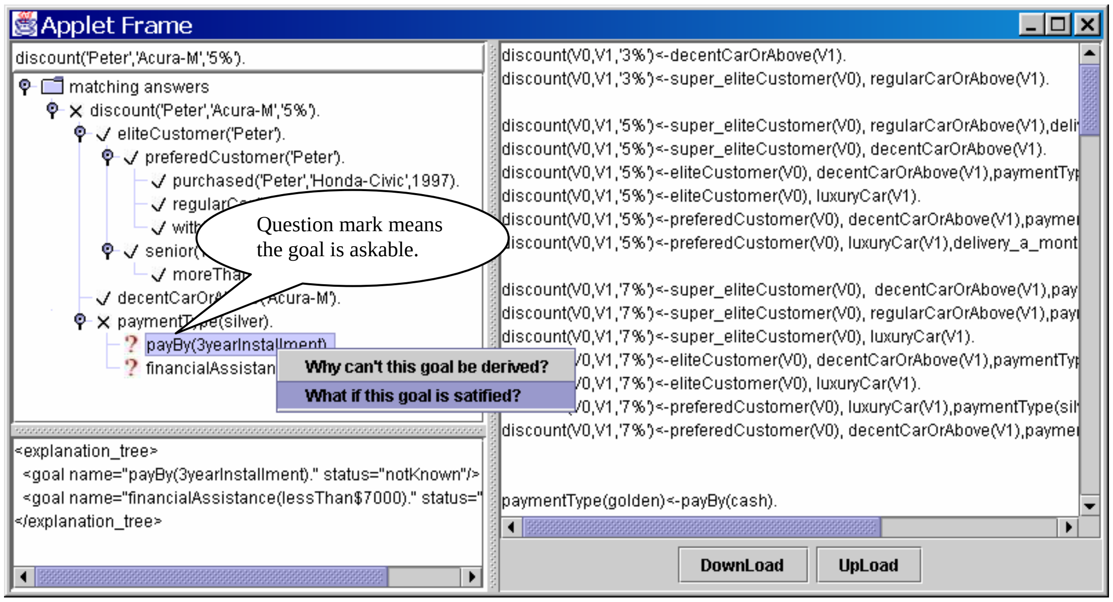


```
discount('Peter','Acura-M','5%').
matching answers
  X discount('Peter','Acura-M','5%').
    ✓ eliteCustomer('Peter').
      ✓ preferedCustomer('Peter').
        ✓ purchased('Peter','Honda-Civic',1997).
        ✓ regularCar('Honda-Civic').
        ✓ withtinLast5Years(1997).
      ✓ senior('Peter').
        ✓ moreThan60YearsOld('Peter').
    ✓ decentCarOrAbove('Acura-M').
    X paymentType(silver).
      ? payBy(3yearInstallment).
      ? financialAssistance(lessThan$7000).
```

```
<explanation_tree>
  <goal name="payBy(3yearInstallment)." status="notKnown"/>
  <goal name="financialAssistance(lessThan$7000)." status="
</explanation_tree>
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),deli
discount(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
discount(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_mont

discount(V0,V1,'7%')<-super_eliteCustomer(V0),  decentCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'7%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'7%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'7%')<-preferedCustomer(V0), luxuryCar(V1),paymentType(sil
discount(V0,V1,'7%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme


paymentType(golden)<-payBy(cash).
```

Figure 5-20 The System Response On the "Why not" Question.

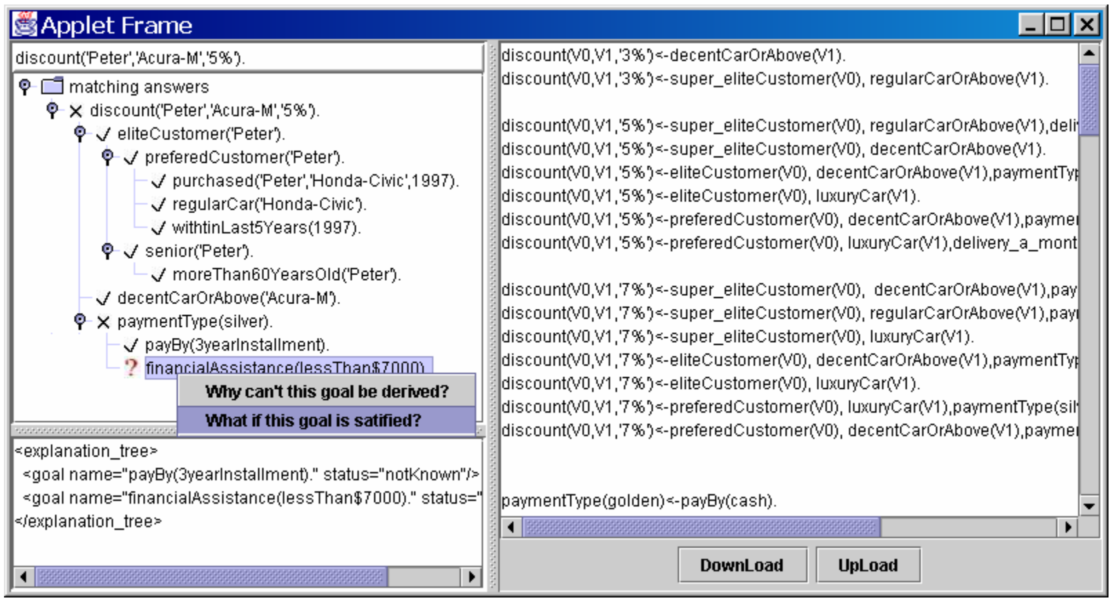

Applet Frame
Question mark means the goal is askable.
Why can't this goal be derived?
What if this goal is satified?
DownLoad
UpLoad

```
discount('Peter','Acura-M','5%').
matching answers
  X discount('Peter','Acura-M','5%').
    ✓ eliteCustomer('Peter').
      ✓ preferedCustomer('Peter').
        ✓ purchased('Peter','Honda-Civic',1997).
        ✓ regular
        ✓ with
      ✓ senior(
        ✓ moreTha
    ✓ decentCarOr          Acura-M').
    X paymentT    e(silver).
      ? payBy(3yearInstallment).
      ? financialAssistan
```

```
<explanation_tree>
  <goal name="payBy(3yearInstallment)." status="notKnown"/>
  <goal name="financialAssistance(lessThan$7000)." status="
</explanation_tree>
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),deli
discount(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
discount(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_mont

discount(V0,V1,'7%')<-super_eliteCustomer(V0),  decentCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), luxuryCar(V1).
          /0,V1,'7%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
          /0,V1,'7%')<-eliteCustomer(V0), luxuryCar(V1).
          /0,V1,'7%')<-preferedCustomer(V0), luxuryCar(V1),paymentType(sil
discount(V0,V1,'7%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme


paymentType(golden)<-payBy(cash).
```

Figure 5-21 The "What if" Question

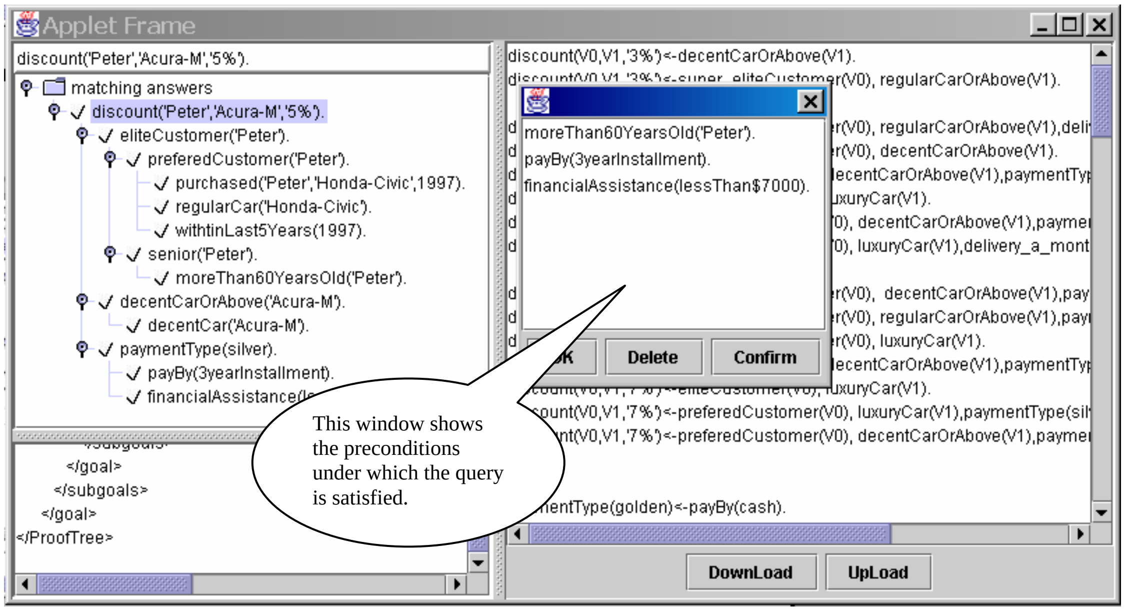

Applet Frame
Why can't this goal be derived?
What if this goal is satified?
DownLoad
UpLoad

```
discount('Peter','Acura-M','5%').
matching answers
  ✗ discount('Peter','Acura-M','5%').
    ✓ eliteCustomer('Peter').
      ✓ preferedCustomer('Peter').
        ✓ purchased('Peter','Honda-Civic',1997).
        ✓ regularCar('Honda-Civic').
        ✓ withtinLast5Years(1997).
      ✓ senior('Peter').
        ✓ moreThan60YearsOld('Peter').
    ✓ decentCarOrAbove('Acura-M').
    ✗ paymentType(silver).
      ✓ payBy(3yearInstallment).
      ? financialAssistance(lessThan$7000).
```

```
<explanation_tree>
 <goal name="payBy(3yearInstallment)." status="notKnown"/>
 <goal name="financialAssistance(lessThan$7000)." status="
</explanation_tree>
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).

discount(V0,V1,'5%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),deli
discount(V0,V1,'5%')<-super_eliteCustomer(V0), decentCarOrAbove(V1).
discount(V0,V1,'5%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'5%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'5%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme
discount(V0,V1,'5%')<-preferedCustomer(V0), luxuryCar(V1),delivery_a_mont

discount(V0,V1,'7%')<-super_eliteCustomer(V0),  decentCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), regularCarOrAbove(V1),pay
discount(V0,V1,'7%')<-super_eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'7%')<-eliteCustomer(V0), decentCarOrAbove(V1),paymentTyp
discount(V0,V1,'7%')<-eliteCustomer(V0), luxuryCar(V1).
discount(V0,V1,'7%')<-preferedCustomer(V0), luxuryCar(V1),paymentType(sil
discount(V0,V1,'7%')<-preferedCustomer(V0), decentCarOrAbove(V1),payme


paymentType(golden)<-payBy(cash).
```

Figure 5-22 The System Response On the "What if" Question (1).


Applet Frame
OK
Delete
Confirm
DownLoad
UpLoad
This window shows the preconditions under which the query is satisfied.

```
discount('Peter','Acura-M','5%').
matching answers
  ✓ discount('Peter','Acura-M','5%').
    ✓ eliteCustomer('Peter').
      ✓ preferedCustomer('Peter').
        ✓ purchased('Peter','Honda-Civic',1997).
        ✓ regularCar('Honda-Civic').
        ✓ withtinLast5Years(1997).
      ✓ senior('Peter').
        ✓ moreThan60YearsOld('Peter').
    ✓ decentCarOrAbove('Acura-M').
      ✓ decentCar('Acura-M').
    ✓ paymentType(silver).
      ✓ payBy(3yearInstallment).
      ✓ financialAssistance(l
```

```
      </goal>
     </subgoals>
    </goal>
</ProofTree>
```

```
moreThan60YearsOld('Peter').
payBy(3yearInstallment).
financialAssistance(lessThan$7000).
```

```
discount(V0,V1,'3%')<-decentCarOrAbove(V1).
discount(V0,V1,'3%')<-super_eliteCustomer(V0), regularCarOrAbove(V1).
```

Figure 5-23 The System Response On the "What if" Question (2).

6 Conclusions and future work

6.1 Conclusions

Business rules are very powerful for the development of enterprise systems that involve management of the policies, regulations and expertise. By separating the rules from the software code and representing them in formal rule languages, businesses can respond to ever changing market rapidly at extremely low cost. In addition, the inference engine takes over most of the deduction work, which makes changing policies or regulations very easy for a well-designed rule base.

One important characteristic of the business rule system is its ability to initialize interactions that are customized for each query. The interactions take the form of questions that remind the user of necessary information input or ask the user to make a choice or decision. Asking a relevant question at a relevant time is a key consideration for building user-friendly interfaces. The prototype uses the predefined predicate to denote if a particular question belongs to the user's domain of knowledge so that all the questions are relevant to the user. Also, the inference engine's ability to select and arrange the rules and facts that are logically related to each proof makes it possible for the system to only ask questions that are relevant to the query. In addition, based on the hierarchical structure of the logical relationships among business rules and facts, we can further optimize the interaction so that the interactions between the system and the user are kept to the minimum. This thesis proposes several methods and strategies to rank each question in order to minimize the number of questions asked by the system.

Explanations are important in the interaction with users, where they can take the initiative. The prototype is designed to give explanations to three kinds of questions. To "how" questions, the system will give an overall answer that shows all the steps involved in the proof. As to the "why not" question, the user is led through an interactive process in which he/she can explore all the situations and get explanations. During the interaction, a "what if" question could be asked at any time to see how a choice of decision or some additional information can affect the whole outcome. The tested demo system illustrates that this approach is a feasible design in terms of flexibility (an concern of usability) and tractability (a concern of system implementation).

Web Service technology gives a strong architectural support for applying business rule technology to the overall environment of e-commerce. SOAP protocol makes the communication between business rule systems completely independent from their implementation specifications and running environments. UDDI provides a standard way to find business rule systems that are deployed as Web Services. WSDL provides a formal way to describe the Web Service interface and invocation method, which makes binding Web Services at runtime possible. In addition, it provides huge potentials for the further automation of the software integration.

In addition to representing the business policies and regulations, rules are also used for the service integration in the prototype system. This approach brings high scalability and more flexible controls over the architectural aspect of the system. The user can change the participating services and partners in the same way as they change policies and regulations. More important, the inherent power of inference engines and rules makes it possible for the system integration also be based on the business policies.

6.2 Future work

The implementation of the prototype system is only the first attempt to incorporate business rule technology into the Web Service architecture. It demonstrates a paradigm through which business rules and Web Services technologies may complement each other for the establishment of distributed e-commerce systems based on policies. This paradigm will see many further developments, and will draw power from a variety of relevant research areas. For example, RuleML, a rule markup language for the semantic web, could be adopted into this paradigm for the purpose of sharing information among the autonomous rule-based Web Services.

Many of the research works have been done on the improvement of the readability of the rules. This research area focuses on the description of the rules in human languages. In the existing systems, the most common approach is to add “comments” to each term and atomic formula and then programmatically assemble them into a human language description. At the current stage, no satisfactory method exists to make the synthesized human language a reliable way to describe rules. Yet, as a supplementary resort to the traditional rule representation, it considerably improves the usability of the rule system.

Another pragmatic improvement of the prototype could be introducing the reaction rules into the system. Reaction rules provide a uniform mechanism to connect the conclusions derived by the rule system to the “real” business actions. Business rules that specify the various steps of a business process can be encoded in the form of reaction rules. With the reaction rules, the inference engine could start a sequence of actions that compose a business activity. In different situations, the same business activity may be proved by different combination of policies and therefore triggers different set of business actions, which gives the business rule technology a strong stand for complex event flow managements [36].

Future work may also include the software agents into the architecture. For example, the current implementation delegates a query to the other business rule services through “$outsource” rules that are established for the purpose of mapping goals to the services that can solve them. An alternative solution is to introduce a software agent to handle all the queries that need to be outsourced. One of the obvious advantages of this solution is that it simplifies the functions and the structure of the system by removing the mapping task and rules to an independent software agent. On the other hand, one or more information agents could be used by the user to provide non-crucial user information on his behalf.

At the next stage of development, performance tests will be introduced for the evaluation of the following aspects of the system:

1. Scalability test: Scalability test gives the performance evaluation of the system under various scenarios. In our particular prototype system, emphasis is put on the following aspects:
    - How does the system perform when handling large amount of requests?
    - How does the system perform when dealing with a huge knowledge or information base?
    - How does the system perform when a hierarchy of services in a large scale is required and called by the local proof procedure?

- Is it easy to add more distributed computing power to the system when necessary?

2. Usability test: The usability test of the prototype system includes two aspects. First, the operation of the system should be concise and effective so that skilled users can use it effectively. On the other hand, the operation of the system should also be straightforward enough so that the naive users can use it easily. Since the operation demands of these two groups of people are contradictory, the test result largely depends on whether the system could let the users choose their preferred representation of rules and explanations, or could automatically choose an appropriate representation for the user.

Extendibility is another important aspect of the system design on the next stage of development. It decides how easily the system could be adapted or configured to satisfy the functionality demands of different application areas. The extendibility concern of the prototype system will focus on the following two questions:

1. Is the interaction mode general and explanatory enough for a large range of application area?
2. How could we add additional function modules to the system without change the interface and the interaction mode of the system?

Bibliography

1. Adelman, Leonard., Sharon, L. Riedel.: *Handbook for evaluating knowledge-based systems: conceptual framework and compendium of methods*, Kluwer Academic Publishers, Boston, 1997.

2. Aissi, Selim., Malu, Pallavi., Srinivasan, Krishnamurthy.: E-Business Process Modeling: The Next Big Step, *IEEE Computer*, May 2002.

3. Antoniou, G., Arief, M.: Executable Declarative Business Rules and Their Use in Electronic Commerce, in *International Journal of Intelligent Systems in Accounting Finance and Management*, v. 10, n. 4, p. 211, 2001.

4. Asuman, Dogac., Ibrahim, Cingil., Gokce, Laleci., Yildiray, Kabak.: Improving the Functionality of UDDI Registries through Web Service Semantics, in *Lecture Notes in Computer Science*, 2002.

5. Booch, Grady. et al.: *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

6. *BRULEE - Business Rule Extraction And Exploitation,* http://www.cs.cf.ac.uk/brulee/bruleehome.html, Sep 2002.

7. Buchanann, B., Shortliffe, E.: Rule-based Expert Systems: *The MYCIN Experiments of the Stanford Heuristic Programming Programming Project*, Addison-Wesley, Reading, MA, 1984.

8. Bumpus, W., Sweitzer, J. W., Thompson, P., Westerinen, A. R., Williams, R. C.: *Common Information Model Implementing the Object Model for Enterprise*

Management, Wiley Computer Publishing, John Wiley & Sons, Inc., New York, 2000.

9. *Business Rules for Electronic Commerce: Project at IBM T.J. Watson Research*, http://www.research.ibm.com/rules/home.html, Oct 2002.

10. Darugar, Parand Tony.: *Web Services -- Component Based Application Revolution (Slides)*, http://www.xmltoday.com/articles/presentations/xmldevcon2001.ppt, April 2002.

11. Dias, D. M., et al.: E-commerce interoperability with IBM's WebSphere Commerce products, in *IBM Systems Journal,* v. 41, n. 2, 2002.

12. Duffy, David A.: *Principles of Automated Theorem Proving*, John Willey & Sons Ltd., England, 1991.

13. *EbXML Message Service Specification*, Version 1.0, UN/CEFACT and OASIS, http://www.ebxml.org/specs/ebMS.pdf, Dec. 2002.

14. Feltovich, Paul J., Ford, Kenneth M.: *Expertise In Context: human and machine*, AAAI Press, Calif., U.S.A., 1997.

15. Ferrand, Gerard., Tessier, Alexandre.: *Declarative Debuggin,* University d'Oleans and INRIA LIFO, France, 1994.

16. Forget, Pascal.: Apache 1.3.23+Tomcat 4.0.2+Load Balancing, Ubeans Technology, http://www.ubeans.com/tomcat/index.html, Jan. 2003.

17. Gallier, Jean H.: *LOGIC FOR COMPUTER SCIENCE – Foundations of Automatic Theorem Proving,* HARPER & ROW Publishers, New York, 1986.

18. Glass, Graham.: *Web Services Building Blocks for Distributed Systems,* Prentice-Hall Inc., 2002.

19. Graham, Steve et al.: *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI,* Sams Publishing, 2001.

20. Gottesdiener, Ellen.: Business Rules Show Power and Promise, *Application Programming Trends*, v. 4, n. 3, Mar. 1997.

21. *Guide Business Rules Project*, Final Report, 11/95, http://www.businessrulesgroup.org/white_paper/br01c0.htm, Sep 2002.

22. *ILOG Jrules White Paper*, http://www.ilog.com/products/rules/engines/jrules40/, Nov 2002.

23. Jia, Keping., Spencer, Bruce.: Negotiating Exchanges of Private Information for Web Service Eligibility, to appear in AI'2003 proceedings.

24. Kannan, P. K., Kopalle, P. K.: Dynamic Pricing on the Internet: Importance and Implications for Consumer Behavior, *International Journal of Electronic Commerce 5*, n. 3, 63–83, 2001.

25. Kreger, Heather.: *Web Services Conceptual Architecture,* IBM Corporation, 2001.

26. Lee, J. et al.: Design and Implementation of an Interactive Visual Analysis System for e-Sourcing, *Research Report RC 22045*, IBM T. J. Watson Research Center, New York, 2001.

27. Marcelo, A. T. de Aragão., Alvaro, A. A. Fernandes.: Characterizing Web Service Substitutivity with Combined Deductive and Inductive Engines, in

Proceedings of the 2nd Biennial International Conference on Advances in Information Systems, Izmir, Turkey, Oct. 2002.

28. Marron, Kevin.: The Quiet Revolution in E-Business, *The Globe and Mail*, 2002.

29. *NARUC Uniform Business Rules Project*, http://www.caem.org/website/archive/Orientation_ Memo_Revised.pdf, July 2002.

30. Merritt, Dennis.: *Building Expert Systems in Prolog*, Springer-Verlag, New York, U.S.A., 1989.

31. Poole, David., Mackworth, Alan., Goebel, Randy.: *Computational Intelligence: A Logical Approach*, Oxford University Press, New York, U.S.A., 1997

32. Purcell, Kenton.: Creating a Web Service: Product Overview, *XML Insider*, Dec. 2001.

33. Ramasundaram, Sethuraman.: WSDL: An Insight Out, *The XML Magazine*, p. 40-44, 2001.

34. Ratnasingam, Pauline.: The importance of technology trust in Web Services security, *Information Management and Computer Security,* v. 10, n.5, p. 255, 2002.

35. Rosca, D., Greenspan, S., Wild, C.: Enterprise Modeling and Decision-Support for Automating the Business Rules Lifecycle, *Automated Software Engineering*, v.9, n. 4, p. 361, 44 p, 2002.

36. Ross – Talbot, Steve., Boley, Harold., Tabet, Said.: *Playing by the rules*, http://www.appdevadvisor.co.uk/Downloads/ADA6_5/ross6_5.pdf, April, 2003.

37. Russell, Stuart J., Peter Norvig.: *Artificial Intelligence –A Modern Approach*, Prentice Hall, New Jersey, U.S.A., 1995.

38. Seely, Scott.: *SOAP – Cross Platform Web Service Development Using XML,* Prentice Hall PTR, 2002.

39. Snell, James et. al.: *Programming Web Services with SOAP,* O'Reilly, 2002.

40. Spencer, Bruce.: The Design of jDREW: A Deductive Reasoning Engine for the Web, in *Proceedings of the First CologNet Workshop on Component-based Software Develoment and Implementation Technology for Computational Logic Systems*, Madrid, Spain, Universidad Politécnica de Madrid, Facultad de Informática TR Number CLIP 4/02.0, Sept 18-19, 2002.

41. *Sun Open Net Environment,* http://www.sun.com/software/sunone/faq.html#2*,* Mar. 2002.

42. Varhol, Peter., Grehan, Rick.: Java Application Server Roundup, *JavaPro*, 2001.

43. Vaughan-Nichols, Steven J.: Web Services: Beyond the Hype, *Computer (IEEE Computer Society)*, 2002.

44. Vlist, Eric van der.: *Using W3C Schema,* O'Reily. http://www.xml.com/lpt/a/2000/11/29/schemas/part1.html, Nov. 2001.

45. *Web page of Internet group,* http://iit-iti.nrc-cnrc.gc.ca/groups/il_e.trx, Mar. 2003.

46. *WebSphere Business Integrator*, IBM Corporation, http://www.ibm.com/software/webservers/btobintegrator/index.html, Nov. 2002.

47. Yasser, Shohoud.: *Real World XML Web Services*, Pearson Education Inc., 2003.

48. Yuan, Michael J.: *Implementing Web Services Presented by developerWorks*, http://www-106.ibm.com/developerworks/, 2001.

VITA

Keping Jia.

Jilin university of technology, MEng, Sep. 1996.

Chongqing university, BSc, Sep. 1989.

Print seperately