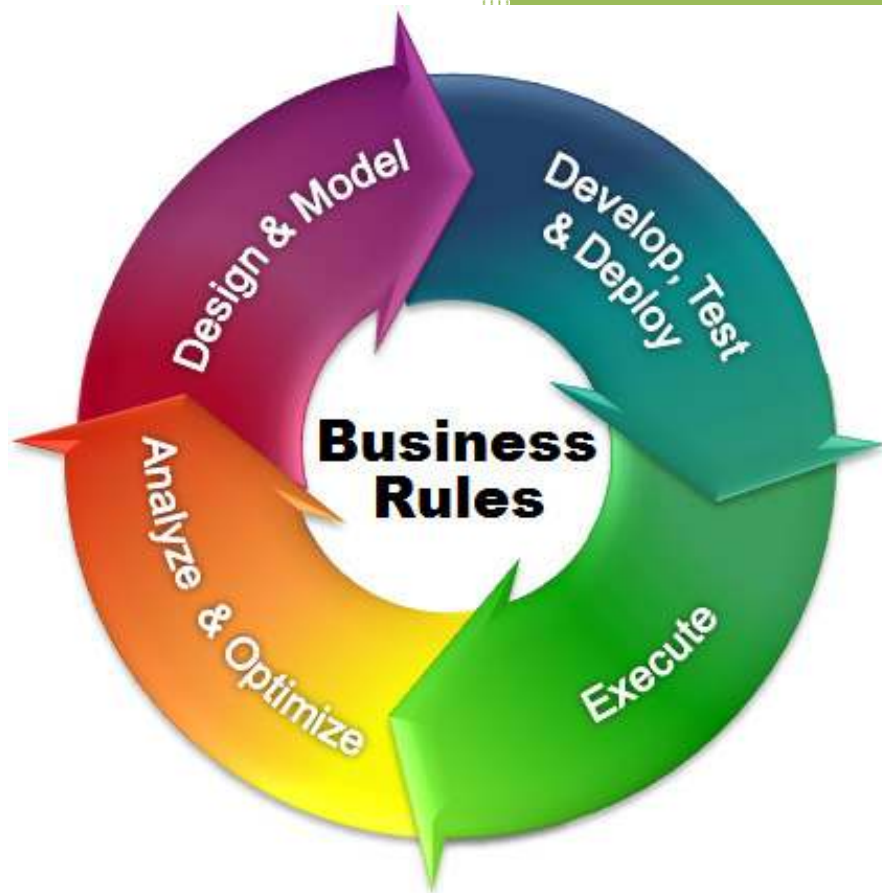


2009

CEng 491

DSK4BRMS

DETAILED DESIGN REPORT



Yetkin KARIŞ

Metin BARIŞ

Erkan AKYOL

Ghassan ALSHANA

MOCKWARE

1/19/2009

Table of Contents

1. INTRODUCTION	4
1.1. PROJECT TITLE	4
1.2. MOTIVATION	4
1.3. PROJECT DEFINITION	5
1.4. PROJECT GOALS	6
2. DESIGN CONSTRAINTS	8
2.1. TIME CONSTRAINTS	8
2.2. LANGUAGE CONSTRAINTS	8
2.3. DATA CONSTRAINTS	8
2.4. EXPERIENCE CONSTRAINTS	9
2.5. PERFORMANCE CONSTRAINTS	9
2.6. USER INTERFACE CONSTRAINTS	9
3. REQUIREMENT ANALYSIS	10
3.1. INTERFACE REQUIREMENTS	10
3.1.1. EXTERNAL INTERFACE	10
3.1.2. INTERNAL INTERFACE	10
3.2. FUNCTIONAL REQUIREMENT	11
4. ARCHITECTURAL DESIGN	14
4.1. MODULES	14
4.1.1. MANAGEMENT MODULE	14
4.1.2. EXECUTION MODULE	14
4.1.3. INTERACTION MODULE	15
4.2. DATA FLOW DIAGRAMS	16
4.2.1. LEVEL 0 DFD	16
4.2.2. LEVEL 1 DFD	17
4.2.3. LEVEL 2 DFD	18
4.3. STATE TRANSITION DIAGRAMS	27

5. DATABASE DESIGN	30
5.1. ENTITY RELATIONSHIP DIAGRAMS	30
5.2. DATA DESCRIPTIONS	33
5.3. ENTITY SETS AND DESCRIPTIONS	35
5.4. DATABASE TABLES	38
6. SYSTEM DESIGN	42
6.1. USE CASE DIAGRAMS	42
6.2. ACTIVITY DIAGRAMS	44
6.3. CLASS DIAGRAMS	47
6.4. SEQUENCE DIAGRAMS	49
7. USER INTERFACE DESIGN	52
8. MISCELLANEOUS	55
8.1. LANGUAGE PATTERNS	
8.1.1. QUERIES	
8.1.2. RULES	
8.1.3. FACTS	
8.2. ENGINE PATTERNS (FORWARD CHAINING)	
9. PROCESS	61
9.1. TEAM STRUCTURE	61
9.2. PROCESS MODEL	61
9.3. GANTT CHART	62
10. TESTING STRATEGY AND PROCEDURES	65
11. SYNTAX SPECIFICATIONS	67
12. CONCLUSION	68
13. REFERENCES	69

1. INTRODUCTION

1.1. PROJECT TITLE

We decided to name our project as BRules. 'B' is stand for business which is our main concern in this project.

1.2. MOTIVATION

Business rules touch our lives in many interesting ways. They can dictate your credit worthiness, what type of loan or insurance rate you qualify for or even why you are overlooked for the last business class upgrade at the airport.

Driving these decisions is a new generation of business rules management systems (BRMS) designed to automate decision making in enterprise IT applications. These systems differ radically from the old 'expert systems' of yesteryear that failed to catch corporate IT attention because they were too complex, expensive to run and maintain and not business-user friendly.

Organizations are now starting to realize that a more hands-on approach is needed. They are looking to a new breed of BRMS technology to empower workers to write their own business policies as the competitive climate demands.

Today rules impact a huge number of target business applications ranging from insurance adjudication, loan approval, claims processing, credit scoring, product/service recommendations, order configuration and fraud detection. A typical application implements between 100 and 1,000 rules. Complex rules are not only difficult to code into applications; they are also a nightmare to maintain using traditional coding.

There are a number of advantages gained by expressing business logic in business rules and using the processing and management facilities included in BRMS to work with them. One is cost and time savings - between 25% to 75% - over the application lifecycle simply through a reduction in modification cycle times. BRMS

are simply more flexible. Changes can be made immediately to rules by business whenever the need arises. In a traditional procedural coding implementation they would have to understand the sequencing of rules programmed in the code, which is incredibly time-consuming and expensive.

Another is consistency across all touch-points that use rules. In most cases a corporation's business rules are defined in manuals or other documents, or possibly not defined formally at all. Many organizations rely on business managers to interpret company policies as business rules, which can be inefficient and lead to inconsistent application of rules. This imperative becomes more important as organizations decide to offer the same services and access to information over multiple channels of communication: to customers, employees, or partners.

BRMS technology continues to mature, with attention now turning to the areas of rules simulation and testing, as well as analytics. Since BRMS is a decision-enabling technology it makes sense to use analytic techniques to optimize them using a scorecard-like approach. For example, routing a customer with a glaring propensity to churn to an experienced customer service agent, or determine if he or she is likely to take up an offer if presented with a rebate incentive.

Rules are also starting to pique the interest of larger IT vendors such as Microsoft, IBM, Oracle and SAP. When these companies start to invest more heavily in rules-based business logic within their applications, industry pundits expect to see a wave of sell-offs.

1.3. PROJECT DEFINITION

BRules is a domain specific kit for business rule management. It consists of three main parts namely Language, Engine and User Interface. We will use existing rule languages to create our structure.

Currently there are many tools to manage business rules such as ILOG, JDREW and Mandarax. They have the ability to execute rules which are internal i.e. are not have to interact with external databases, web services or other applications. Our aim is to make a generic language and required engine which will also be able to send a query to external sources and request response from external sources.

Users can define, classify and manage rules on the other hand they can send queries to the engine and engine will respond according to defined rules. But the main issue that we will be working on is external sources that BRules interacts with.

1.4. PROJECT GOALS

When our project is done, we will like to have created a domain specific language (DSL), a domain specific engine (DSE) and a domain specific toolkit (DST).

- User will be provided a graphical interface with a toolset.

The opening screen in our program will be flexible. According to type of the user opening screen will be changed. All type of users will be able to manage or execute rules without any coding. The toolset of our program will assist user.

- This toolset will provide a module for testing results.

The regular user will behave like a tester, so rule manager will. This user can test result set of queries.

- The programmer will be able to manage rule sets.

New rules can be created, or deleted by rule manager. The program will be flexible about managing rules.

- A powerful domain specific language will be created.

We will create a markup language that will be more powerful than most present markup languages. This language will have extra keywords and properties to connect external sources.

- The engine will be able to interact with external sources.

If the user needs to connect an external source, our program will supply this property in a wide area of sources. Our engine will be able to be in touch with web

services, databases and some application programming interfaces. This is the most powerful part of our program since most present markup languages do not support this opportunity.

- The program that we will create will be open source.

Developers can create new interactions with other sources by developing our program. Since we want our program to be generic, it will be adaptable to most development tools.

2. DESIGN CONSTRAINTS

In this part we have divided that topic into five constraints that will affect us during our project. These constraints are: Time constraints, language constraints, data constraints, experience constraints, performance constraints and user interface constraints. We will explain all of them one by one in details in order to give a clear view about those constraints.

2.1. TIME CONSTRAINTS

Timing constraints play an important role for the successful completion of the project by the end of the year. Since senior project is a two semester course, the project will have to be finished by the end of May 2009 including all of its designs. Also, according to Gantt chart we have prepared, we have a prototype demo at the end of this semester. So in order to complete our project efficiently, we should use our time carefully in order not to fall behind the schedule.

2.2. LANGUAGE CONSTRAINTS

At the beginning of the semester and the project, we were confused a little bit about the language we will use for our project. Firstly, we decided to do our project in Ruby but after that we all agreed that we will do our project in C++ because all of us are familiar with C++ more than Ruby. Thus all C++ code for this project will conform to ISO C++ standard. Also, for our Domain Specific Engine (DSE), we are going to use SQL in order to deal with databases and rules and then show it to the user as a table. Moreover, we are going to use Prolog which we will define the rules, logic and facts inside the DSE.

2.3. DATA CONSTRAINTS

Because we are going to deal with database a lot and the queries that the user will send to the engine and respond from it, the integrity constraints are very

useful are will be forward-oriented, i.e. it will be updated and modified according to specific conditions for efficiency reasons.

2.4. EXPERIENCE CONSTRAINTS

Although we have participated in many software projects and homeworks, our current project is harder than them because of new concepts. Our group is getting familiar to the framework every day, but some detailed usages must be examined correctly. Also, it can be useful to handle some unexpected problems about this new concept.

2.5. PERFORMANCE CONSTRAINTS

One of the important constraints of our project is a satisfying performance and optimizations must work efficiently. In order to accomplish a good performance (i.e. fast and efficient execution), we decided to do three main things for our modules (Management Module, Execution Module, Interaction Module and User Interface Module). Firstly, we should search and inspect all algorithms, methods and approaches carefully related to the module. Secondly, implement the module in an accurate way and finally testing the module to confirm that it is working well.

2.6. USER INTERFACE CONSTRAINTS

As it is illustrated in our project, we have two modules that can be opened according to the user's choice. These two modules are Management Module which user can insert, delete, update and create new rules and Execution Module which the user can request and be responded a rule set as a table by the engine. So user interface should be easy and understood by the user to choose one of them and not to confuse between them. Also it has many properties that should be taken into consideration. These properties are:

- Flexible specification of layout and other aspects of presentation for graphical user interfaces.
- Names of *menus* and other *GUI* elements will be easy to understand and straightforward.

- Establishment of automatically updated connections between application data and user interface components.
- Convenient maintenance of multiple presentations or views of the same data.

3. REQUIREMENT ANALYSIS

3.1. INTERFACE REQUIREMENTS

The business rule system in our program will interact with user, get information from user and according to user's requests the business rules will be executed or managed including creating basic rules and composite rules and updating rules. So our program will include external and internal interfaces. The external interface will be available since we want user not to deal with exhaustive codes, so the user will have a graphical user interface for managing business rules easily.

3.1.1. EXTERNAL INTERFACE (UI)

The first part of our program is external user interface which is a graphical user interface directly in touch with user. Our external interface will have log in part according to authentication. This interface will have 2 different screens according to authorization of user. In this part there will be 2 types of users respectively regular user and rule manager. When logged in as a regular user, the user will be redirected to execution module. In this module the user will be able to send queries and get response. When logged in as a rule manager, the user will be redirected to management module. This user will be generally the business analyst. Rule manager will be capable of creating new basic rules, creating new composite rules from existing ones, deleting existing rules, inserting and updating rules.

3.1.2. INTERNAL INTERFACE

In internal interface there will be three modules respectively management module, execution module and interaction module. According to user's requests the

business rules will be managed in management module which only the business analyst will have permission to manage rules, or the rules will be executed in execution module, or interact with fact set when necessary.

The business system in internal interface will have architecture. This system has a rule editor that provides a user interface, an inference engine that applies the rules using its algorithms, a rule repository that saves information related to the rule, a rule object model handler which controls over the rule object model, the rule object model that express the rule class data, rule adapter that supply an interface between rule editor and rule object model handler, rule repository handler that control rule repository, and so on.

Lastly, we have an API for users that only can use the engine. API will be a web service for demonstration. Clients can make their own API for using our engine. For example in a hospital API can be implemented with C++ and its interface may vary according to user.

To be convenient some frequently used rules, facts and queries will be embedded to this API. The binding between the API and our engine will be similar to UI and DSE connection. The query sent by API will be transformed into our language and executed like an SQL query in our engine.

The transformation of information of API and external sources is a key concept that we are still working on.

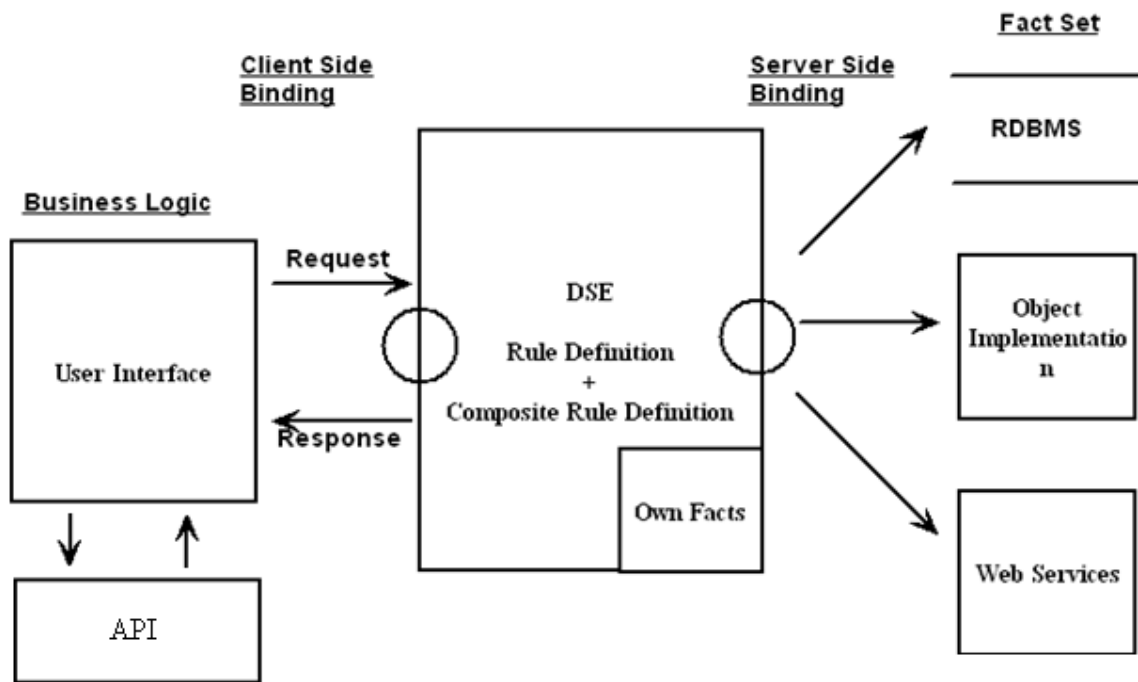


Figure 1. General Architecture of BRules

3.2. FUNCTIONAL REQUIREMENTS

Our program will have three main parts namely user interface, domain specific engine and fact set.

User Interface:

This is the first part of our program which interacts with user. There will be a graphical user interface which will have two types of users. According to authorization of user, this interface will be in touch with domain specific engine.

Domain Specific Engine:

All rule related operations will be done in this part. When user wants to do operations using graphical interface, this request will be granted in this part and make operations such as executing or managing rules according to information coming from user. This part will also have rulebase and its own facts. While making rule

related operations, engine will interact with its own fact set or external fact set such as web services, some application programming interfaces if necessary.

Fact Set:

This part is the last part of our program. This fact set will include web services, some application programming interfaces, relational database management system (RDBMS), and domain specific engine's own facts. Our engine will interact with this part if necessary according to granted information from user.

Our interface will get information from user; make C++ binding to make operations in our domain specific engine for business rules management. If the user is a regular user:

- There will be a log in screen.
- When user log in as a regular user, there will be an execution screen for the user.
- Regular user will send queries using a graphical interface.
- The query will be got with C++ binding and verified in domain specific engine.
- By using query, our program will get initial rule.
- By forward chaining, all related rules will be evaluated and according to corresponding facts.
- If our domain specific engine's own facts are sufficient for evaluating, our program will use C++ binding to use facts inside domain specific engine.
- If our own fact set is not sufficient to evaluate rules and extra fact set is needed, our program will interact with some external fact set among web services, application programming interfaces or relational database management systems where the corresponding facts for the rule in.

If the user is a rule manager:

- Also called as business analyst.

- This user will already have authorization as much as regular user have and will be able to send queries to program.
- After logging in as a rule manager, the user will see a management screen.
- Rule manager will use a toolset in management screen.
- This toolset will enable rule manager to create basic rules, create composite rules from existing rules, delete existing rules, and modify rules.
- Domain specific engine will gather information with C++ binding from the user.
- Then the request will be verified by syntactic and semantic checking such that the user can create a rule that does not exist or can delete a rule that exists, and so on.
- To make rule related operations, domain specific engine (DSE) will interact with only rule repository.
- If creation or update needed, domain specific engine will do corresponding operations and update rule repository.
- If deletion needed, the corresponding rule will be found in rule repository and will be deleted by our domain specific engine. So on rule repository will be updated.
- The user who connects our engine with API can only send queries via this API to our engine.

4. ARCHITECTURAL DESIGN

4.1. MODULES

BRules engine is composed of Manager, Executor and Connector Module.

4.1.1. MANAGER MODULE

Management module has 3 classes. These classes are:

- Parser
- Analyzer
- Organizer

Parser gets the rule or fact input in xml format and sends it Analyzer module right after checking validity and changing format of input.

Analyzer gets FormattedRule from Parser and by combining it with data from rulebase it gives a RuleSet (family tree of rule).

Organizer should take the RuleSet and insert it into right place and in right form to Rule Repository and if needed it makes changes in other rules.

4.1.2. EXECUTOR MODULE

This module has 4 classes. These classes are:

- Query_Listener
- Recognizer
- Trigger
- Engine

Query_Listener is responsible for getting queries from the user. It checks the query and sends it to Recognizer.

Recognizer converts the Verified_Query into Logic which has a tree-like data structure.

Trigger is for getting corresponding rules from rulebase, ask for related data from Connector module, and fire the Engine.

Engine is the heart of the structure. When it is fired it finds the result of query. We will use Rete algorithm or forward chaining to achieve this process.

4.1.3. CONNECTOR MODULE

This module is responsible for connection and data transfer from external sources and has 2 classes which should be extended.

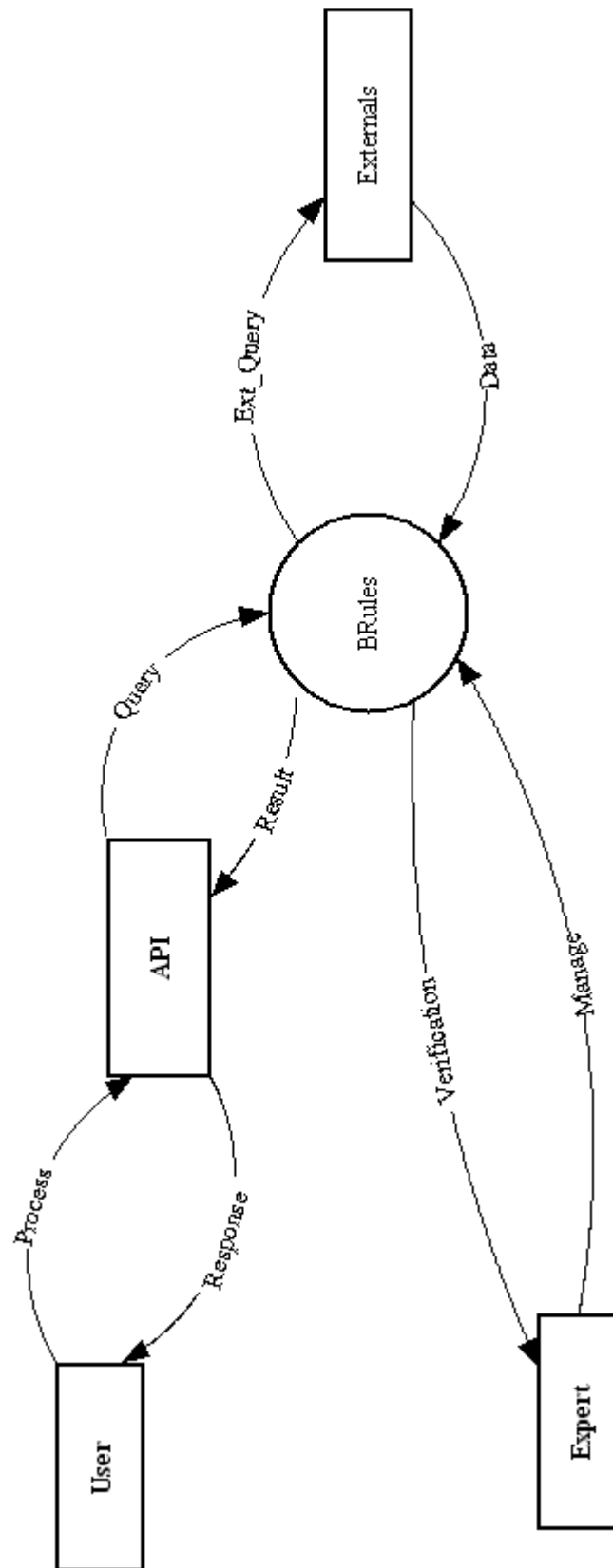
- Solver
- Adapter

Solver is to define the needed data from external sources and determine the server or source to be connected.

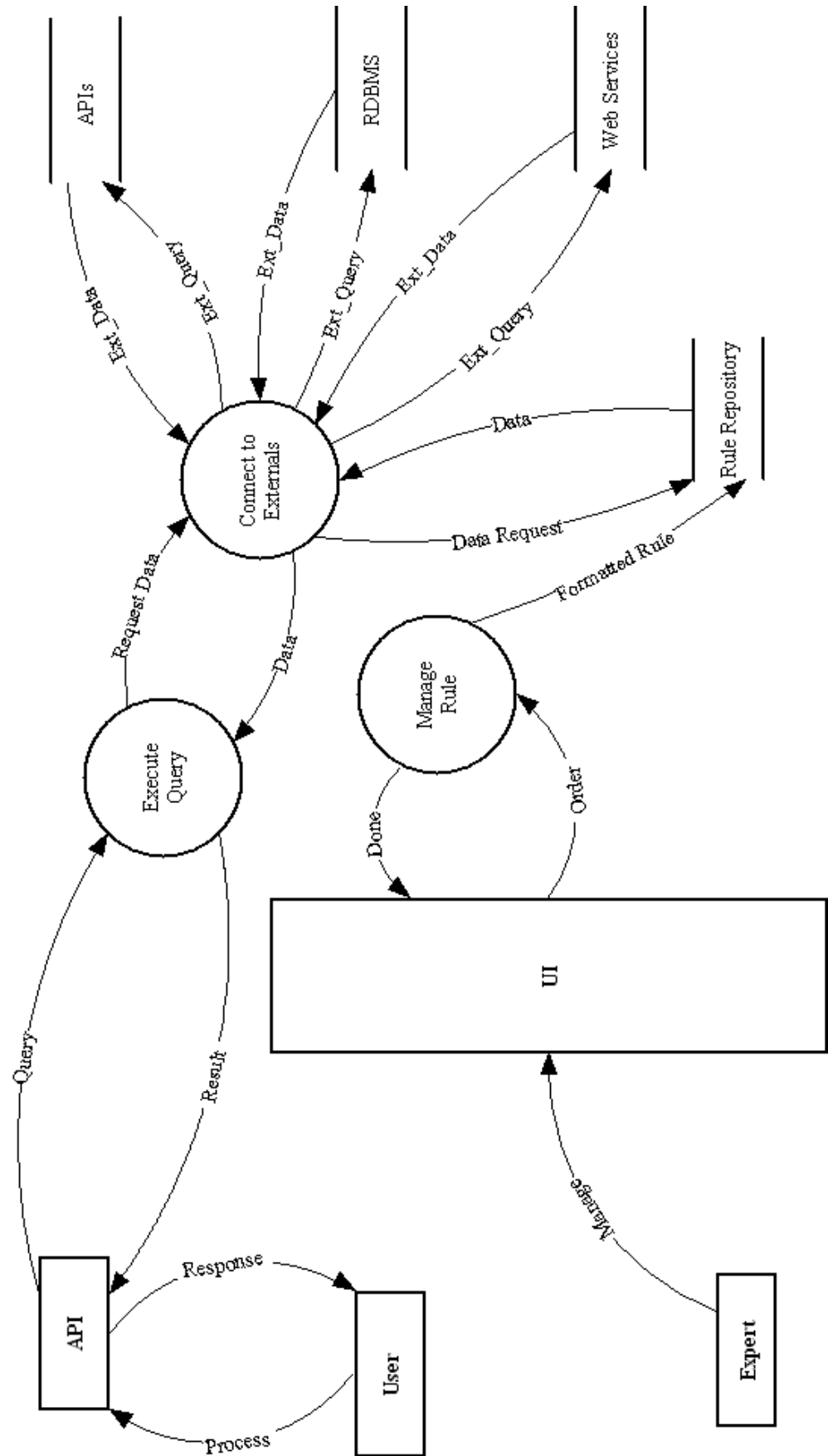
Adapter makes a connection between our Engine and external source and gets required data.

4.2. DATA FLOW DIAGRAMS

4.2.1. LEVEL 0 DFD

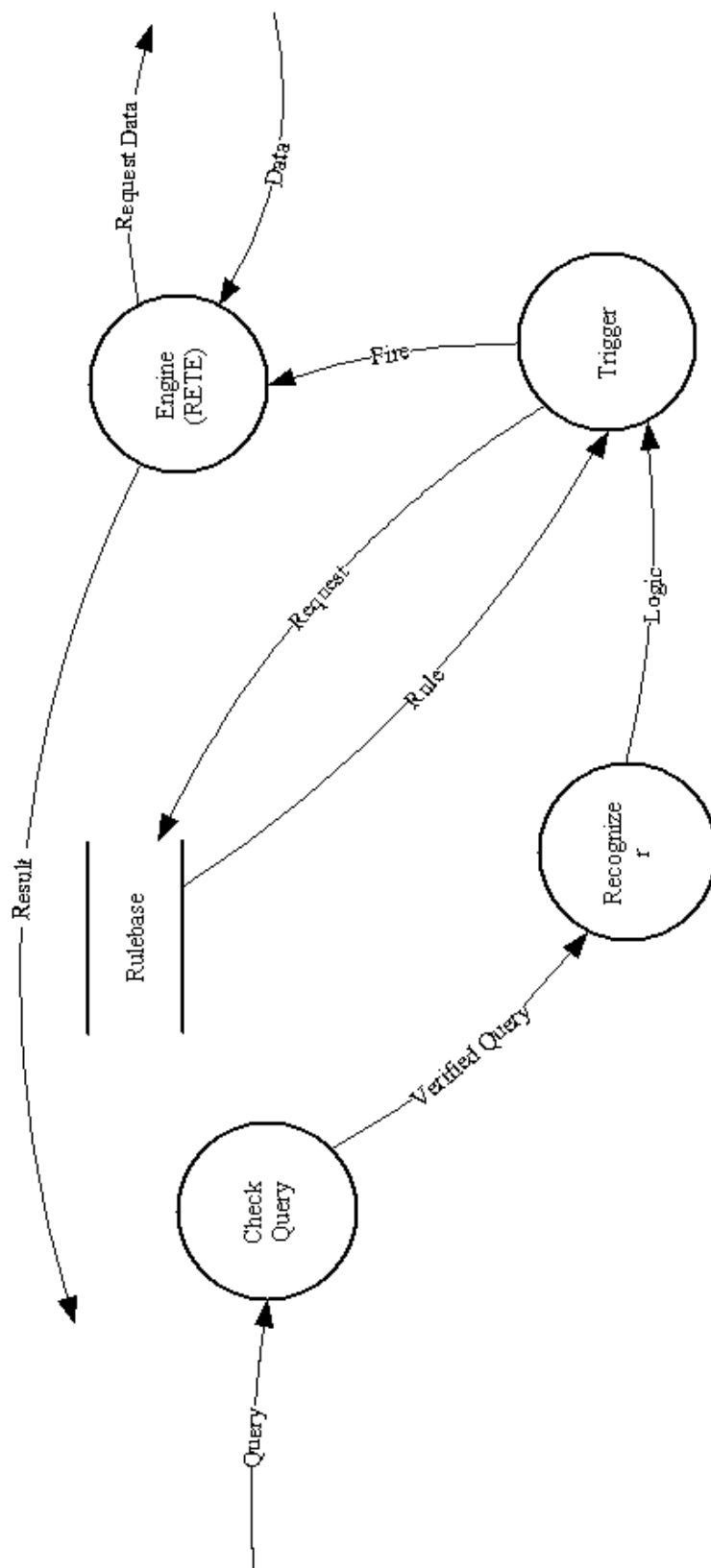


4.2.2. LEVEL 1 DFD



4.2.3. LEVEL 2 DFD

Executor Module



Name	Query
From	User
To	Query_Listener
Description	Data that is sent as input to our program. It is sent by User

Name	Verified_Query
From	Query_Listener
To	Recognizer
Description	This data is qualified to be query for input of our program.

Name	Logic
From	Recognizer
To	Trigger
Description	This data is an instance of a class that will be our logical item like conditions and rule products (under maintenance).

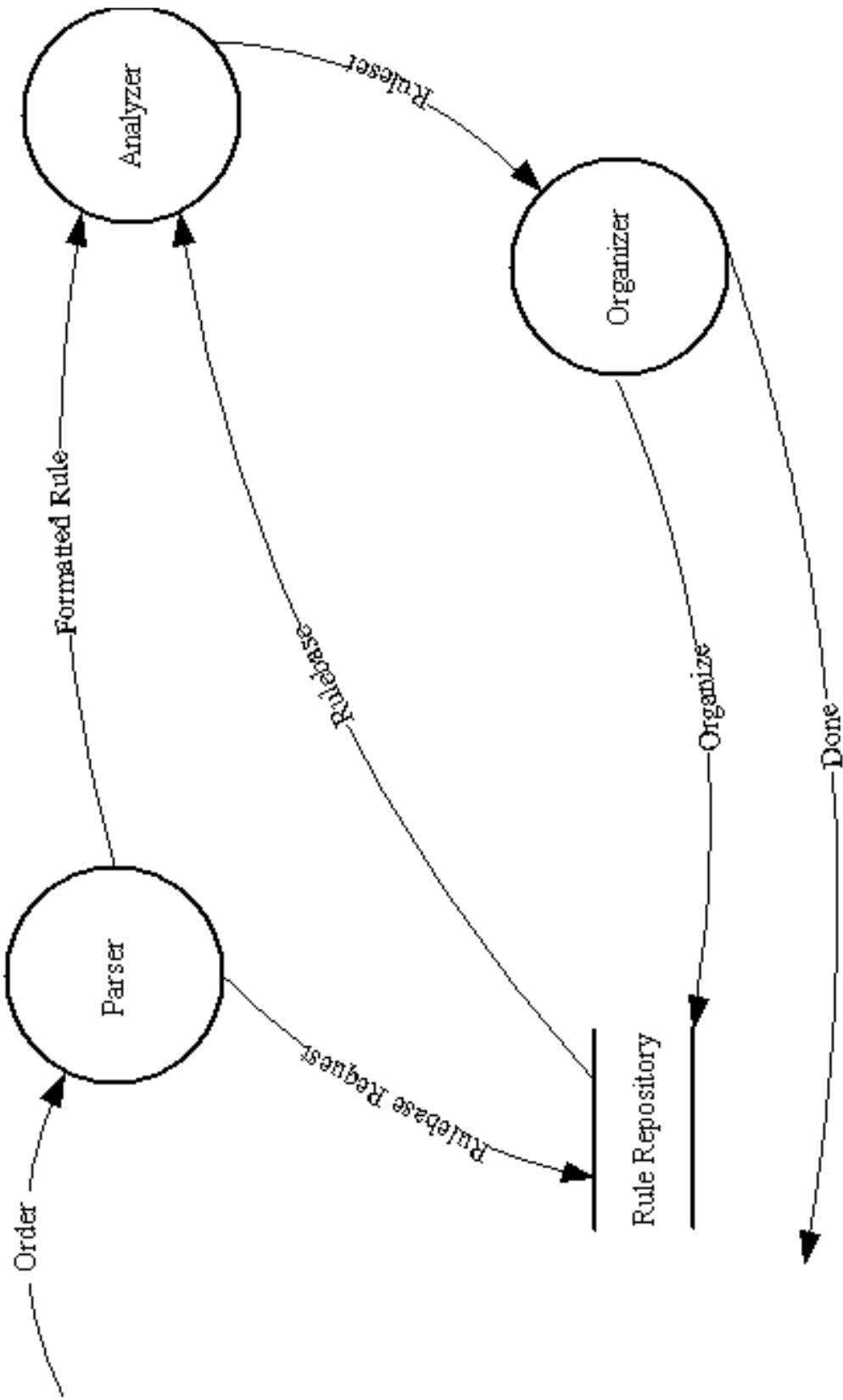
Name	Request
From	Trigger
To	Rulebase(Repository)
Description	This data asks a rule from rulebase

Name	Rule
From	Rulebase
To	Engine
Description	This data is our rule object that asked by trigger for engine to iterate

Name	Data
From	Connector
To	Engine
Description	Data is a needed data from connector to find from to complete engine stage

Name	Data_Request
From	Trigger
To	Connector
Description	It is a query-like data that will be recognized by connector

Manager Module



Name	Rule
From	Expert
To	Parser
Description	String-like data with keywords and it is nested

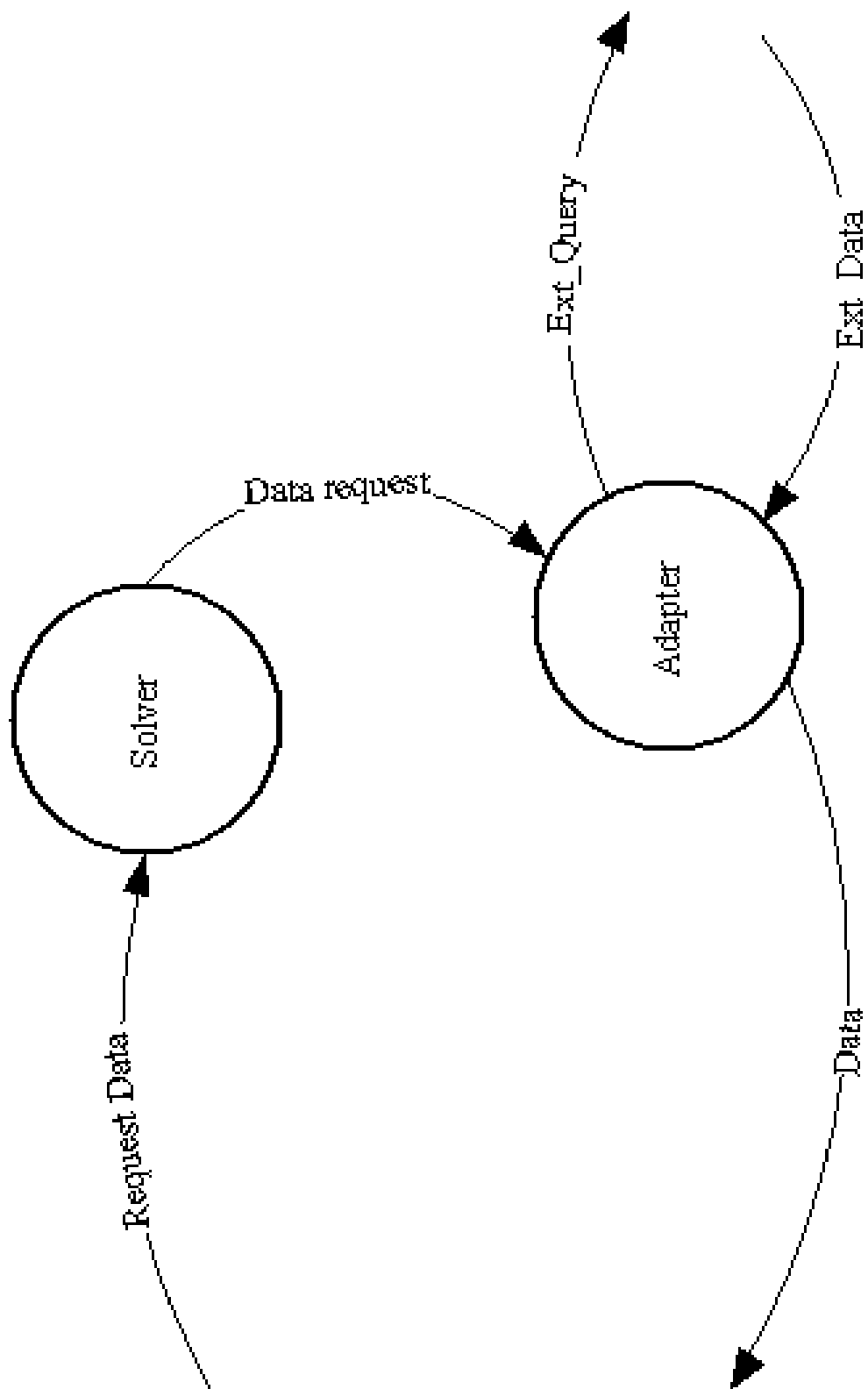
Name	Rulebase_Request
From	Parser
To	Rulebase(repository)
Description	Data that asks for rulebase and it is string type

Name	Formatted_Rule
From	Parser
To	Analyzer
Description	Rule class data created by parser from input rulestring, it is verified.

Name	Rulebase
From	Rulebase
To	Analyzer
Description	Root of a family rule tree can be used for reaching rule nodes.

Name	RuleSet
From	Analyzer
To	Organizer
Description	A family rule tree for inserting and organizing rulebase

Name	Organize
From	Organizer
To	Rulebase
Description	Organized ruleset to be inserted to rulebase and extra data to change organization(will be separated in next phase)



Name	Condition
From	Executor
To	Solver
Description	This data is to create server and Data information

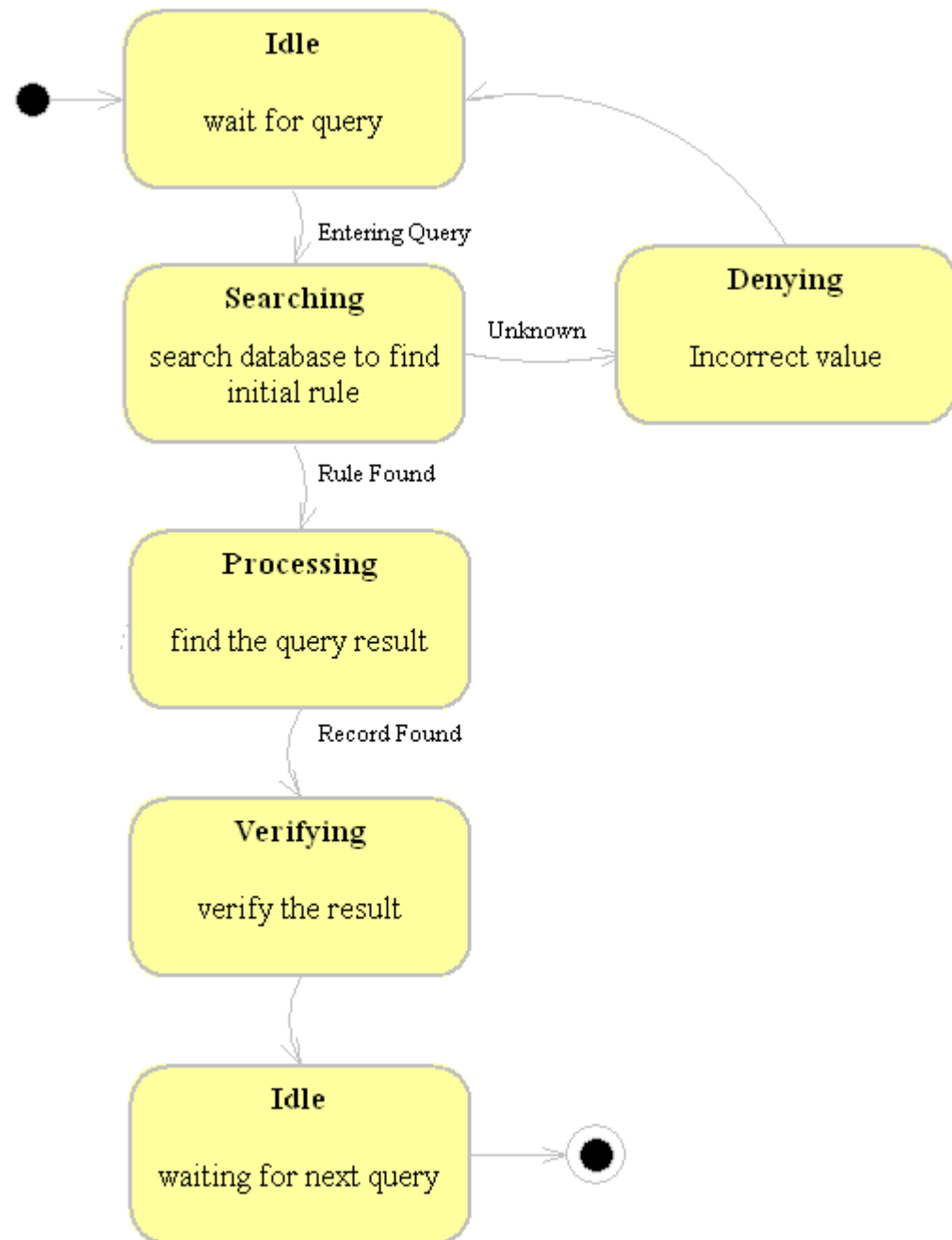
Name	Data_Request
From	Solver
To	Adapter
Description	Data that carries server and data information

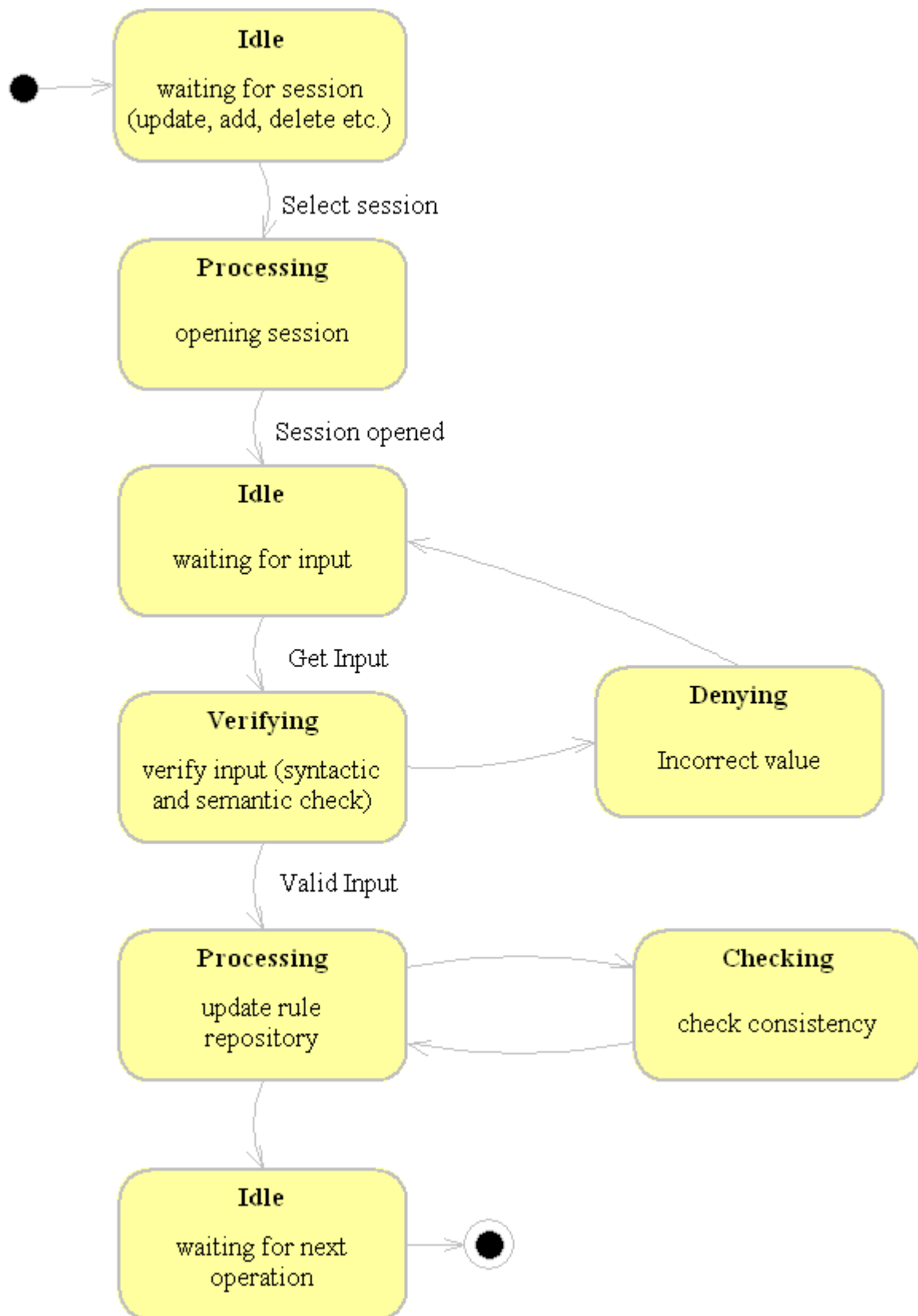
Name	Adapted_Request
From	Adapter
To	External
Description	The data request that can be recognized by externals

Name	Raw_Data
From	External
To	Adapter
Description	Incoming raw data from external

Name	Data
From	Adapter
To	Executor
Description	Needed data in known format to complete execution

4.3. STATE TRANSITION DIAGRAMS





5. DATABASE DESIGN

5.1. ENTITY RELATIONSHIP DIAGRAMS

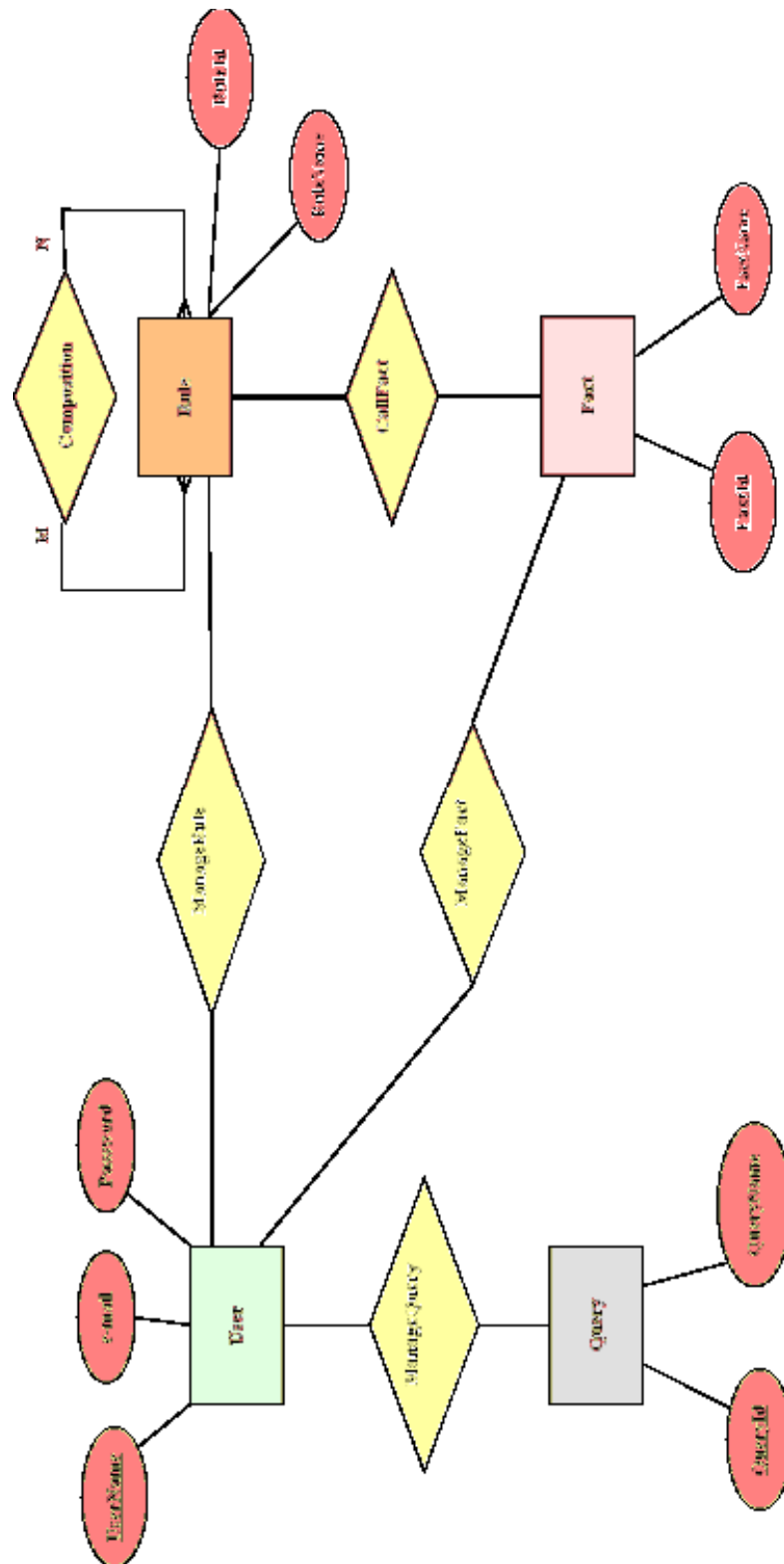


FIGURE 5.1.1

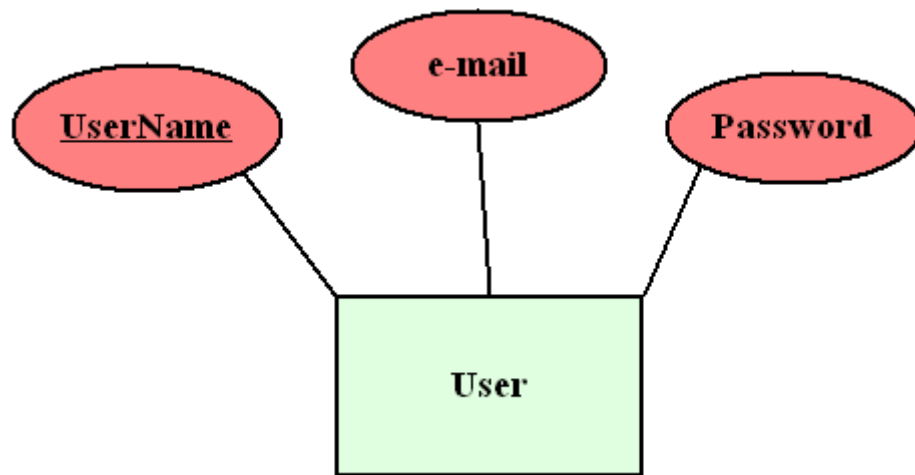


FIGURE 5.1.2

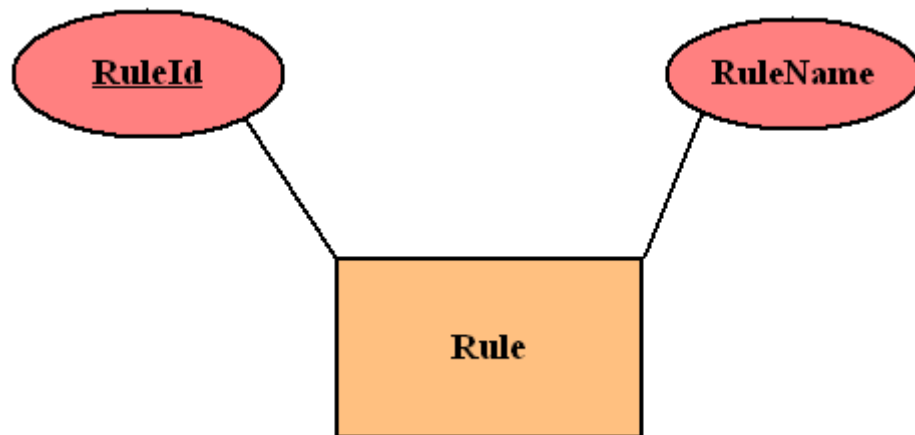


FIGURE 5.1.3

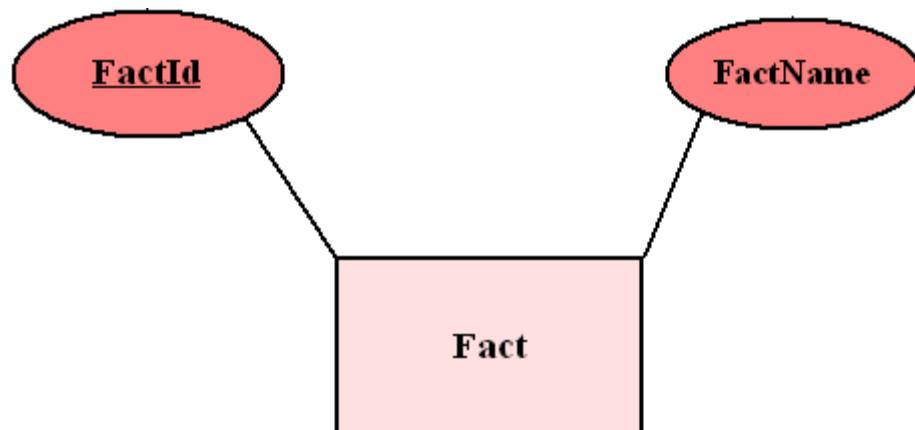


FIGURE 5.1.4

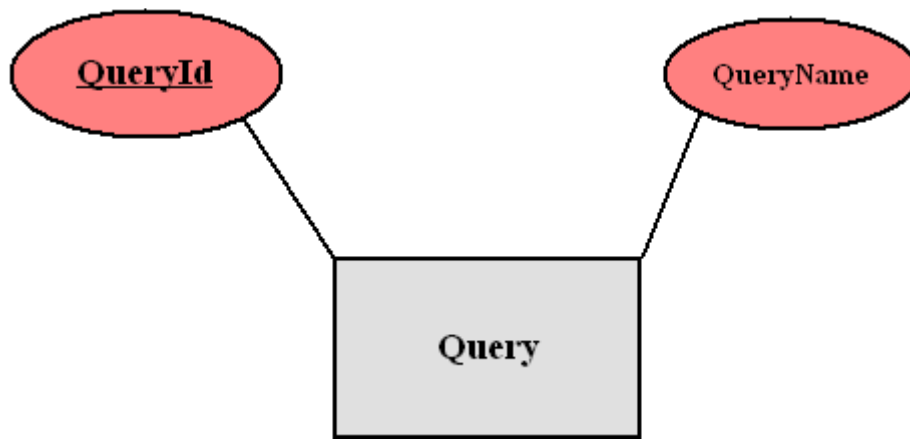


FIGURE 5.1.5



FIGURE 5.1.6



FIGURE 5.1.7

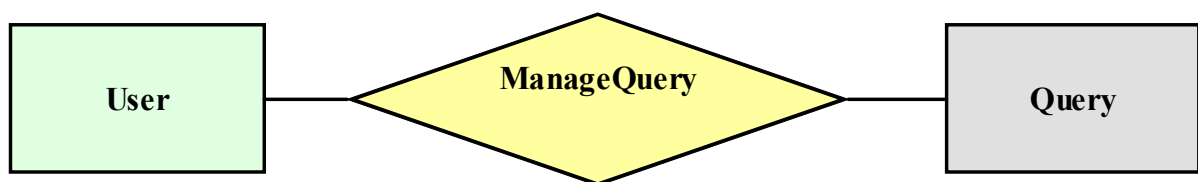


FIGURE 5.1.8



FIGURE 5.1.9

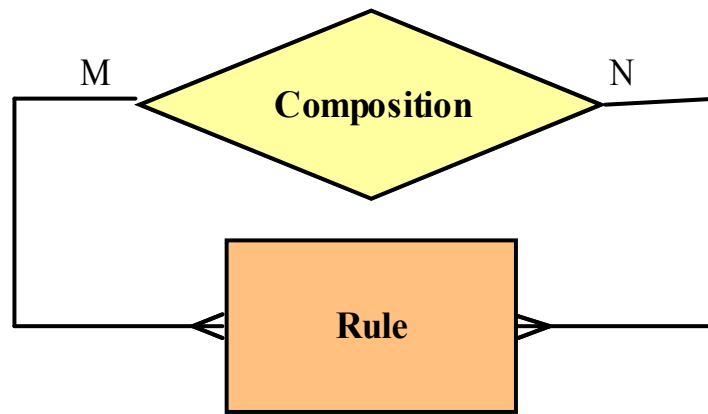


FIGURE 5.1.10

5.2. DATA DESCRIPTIONS

Attributes with "*" are can not take null value.

○ User

Data	Type&Size	Format
<u>UserName</u> *	VARCHAR-20	Text
Password*	VARCHAR-20	Text is hidden
e-mail*	VARCHAR-40	Text

○ Rule

Data	Type&Size	Format
<u>RuleId</u> *	INTEGER	Number
RuleName*	CLOB	Text

○ Fact

Data	Type&Size	Format
<u>FactId</u> *	INTEGER	Number
FactName*	CLOB	Text

○ Query

Data	Type&Size	Format
<u>QueryId</u> *	INTEGER	Number
QueryName*	CLOB	Text

- ManageRule

Data	Type&Size	Format
<u>UserName</u> *	VARCHAR-20	Text
<u>RuleId</u> *	INTEGER	Number

- ManageFact

Data	Type&Size	Format
<u>UserName</u> *	VARCHAR-20	Text
<u>FactId</u> *	INTEGER	Number

- ManageQuery

Data	Type&Size	Format
<u>UserName</u> *	VARCHAR-20	Text
<u>QueryId</u> *	INTEGER	Number

- CallFact

Data	Type&Size	Format
<u>RuleId</u> *	INTEGER	Number
<u>FactId</u> *	INTEGER	Number

- CompositeRule

Data	Type&Size	Format
<u>RuleId</u> *	INTEGER	Number
Rule1*	INTEGER	Number
Rule2*	INTEGER	Number
Operator*	VARCHAR-20	Text

5.3. ENTITY SETS & DESCRIPTIONS

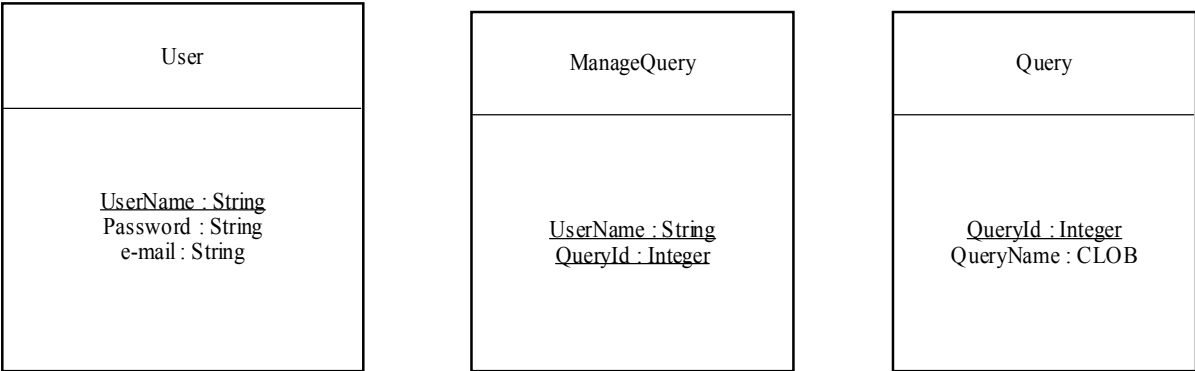


FIGURE 5.3.1

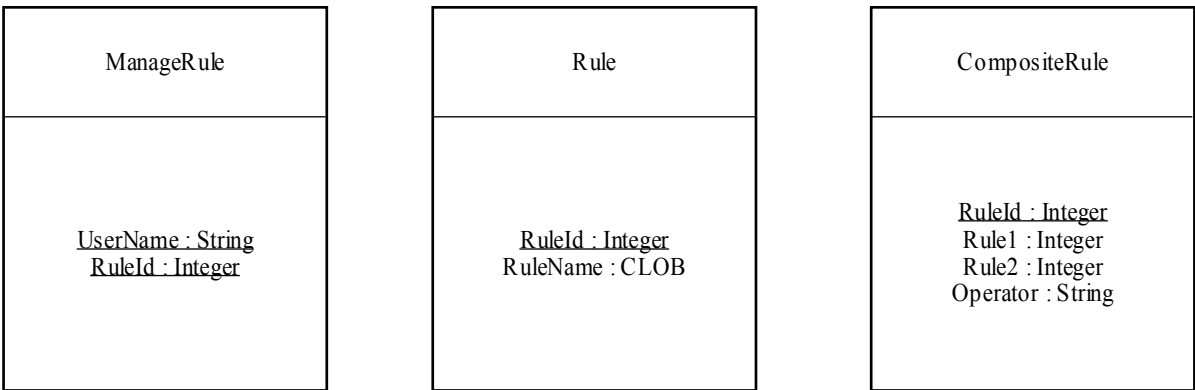


FIGURE 5.3.2

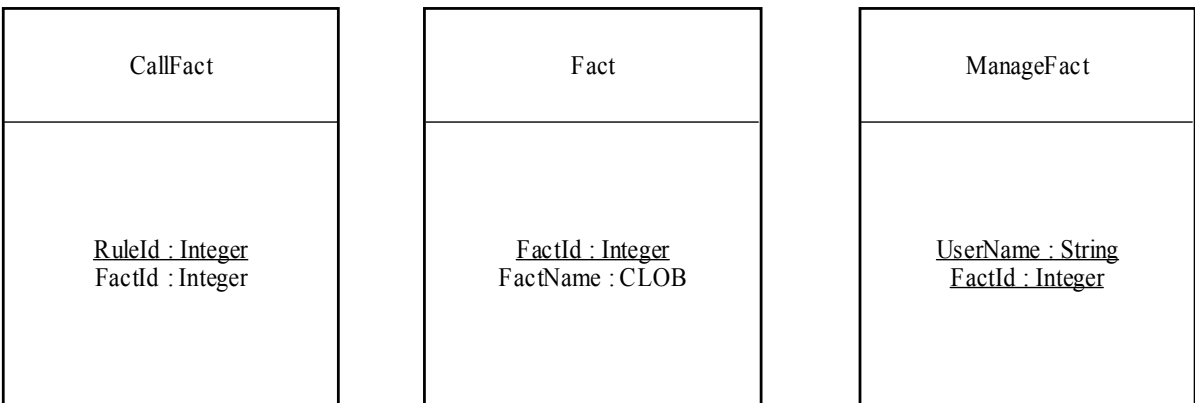


FIGURE 5.3.3

- ENTITY DESCRIPTIONS
 - User

In the database of BRules; the account information and e-mail of users are kept in the 'User' entity.

UserName: Each user of the system has a unique user name in BRules; thus the attribute 'UserName' which holds the user names is the primary key of the entity 'User'.

Password: This string field is the matched password for the user name of the user.

Email: The electronic mails of the users are kept in this field.

- Rule

The user has capability to manage rules. While the user manages the rules, they are stored in BRules database. Rule entity in the database has the rule information and a key for the rule.

RuleId: In Rule entity every rule has a unique id.

RuleName: This attribute involves the rule having a name as RuleName.

- Fact

Facts are used for executing rules by the user. The user also can add or delete facts. Fact entity has the fact itself and a pointer to it.

FactId: Every fact has a unique id in fact entity.

FactName: The fact is kept in.

- Query

In BRules database, the user can save and load the queries. This is useful for a user who uses many common queries. Query entity has the key value of query and the query itself.

QueryId: Each query has a unique id in database.

QueryName: Query is stored here.

- RELATION DESCRIPTIONS
 - ManageRule

Brules has a feature which allows the users to manage rules such as creating new rules, deleting existing rules and updating rules. ManageRule relation has the name of the user and the id of the rule.

UserName: The name of the user who manages rules.

RuleId: The id number of rule which is used to be managed by the user.

- CompositeRule

This is a recursive relation for Rule entity. The user is able to create composite rules by using existing rules. Rules can recursively compose new rules. This relation has the id's of composite rule, the rules that compose the existing rule and logical operator of these rules.

RuleId: The id of the composite rule.

Rule1: The id of the first of composer rules.

Rule2: The id of the second of composer rules.

Operator: This is the logical operator for the operation between two rules.

- ManageFact

The user can also manage facts in BRules database. New facts can be created or existing facts can be deleted by the user. ManageFact has the name of the user and id of the fact.

UserName: The username information of the user.

FactId: The id of the fact to be managed.

- ManageQuery

When a user wants to use a query many times, he/she can save this query and use it at any other time. This feature is useful for the user to save time when wants to send queries which he/she uses commonly. This relation has username of the user and id of the query.

Username: This attribute involves the username.

QueryId: The id of the query which is already saved.

- CallFact

When rules are executed, the user need to get information from fact set. According to result syntactic and semantic checking, rules are executed. CallFact has the attributes RuleId and FactId.

RuleId: The id of the rule been executed.

FactId: This attribute involves the id of the fact.

5.4. DATABASE TABLES

createDatabase.sql

```
\i User.sql
\i Rule.sql
\i Fact.sql
\i Query.sql
\i ManageRule.sql
\i ManageFact.sql
\i ManageQuery.sql
\i CallFact.sql
\i CompositeRule.sql
```

deleteDatabase.sql

```
DROP TABLE User
DROP TABLE Rule
DROP TABLE Fact
DROP TABLE Query
DROP TABLE ManageRule
DROP TABLE ManageFact
DROP TABLE ManageQuery
DROP TABLE CallFact
DROP TABLE CompositeRule
```

 User.sql

```
CREATE TABLE User (
  UserName VARCHAR (20) NOT NULL,
  Password VARCHAR (20) NOT NULL,
  e-mail VARCHAR (40) NOT NULL,
  UNIQUE (Email),
  PRIMARY KEY (UserName)
```


);

 Rules.sql

```
CREATE TABLE Rule(  
RuleId INTEGER NOT NULL,  
RuleName CLOB NOT NULL,  
PRIMARY KEY(RuleId)  
);
```

 Fact.sql

```
CREATE TABLE Fact(  
FactId INTEGER NOT NULL,  
FactName CLOB NOT NULL,  
PRIMARY KEY(FactId)  
);
```

 Query.sql

```
CREATE TABLE Query(  
QueryId INTEGER NOT NULL,  
QueryName CLOB NOT NULL,  
PRIMARY KEY(QueryId)  
);
```

 ManageRule.sql

```
CREATE TABLE ManageRule(  
UserName VARCHAR(20) NOT NULL,  
RuleId INTEGER NOT NULL,  
PRIMARY KEY(UserName,RuleId)  
FOREIGN KEY (UserName) REFERENCES Users (UserName),  
FOREIGN KEY (RuleId) REFERENCES Rule (RuleId)  
);
```


ManageFact.sql

```
CREATE TABLE ManageFact(  
  UserName VARCHAR(20) NOT NULL,  
  FactId INTEGER NOT NULL,  
  PRIMARY KEY(UserName,FactId)  
  FOREIGN KEY (UserName) REFERENCES Users (UserName),  
  FOREIGN KEY (FactId) REFERENCES Fact (FactId)  
);
```

ManageQuery.sql

```
CREATE TABLE ManageQuery(  
  UserName VARCHAR(20) NOT NULL,  
  QueryId INTEGER NOT NULL,  
  PRIMARY KEY(UserName, QueryId)  
  FOREIGN KEY (UserName) REFERENCES Users (UserName),  
  FOREIGN KEY (QueryId) REFERENCES Query (QueryId)  
);
```

CallFact.sql

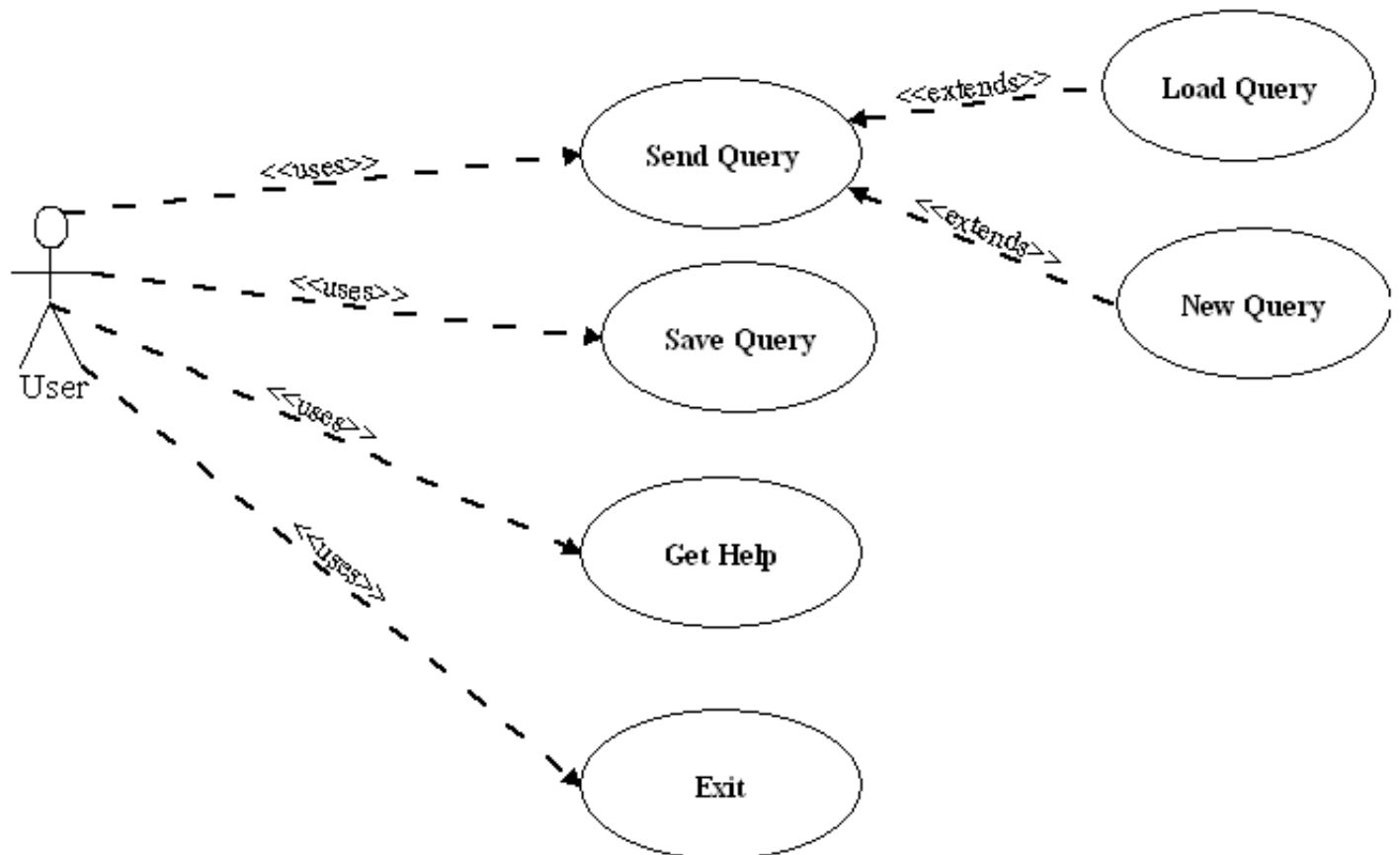
```
CREATE TABLE CallFact(  
  RuleId INTEGER NOT NULL,  
  FactId INTEGER NOT NULL,  
  PRIMARY KEY(RuleId),  
  FOREIGN KEY (RuleId) REFERENCES Rule(RuleId),  
  FOREIGN KEY (FactId) REFERENCES Fact(FactId)  
);
```

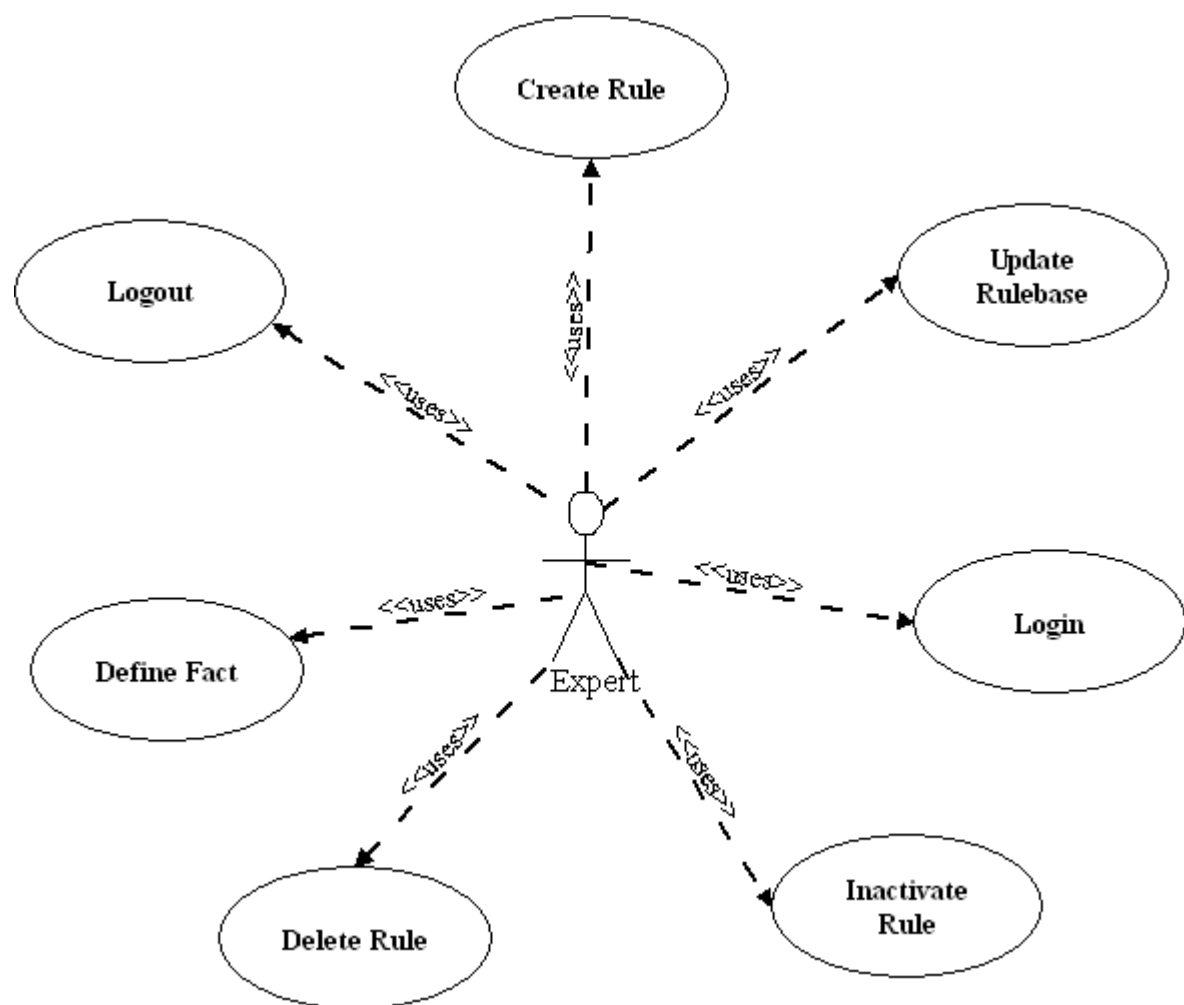
CompositeRule.sql

```
CREATE TABLE CompositeRule(  
  RuleId INTEGER NOT NULL,  
  Rule1 INTEGER NOT NULL,  
  Rule2 INTEGER NOT NULL,  
  Operator VARCHAR(20) NOT NULL,  
  PRIMARY KEY(RuleId),  
  FOREIGN KEY (RuleId) REFERENCES Rule(RuleId),  
  FOREIGN KEY (Rule1) REFERENCES Rule(Rule1),  
  FOREIGN KEY (Rule2) REFERENCES Rule(Rule2)  
);
```

6. SYSTEM DESIGN

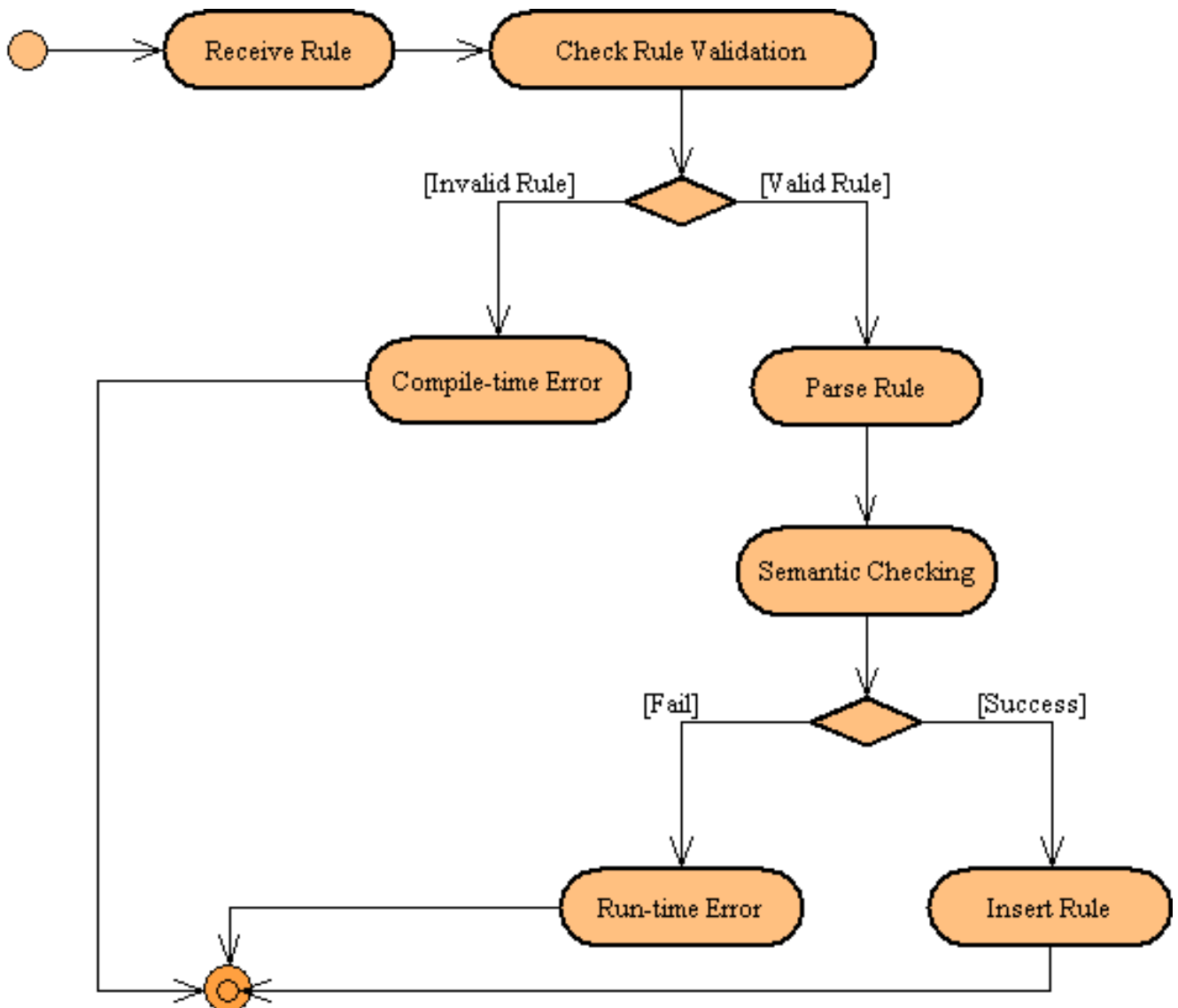
6.1. USE CASE DIAGRAMS





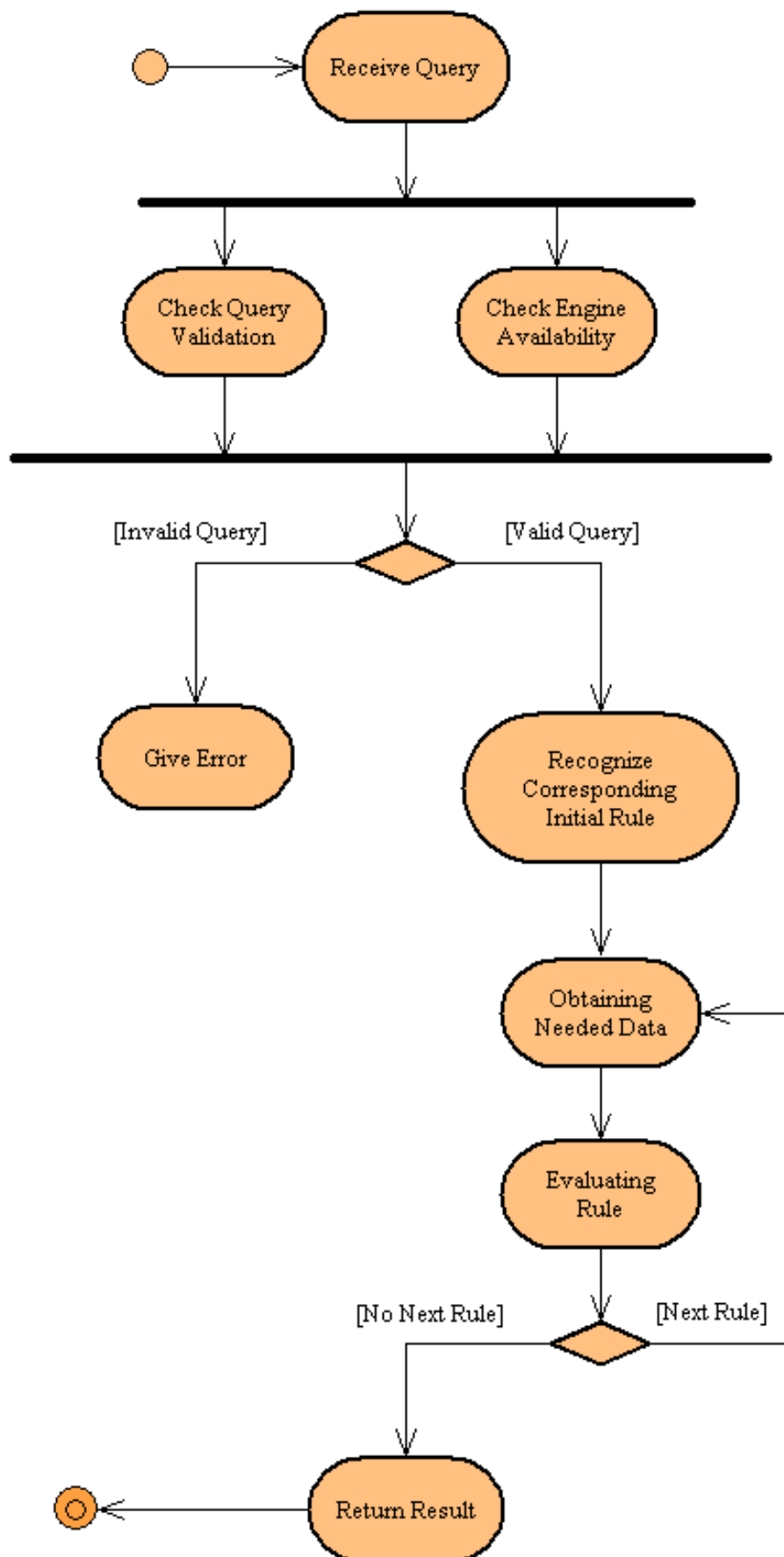
6.2. ACTIVITY DIAGRAMS

Expert Session



- Expert enters a rule or fact to the program. When rule is received it is checked for validation and if it is valid to be a rule program continue running, terminating (compile time error) otherwise.
- Program will parse the valid rule and test it in next step.
- If test is failed then program gives run time error if not continue.
- Then rule will be inserted and rulebase is organized accordingly.

User Session

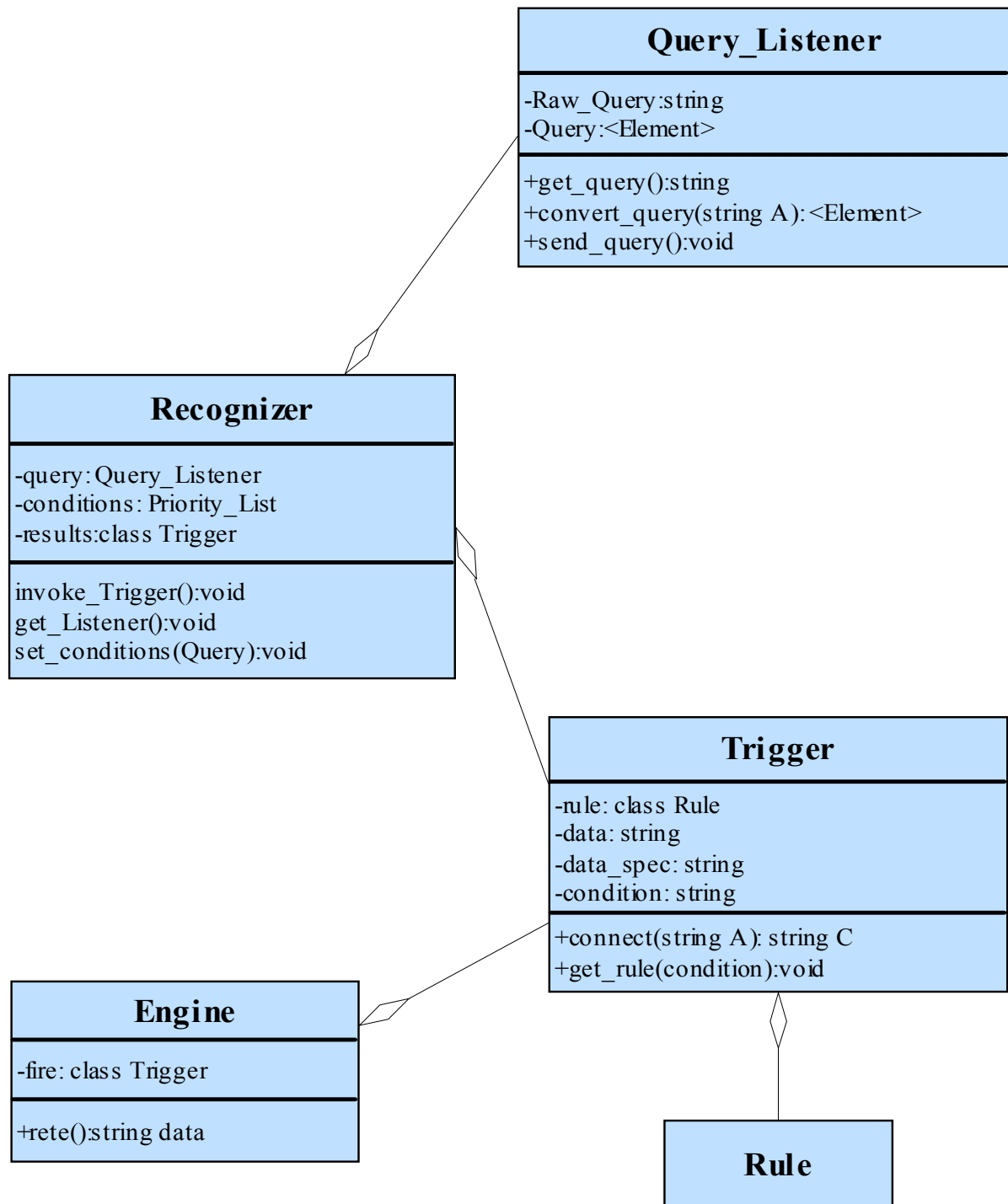


Users (or testers) may write a query (with our specifications) or load a saved query in our interface and get the result of the query visually as a table like SQL tables. Below is the description of the path that our program will follow.

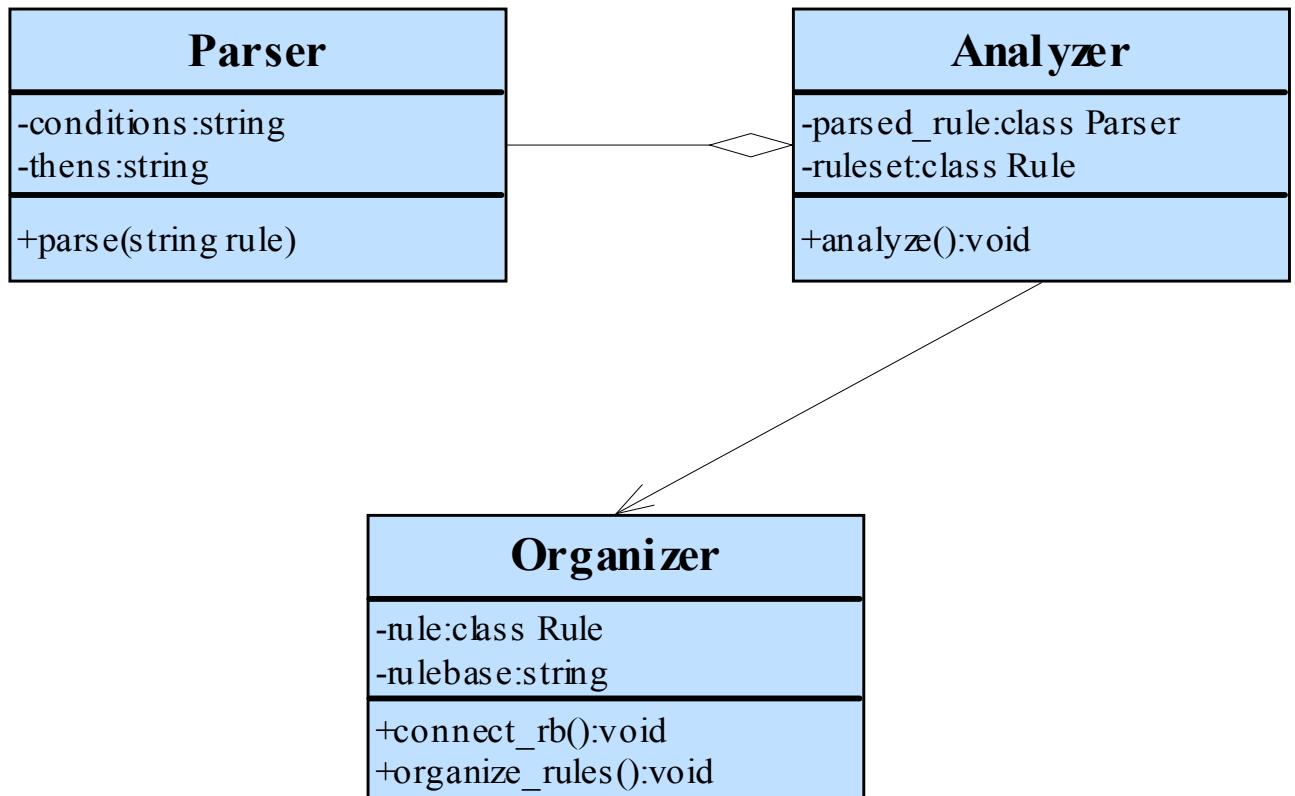
- When query entered the program will check the validity of the query and the engine. If it is a valid query and engine is available then program will continue to execute query otherwise it gives an error and terminate.
- Valid query will be recognized by the program and a rule or a ruleset will be obtained from rulebase to initiate the engine.
- After engine was initiated it starts to iterate until there are no rule will be executed.
- The output will be the result of the query.

6.3. CLASS DIAGRAMS

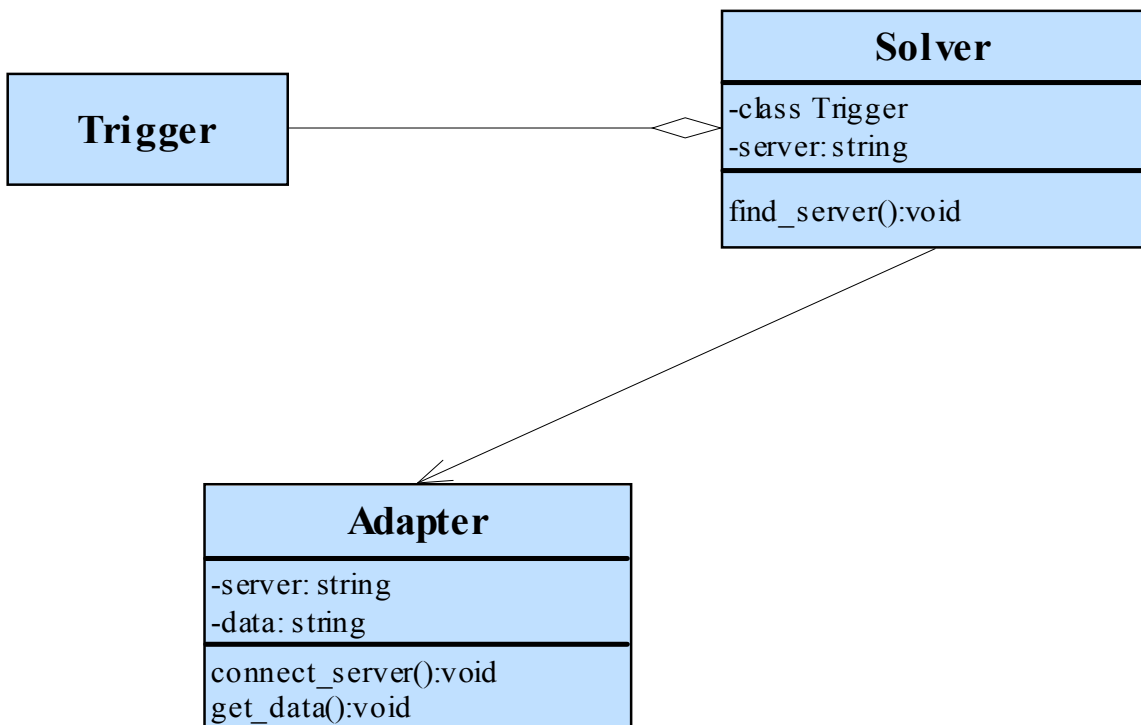
Execution Module



Management Module

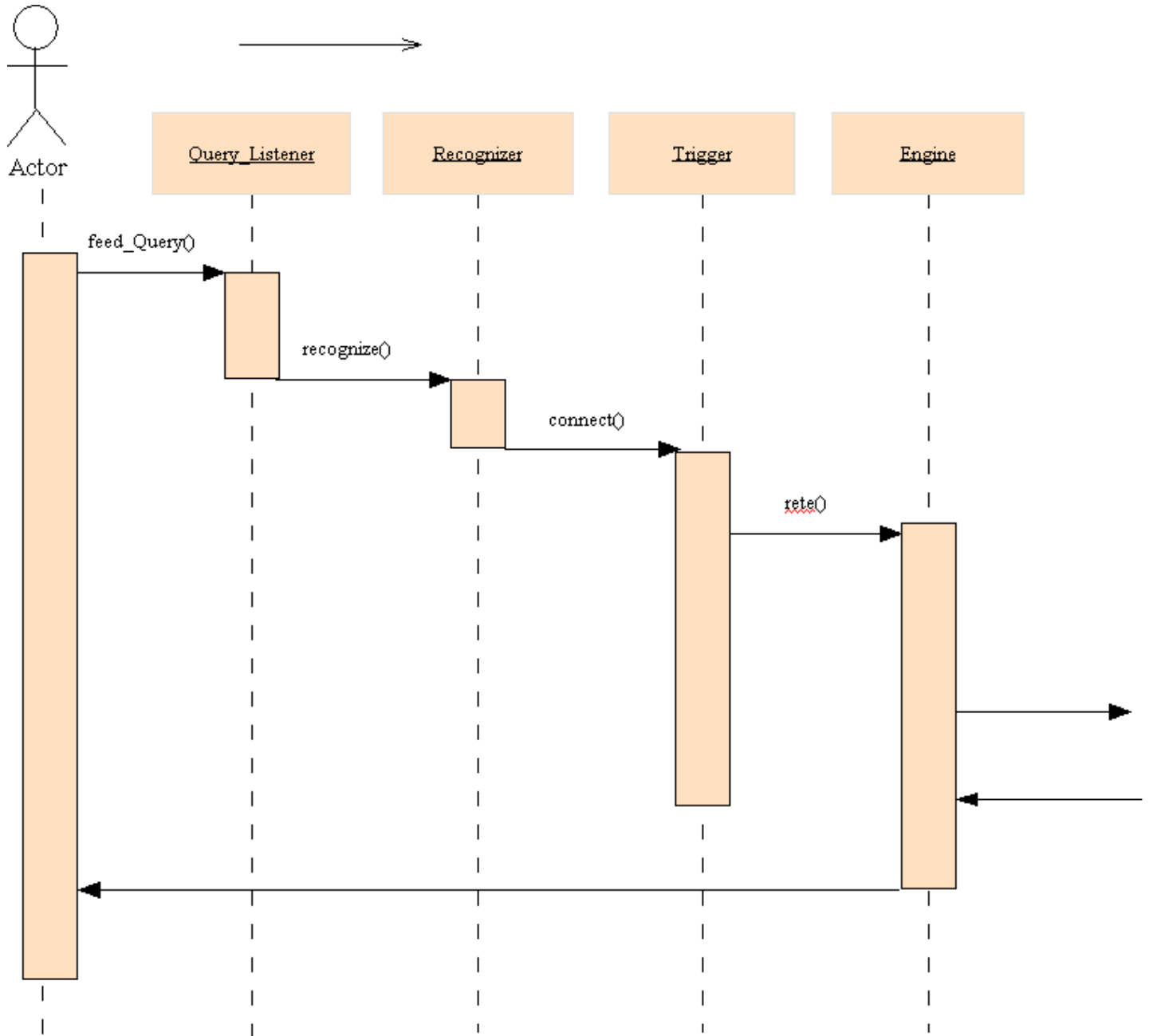


Connector Module

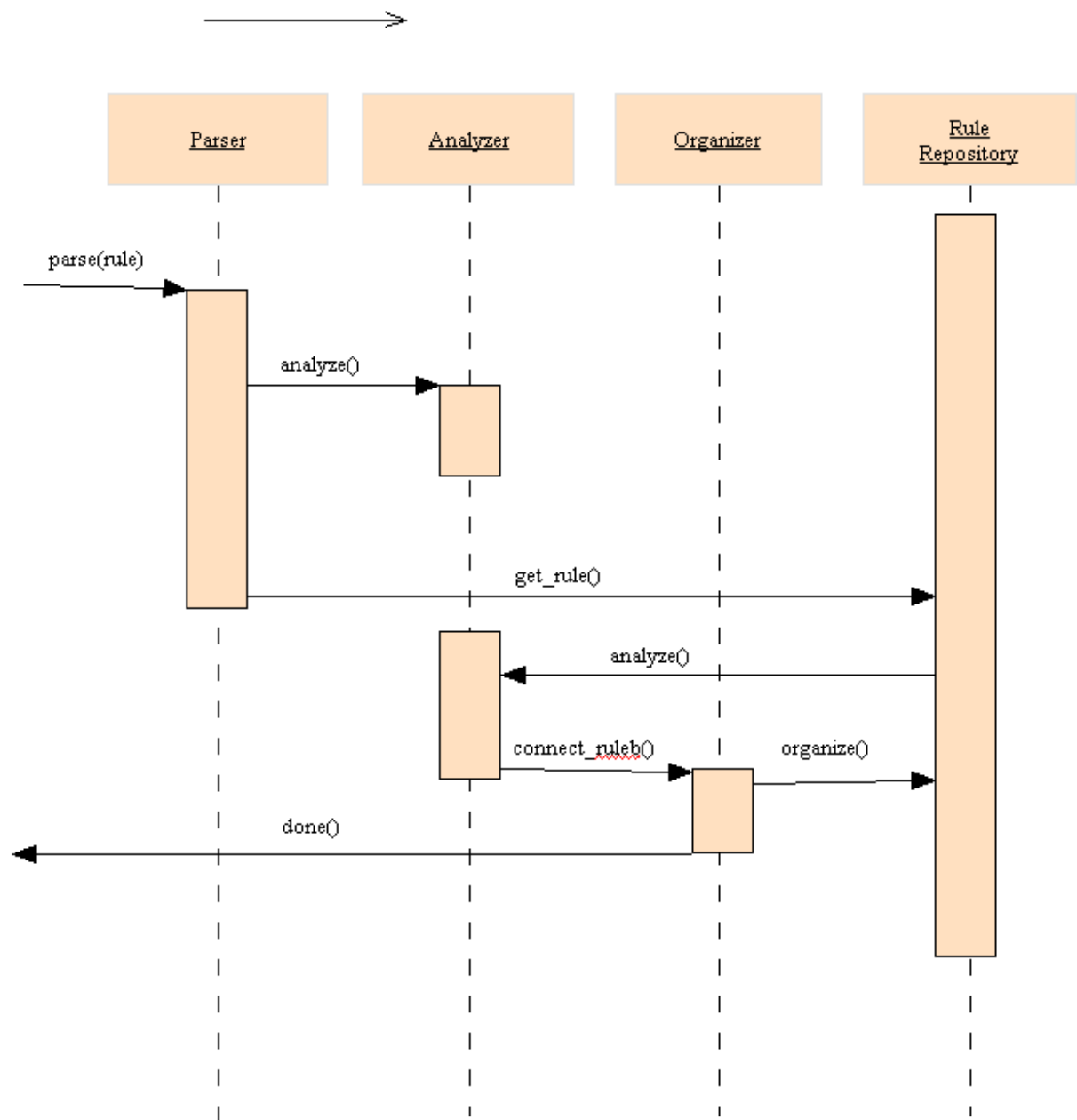


6.4. SEQUENCE DIAGRAMS

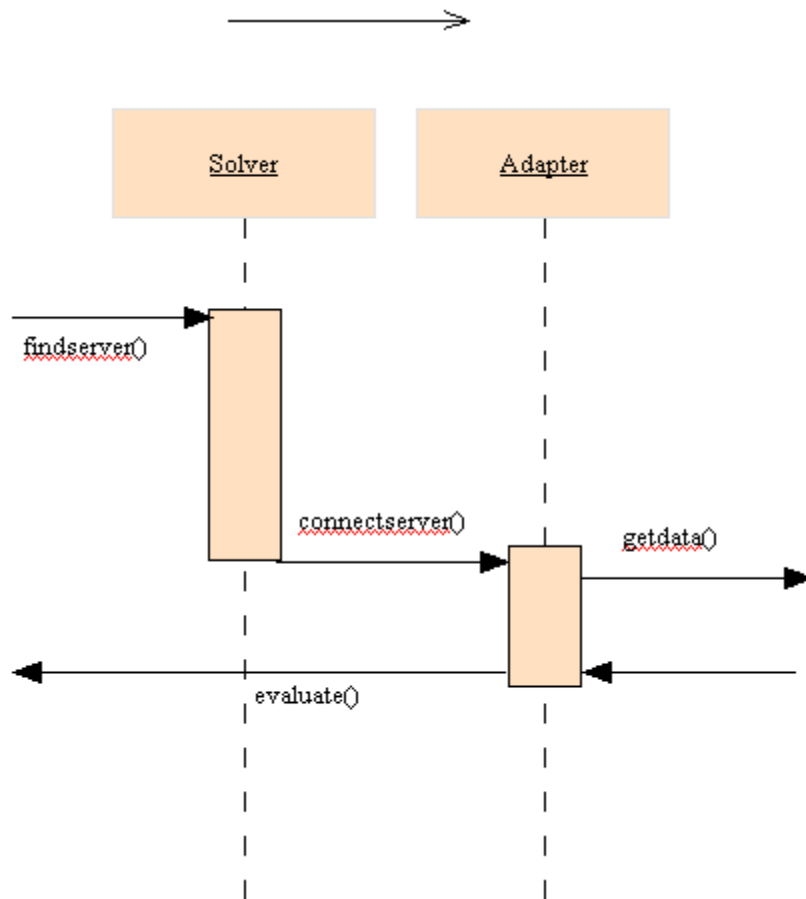
- Execution Module



- Management Module



- Connector Module



7. GRAPHICAL USER INTERFACE DESIGN

We have developed our user interface as a result of our needs and to make more usable for the users.

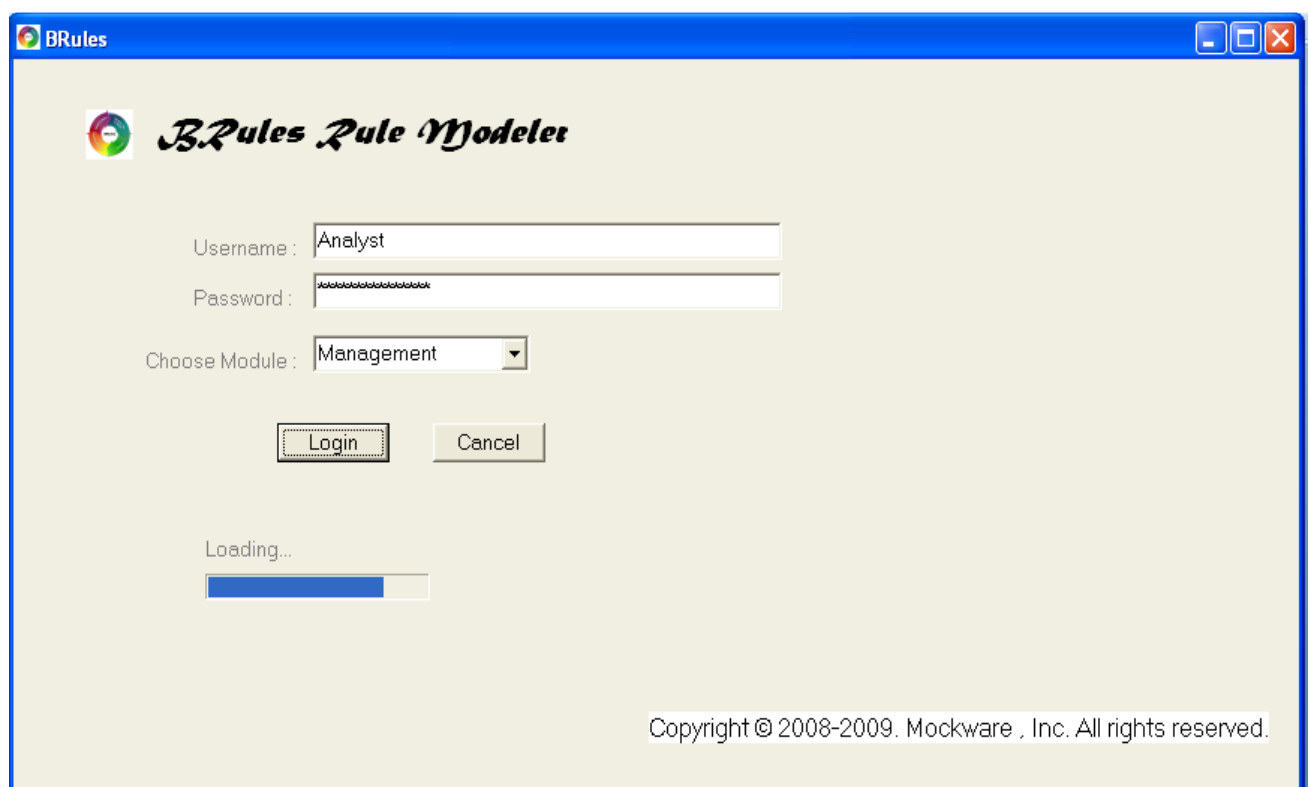


Figure 7.1- Login to Brules Rule Modeler

According to authentication of user, the user can login to one of BRules's modules. There are two modules can be opened according to the user's choose. In general, in these two screens are Management Module which user can insert, delete, update, and create new rules and Execution Module which the user can request and be responded a rule set as a table by the engine. When a user logs in to the system via Management Module, this means that the user will have all the rights

of managing the rule models. Otherwise, managing rules properties of our software will be disabled and just this user will have testing rights over existing rule sets.

When a user logs in to the system , the below screen will be opened ,that is, Brules Rule Modeler. There are 5 different parts in that menu , namely, Project Explorer, Rule Modeler, Rule Context, Outline, and lastly 3-tabbed menu with tabs ; Properties, Problems, Tasks. There are 8 options , File, Edit, Search, View, Project, Run, Window, Help .

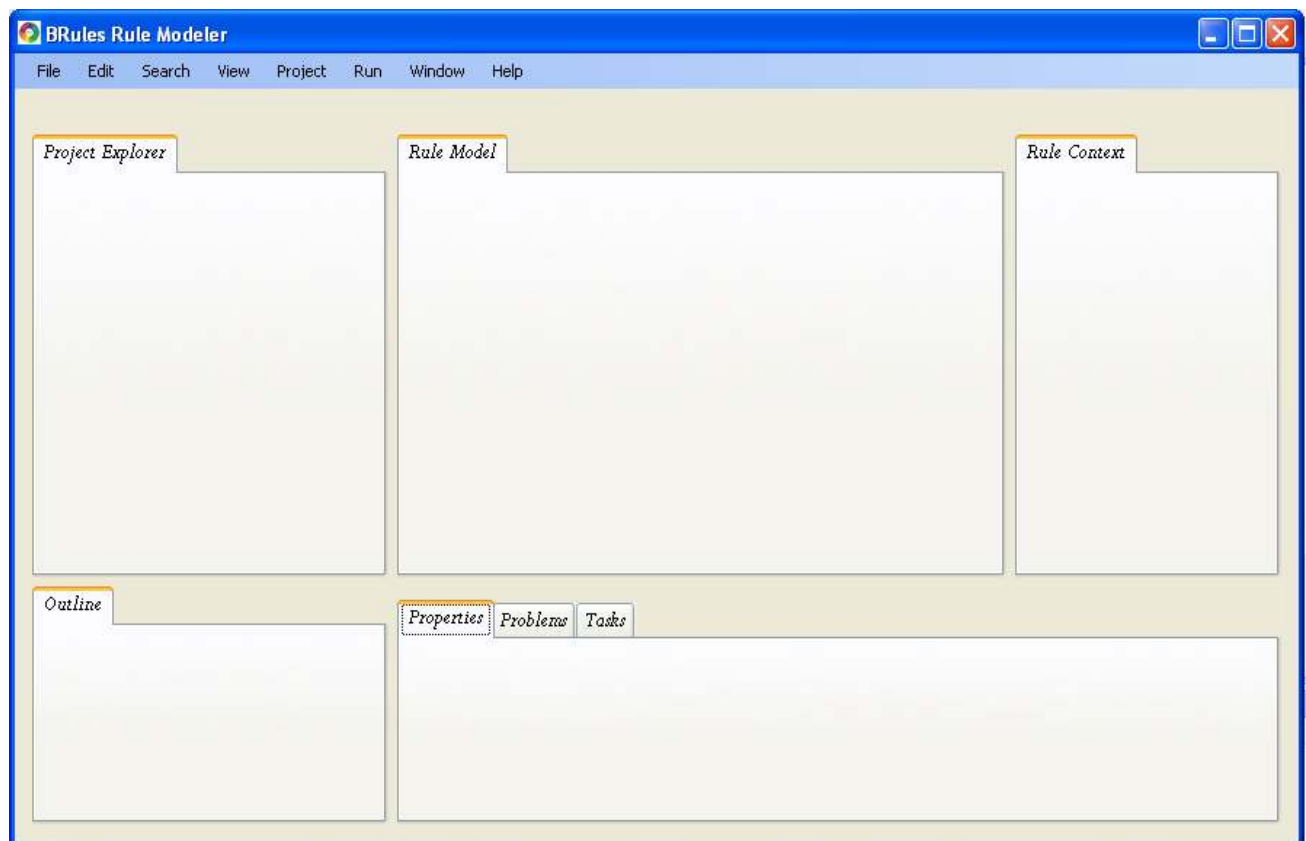
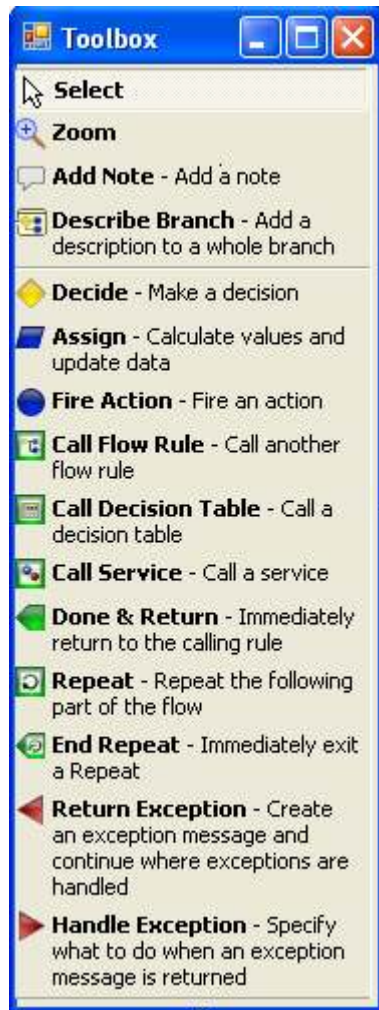


Figure 7.2- Brules Rule Modeler's Main Screen.

Moreover, to create new rules and change existing ones , there is a toolbox that contains all required tools with their explanations .This toolset has Select, Zoom, Add Note, Describe Branch, Decide, Assign, Fire, Action, Call Flow Rule, Call Decision Table, Call Service, Done & Return, Repeat, End Repeat, Return Exception and Handle Exception.

Below is the toolset :



We will have such a Toolbox to make our Rule Modeler capable of managing rules visually. At the end of our project our toolbox can be different this one according to our needs.

Figure 7.3- Brules Rule Modeler's Toolbox.

8. MISCELLANEOUS

In this section some determined features about our DSL and DSE are introduced.

8.1. LANGUAGE PATTERNS

Structures that we will use in our DSL are introduced in this section. These structures are namely queries, rules and facts. Brief descriptions, tags and examples of these structures are given in following subsections.

8.1.1. QUERIES

Description:

Queries allow one to query business rules using any rule property, including user-defined properties such as business rule author, effective/expiration date, and business rule status. You can also query on classes, attributes and methods referenced in the rules.

Queries are regarded as headless implications, symmetrically to regarding facts as bodiless implications. They enumerate the bindings of all their free variables.

Tags:

`<query::query_id:query_name>`

This tag is used to determine query name, query id and start of the query block.

`<_body>`

This tag is used to determine the body of the query

`<atom>`

This tag is used to encapsulate an atom expression

`<_opr>`

This tag determines which relations are included in the operation

`<rel>`

This tag determines relations between variables

`<var>`

This tag determines variables

<ind>

This tag determines values that correspond to a variable

Example:

```
<query::4:discount:>
  <_body>
    <atom>
      <_opr>
        <rel>discount</rel>
      </_opr>
      <var>customer</var>
      <var>product</var>
      <var>amount</var>
    </atom>
  </_body>
</query>
```

Give the discount amounts for all customers buying any products.

8.1.2. RULES

Description:

If the <body> of the rule structure satisfies then the <_head> part is evaluated.

Except for <query> tag in the query format all tags are used in rule format with same specifications.

Tags:

<imp::rule_id:rule_name>

This tag is used to determine rule name, rule id and start of the rule block.

<_head>

This tag is used to determine “then statement” of a rule.

<body>

This tag is used to encapsulate “if statement” of a rule.

<and>

This tag provides “and” logical operation.

<or>

This tag provides “or” logical operation.

Example:

```
<imp::5:premium>
  <_head>
    <atom>
      <_opr>
        <rel>premium</rel>
      </_opr>
      <var>customer</var>
    </atom>
  </_head>
  <_body>
    <atom>
      <_opr>
        <rel>spending</rel>
      </_opr>
      <var>customer</var>
      <ind>min 5000 euro</ind>
      <ind>previous year</ind>
    </atom>
  </_body>
</imp>
```

A customer is premium if their spending has been min 5000 euro in the previous year.

8.1.3. FACTS

Description:

Facts are most simple structures in our DSL. It is just like `<_head>` part of the rule syntax.

Tags:

`<fact::fact_id:fact_name>`

This tag is used to determine fact name, fact id and start of the fact block.

`<head>`

This tag is used to determine head of the fact.

Example:

```
<fact::4:honda>
  <_head>
    <atom>
      <_opr>
        <rel>regular</rel>
      </_opr>
      <ind>Honda</ind>
    </atom>
  </_head>
</fact>
```

A Honda is regular.

8.1.4. EXTERNALS

Description:

This structure is used to get data which are absent in our database from external sources.

Tags:

`<ext::ext_type>`

This tag determines the start of ext block and type of the external source. "ext_type" can be db, ws and api.

`<function_name>`

This tag identifies the function name which can return the related data with the information fed.

`<ext_query>`

This tag encapsulates the SQL query as a string that is used to get data from database.

`<data_type>`

This tag determines the data type of the needed data.

`<data_name>`

This tag determines the name of data that is requested from an external source.

Example:

`<ext::db>`

`<ext_query>`

`<Ind> "Select * From Employee" </Ind>`

`</ext_query>`

`</ext>`

8.2. ENGINE PATTERNS(FORWARD CHAINING)

Consider a system whose knowledge is represented in the form of production rules and whose domain is the truth of abstract propositions: A, B, C,

The knowledge base consists solely of rules as follows.

Rule 1: A and B and C implies D

Rule 2: D and F implies G

Rule 3: E implies F

Rule 4: F implies B

Rule 5: B implies C
 Rule 6: G implies H
 Rule 7: I implies J
 Rule 8: A and F implies H

To start with, assume that the system has been asked whether proposition H is true given that propositions A and F are true. We will show that the system may approach the problem in two quite distinct ways. Assume for the present that the computer stores these rules on a sequential device such as magnetic tape, so that it must access the rules in order unless it rewinds to rule 1.

What I am about to describe is a basic forward chaining inference strategy. This itself has several variants. We may pass through the rules until a single rule fires, we may continue until all rules have been processed once, or we may continue firing in either manner until either the conclusion we desire has been achieved or until the database of proven propositions ceases to be changed by the process. A little thought shows that this gives at least four different varieties of forward chaining. This will become clearer as we proceed.

The assumption is that A and F are known to be true at the outset. If we apply all the rules to this database the only rules that fire are 4 and 8 and the firing of rule 8 assigns the value true to H, which is what we were after. Suppose now that rule 8 is excised from the knowledge base. Can we still prove H? This time only rule 4 fires, so we have to rewind and apply the rules again to have any chance of proving the target proposition. Below we show what happens to the truth values in the database on successive applications of the rules 1 to 7.

Proposition	0	1	2	3	4	5	6	7
A	T	T	T	T	T	T	T	T
B		T	T	T	T	T	T	T
C			T	T	T	T	T	T
D				T	T	T	T	T
E								
F	T	T	T	T	T	T	T	T
G					T	T	T	T
H						T	T	T
I								
J								

Table 8.1 Naïve forward chaining.

So, H is proven after 5 iterations. Note, in passing, that further iterations do not succeed in proving any further propositions in this particular case. Since we are considering a computer strategy, we need to program some means by

which the machine is to know when to stop applying rules. From the above example there are two methods; either 'stop when H becomes true' or 'stop when the database ceases to change on rule application'. Which one of these two we select depends on the system's purpose; for one interesting side effect of the latter procedure is that we have proved the propositions B, C, D and G and, were we later to need to know their truth values, we need do no more computation. On the other hand, if this is not an important consideration we might have proved H long before we can prove everything else.

It should be noted that we have assumed that the rules are applied 'in parallel', which is to say that in any one iteration every rule fires on the basis that the data are as they were at the beginning of the cycle. This is not necessary, but we would warn of the confusion that would result from the alternative in any practical applications; a knowledge-based, and thus essentially declarative system, should not be dependent of the order in which the rules are entered, stored or processed unless there is some very good reason for forcing modularity on the rules. Very efficient algorithms, notably the rete algorithm, have been developed for this type of reasoning.

These strategies are known as **forward chaining** or data directed reasoning, because they begin with the data known and apply the rules successively to find out what results are implied.

This strategy is particularly appropriate in situations where data are expensive to collect but potentially few in quantity. Typical domains are loan approval, financial planning, process control, scheduling, the configuration of complex systems and system tuning.

In the example given, the antecedents and consequents of the rules are all of the same type: propositions in some logical system. However, this need not be the case. For example, the industrial control applications the inputs might be measurements and the output control actions. In that case it does not make sense to add these incommensurables together in the database.

Variations on forward chaining now include: 'pass through the rules until a single rule fires then act'; 'pass through all the rules once and then act'.

9. PROCESS

The most important elements that define a process are process model and team organization choices. They are explained below in detail.

9.1. TEAM STRUCTURE

The most proper team organization category for our group is *Democratic Decentralized (DD)*.

Our reasons to choose DD are:

- Consensus plays an important role in our decision making process
- We have no permanent team leader
- Every member of the team should understand how things are handled
- Communication is horizontal
- We have coordinators for tasks, namely rotating task coordinators, but these coordinators may change as tasks change

9.2. PROCESS MODEL

After discussing among group members, we decided to use waterfall software model depending on the characteristics of our project. This software model was the most suitable one for our project since in this model software evolution proceeds through an orderly sequence of transitions from one phase to the next in order. Furthermore, time constraints had affects to choose this model since there is a strict schedule with deadlines that is given at the beginning of the project.

Waterfall model was best matched model with our case. No overlap or iteration is allowed during any period of the process. After a phase is completed we will not turn back for large scale modifications. This classic model has been widely characterized as both a poor descriptive and prescriptive model of how software development "in-the-small" or "in-the-large" can or should occur. Alternatively, the most criticized property of waterfall model is the absence of evolutionary approach and supplying

the product only after all the phases are over. But in the implementation phase we will provide several prototypes in order to get rid of the mentioned disadvantage.

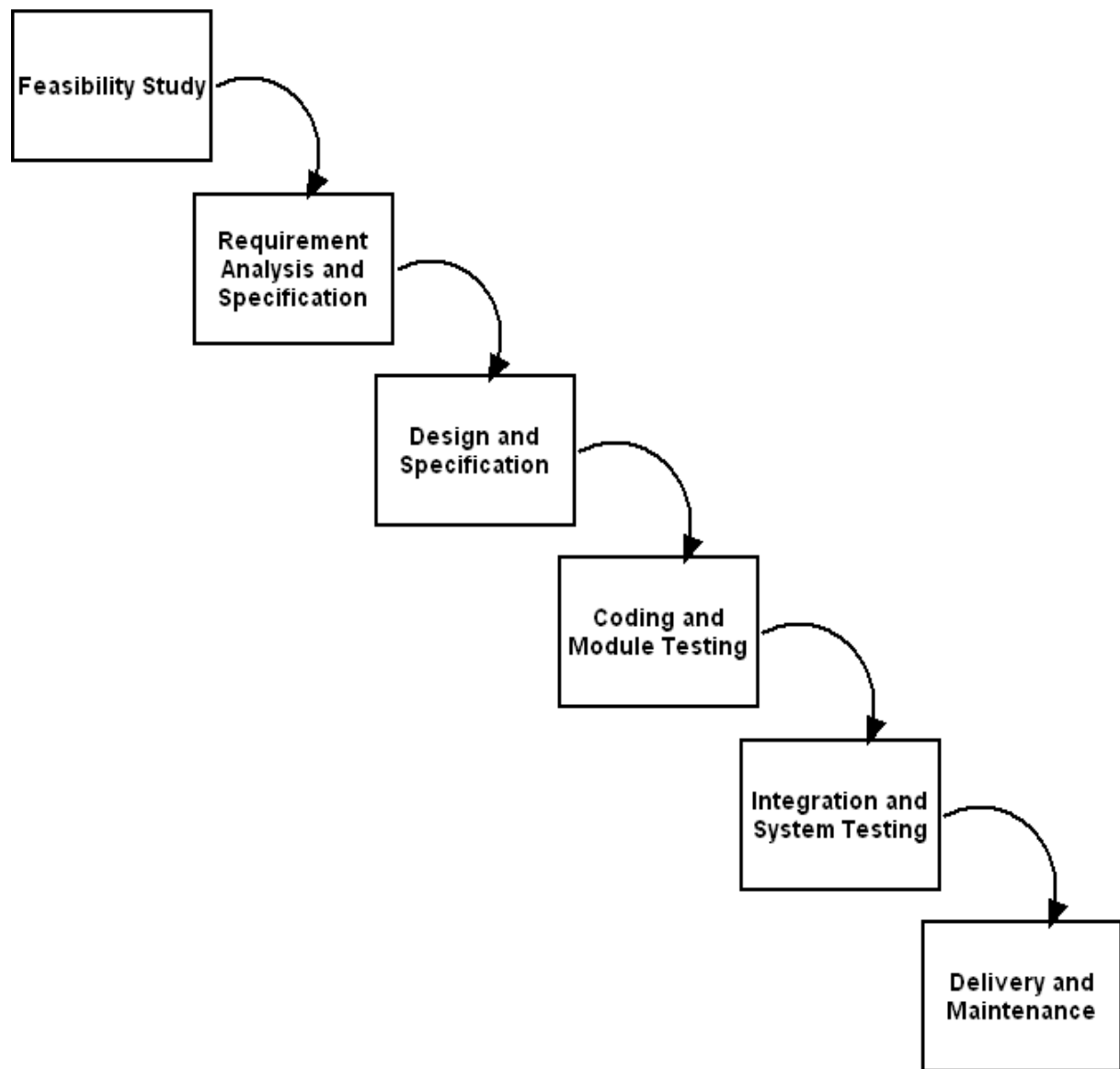


Figure 1 - Modified Waterfall Model

9.3. GANTT CHART

GANTT CHART



GANTT CHART



10. TESTING STRATEGIES AND PROCEDURES

As Edsger W. DIJKSTRA states, "Program testing can be used to show the presence of bugs, but never to show their absence!" Keeping this in mind, in order to have a program as free of bugs as possible, it is a must to have a good testing plan. Testing is a crucial process in our project, since the system will comprise of many components and modules, it can consume much time to detect an error after the system is up and running as a whole.

Unit Testing

We have chosen the strategy of white box testing because it considers the internal details of a module and tries to test every unit inside. Also, white box testing ideally would try any possible execution sequence for a module; that is every branch in the code is taken and every combination of input values is tried. So the module and classes which we are going to test by using white box testing are:

➤ Manager Module

- Parser class
- Analyzer class
- Organizer class

➤ Executor Module

- Query_listener class
- Recognizer class
- Trigger class
- Engine class

- Connector Module
 - Solver class
 - Adapter class

Also, we found that for decreasing the errors of our codes and making test easier for us, we have to put into our consideration these characteristics of codes that we will test.

- ✓ Understandability
- ✓ Simplicity
- ✓ Stability
- ✓ Observability
- ✓ Operability
- ✓ Controllability
- ✓ Decomposability

Integration Testing

After we have tested the modules individually, we are going to test all the units of the project as a whole and test for any collision between them. Also, We will test all the expected features of the system such as related data of the users of the program is managed, user interface is prepared for easy use, the module functionalities work properly. Moreover, we will carry out the integration test from the perspective of the possible users of our program. Finally, We will look for if all the modules are integrated successfully and the functionalities are interacting properly.

Validation Testing

Validation test asks for if the product is going in the right way or not and the answer specifies whether our program will be preferred by the form users or not. Therefore validation is important. So, we will perform a black box testing too in order to specify all the needed requirements and obtained possible errors and solve them.

11. SYNTAX SPECIFICATIONS

Because we are dealing with a project and not a simple work, we have decided to determine syntax specifications for our project in order to read, understand, add to, maintain and debug easily. The specifications are listed below:

Function names:

If a function name consists of more than one word, ‘_’ will be put between words to connect and form a one word function name.

Variable names:

Appropriate choices for variable names are seen as the keystone for good style. Poorly-named variables make code harder to read and understand. As a result, all variables begin with a lower-cased word, and if consisting of multiple words, the rest is underscored. Some variable examples are: “protocol_name”, “comment_line”.

Classes names:

- All classes must begin with a capital letter.
- Class methods begin with a lower case.

Comments:

Increasing the understandability of the codes, we write comments after some lines. We will use C++ comments to reduce the complexity.

It will be like this;

```
// this function does that
```

```
// this call does that
```

12. CONCLUSION

At the beginning of the project we didn't know anything about the project. After making so much research we improved our knowledge about the project and got idea about the case. From beginning to now we gained too much experience and improved our skills.

We had a great motivation for initial design report. After checking our early works, especially in requirement analysis report, we saw our missing parts. We fulfilled our lack of knowledge by doing so much research. Especially finding corresponding classes and methods was very hard and diagrams took most of our time. Keywords were also a big deal. Creating user interface also took much time. However, we believe that we overcame all problems and did a good job.

We are much aware of the significance of initial design since it is the main step to detailed design. So we did our best while working on initial design and created this document. We changed some parts in our plans that we did before while preparing requirement analysis report. We know that this initial design report reflects our idea and knowledge about the project better than early works. Although having some small problems in some details of project, we are now confident about the project.

13. REFERENCES

RuleML Homepage, Realize your knowledge, September 2008.

URL:<http://www.ruleml.org/>

jDrew Homepage, A Java Deductive Reasoning Engine for the Web, January 2004

URL:<http://www.jdrew.org/jDREWebsite/jDREW.html>.

CLIPS: A Tool for Building Expert Systems, May 2008

URL:<http://clipsrules.sourceforge.net/>

Jess, the Rule Engine for the Java Platform, November 2008

URL:<http://herzberg.ca.sandia.gov/>

Business Rules Management System, Enterprise Decision Management: Visual Rules, 2008

URL: http://www.visual-rules.com/business-rules-management-enterprise-decision-management.html?utm_source=GoogleAd&utm_medium=PPC&utm_content=G_W-Klicker_BRMS&utm_campaign=G_W-Klicker&gclid=CLqe3IfJyJcCFUwb3godU0cUSA

Business Rules Engine, Microsoft Corporation, 2008

URL: <http://msdn.microsoft.com/en-us/library/aa561216.aspx>

IBM Education Assistant-WebSphere software, November 2008

URL: http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.wpi_v6/wpswid/6.0/BusinessRules.html

CodeProject: Externalize your business rules, May 2007

URL: <http://www.codeproject.com/KB/dotnet/BRMS.aspx>

Jboss Drools, How do Jboss projects work together? 2008

URL: <http://downloads.jboss.com/drools/docs/4.0.3.15993.GA/html/index.html>