5-1-2005

A Rule Based Expert System Framework for Small Water Systems

Suresh Jayanty
Western Kentucky University

Follow this and additional works at: http://digitalcommons.wku.edu/theses
Part of the Computer Sciences Commons

Recommended Citation
http://digitalcommons.wku.edu/theses/502

This Thesis is brought to you for free and open access by TopSCHOLAR®. It has been accepted for inclusion in Masters Theses & Specialist Projects by an authorized administrator of TopSCHOLAR®. For more information, please contact topscholar@wku.edu.
A RULE BASED EXPERT SYSTEM FRAMEWORK FOR SMALL WATER SYSTEMS

A Thesis
Presented to
The Faculty of the Department of Computer Science
Western Kentucky University
Bowling Green, Kentucky

In Partial Fulfillment
Of the Requirements for the Degree
Master of Computer Science

By
Suresh C Jayanty

May 2005.
A RULE BASED EXPERT SYSTEM FRAMEWORK FOR SMALL WATER SYSTEMS

Date Recommended

Dr. Uta Ziegler
Director of Thesis

Dr. Andrew Ernest

Dr. Guangming Xing

Dr. Elmer Gray
Dean, Graduate Studies and Research, Date
Acknowledgement

I would like to thank, Dr. Uta Ziegler for her excellent guidance and valuable feedback at various stages of this work, Dr. Andrew Ernest for providing this opportunity and guidance, and Dr. Guangming Xing for his invaluable comments. I would also like to thank my parents for their unabated love and support.

Suresh C. Jayanty
### Table of Contents

Acknowledgement ........................................................................................................ iii

List of Figures ............................................................................................................... v

Chapter 1: Introduction ................................................................................................. 1

Chapter 2: Background ................................................................................................. 4
  2.1 Expert Systems ..................................................................................................... 4
  2.2 Forward and backward chaining ........................................................................... 10

Chapter 3: Framework and prototype ......................................................................... 20
  3.1 Design criteria ..................................................................................................... 20
  3.2 General structure ................................................................................................. 21
  3.3 System interaction ............................................................................................... 29
  3.4 Total Coliform Rule ............................................................................................ 31
  3.5 TCR expert system GUI ....................................................................................... 37

Chapter 4: Conclusions .............................................................................................. 41

Appendix I .................................................................................................................. 42

Appendix II ................................................................................................................ 44

Appendix III ............................................................................................................... 48

Appendix IV .............................................................................................................. 49

References .................................................................................................................. 52
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Knowledge-Based Expert System architecture</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Example IF-THEN type of rule</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Inference process in a Rule-based Expert System</td>
<td>7</td>
</tr>
<tr>
<td>Figure 4</td>
<td>An example course prerequisite structure</td>
<td>8</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Temporal redundancy</td>
<td>12</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Structural similarity</td>
<td>12</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Example RETE network</td>
<td>15</td>
</tr>
<tr>
<td>Figure 8</td>
<td>The Framework Architecture</td>
<td>22</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Example XML data</td>
<td>25</td>
</tr>
<tr>
<td>Figure 10</td>
<td>RULEML design</td>
<td>25</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Interactions between knowledge engineer, JESSGUI and ORYX, and NXD</td>
<td>30</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Interactions between NXD, Rule Transformer module and the Inference engines</td>
<td>30</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Interactions between the User, GUI and the Inference engines</td>
<td>30</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Total coliform rule flowchart</td>
<td>33</td>
</tr>
<tr>
<td>Figure 15</td>
<td>TCR facts</td>
<td>35</td>
</tr>
<tr>
<td>Figure 16</td>
<td>TCR Monthly violation rule 1</td>
<td>35</td>
</tr>
<tr>
<td>Figure 17</td>
<td>TCR Monthly violation rule 2</td>
<td>36</td>
</tr>
<tr>
<td>Figure 18</td>
<td>TCR Acute violation rule 1</td>
<td>36</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Rule Selector sample screen</td>
<td>37</td>
</tr>
</tbody>
</table>
Figure 21: Total Coliform Rule routine samples screen ................................................. 38

Figure 22: Repeat sample screen. ................................................................................. 39

Figure 23: An example result dialog screen. ................................................................. 40
Using an expert system to make decision making more reliable has been well studied and implemented over the years. For effective use, both data-driven questions (forward chaining) and goal-driven questions (backward chaining) need to be supported. Similarly, an avenue to update rules in the system as and when they change without major recompilation should be available.

In this thesis we present an expert system framework that can help small water system operators make informed decisions regarding compliance with various EPA rules that may apply to them. To support both types of questions mentioned earlier, the system incorporates two expert system shells: JESS for answering data-driven questions such as “This is my reading for sample X. What needs to happen next?” and MANDARAX for goal-driven questions such as “We want to be compliant with the Total Coliform Rule. What do we need to do?” To make sure that rules are consistent and to support a straightforward rule-updating process, we use a native xml database to store the rules. All the rules are in XML format which ensures better symbiosis with other tools that support XML and allows one set of rules to be used for both JESS and MANDARAX.
Chapter 1: Introduction

This thesis is about the design and implementation of a rule-based expert system framework in the domain of EPA rules for small water systems. A water system is a company that is entrusted the responsibility of supplying drinking water to communities or individuals. A small water system is a water system that supplies water to a population of 3300 people or less. About 95% of the water systems in the United States are small water systems [16]. Small water systems supply water, which is one of the main criteria for economic development in the areas they serve.

Even though the small water systems play such a vital role they face managerial, technical and financial challenges. The work in this thesis is aimed at supporting the managerial capacity of the small water system operator. Small water systems cannot afford hiring a knowledgeable expert [16] to help maintain compliance to EPA guidelines. A knowledge-based expert system, whose domain of knowledge is EPA regulations, can be a cost effective solution to the problem. By using such a system a small water system operator will be able to make decisions regarding compliance to EPA guidelines more effectively.

An expert system is a computer program which can emulate the problem solving skills of an expert in a particular domain. Expert systems have been used before, to support decision making regarding water and/or environmental issues. A number of expert systems: Residential water conservation techniques [8], Cornell Mixing Zone Expert System (CORMIX) [2] and Solvent Alternative Guide (SAGE) [9] built for similar domains were reviewed. The expert system is described in [8] can be used to calculate the
amount of water as well as the monetary savings possible by installing water efficient
devices in homes. CORMIX is an expert system that can be used by water quality
analysts to simulate effects of various types of discharges into water bodies. SAGE is an
expert system that can evaluate industrial parts cleaning and degreasing processes and
gives advice on environmentally safe alternatives for those processes.

As part of the thesis, a framework for expert systems in the EPA-rule domain was
developed. This framework was applied to build a user-friendly expert system. The
development of the framework to meet certain design criteria is a major part of the work
completed for the thesis. Some of the components used in the framework were taken off-
the-shelf. One of the reasons for doing so is that these components are already very
mature and serve their purpose very well. A better system can be built in the time
available by using existing components rather than by implementing everything.

Upon completion of the framework design, the framework was applied to the
problem domain by developing a prototype that can give advice regarding one of the EPA
rules that small water systems must satisfy. Implementing this application is the second
major contribution of this thesis. It required the development of modules:

- to coordinate the working of the off-the-shelf components (including all
  extra code needed to support the exchange of information between
  components);
- to represent the EPA-rule related knowledge in a form suitable to the use
  in the framework;
- and a GUI to make the system easy to use.
As a general outline for the rest of the document, chapter two gives the background information and literature overview. Chapter three is divided into five sections: the first three sections introduce the framework and explain each of the components that were used or developed and their interactions. The last two sections describe the prototype built using this framework for the domain of Total Coliform Rule of the EPA.
Chapter 2: Background

This chapter contains a general outline of various topics of interest in this thesis. Section 2.1 is an introduction to expert systems explaining the general structure and a few issues related to them. Section 2.2 is an outline about the forward and backward chaining mechanisms. An expert system uses one or both of these to infer an answer when asked a question.

2.1 Expert Systems

An expert system is a computer program that can emulate the problem solving ability of an expert in a particular domain [14]. With a front-end which has natural language processing capability, it can mimic a human expert quite closely. There are basically four parts to an expert system:

a) Knowledge base: The Knowledge base is the knowledge, inside the expert system. The most popular form for storing knowledge is If-Then rules.

b) Working memory: The working memory is a storage space for facts in an expert system. The content of the working memory changes as the inference process progresses in the expert system.

c) Inference engine: The inference engine is the brain behind the expert system. It takes rules from the knowledge base and facts from the working memory, and draws conclusions from them. The conclusions represent the response of the expert system to a user’s query.

d) GUI and Explanation Facility: The GUI (Graphical User Interface) and explanation facility acts as the interface between the user and the expert system.
Figure 1 is a pictorial representation of an Expert System.

![Figure 1: Knowledge-Based Expert System architecture. The left pane represents the actual expert system and the middle pane the interface to the user at the right.](image)

The knowledge base represents the expertise that is entered into the system [14]. This expertise may be collected from a variety of sources: interviewing an expert in that particular domain, reading literature related to the proposed domain of expertise, etc. This collected expertise is then stored in the knowledge base. This process is called knowledge engineering. The most popular way of storing knowledge is in the form of IF-THEN statements. The IF-THEN statements represent actions to be taken "if" a condition is satisfied. Figure 2 gives an example of knowledge represented as a rule. Other modes of storing knowledge are semantic nets, schemas, and frames among other methods [14]. Semantic nets are a representation of declarative knowledge. In simple words they represent knowledge using constructs that state or declare something. For example the statements "All mammals give birth to fully born babies" and "A whale is a kind of mammal" can be represented in a semantic net. Schemas are a way to store causal knowledge. This type of knowledge cannot be represented using semantic nets or rules
because they do not support the notion of representing knowledge as to why a certain thing happens. For example, IF-THEN rules and semantic nets are used to store the knowledge “If person has a fever then take aspirin.” These representation methods do not have a way of representing why taking aspirin reduces fever. This information is useful if the fever is not subsiding, even after the use of aspirin. This type of causal knowledge can be stored using schemas [14]. A frame is another type of knowledge representation structure. Frames are actually a type of schema that can be used to simulate common-sense knowledge.

![IF CONDITION(S) THEN ACTION(S)](image)

General format of a rule

![IF "CURRENT SAMPLE EXCEEDS THRESHOLD" THEN "INCREASE SAMPLE COUNT NEXT MONTH"](image)

An example rule

Figure 2: Example IF-THEN type of rule. The condition is also called ‘antecedent’ and the action is also called ‘consequent’

The inference engine is the part of the expert system that “infers” or concludes a solution based on the knowledge base and the facts supplied by the user [14]. It does so by deciding which rule to execute next from a set of rules which are ready to be executed. When all the facts that a rule depends on are available, the rule is said to be ready for execution. The technical term is activation. The set of activated rules is called agenda. The inference engine applies a conflict resolution strategy to select one rule out of the activated set to “fire.” The firing of a rule results in the consequent part of the rule being
executed. Once a rule that is on the agenda is fired, it is removed from the agenda; after updating the system with the consequences of this firing, the inference engine moves on to the activated rule with the next highest priority. This process is cycled until a condition that makes the execution stop is met. Figure 3 gives an algorithmic view of the above process [14].

```
WHILE at least one rule is activated

**Conflict Resolution:** Select the rule with the highest priority among the activated rules and fire it.

**Act:** Sequentially perform the consequents of the just fired rule. If a rule changes the working memory, the effects are immediately visible. Remove the rule that was just fired from the list of activated rules.

**Match:** Check if all conditions on the antecedent part of any rule are satisfied. If so add that rule to the activated rules list. If any condition of the antecedent of an activated rule is missing from the working memory, remove that rule from the agenda.
```

Figure 3: Inference process in a Rule-based Expert System.

A process called refraction is used by the inference engine to take care of trivial looping of rules. A trivial looping occurs when the same fact(s) cause a rule to fire again and again. For example consider the rule- If “fire-detected” then “start sprinkler.” Now if the fact “fire-detected” is asserted into the system, the above rule fires (pun unintended) and the sprinkler is started. Now since a fact, when asserted into the system, stays there until it is removed explicitly, the start-sprinkler rule would execute again and again, which is definitely undesirable. So the process of refraction is used in which once a fact(s) causes a rule to fire, that fact(s) cannot make the rule fire again unless the fact(s) are removed and reinserted again.
There are two types of inference mechanisms. Fact driven inference, called forward-chaining, and goal-driven, called backward-chaining. The internal working of both paradigms is explained in section 2.2. As a general overview of the process, the forward-chaining engine starts with a set of facts and continues on until it can no longer find rules to fire from the present set of facts. A backward chaining engine takes a goal or hypothesis and works backwards to the facts that can support the hypothesis. In terms of rules and facts, the system tries to find facts which support the goal given to it. If it cannot find the facts it needs, it searches for rules which when executed would make those facts available. This process continues until the system is either able to show that the goal is achievable or not achievable. If the goal is achievable, the expert system will let the user know which steps must be taken (that is which facts must be provided) to reach the goal. The forward chaining and backward chaining processes are explained based on a sample course prerequisite structure shown in Figure 4.

![Figure 4: An example course prerequisite structure.](image-url)
The forward chaining process begins on the left side of the picture above and continues towards the right and the backward chaining starts on the right side and continues towards the left side. For example if the user asks the system “What are the courses for which I have taken the prerequisites already?” the inference engine has the information that CS 240, CS 241, CS 244, CS 250, and CS 242 have already been taken (that a course is already taken is represented in the system as a fact), so it can specify to the user that she has all the prerequisites satisfied for courses CS 444, CS 360 and CS 338 (no prerequisites is equivalent to having prerequisites satisfied). The system is able to get to this solution by taking the facts already in the system and finding prerequisite rules which can be fired from the present set of facts.

For a backward chaining engine example, consider the user asking the system “What are the courses I have to take before I take CS 425?” The inference engine begins by making CS 425 the goal. It determines whether any fact already in the system supports the goal, which is whether the student has already taken CS 425. In this example the inference engine does not find such a fact. So from the prerequisite structure it infers that the student needs to complete CS 340 and CS 360 before she can take CS 425. It begins looking for facts which support the hypothesis that the student has already taken CS 340 and CS 360. Because it cannot find these supporting facts the inference continues working backwards by taking CS 340 and CS 360 as sub-goals in turn. This process continues until the engine can no longer work backwards. In this example the engine halts by outputting CS 340, CS 360, and CS 338 as the courses the student needs to take before CS 425 can be taken.
The working memory is the place where all the facts are stored when the expert system is running. Facts may be added or removed from the working memory while the expert system is going through the inference process. The expert system reasons on the facts present in the working memory. There are basically two types of facts namely unordered facts and ordered facts. Unordered facts are also called slotted facts, while an unslotted fact is the other name for ordered facts. An example of an unslotted fact is “(RAINING),” and an example of a slotted fact is “(NAME JOE).” Slotted facts are useful when multiple facts which follow a certain template are needed. For example all the facts (NAME ALI), (NAME JOE), (NAME SURESH) satisfy the template (NAME ?x). The process of applying a value to the variable in a slotted fact as is done above is called variable binding. Unslotted facts on the other hand are useful when a temporary assertion is needed to be put in the working memory.

2.2 Forward and backward chaining

This section is an introduction to the internal working of pattern matching in forward chaining and backward chaining engines like JESS and MANDARAX, respectively. A pattern is a construct on the antecedent part of a rule which needs to be satisfied for the rule to be activated. Thus a pattern can be a combination of slotted facts and/or unslotted facts. The system tries to find facts in the working memory which satisfy these pattern(s). This process is called pattern matching. A straightforward algorithm for pattern matching is to look at a rule’s pattern(s) and check all combinations of fact(s) which might satisfy the pattern(s). More than one combination of facts may be found which satisfies the pattern. The inefficiency of this algorithm is that it cannot scale to larger number of rules.
Figure 5 shows the inefficiency of the simple pattern matching algorithm. As fired rules may add or delete facts from the working memory, new rules may be activated or already activated rules may need to be removed from the activated list because the facts that satisfied their patterns are no longer in the working memory. So after each cycle the system needs to check which activated rules are still valid and which rules have new activations. Two properties of rule based systems are not taken advantage of by the above algorithm: they are temporal redundancy and structural similarity.

At any time, the number of facts added or removed is a small subset of the total facts. This property is called temporal redundancy. In other words, the majority of the facts are not changed from one cycle to another. It can be seen that time and processing power can be saved by identifying patterns that are affected by the changes in the facts in the working memory and by not wasting time on patterns which are not affected by the changed facts. So the process of updating the agenda should be driven by the changed facts and by not by checking the antecedent of each rule. See Figure 5.

Rules may share the same patterns: thus a set of facts which satisfies the pattern of a particular rule can also satisfy the same pattern of another rule. This concept is called structural similarity. So if a combination of facts is found which satisfies a pattern in one rule, this combination information - if saved - saves the inference engine the resources needed to find the same combination to satisfy the same pattern for some other rule. See Figure 6.
A better algorithm is to do the pattern matching in an incremental manner: save the configuration of facts created so far and check only with facts which were added or
removed since the last check. For this algorithm to work there must be a way to store information about which rules are activated by what facts and which rules require what facts to be activated. The algorithm used by JESS and many other forward chaining engines to accomplish the storing is called RETE. The RETE algorithm [13] is an efficient pattern matching algorithm for forward chaining engines. The algorithm takes advantage of temporal redundancy and structural similarity of rules, which is analogous to the facts finding the rules as opposed to the rules finding the facts, logic which the simple pattern matching algorithm implements.

The idea of RETE is to construct a network of nodes, where each node represents a pattern occurring on the left hand side of a rule in the knowledge base. The facts (both asserted and removed) move through this network. A fact activates a node and moves to the next level if it satisfies all the conditions that the node imposes. Once all the nodes which represent the patterns of a rule are active, the rule is said to be activated.

So in this case when a new fact is asserted into the system, all that the pattern matching routine has to do is to add the new fact to the network and the movement of the fact through the network leads to the updated configuration. Similarly if a fact is removed from the working memory, all the nodes which were activated by this fact must be deactivated. In order to perform these operations abstractly, inference engines use the concept of tokens. Tokens are an association of facts from the working memory with an instruction. JESS has four types of instructions namely, ADD, REMOVE, UPDATE, and CLEAR. ADD is associated with a fact when that fact is asserted; REMOVE when that fact is retracted; UPDATE when a duplicate fact is circulated; and CLEAR is associated with a fact when the rule engine is flushing all rules and the working memory. Thus when
a node sees an ADD type token, it knows it can use the associated fact if it satisfies any condition of that node. Similarly when a REMOVE type token is seen, the node knows that if the associated fact is needed to satisfy its pattern, then the node is deactivated.

Consider the following example rule from the example course prerequisite structure in the previous section:

If
(\text{CS} \ ?\text{course\_name})
And
(\text{PREREQ} \ ?\text{course\_name}, \ ?\text{prereq\_name})
And
(\text{DONE} \ ?\text{prereq\_name})
Then
\text{CanTake} (\ ?\text{course\_name})

The rule basically takes a CS course fact and checks if there is a PREREQ fact with the same course name and a DONE course fact with the prerequisite name. If the system can find a combination of these three facts which match these patterns then the system tells the user that the student has satisfied a particular prerequisite of a subject she wants to take.

The RETE network for this rule is as shown in Figure 7:
The diamond shaped nodes are called one-input nodes and the trapezoidal nodes are called two-input nodes [13].

The main task of a one-input node is to check if the name of the token which is going through it has the same name as itself. For example the leftmost one-input node only allows tokens to pass through whose name (technically head) matches 'CS.' When a one-input node allows a token to go through it, the token becomes the output of that node.

The two-input nodes are also called join nodes. The reason is that their main task is to join the results of pattern matches coming in from the left input with the right input. A join operation is an intersection of two sets (called inner join; there are other join operations which are not exactly intersection). So the join-node generates as output all

![Example RETE network](image)
facts from the left and right input which have a common field. In the example above, all CS and PREREQ facts with same \( ?\text{course\_number} \) value are filtered through the first join-node. While a join-node can receive more than one input on its left channel, it cannot process more than one input on its right channel at the same time. Join-nodes remember the input they get on both their input channels. In order to remember they maintain two memories called alpha and beta memories, left and right memories non-technically. When one of potentially several tokens coming from the left channel matches with that from the right channel, an output is generated from the two-input node. Then the two-input nodes moves on to determine whether there is a match between the left channel inputs and the next right channel input. A two-input node generates an output for each match.

The oval node at the bottom of the network is the terminal node. Terminal nodes represent individual rules. When a terminal node receives input, the rule that it represents can be sent to the activation list because all the tests on the LHS of the rule were successful.

While RETE is a pattern-matching algorithm used by forward chaining engines, the Unification algorithm is the pattern-matching algorithm used by backward chaining inference engines. They use unification to find a set of bindings to variables to equate a goal with the right hand side (consequent) of some rule. In general the backward chaining inference process begins with taking a goal pattern and searching the knowledge-base for rules which have the goal pattern on their RHS (consequent). Once such a rule(s) is found, its LHS or antecedent part is taken and the working memory is searched to see if any facts match the patterns on the consequent. If no facts are found the patterns on the
antecedent become the new sub-goals, and the process continues finding rules which
have these patterns on the consequent side. This process continues until either there are
no more rules that have the required patterns on their RHS or facts are found in the
working memory to support all the patterns. The unification algorithm finds a consistent
set of fact substitutions that would make two patterns look alike. Mathematically, for two
patterns \( p \) and \( q \), if \( \text{UNIFY}(p, q) = \partial \) then \( \text{SUBST}(\partial, p) == \text{SUBST}(\partial, q) \) where \( \partial \) is the
set of fact substitutions and \( \text{SUBST} \) is a function which applies \( \partial \) to variables in \( p \) or \( q \).
As an example consider giving the UNIFY algorithm the two patterns: \( \text{PREREQ} (\text{CS445}, x) \) and \( \text{PREREQ} (\text{CS445}, \text{CS338}) \) results in the fact substitution list with the
following single entry: \( \{\text{CS338}/x\} \). This means if every \( x \) is replaced by \( \text{CS 338} \), then the
two patterns are equal.

The following is the unification algorithm from [17]:

\{L1 and L2 are patterns to be unified, L1 and L2 may contain variables.\}

\text{UNIFY}(L1, L2)

1. If \( L1 \) and \( L2 \) are both variables or constants
   a. If \( L1 \) is equal to \( L2 \) then return Nil (No substitutions are needed to make
      \( L1 \) equal to \( L2 \))
   b. If \( L1 \) is a variable and
      i. If \( L1 \) occurs inside \( L2 \) then return \{FAIL\}. (\( L1 \) occurring within
         \( L2 \) means that \( L1 \) is used with in \( L2 \) as in \( L1 = x \) and \( L2 = f(x) \).
         We return FAIL when one variable occurs inside another
         variable since replacing \( x \) with \( f(x) \) does not eliminate \( x \)).
      ii. Else return \( \langle L2/L1 \rangle \) (That is we substitute \( L2 \) for \( L1 \))
c. If L2 is a variable and
   i. If L2 occurs inside L1 then return \{FAIL\}. (Same reason as above.)
   ii. Else return (L1/L2) (Substitute L2 for L1.)

d. Else return \{FAIL\}. (Control comes here when both L1 and L2 are constants and they are not similar.)

2. If the initial predicate symbols are not same return \{FAIL\}. (If the heads or the names of the patterns do not match, they cannot be unified. For example (PREREQ (SUB X)) and (CS (SUB CS445)) are two different facts and we cannot unify them.)

3. If the initial predicate symbols are the same but the number of arguments is different, then returns \{FAIL\} ((CS (SUB X)) and (CS (SUB X) (SUB Y)) are different.)

4. Initialize S, the set of substitutions to make to be NIL. After step 5 is completed successfully S contains the list of substitutions needed to make L1 and L2 equivalent.

5. For I ← 1 to number of arguments in L1 (Control comes here when L1 and/or L2 are multi-slotted patterns, so each slot needs to be unified to get the total substitution set.)
   a. Call Unify routine with j\textsuperscript{th} argument of L1 and the j\textsuperscript{th} argument of L2 storing the result to S. (This step finds the substitution needed for the j\textsuperscript{th} argument between L1 and L2.).
b. Is $S$ contains FAIL then return $\{\text{FAIL}\}$. (There exists no substitution for the $j^{th}$ arguments between $L_1$ and $L_2$ so the unification failed).

c. If $S$ is not equal to NIL (If both arguments from $L_1$ and $L_2$ are not identical) then

   i. Apply $S$ to the rest of the $L_1$ and $L_2$ (Apply the substitutions in the set $S$ to the rest of the $L_1$ and $L_2$. For example if $S$ is $\{\text{CS340/x}\}$ then all occurrences of $x$ in the rest of $L_1$ and $L_2$ are replaced with CS340).

   ii. $\text{SUBST} = \text{SUBST} + S$ (Append $S$ to the final output).

6. Return $\text{SUBST}$. ($\text{SUBST}$ is the final substitution set that is returned. It contains all the substitutions necessary to make $L_1$ identical to $L_2$).
Chapter 3: Framework and prototype

This chapter is a description of the design and implementation of the framework that was built as part of the thesis. The first section deals with the design criteria. The second section is an introduction to the different parts of the framework. The third section gives a brief picture of the interactions between each of the components of the framework. The fourth section begins with a brief introduction to the Total Coliform Rule (TCR), the example domain of expertise that was used for the expert system built and continues with a description of the knowledge engineering process for the example domain above for small water systems. The final subchapter uses a set of sample screens to describe the application that was built on the proposed framework. The prototype can give advice on the Total Coliform Rule of the EPA.

3.1 Design criteria

We have set for ourselves a few design criteria for the framework and the application:

**Usability:** The application interface should look similar to the standard paper worksheet.

**Easy rule updateability:** EPA rules or related information and data change. When this happens the application should not become out-of-date. The change in rules should be propagated easily with little or no code modification. Also in order to cut inconsistency, all the knowledge should be in a single location.

**Clean separation of logic and GUI:** The application should follow the Model-View-Controller (MVC) architecture [15]. The model contains the application code or business logic, what we are trying to achieve; the view is what the users see on their
screen and the controller is an interface between the view and the model that is responsible for updating the view as specified by the model. MVC also allows for better debugging and updateability.

**Type of Questions:** The application should support different types of questions, both forward and backward chaining type questions.

### 3.2 General structure

The framework is an integration of available software with inter-lying glue and some of the components being written by the author. Specifically the software that were used off-the-shelf are:

- **JESS**, a forward chaining expert system shell.
- **MANDARAX**, a backward chaining expert system shell.
- **dbXML**, a native XML database.
- **JESSGUI**, a GUI (Graphical User Interface) based knowledge engineering tool for JESS.
- **ORYX**, a GUI based knowledge engineering tool for MANDARAX.

Figure 8 gives a graphical representation of the framework. The forward chaining engine (JESS [4]) and backward chaining engine (MANDARAX [6]) together represent the inference engine: the rules and the base facts represent the knowledge base. The user gives the system the facts it needs and her/his queries through the GUI and receives as output conclusion(s) and the explanation of how the system has come to the conclusion(s). The Native XML database acts as a rule store. Rules which are in RULEML [10] format are extracted from the native XML database and translated into the
format of either of the rule engine. Additionally the native XML database acts as a permanent place to store data that may be later used to create reports. Finally the native XML database can be used to store some base facts that are used to initialize the system.

A brief explanation of each part of the architecture above:

**GUI and explanation facility:** The GUI and the explanation facility is the user's interface to the system proposed. The GUI is used by the user to query the expert system
and get the conclusion(s). The explanation facility is used to get information about how the system came to the conclusion(s). The user can use both the backward chaining and forward chaining mechanisms from the interface, though the expert system decides which inference engine to use based on the type of question that she/he asks. In other words, there are two paths that the user can take upfront. She/he either starts out stating a few initial conditions/facts and asking the system where that leads to. Or they could give a goal and ask the system how to get to the goal. Because we did not include natural language processing in the GUI, the information needed by the system is provided by entering the data into the GUI. The GUI also includes a decision explanation facility that explains to the user how it reached a particular conclusion. The user can use this information to confirm if the conclusion is logically correct.

The GUI follows the MVC architecture. The View or GUI is created at runtime based on a configuration file. This configuration file is changed based on the results produced by the inference engine. Thus making the inference engine (forward chaining or backward chaining) the Model, the Controller is a module that reads this configuration file and updates the View (GUI) and captures GUI events and passes them onto the inference engine in the form of facts or goals.

**JESS:** JESS [4] stands for Java Expert System Shell. An expert system shell is a combination of an inference engine and a place holder for the knowledge base. An expert system shell can be viewed as being a complete expert system with the knowledge base removed. Knowledge base represents rules and facts.
JESS is a forward chaining expert system shell as well as a scripting language. It implements a very efficient version of the RETE algorithm. It was developed by Ernest Friedman-Hill at Sandia National Laboratories. The JESS scripting language is the native language for the JESS expert system shell. It is easy to embed JESS in another java program.

MANDARAX: MANDARAX [6] is an open-source backward chaining expert system shell written by Jens Dietrich. Just like JESS, it provides an inference engine and a place holder for knowledge base. The native format for rules and facts is similar to prolog syntax. MANDARAX is also easily embeddable in another java application.

RULEML: The Rule Markup Language is a standard for encoding rules in an inference engine neutral manner. “The Rule Markup Initiative has taken steps towards defining a shared Rule Markup Language (RuleML), permitting both forward (top-down) and backward (bottom-up) rules in XML for deduction, rewriting, and further inferential-transformational task” [10]. The output of the knowledge engineering process is transformed into RULEML. The advantage of this process is that the same knowledge base can be used by both JESS and MANDARAX. RULEML is a variant of the eXtensible Markup Language (XML). XML is a markup language like HTML, but with the advantage of user-defined tags. XML is fast becoming the standard mode of information exchange on the Internet because of the endless customizations that are possible with it. XML is also a way of data storage with semantics attached. For example we could store an address book entry in XML in this way (Figure 9):
Just like contacts were represented above in XML, rules can be represented too. Figure 10 shows the specification of RuleML. (Source: www.ruleml.org/indesign.html).

Figure 9: Example XML data.

Figure 10: RULEML design (SOURCE: www.ruleml.org/indesign.html)
The reaction rules represent the forward chaining rules that are given to JESS (the forward chaining engine) and the transformation rules represent the backward chaining rules that are given to MANDARAX (the backward chaining engine). Derivation rules are a subset of the transformation rules which do not exactly follow the template of a backward chaining rule. Facts, queries, and integrity constraints all fall under derivation queries category because of this property. Facts can be seen as rules whose antecedent is always true because it is empty and whose consequent is the fact itself. The queries can be thought of as derivation rules whose consequent is empty. So they represent a test for the existence of their antecedent constructs like facts in the working memory. Integrity constraints represent a subset of the queries, whose main task as the name suggests is to report any inconsistencies in the working memory. They achieve this task by querying the working memory for conflicting facts. Please Appendix I see for an introduction to RULEML.

**Native XML Database:** A native XML database (NXD) is to XML data, as a Relational Database Management System (RDBMS) is to relational data. The smallest unit of storage in a native XML database is an XML document just like a table is for an RDBMS. A native XML database both stores and retrieves the XML data according to a logical format rather than as they are, without loosing the semantics of the document.

Because we wanted to encode the rules and the base facts in XML, we need an efficient way to store XML data. A native XML database is more advantageous to use here than an RDBMS. One of the advantages of using a NXD is that no extra processing is needed to convert XML data into relational tables. The second advantage is that it is much easier to get output as XML from an NXD than an RDBMS if the input is an XML file, because
all we have to do is to follow the logical format and extract the data. Our idea is to store the rules among other things in the XML database and extract them as needed by the application, which helps in easier management of rules as all the rules are updated in one procedure module of the framework. After an update the transformation mechanism is executed again to update the rules for each of the engines.

An open source native XML database called dbXML [3] was used. The native XML database (NXD) was used in the project to store particularly two things:

- Rules
- Base-facts

The rules are to be encoded into RULEML format before storing into the NXD. The facts are encoded in XML format using the JESSGUI tool. The following is part of the application, but it is discussed here as an introduction to how the NXD is used. A very simple method was used for storing and retrieval of the above entities in the NXD. When the application starts off, the base-facts and rules are pulled from the database and manipulated upon as required. When the application is being closed, though the rules may not be updated, the base-facts may be updated. XML:DB API [12] was used for accessing the dbXML database. XML:DB API represents a standard for accessing and managing native XML databases. For example the following java code fragment shows the procedure to retrieve an XML file from the NXD.

```java
XMLResource resource = (XMLResource) col.getResource(filename);
String doc = (String) resource.getContent();
```
The String object doc now contains the complete XML tree. This tree can be manipulated later to get the required rules or facts. Please see Appendix III for an introduction to XML:DB API

**Rule translator:** The rule translator is a combination of two modules namely, a module to extract rules from the NXD and another to apply a set of XSL (XML Stylesheet Language) transformations to transform the rules in RULEML into native format for each of JESS and MANDARAX. Both these modules were developed as part of this work. Please see Appendix II for an explanation with an example.

**Rule inserter:** Knowledge engineering is one of the most important tasks in the construction of an expert system since the “intelligence” of the system depends on the rules that are constructed as an output of the process. Because of this importance, care must be taken to make sure that the rules base is consistent. The general procedure is that the expert system builder interviews human experts in the domain, reads literature related to the domain and creates a set of rules that represent the expertise in the native format of the expert system. Because the last step requires learning a new language which at times has esoteric syntax, this process is not generally easy.

Two programs JESSGUI and ORYX, were used for the purpose of knowledge engineering in this work. JESSGUI [5] is a tool that can be used by novice users as well as advanced users of JESS to add rules to the JESS engine. Another feature of JESSGUI is that one can write rules in XML format, which can later be added to the NXD. ORYX [7] is for MANDARAX as what JESSGUI is for JESS. ORYX supports RuleML, so the rules created using ORYX can be directly inserted into the native XML database.
3.3 System interaction

This subchapter describes how components of the system interact with one another. There are broadly three interactions:

• Interaction between the knowledge engineer, the knowledge engineering tools (JESSGUI and MANDARAX), and the Native XML database.

• Interactions between the NXD, the rule transformer module and the inference engines.

• Interactions between the user, the GUI and explanation facility and the inference engines.

Of these the first interaction takes place either when the expert system is being built or when EPA changes a rule that the expert system encodes. The second interaction takes place whenever the user starts the expert system. The third is the interaction between the user and the expert system. The following figures show these in detail.

Figure 11 depicts the first of the interactions. The knowledge engineer uses the JESSGUI and ORYX tools to encode rules and base-facts in XML. These are then inserted into the native XML database.

Figure 12 represents the interactions between the inference engines, the rule transformer module and the NXD. The rules and base-facts in XML are extracted from the NXD by the transformer module, a set of XML transformations are applied to them and the resulting rules and facts are placed in the inference engines.

The third interaction is shown in Figure 13. The user queries the inference engines and gets the answers and explanation using the GUI.
Figure 11: Interactions between knowledge engineer, JESSGUI and ORYX, and NXD

Figure 12: Interactions between NXD, Rule Transformer module and the Inference engines

Figure 13: Interactions between the User, GUI and the Inference engines
3.4 Total Coliform Rule

This section describes an example domain of expertise specifically the Total Coliform Rule as applied to small water systems, followed by the knowledge engineering process for the same. The first part gives an introduction to the Total Coliform Rule (TCR) [11] as published by the EPA. The final part describes the process of developing a knowledge base which can be used by small water systems to maintain compliance to the Total Coliform Rule (TCR).

**Purpose of TCR:** The TCR is aimed at reducing the presence of fecal pathogens or coliforms in drinking water. Fecal pathogens include total coliforms and other fecal coliforms including E.Coli. Samples of drinking water are collected by the small water system operator and sent to a lab to be analyzed for the presence of fecal pathogens. The testing lab informs the small water system operator of the results. The small water system operator now has to follow the stipulations laid out by the EPA to decide on the course of actions to take. This is the point where the application of the framework developed in this work comes in. It can help the small water systems manage data and inform the operator of any additional steps that need to be taken. Thus the system supports the managerial capabilities of small water system as encouraged by the EPA.

The rule has the following stipulations for small water systems [1]:

a) A minimum number of routine samples need to be taken per month. The exact number depends on the size of the population served (see Table 1).

b) If any sample tests positive for total coliforms, repeat samples must be collected within 24 hours of the operator knowing the results. The number of
repeat samples per positive sample also depends upon the population size served (see Table 1).

c) Any total coliform positive routine sample results in increased monthly samples next month. For small water systems, this number is 5.

d) The stipulation concerning repeat samples is that one of the samples is to be taken within 5 service locations before the affected service location (the service location which tested positive for Total Coliform), another at the affected service location and one more sample within 5 service locations after the affected service location. For water systems serving populations between 25 and 1000, the fourth sample can be taken anywhere.

e) A Monthly Maximum Contaminant Level (MCL) violation is triggered if any or all of the conditions below hold:

- There are multiple total coliform positive routine samples but no fecal coliform routine or repeat samples.
- A routine sample tests positive for total coliform and negative for fecal coliform and one or more repeat samples test positive for total coliform but negative for fecal coliform.

f) An Acute MCL violation is triggered if any or all of the conditions below hold:

- A routine sample tests positive for total coliform and fecal coliform and at least one of the repeat samples associated with this routine sample tests positive for total coliform.
- A routine sample tests positive for total coliform but negative for fecal coliform and at least one associated repeat sample tests positive for both total coliform and fecal coliform.

Figure 14: Total coliform rule flowchart
The above pseudo-flowchart (Figure 14) summarizes the rules given before about the MCL violations. The reader may realize that a sample may be fecal coliform positive only if that sample is already total coliform positive. The case that a sample is fecal coliform positive without it being total coliform positive does not arise.

<table>
<thead>
<tr>
<th>POPULATION SERVED</th>
<th>ROUTINE SAMPLES PER MONTH</th>
<th>REPEAT SAMPLES PER POSITIVE ROUTINE SAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>25-100</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1001-2500</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2501-3300</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Required numbers of routine and repeat samples to be taken based on the population size.

**Representation of TCR in the expert system:** The process of knowledge engineering for the example domain (TCR) is explained here. Once the required information is gleaned from experts and literature among other sources, the TCR is implemented for the system using the two knowledge engineering tools mentioned earlier (JESSGUI and ORYX). Please note that the knowledge is shown below in JESS only because it is more concise than RULEML. In the actual application the knowledge below is represented in RULEML. Also the rules here are separate from another set of rules which are used to control the GUI. The rules here represent the expertise in the domain of TCR while the rules that control the GUI represent a bridge between the inference engine and the GUI. The Total Coliform Rule is captured by the expert system in two separate modules. The first module (Figure 15) defines the information in Table 1 as follows in JESS’s clp language:

Please note that min_pop=Minimum Population served, max_pop=Maximum Population served, spm=samples per month, rspm=repeat samples per positive routine sample.
The facts above can be correlated to Table 1. “MAIN” above stands for the default module.

The second module deals with capturing rules that specify when a violation is detected.

The two monthly MCL violation varieties were encoded as follows in JESS’s clp language as (Figure 16 and Figure 17):

```lisp
(defrule monthlyViolation1
  (tc (num ?number))
  (rtc (num ?number)(subnum ?sub))
  (not(fc (num ?any))) (not(rfc (num ?atall)))
  (GETSTATUS) =>
  (assert (monthlyviolation)) (assert (increasedmonthlyviolation))
  (printout t “A message to tell the user which sample(s) caused a violation”))
```

Figure 16: TCR Monthly violation rule 1

tc represents total coliform test, fc represents fecal coliform test, the num represents the routine sample number, ?number represents a variable binding in JESS, sub represents the number of the repeat sample that tested positive for total coliform. The any and atall represent sample numbers which are not the same as the sample number which tested positive for total coliform. Variable binding means binding a variable name to a value. When a single variable is used multiple times in the left-hand side (LHS/antecedent) of a rule, it means that each time the value of the variable is the same...
as the other. For example in the above rule to test if a routine sample and a repeat sample are having the same sample number we use the same variable twice.

The “assert” construct is used to put a fact into the system while “printout” just prints to the standard output. The “not” construct is used when an absence of a fact is to be tested.

This second monthly violation here can be correlated to the second variety of monthly MCL violation described earlier.

```
(defrule countTCPs1 ?z<- (control_fact) (test (> ?TCPcount* 1)) =>
  (assert (multiple_TCPs)) (retract ?z))

(defrule monthlyViolation2
  (multiple_TCPs)
  (not(fc (num ?any)))
  (not(rfc (num ?atall)))
  (GETSTATUS) =>
  (assert (monthly_violation)) (assert (increased_monthly_violation))
  (printout t “A message to tell the user which sample(s) caused a violation”))
```

**Figure 17: TCR Monthly violation rule 2**

One of the two cases for Acute MCL violation is encoded in JESS as follows (Figure 18):

```
(defrule acuteViolation1
  (tc (num ?number))
  (fc (num ?number))
  (rtc (num ?number))
  (GETSTATUS) =>
  (assert (acute_violation))
  (assert (increased_monthly_samples))
  (printout t “A message to tell the user which sample(s) caused a violation”))
```

**Figure 18: TCR Acute violation rule 1**
3.5 TCR expert system GUI

This section is an introduction to the GUI details of the expert system. The expert system presently can answer questions related to the Total Coliform Rule. The general procedure is:

(a) The user selects which EPA rule to work with (Figure 19).

(b) The system, based on the selection and information if it exists from prior sessions, generates a GUI form. The system may fill some of the fields in the GUI form if they were already filled by the user in previous sessions.

(c) After filling the required fields the user begins querying the system.

\[
\text{Figure 19: Rule Selector sample screen. User selects one of the rules shown}
\]

Once the user selects a rule, the form screen for that rule is generated and is shown to the user. The GUI has the ability to save state, which means it can remember what was entered in the previous sessions. Suppose the user fills in some of the fields and closes the GUI, it can remember the values entered and when the user restarts the application it can refill the already entered data. A sample screen for the Total Coliform Rule is shown in Figure 20. As seen from the sample screen, the data which the user enters to satisfy the rule requirements is separated by month. The GUI remembers state by maintaining what
is called a template for each month. A template is a data structure with place-holders for each of the fields in the GUI. When the GUI is closing it collects all such templates for all the months and saves them onto the disk in the form of an XML file. This XML file is then stored in the NXD. When the application restarts this file is then extracted from the NXD and the GUI fields for each month are initialized with the values in the template of that month. The templates XML file can also be used for other purposes like printing reports etc.

The fields in the routine sample screen are broadly of two types: fields which are purely for reporting requirements like the date sample was tested for coliforms, the location at which the sample was taken, and the date the results of coliform tests were known. Fields of the other type are used by the expert system to come to a conclusion. These include the total coliform result field and the fecal coliform result field.

A total coliform positive sample requires a set of additional repeat sample be taken. The repeat samples have a separate dialog screen. Figure 21 shows an example of a repeat sample dialog screen.
Date sample was tested
Date results were known.

![Repeat Sample Number: 1](image)

Figure 21: Repeat sample screen that appears when the user specifies that a routine sample is total coliform positive.

The visibility of this dialog screen is controlled with the help of a set of rules and facts, an attribute of the MVC architecture defined earlier. The rules and the facts are the means through which the inference engine controls the view or the GUI. Whenever the user specifies using the GUI that a positive total coliform routine result is detected, the GUI asserts a fact into the inference engine. The inference engine sees the presence of this fact as a cue that it needs to ask the user to enter results of the repeat samples by using some rules. These rules check for the presence of the fact and make the repeat dialog box visible. Each routine fecal coliform result puts a fact specifying the routine sample number into the working memory. Similarly each repeat total coliform positive or fecal coliform positive routine puts a fact into the working memory specifying the routine sample number and the repeat sample number. The following are examples of what these facts would look like (in JESS language):

\[(tc \text{ (num 3)}) \text{ and } (fc \text{ (num 3)}), (rtc \text{ (num 3) (subnum 2)}) \text{ and } (rfc \text{ (num 3) (subnum 2)})\].
The tc fact above represents that the third routine sample is total coliform positive. Similarly fc represents that the third routine sample is fecal coliform positive. rtc fact above represents that the second repeat sample of the third routine sample is total coliform positive. rfc can be similarly interpreted by the reader.

Once the user has entered all the information she has she can ask the expert system to draw a conclusion based on the information entered so far. The user presses the get status button to do this. The status is shown in another dialog which also contains an explanation for the conclusion that the expert system drew. Figure 22 shows an example screen of the result dialog.

![conclusion/explanation dialog](image)

Figure 22: An example result dialog screen.
Chapter 4: Conclusions

Rule based expert systems have been around since mid-1960s and to date there are a multitude of expert systems for myriad fields. A single rule based system which can offer expertise on many domains is very hard to build. It is no accident that the well-constructed expert systems are domain specific. The domain of expertise in this project is EPA guidelines for small water systems. An expert system framework was constructed which helps a small water system operator make informed decisions regarding compliance with various EPA rules that may apply to her/him. An expert system prototype which can give expert answers on the Total Coliform Rule of the EPA was successfully built and developed on the framework. This framework can be further enhanced by:

- Supporting rules to help a small water system operator to select sampling areas.
- Implementing a liaison between the forward chaining engine and backward chaining engine so that they can work in synergy to solve a problem. This can be a message based system in which each of the engines communicates with the other using messages.
- Implementing other EPA rules for small water systems can be implemented on the framework.
- Field testing the prototype with small water system operators to get feedback on how to improve the GUI and the functionality of the system.
Appendix I

A small introduction to show how rules are encoded in RULEML.

The rule being encoded is:

IF  
Customer X is a platinum member  
THEN  
Discount for X is 12.5%

The RULEML representation is:

```xml
<IMP>

<_HEAD>

<ASSERT>

<ATOM>

  <_OPR><REL>Discount</REL></_OPR>
  <VAR>X</VAR>
  <IND>12.5%</IND>

</ATOM>

</ASSERT>

<_HEAD>
```

This part encodes the part of the rule that says "Discount for X is 12.5%."  
VAR stands for variable.  
IND tag specifies constants.  
The `<_OPR><REL>` tags specify the predicate being applied.

The assert tags result in the fact embedded with in to be inserted into the working memory.

The head is the one that comes after the THEN part in the rule given above.
For both the forward as well as the backward chaining rules, the rule format does not change. The way in which the rule is interpreted by the transformation mechanism changes.

For a forward chaining rule the BODY part of the rule above is taken to be the condition to check and the part in the HEAD to be the result if the IF part is true. The forward chaining engine looks for facts that support the pattern “(X, platinum member)” and substitute the value of X it finds in the result to get a discount fact. Where as for a backward chaining rule, it is interpreted in this way, the head part is said to be true if the body part is true. So in this case the backward chaining engine begins by checking what are the values that when substituted in X would satisfy the proposition (X, platinum member). Once it finds the one that satisfies the proposition, the engine proceeds to state that X gets a discount of 12.5%.
Appendix II

A sample rule given in RuleML format is transformed into `.clp` format of JESS. MANDARAX can use RULEML encoded rules directly, so no further transformations are needed.

XML transformations, an introduction:

XML transformations are a set of rules that are applied on a XML document to transform it into a different document. That is, each rule in the transformations has some information as to how to interpret a particular entity in the XML document and the action to take when one such entity is seen. For example the following transformation rule tells the action to take when the word “retract” is seen:

```
<xsl:template match="retract">

  <xsl:choose>
    <xsl:when test="./var">
      <xsl:text>(retract ?</xsl:text>
      <xsl:value-of select="var" />
      <xsl:text>)</xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-templates select="./fact/atom" />
      <xsl:text>)</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

This part of the rule says that when there is an element with the name “var” then this part of the rule is run. So for the example:

```
<retract>
  <var>
    X
  </var>
</retract>
```

The transformation would result in the text(retract X) being outputted.

This part says that apply the templates fact/atom just as we have just applied the template retract.
A complete transformation example

To transform the rule

```
<IMP LABEL="RULE 1">
  <_HEAD>
  <ASSERT>
    <FACT>
      <ATOM>
        <_OPR><REL>DISCOUNT</REL></_OPR>
        <VAR>X</VAR>
        <IND>12.5%</IND>
      </ATOM>
    </FACT>
  </ASSERT>
  </_HEAD>
  <_BODY>
    <AND>
      <FACT>
        <ATOM>
          <_OPR><REL>ISA</REL></_OPR>
          <VAR>X</VAR>
          <IND>PLATINUM_MEMBER</IND>
        </ATOM>
      </FACT>
    </AND>
  </_BODY>
</IMP>
```

applying the transformation rules given in Appendix IV (and copied from http://www.ruleml.org/jess/RuleMLTransform.xsl), we get the following output:

```
(defrule rule1
  (declare (salience))
  (isa ?x platinum_member)
  =>
  (assert (discount ?x 12.5%))
)
```

walking through the transformation:

The template for imp is matched. The action that takes place is

a) The attribute label is taken and kept in a variable called RuleLabel.
b) The text "(defrule " is outputted.

c) The value in RuleLabel is outputted. So at the end of this step the following have been outputted "(defrule rule1 ".

d) The template next says to apply the template "_body".

e) That template is applied.

f) The body template has the following line: <xsl:apply-templates />. This line has the effect that all the templates are tried on the present XML node and each of the children in document order (the order in which the children appear in the XML document.)

g) So the first template applied is "fact". Apply <xsl:apply-templates select="atom" /> which is present in fact template (the first "if" statement).

h) Applying "atom" template, then apply the template for "_opr" as given in the rule.

i) The first if statement is selected because it takes three steps up to find a node whose child (ren) has/have the nodes fact/atom in that order. So the text "(" and "isa" is outputted. The output at this stage is "(defrule rule1 (isa ". Next template var is called.

j) The var template results in "?X" being added to the output; so the output is : (defrule rule1 (isa ?X. next ind template is called and the output changes to (defrule rule1 (isa ?X platinum_member)

k) Next the transformer comes back to _body template and outputs a "=>". And then returns to imp.

l) Next imp calls "_head".
m) The head template calls the assert template which then outputs "(assert " and calls atom template. The fact template calls _opr template. The opr template outputs "(discount ". And calls var template.
n) The var template outputs ?X next ind is called and the output is appended with 12.5%.
o) That completes the _head part and it returns to imp. At this point the transformation stops.
Appendix III

XMLDB: API

A small introduction to XML:DB api.

Inserting an XML document into a native XML database:

A new XML file can be inserted into the NXD by first converting the XML into a stream in java String format.

```java
XMLResource resource = (XMLResource) collection.createResource("myXMLFileKey", XMLResource.RESOURCE_TYPE);
resource.setContent("<XMLFILE>an xml file stream as String</XMLFILE>" restricts);
collection.storeResource(resource);
```

Retrieving an XML document from a native XML database:

An XML document already in the database can be retrieved by using the getContent method of XMLResource class as follows:

```java
XMLResource resource = (XMLResource) collection.getResource("myXMLFileKey");
String document = resource.getContent(resource);
```

Updating an XML document in the NXD:

An XML document already in the NXD can be updated by using a combination of getContent and setContent methods of the XMLResource class:

```java
XMLResource resource = (XMLResource) collection.getResource("myXMLFileKey");
String document = resource.getContent(resource);
//modify the document
resource.setContent (document);
collection.storeResource(resource)
```
Appendix IV

The RULEML to clp conversion XSLT file given at

http://www.ruleml.org/jess/RuleMLTransform.xsl

<?xml version="1.0" encoding="UTF-8" ?>
- <!--
  ____________________________
  ____________________________
  <!--
    RuleML stylesheet for 0.8
  -->
  <!--
    Said Tabet, The RuleML Initiative
  -->
  <!--
  ____________________________
  ____________________________
  -->
  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <xsl:output method="text" />
    <xsl:template match="rulebase">
      <xsl:variable name="KBLabel">
        <xsl:value-of select="@label" />
      </xsl:variable>
      <xsl:apply-templates />
    </xsl:template>
    <xsl:template match="imp">
      <xsl:variable name="RuleLabel">
        <xsl:value-of select="./@label" />
      </xsl:variable>
      <xsl:text>
        (defrule
          <xsl:value-of select="translate($RuleLabel,'',")" />
          :
          "add rule comment here."
        )
      </xsl:text>
    </xsl:template>
  </xsl:stylesheet>

49
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="conclusions">
    <xsl:apply-templates />
  </xsl:template>
  <xsl:template match="boundfact">
    <xsl:text>?</xsl:text>
    <xsl:value-of select="var" />
    <xsl:text><</xsl:text>
    <xsl:apply-templates select="fact" />
  </xsl:template>
  <xsl:template match="fact">
    <xsl:if test="./atom" />
    <xsl:text>- </xsl:text>
    <xsl:if test="./rulebase">
      <xsl:text>asserting a ground fact: </xsl:text>
      <xsl:value-of select="./atom" />
    </xsl:if>
  </xsl:template>
  <xsl:template match="assert">
    <xsl:text>(assert</xsl:text>
    <xsl:apply-templates select="./fact/atom" />
    <xsl:text>)</xsl:text>
  </xsl:template>
  <xsl:template match="retract">
    <xsl:choose>
      <xsl:when test="./var">
        <xsl:text>(retract ?</xsl:text>
        <xsl:value-of select="var" />
        <xsl:text>)</xsl:text>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>(retract</xsl:text>
        <xsl:apply-templates select="./fact/atom" />
        <xsl:text>)</xsl:text>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
References


[7]. ORYX, www.jbdietrich.com/standalone.html


[10]. The RULEML Initiative, www.ruleml.org


