FRAMEWORK FOR AN EXPERT SYSTEM GENERATOR

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Jacob Cernik

May, 2009

FRAMEWORK FOR AN EXPERT SYSTEM GENERATOR

Jacob Cernik

Thesis

Approved:	Accepted:
__________________________	__________________________
Advisor Dr. Chien-Chung Chan	Dean of the College Dr. Chand Midha
__________________________	__________________________
Committee Member Dr. Kathy J. Liszka	Dean of the Graduate School Dr. George R. Newkome
__________________________	__________________________
Committee Member Dr. Zhong-Hui Duan	Date
__________________________	
Department Chair Dr. Wolfgang Pelz	

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION AND MOTIVATION

With the growth of knowledge management in the business world, it has become crucial to codify knowledge into the work process. Codified knowledge is knowledge that has been obtained from a domain expert and transferred into electronic form. This transference of knowledge then helps businesses automate, streamline, and preserve knowledge into a central shareable source. Decisions and training specifics can then be derived from the knowledge using analysis tools or other domain experts. One way to implement this knowledge is through the use of rule-based expert systems. A rule based expert system is a program or hardware configuration that uses a set of preconditions and rules to come up with expert answers to a situation. These programs have already found a niche in the market place. Hospitals have used expert systems to help then diagnosis patients. Doctors can supply an expert system a set of symptoms and it will return back a diagnosis to aid in the final decision making process. These systems are not limited to the medical industry. These systems can be useful to companies who wish to automate various forms of validation, statistical analysis, and diagnostics within their company or organization.

However, creating and updating an expert system can be a slow and painful process. It involves understanding various syntax and semantics of knowledge sources and expert system language. This fact alone makes it hard to generate an expert system and even harder to teach a computer to build one. In a competitive atmosphere this is enough to turn potential users away, because of the time they would need to invest in building just one expert system.

The goal of this research is to teach a computer how to generate an expert system and remove the end user from having to worry about specific syntax, semantics, and in-depth coding involved in creating an expert system. By removing the end user from having to worry about coding and translating rules, a rule-based expert system can be generated and updated faster and without the help of a programmer. The ability to quickly and easily create or update an expert system would allow a business to codify and automate or assist new business decisions quickly and as close to real time as possible. Therefore, the main beneficiaries of such a system are people who want to create or constantly update an expert system with very little prior knowledge about expert systems

CHAPTER II

EXPERT SYSTEMS AND THEIR DRAWBACKS

2.1 What is A Rule Based Expert System

There are three parts to an expert system: a rule base, working memory, and an inference engine.

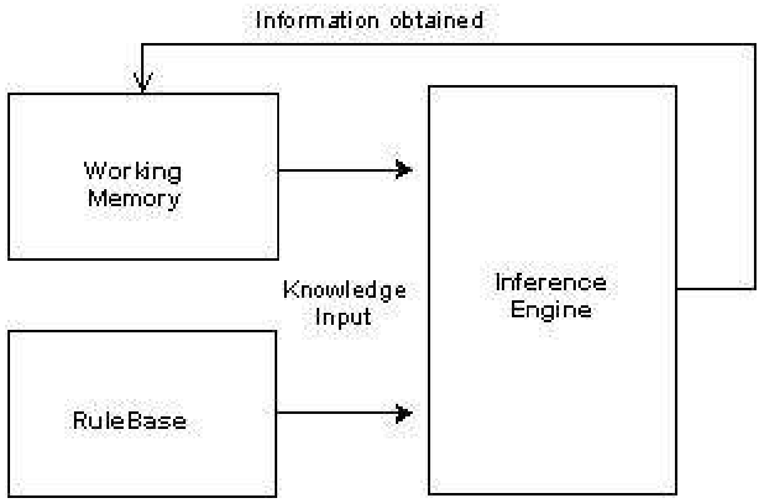


Figure 1 Expert System Architecture

The first part in an expert system is the rule base. This is comprised of two parts: a rule set and a dictionary. The rule set defines expert knowledge in the form of rules that can be thought of as a collection of cause and effect sentences. An example rule is "if your eyes are brown and your hair is brown then you are likely to have glasses". Furthermore, rules can also carry uncertainty information [8]. Uncertainty information contains information about how the rules were obtained and how accurate or plausible the rule assertions are. One example form of uncertainty uses rough set theory [4, 5]. While expert systems can differ on what form of uncertainty they use, this research will focus on using rough set theory but be implemented in a way to accommodate for future changes.

The dictionary defines the metadata for all of the knowledge variables, such as type, domain, and name. In the previous example the dictionary would contain at least three variables eye color, hair color, and whether or not they need glasses. Specifically the dictionary would define hair color is a discrete value and it can be white, blonde, brown, or black.

The second part of the expert system is the working memory. This is a collection of preconditions, which are a collection of variables defined in the dictionary. Using the previous example, a sample working memory could assert that student A's eyes are blue and their hair is blond.

The working memory and rule base are then used by an inference engine and which will attempt to assert new conditions. The process of processing rules can be different depending on what type of inference engine you are using. Two of the most commonly used algorithms are forward chaining and backwards chaining. Forward chaining attempts to take the preconditions and see if the rule base can assert anything from them. Backwards chaining works by supplying the post condition and then asserts the preconditions. If the preconditions match what is in working memory it can assert that something is true. Regardless of the inference technique if a rule is satisfied, new facts will be added to working memory. This process usually continues until no new assumptions can be made. At this point working memory can be parsed to find out what information can be derived from the preconditions. Using the preconditions "blonde hair" and "blue eyes" it could assert statements such as "is German" and/or "likely to be born in April".

The rule base is the cornerstone of the expert system. The rule base defines a domain experts knowledge into a codified form, and defines what attributes are important. These attributes are then defined in the dictionary and used in working memory. Due to its importance the goal of the expert system generator will be to create the rule base for a given expert system.

2.2 Creating An Expert System

To create an expert system a user must have: an expert source of knowledge, an inference engine, an understanding on how to build a rule base, and knowledge of how to enter and retrieves IO from the expert system. The hardest part is obtaining the knowledge to create the rule base. These knowledge sources can come from various places, such as domain expert, data mining, and other legacy devices. Examples of such knowledge sources can be seen in Appendix A and B.

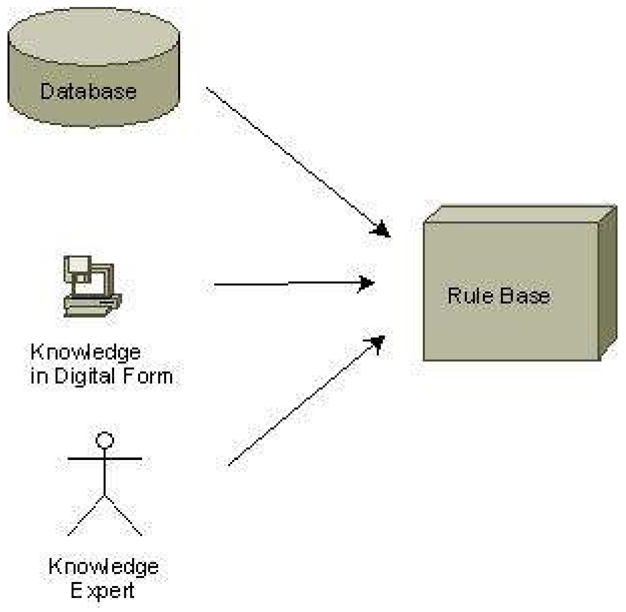


Figure 2 Creating a Rule Base

To currently create an expert system a programmer must take the knowledge source and translate it into rule form. While this may sound easy, it involves the programmer having a partial understanding about the knowledge that is being codified and the expert system language you are coding in. After the knowledge has been transferred to a rule base, the user must supply input into the expert system, in the form of the working memory. This input can come from a GUI, console, or script depending on the type of application. Once this is complete the user can run the expert system and translate the answer from working memory.

2.3 Drawbacks To Expert Systems

While Expert systems in general are useful in knowledge codification, they do have several drawbacks. The first and main drawback to an expert system is that expert system programmers need to have an understanding on how to create the expert system and domain knowledge. To create an expert system the user must understand how to: mine rules from a dataset, transform those rules into an expert system, and run those rules. This alone is a major deterrent and time-consuming endeavor. Most companies don’t want to spend the time and money teaching their employees how to code, understand, and maintain expert systems.

The second drawback is that end users are sometimes overwhelmed by expert systems. Even after it has been created they are unsure on how the system comes up with its decisions or maintains them as knowledge changes. This is only compounded by the fact that expert systems are hard to create and develop [7, 9]. However if expert systems were easier to implement, end users could spend more time understanding the mechanics of how the system works and less about its syntax and grammar.

The final problem is that even if you decide to use an expert system, the process of turning expert knowledge into an expert system lacks standardization. First of all, the user can mine knowledge from various sources, such as learning strategies and interviewing. After the knowledge has been obtained it must be converted into the expert system of the user choice such as Clips or Jess. Furthermore, each of these choices can affect the reliability, speed, and reusability of an expert system.

2.4 Solution: An Expert System Generator

These drawbacks are often enough to keep a user from using or creating an expert system, because of its complexity and the initial startup cost associated with training employees to use them. The solution to this problem is to abstract the end user from as much of the complexity of expert system programming language and knowledge transformation as possible. The problem can be solved by creating an expert system generator.

An expert system generator would be responsible for taking in a knowledge source and creating a ready to use expert system with very little input or knowledge from the creator. However, the main problem with creating an expert system generator is the lack of a unified starting point. The knowledge used to create the expert system can come in many forms: learning strategies, databases, personal knowledge, xml files, and many more. Each of these forms has a different syntax and semantics for its rules and a user will have to translate those rules to an expert system rule base. This is a problem for the computer because unlike humans, it is harder for a computer to learn and adapt to each new form. The solution to this problem is to create a standard to be used as an input to the expert system generator. While this may add an additional step to the process it will also help create a standard in all forms of rule-based knowledge within a company.

The second problem to address is how to create an expert system generator. An expert system generator must be able to parse the standard file and store all of the information in memory and perform a transformation on that data to a specific expert system.

CHAPTER III

CREATING AN EXPERT SYSTEM GENERATOR

While simple expert system generators exist, they are tailored strictly from knowledge source A to expert system B. This means that there could be a countless number of generators all differing in implementations [3, 6]. When creating a generic expert system generator the goals are to standardize all of the implementations, to eliminate repeated work and to centralize many different generators into one. There are two steps to creating this expert system generator: creating a standardized input file and generating an expert system from that standard.

3.1 PMML

Predictive Model Markup Language, or PMML for short, was found to be the best candidate for the standard input file. PMML is an xml extension written by the Data Mining Group, and was created to be a standard between various data mining models. PMML was originally created in the late

1990's. Since then it has evolved and grown to cover various types of data mining models in version 3.1. Since PMML's conception, it has found itself a useful asset in data mining and manipulation.

PMML is a prime candidate for input to an expert system generator for two reasons: PMML was designed to be a standard in data mining models and it is easily parsed. Since PMML was designed to be used for different models, its design is flexible enough to be used for a rule based expert system or even a tree based expert system.

A PMML file consists of many different tags and attributes. A PMML document is comprised of two parts: a DataDicationary and the RuleSet. The first section in a PMML document is the Dictionary, which holds all of the information about the attributes that are used in the rule set.

```
- <DataDictionary numberOfFields="2">
  - <DataField name="Size" optype="categorical" dataType="String">
      <Value value="small" />
      <Value value="medium" />
      <Value value="big" />
    </DataField>
  - <DataField name="Attitude" optype="categorical" dataType="String">
      <Value value="positive" />
      <Value value="negative" />
    </DataField>
  </DataDictionary>
```

Figure 3 Sample PMML Dictionary

The dictionary is composed of 3 main tags: DataDictionary, DataField, and Value. A DataDictionary is the opening tag for the dictionary and symbolizes the start of the dictionary. The DataDictionary tag has one attribute that specifies the number of variables described in the dictionary. In Figure 3 the dictionary contains two attributes: size and color. The DataField tag defines the name, type, and domain of the attribute. The first attribute is called color and its domain is categorical, with its values represented as strings. The final tag is the Value tag. The value tag defines the possible values that the parent tag takes on. In this example the possible values of “size” are “small”, “medium”, and “large”.

The next portion of a PMML document is the portion of the document describing how the rules were obtained and how they should be interpreted.

```
<RuleSetModel modelName="ESCore Generated">
<MiningSchema>
  <MiningField name="Size" usageType="active" />
   <MiningField name="Attitude" usageType="predicated" />
</MiningSchema>
```

Figure 4 Sample PMML Rule Set Model

The RuleSetModel tag tells how the rules will be displayed as a rule set. Since PMML covers various mining models there are other forms that rules could be displayed in, such as trees. The mining schema defines what attributes will be used from the dictionary and which is the decision attribute. Each Attribute will be represented as a

MiningField tag. The decision attribute will also be displayed as having a usage type of "predicated".

```
<RuleSet>
  <RuleSelectionMethod criterion="firstHit" />
        <SimpleRule id="0" value="positive" weight="3" confidence="1">
          - <CompoundPredicate booleanOperator="and">
                <SimplePredicate field="Size" operator="equal"
                   value="big" />
             </CompoundPredicate>
          </SimpleRule>
</RuleSet>
</RuleSetModel>
```

Figure 5 Sample PMML Rule Set

The RuleSet tag delimits the beginning of the rule section. The RuleSelectionMethod defines how the rules are to be selected. In this case we have declared that rules are fired by first hit. The SimpleRule tag specifies the rules unique id, the value that the decision value takes on, and the rule's uncertainty. The next tag is the CompoundPredicate tag. Every child of this tag is connected by the booleanOperator, which is usually "and" or "or". Next, each value in the rule is specified using SimplePredicate. Using the and/or connectives complex rules can be made, such as "if you have studied and your GPA is 4.0 then you will get an A on the final". The SimplePredicate specifies a rule's preconditions. For this instance, Rule 0 is defined to be: the attitude is positive where the Size is equal to big. Finally all of the tags can be closed off to finish off the PMML file. A full PMML example can be seen in Appendix C.

The second reason PMML was chosen was its parse ability. Since the PMML is an extension of XML, many libraries have been developed to aid in parsing XML syntax. This will have two benefits: accuracy and repeatability. The most important is accuracy. When retrieving the data from the PMML, it will be easier to parse out the data using language-supplied libraries. This will help eliminate errors while parsing a messy regular expression. Secondly, since most languages include a library to parse XML files, this approach to an expert system generator can be duplicated in another programming language.

While PMML offers the best solution as an input, it lacks the ability to represent all forms of uncertainty. The current PMML language only allows for certainty factor theory as its form of uncertainty. To be an accurate input file for rough set theory, several attributes will have to be added to the rule tag: certainty, coverage, strength, probability, and support. With a standard rule form, all that is left is to define a transformation that transforms an original rule source into PMML [1, 2].

3.2 Creating A PMML Generator

To create a PMML generator one needs to know the syntax and semantics of the rules and metadata/dictionary information. Once the syntax of the original rule set has been identified, one can parse the rule and metadata information, and then translate it into a form understandable to a computer.

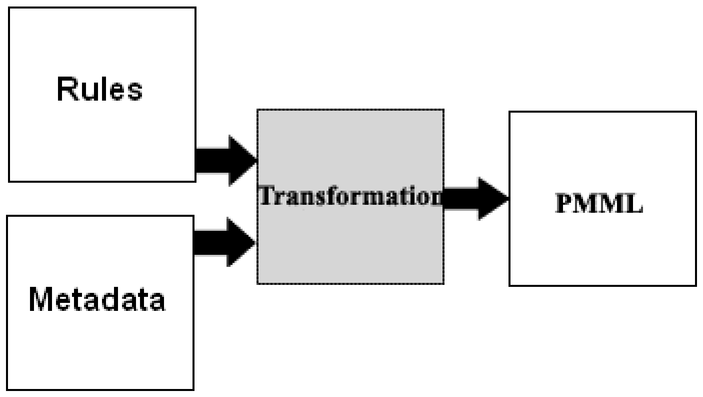


Figure 6 Defining a PMML Transformation

This can be accomplished by storing the dictionary and rule base as objects in an object oriented language. Once the information is stored in a computer as objects, it can be translated into PMML for permanent storage.

3.3 Expert System Generator

With a well-defined input it is now possible to create an expert system generator. The computer can parse through the PMML information and store it back into its object form, and then translate the objects into an expert system.

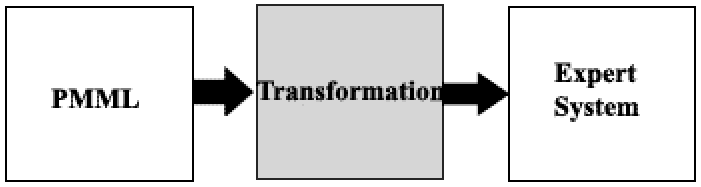


Figure 7 Defining an Expert System Transformation

3.4 Bringing It All Together

Based on these assumptions two things are needed to create an expert system: transformations and objects. The framework will need two transformations: one to transform knowledge sources to PMML and another to convert the PMML file to an expert system. There can be many types of these two transformations depending on your input and output. For example there can be a transformation that transforms rules in a database to PMML and another that transforms rules from a learning strategy to PMML. While the user will have to create the original transformation, once it is defined, the end user is abstracted from it when defining new rules.

The second thing needed is a list of objects to hold the information between transformations and interact with PMML. The object list will be defined as all objects used as inputs and outputs to an expert system. Since PMML is the central pillar the expert system generator, the objects should also be able to interact with a PMML document.

CHAPTER IV

IMPLEMENTATION

4.1 Design Requirements

Expert systems are embedded into many types of applications. While most are simple windows applications, they can be web applications or embedded into other mechanical devices. To account for this the generator framework should be as portable as possible. The solution is to create an external library called ESCore. This library will provide the framework for creating an expert system generator: the objects to be used and the transformations. When an end user wants to create an expert system generator all that is needed is to include the ESCore library.

There are also different goals for an expert system generator. The first approach is to embed a generator into an existing application. In this case, users will create a generator from rule form A to expert system B. This is useful if an end user never changes rule form A or expert system B, but constantly needs to update their rule base. The other approach is to create a generic expert system generator where users can change the transformations at will. Both requirements can be met by creating the transformation external libraries to ESCore. While ESCore will provide the interface for these transformations, as libraries they can be easily swapped in and out using reflection.

4.2 ESCore Library

ESCore can be thought of in two different parts: transformations and objects. The object list was selected from all of the physical objects in an expert system. There are seven ESCore objects that are used to hold information between the transformations. The seven objects can be seen in Figure 8 as the first seven objects.

Object	**Description**
Rule	Object that represent one rule in a rule file or expert system, and its certainty.
RuleSet	An object that represents a collection of rules to be used in the expert system.
Fact	A knowledge fact.
FactList	A collection of facts.
Attribute	A representation of an attribute and its values.
Dictionary	A collection of Attributes to be used in the expert system.

Figure 8 ESCore Objects

Certainty	Certainty of a rule.
ExpertSystemGenerator	Defines the interface for the transformation between PMML and the expert system.
PmmlGenerator	Defines the interface for the transformation between a rule file and PMML.
InferenceEngine	Defines the structure for an inference engine.

Figure 8 ESCore Objects (Continued)

While these objects can probably be found in any expert system program what makes them unique is their interaction with PMML. All of the objects can be translated into or from their PMML forms. For example, passing in a PMML document can populate a RuleSet or Dictionary object. Similarly if an empty RuleSet or dictionary is created and filled later, the object will have functions defined to translate it to its PMML equivalent.

The second part of ESCore is its transformation interfaces. ESCore provides the interface for all transformations in an expert system generator. The first transformation is a conversion from a knowledge source to its PMML form. The PmmlGenerator class covers this type of transformation. To create a PMML generator all one has to do is extend the PmmlGenerator class and override the abstract methods. There are two methods that need to be overridden. The first one takes in metadata information and returns a dictionary. The second function takes in the dictionary and rule information and returns a rule base object. After these functions have been written the user can call the parent code to create the PMML file.

The second transformation is the expert system generator. This interface is defined in the ExpertSystemGenerator class. When creating a transformation that extrends this class the user must overide one method createExpertSystem. This method will take in a dictionary, RuleSet, and an expert system indentifier. The transformation will use this information to create the expert system, which can be identified by supplied indentifier.

To accomodate testing purposes, ESCore also includes another set of objects to run the expert system. These objects are the factlist, fact, and inference engine object. A user can extend the inferenceEngine object to encapsulate the workings of an inference engine of a specific expert system. The working memory can then be supplied as a factlist.

CHAPTER V

DEMO: A RULE BASED EXPERT SYSTEM GENERATOR

There are two applications associated with the expert system generator: the actual generator and the expert system viewer to test the results. The expert system generator will generate the expert system, and standardize the rule base into PMML. The viewer will allow you to enter facts into working memory and run the expert system.

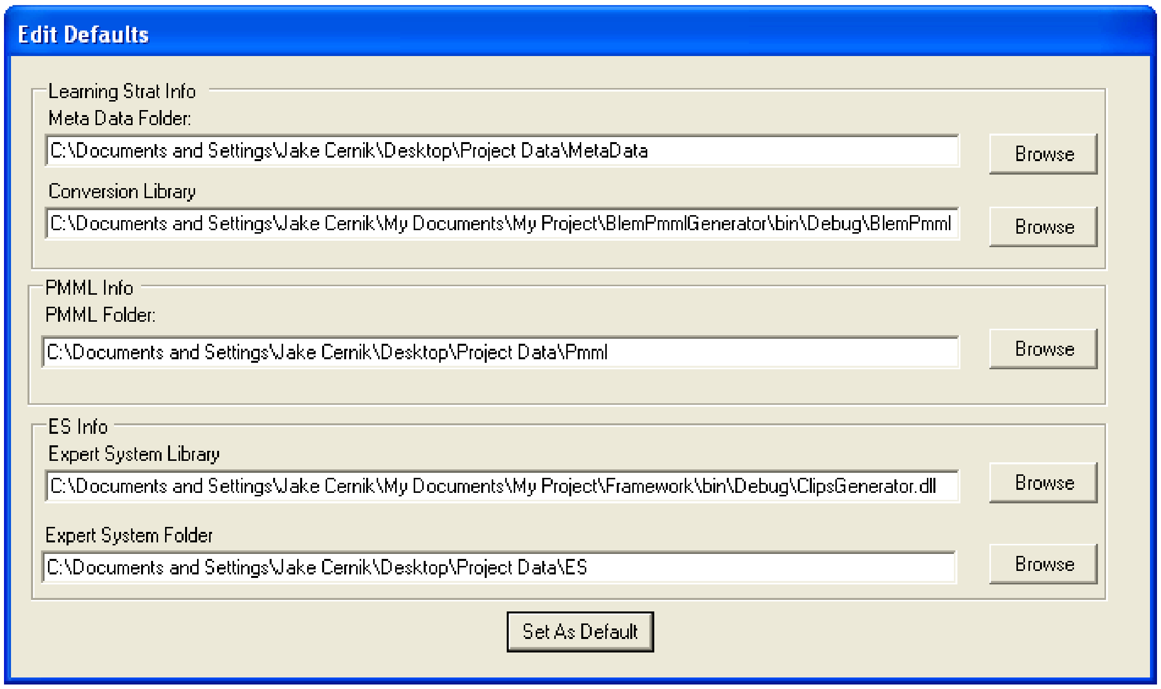
5.1 Creating Transformations To Be Used

The expert system generator currently has two transformations: a transformation between a BLEM2 learning strategy file to PMML and another that converts a PMML file to a Clips rule based expert system. The CLIPS expert system is a prime candidate for the first transformation. The clips inference engine forward chaining expert system that is both transformations, are stored in their own dll

Files, separate from the ESCore library. These transformations can be found in Appendix D and E.

5.2 Using The Expert System Generator

When launching the expert system generator, the first step is to set up the default information for the generator. The defaults specify the default location where the rule base and the PMML file. The defaults page also specifies what transformations to use.


Edit Defaults

Learning Strat Info
Meta Data Folder: C:\Documents and Settings\Jake Cernik\Desktop\Project Data\MetaData
Conversion Library: C:\Documents and Settings\Jake Cernik\My Documents\My Project\BlemPmmlGenerator\bin\Debug\BlemPmml

PMML Info
PMML Folder: C:\Documents and Settings\Jake Cernik\Desktop\Project Data\Pmml

ES Info
Expert System Library: C:\Documents and Settings\Jake Cernik\My Documents\My Project\Framework\bin\Debug\ClipsGenerator.dll
Expert System Folder: C:\Documents and Settings\Jake Cernik\Desktop\Project Data\ES

Figure 9 Expert System Generator Edit Defaults Screen

After the defaults have been specified, an expert system can be generated. By specifying the metadata file, the rule file, and the name of the PMML document, a PMML file can be generated. Then the generator can load the PMML file and create an expert system from it with the given expert system identifier. The status window will display the status of each transformation.

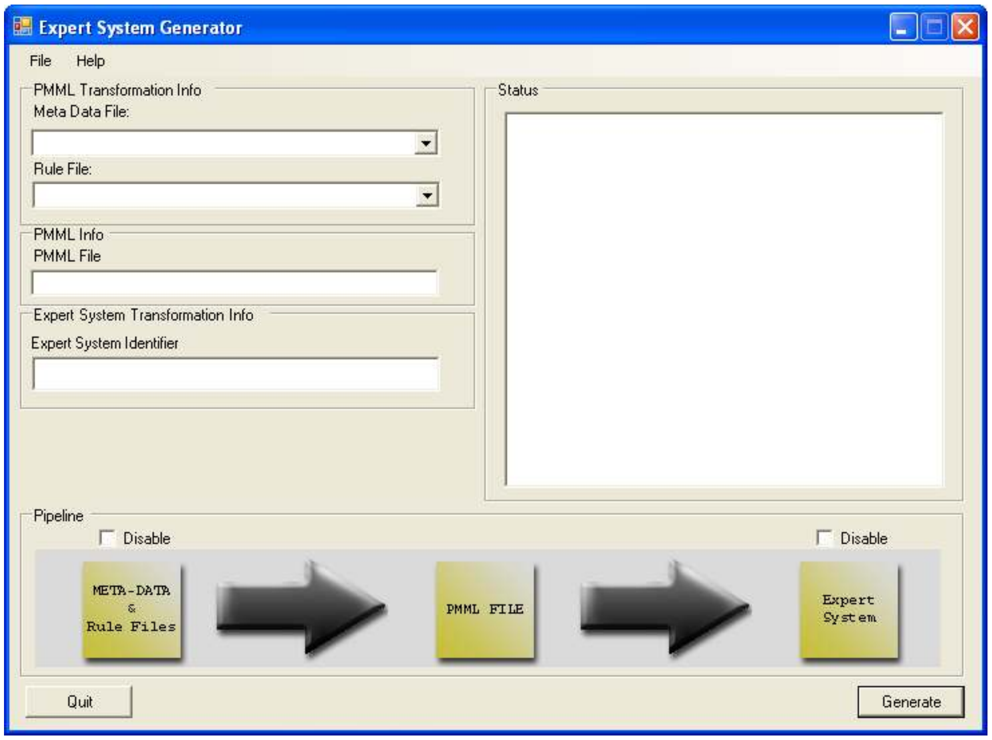


Figure 10 Expert System Generator Main Screen

Once the expert system is created it can be tested using the expert system viewer. The viewer allows users to add facts to the working memory by filling out the fields and pressing the add button. After the working memory has been populated, the expert system can be executed by clicking the run button.

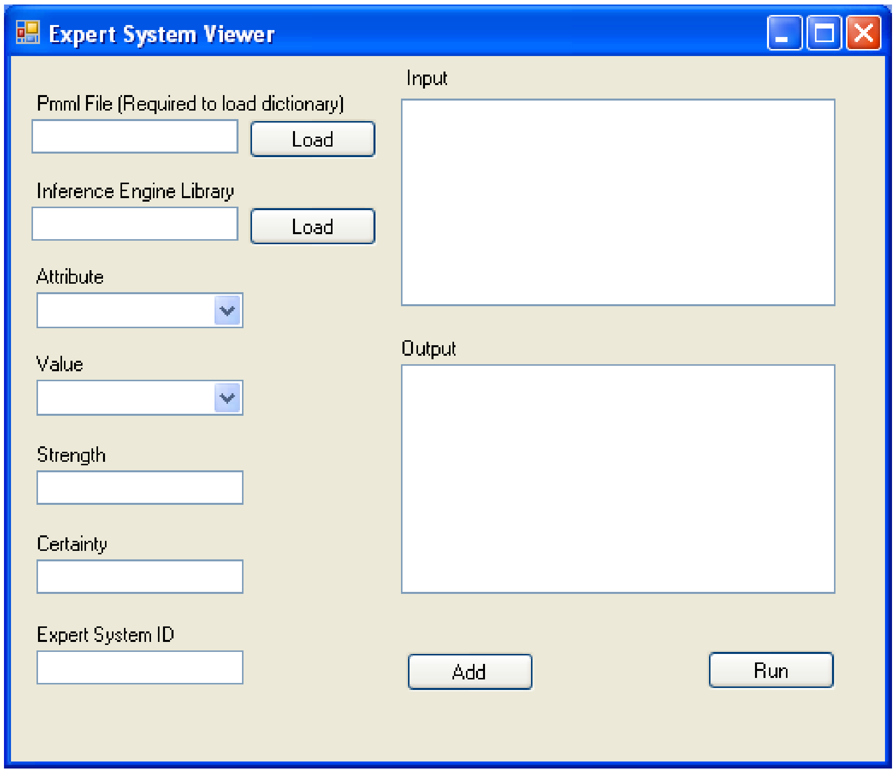


Figure 11 Expert System Viewer Main Screen

The results of the expert system will be displayed in the output window. The user will then be able to evaluate and process the output of their expert system.

CHAPTER VI

FUTURE WORK

While the basic principles are laid down in this research, there are many enhancements that can be done. The main work left to do is to create more transformation libraries. In its current state, the general expert system generator can only transform BLEM2 files to a Clips rule based expert system. Future libraries will allow users to input other sources of knowledge and create other types of expert systems.

Secondly, the library only accommodates rough set theory as the main form of uncertainty. While the uncertainty object can easily be swapped out and changed with a new form. It is not done in this research. However this would require multiple versions of ESCore. This would also means there should be an XML validation for each version of PMML. This would prohibit a transformation that uses one form of uncertainty from loading in another form.

When a transformation picks up a PMML file, it would need to verify that the file meets the standard for that specific transformation.

Another enhancement is to improve how the first transformation is selected. In the current implementation, a user must specify the dll transformation they wish to use. A better answer is to create a chain of transformations. In this scenario the user would only need to supply the input files, then the generator would select which transformation to apply. By creating a validation method in each of the libraries, a library can check to see if it can understand the data. If the transformation can validate the data it will do the transformation. Otherwise, it will see if the next transformation is valid.

The final enhancement is to extend the expert system generator to cover other forms of expert systems. PMML also allows for expert knowledge to be stored in tree form. Using the provided framework, new methods and objects can be created to account for these changes.

CHAPTER VII

CONCLUSIONS

The expert system generator is a valuable asset for anyone who wishes to create or update an expert system generator. An end user now has the ability to quickly create a new expert system using the current generator. The work also allows users to embed the generator framework into their own applications to create their own custom generator. Furthermore, once a set of transformations has been defined the end user should be abstracted form having to worry about the actual coding and updating process. This will open up a wider opportunity for expert systems to be used in the business world by spending less time creating the expert system and more time utilizing them.

However at, this time, the number of transformations provided is limited. Anyone who wishes to create a new transformation must take on the burden of writing the transformations. While this may be unappealing at first, the idea of stream lining the creation of expert system should help offset the transformation creation process. Once the transformations are created, they can be reused and shared so that others can benefit from them.

BIBLIOGRAPHY

[1] Data Mining Group , PMML Version 3.0 .<http://www.dmg.org/pmml-v3-0.html> .

[2] Stefan Raspl, An Overview of PMML Version 3.0

[3] Saima Rahman. “RuleBased Generator Using JESS” May,2002.

[4]Pawlak Z. “Rough sets: basic notion”, Int. J. of Computer and Information Science 11.

[5] Chien-Chung Chan, “Rough Sets Theory and Its Applications” Department of Mathematics and Computer Science University of Akron.

[6] Chien-Chung Chan, Zhicheng Su. ”From Data to Knowledge: an Integrated Rule-Based Data Mining System”, Department of Computer Science University of Akron.

[7] John McCarthy. “SOME EXPERT SYSTEM NEED COMMON SENSE” Computer Science Department Stanford University Stanford, CA 94305.

[8] BRUCE G. BUCHANAN, RICHARD 0. DUDA. “Principles of Rule-Based Expert Systems” Department of Computer Science Stauford University Stanford, CA.

[9] DANIEL G. BOBROW, SANJAY MITTAL, and MARK J. STEFIK “EXPERT SYSTEMS: PERILS AUD PROMISE”, Communications of the ACM September 1986 Volume 29 Number 9.

[10] Gary Riley “What is CLIPS” <http://www.ghg.net/clips/WhatIsCLIPS.html> October 22, 2007

APPENDICES

APPENDIX A

WEKA EXAMPLE

Dictionary	Rules
@relation golf @attribute outlook { sunny, overcast, rain} @attribute temperature real [0.0,100] @attribute humidity real @attribute windy { true, false} @attribute class { Play, 'Dont Play' } @data % 14 instances follow sunny, 85, 85, false, 'Dont Play' sunny, 80, 90, true, 'Dont Play' overcast, 83, 78, false, Play rain, 70, 96, false, Play rain, 68, 80, false, Play rain, 65, 70, true, 'Dont Play' overcast, 64, 65, true, Play sunny, 72, 95, false, 'Dont Play' sunny, 69, 70, false, Play rain, 75, 80, false, Play sunny, 75, 70, true, Play overcast, 72, 90, true, Play overcast, 81, 75, false, Play rain, 71, 80, true, 'Dont Play'	% Rule 1 - Length 1 'class'('Play') :- ('outlook'('overcast')). % Rule 2 - Length 2 'class'('Play') :- ('humidity'(X_2), X_2 =< 75), ('outlook'('sunny')). % Rule 3 - Length 2 'class'('Dont Play') :- ('humidity'(X_4), X_4 > 75), ('outlook'('sunny')). % Rule 4 - Length 2 'class'('Dont Play') :- ('outlook'('rain')), ('windy'('true')). % Rule 5 - Length 2 'class'('Play') :- ('outlook'('rain')), ('windy'('false')).

APPENDIX B

BLEM EXAMPLE

```
RULE FLIE
big,?,?,?,positive,3,1.0000,0.2308,0.4286
?,red,?,?,positive,1,1.0000,0.0769,0.1429
small,yellow,soft,?,positive,2,1.0000,0.1538,0.2857
?,blue,?,?,negative,2,1.0000,0.1538,0.3333
?,green,?,?,negative,1,1.0000,0.0769,0.1667
?,?,moderate,?,negative,1,1.0000,0.0769,0.1667
?,?,?,metal,negative,1,1.0000,0.0769,0.1667
?,?,hard,wood,positive,1,0.5000,0.0769,0.1429
?,?,hard,wood,negative,1,0.5000,0.0769,0.1667

META DATA FILE
5
5
Size c 3
small medium big
Color c 4
yellow green red blue
Feel c 3
soft moderate hard
Material c 3
plastic metal wood
Attitude c 2
positive negative
```

APPENDIX C

PMML EXAMPLE

```
- <PMML version="3.1" xmlns="http://www.dmg.org/PMML-3_1"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Header copyright="Example.com" />
  - <DataDictionary numberOfFields="5">
    - <DataField name="Size" optype="categorical" dataType="String">
        <Value value="small" />
        <Value value="medium" />
        <Value value="big" />
      </DataField>
    - <DataField name="Color" optype="categorical" dataType="String">
        <Value value="yellow" />
        <Value value="green" />
        <Value value="red" />
        <Value value="blue" />
      </DataField>
    - <DataField name="Feel" optype="categorical" dataType="String">
        <Value value="soft" />
        <Value value="moderate" />
        <Value value="hard" />
      </DataField>
    - <DataField name="Material" optype="categorical" dataType="String">
        <Value value="plastic" />
        <Value value="metal" />
        <Value value="wood" />
      </DataField>
    - <DataField name="Attitude" optype="categorical" dataType="String">
        <Value value="positive" />
        <Value value="negative" />
      </DataField>
    </DataDictionary>
  <RuleSetModel modelName="ESCore Generated" functionName="Classifciation"
        algorithmName="RuleSet">
    - <MiningSchema>
        <MiningField name="Size" usageType="active" />
        <MiningField name="Color" usageType="active" />
        <MiningField name="Feel" usageType="active" />
        <MiningField name="Material" usageType="active" />
        <MiningField name="Attitude" usageType="predicated" />
      </MiningSchema>
    -
    <RuleSet>
        <RuleSelectionMethod criterion="firstHit" />
      - <SimpleRule id="0" value="positive" weight="3" confidence="1">
        - <CompoundPredicate booleanOperator="and">
```

```
          <SimplePredicate field="Size" operator="equal" value="big" />
        </CompoundPredicate>
      </SimpleRule>
    - <SimpleRule id="1" value="positive" weight="1" confidence="1">
      - <CompoundPredicate booleanOperator="and">
          <SimplePredicate field="Color" operator="equal" value="red" />
        </CompoundPredicate>
      </SimpleRule>
    - <SimpleRule id="2" value="positive" weight="2" confidence="1">
      - <CompoundPredicate booleanOperator="and">
          <SimplePredicate field="Size" operator="equal" value="small" />
          <SimplePredicate field="Color" operator="equal" value="yellow"
              />
          <SimplePredicate field="Feel" operator="equal" value="soft" />
        </CompoundPredicate>
      </SimpleRule>
    - <SimpleRule id="3" value="negative" weight="2" confidence="1">
      - <CompoundPredicate booleanOperator="and">
          <SimplePredicate field="Color" operator="equal" value="blue" />
        </CompoundPredicate>
      </SimpleRule>
    - <SimpleRule id="4" value="negative" weight="1" confidence="1">
      - <CompoundPredicate booleanOperator="and">
          <SimplePredicate field="Color" operator="equal" value="green" />
        </CompoundPredicate>
      </SimpleRule>
    - <SimpleRule id="5" value="negative" weight="1" confidence="1">
      - <CompoundPredicate booleanOperator="and">
          <SimplePredicate field="Feel" operator="equal" value="moderate"
              />
        </CompoundPredicate>
      </SimpleRule>
    - <SimpleRule id="6" value="negative" weight="1" confidence="1">
      - <CompoundPredicate booleanOperator="and">
          <SimplePredicate field="Material" operator="equal"
              value="metal" />
        </CompoundPredicate>
      </SimpleRule>
    - <SimpleRule id="7" value="positive" weight="1" confidence="0.5">
      - <CompoundPredicate booleanOperator="and">
          <SimplePredicate field="Feel" operator="equal" value="hard" />
          <SimplePredicate field="Material" operator="equal" value="wood"
              />
        </CompoundPredicate>
      </SimpleRule>
    </RuleSet>
  </RuleSetModel>
</PMML>
```

APPENDIX D

BLEM2 TO PMML TRANSFORMATION

```
using System;
using System.Collections.Generic;
using System.Text;
using ESCore;
using System.IO;
namespace BlemPmmlGenerator
{
   class GenerateFromBlem : PmmlGenerator
   {
      //Call the base constructor
     public GenerateFromBlem(String RuleFile, String MetaDataFile, String PmmlFileName)
          : base(RuleFile, MetaDataFile, PmmlFileName)
      { }

      //Create a RuleSet object by reading in the rule and metadata file
      public override RuleSet createRuleSet(String ruleFile, Dictionary blemDictionary)
      {
         //Open the rule file
         StreamReader ruleReader = new StreamReader(ruleFile);
         String line = ruleReader.ReadLine();
         RuleSet myRuleSet = new RuleSet("BLEM Generated RuleSet");

         char[] seps ={ ',' };
         int ruleNumber = 0;
         String[] aRuleForm;
         String[] fullForm;
         Rule aRule;
         Certainty ruleCert;
         Boolean wildcardExists = false;

         //Make sure wildcard isn't in dictionary as a possible value
         for (int j = 0; j < blemDictionary.getSize(); j++)
         {
            if (blemDictionary.getAttributeAt(j).containsValue("?"))
               wildcardExists = true;
         }

         //Read in the Rules
         while (line != null)
         {
            //Read and Add to ruleset
            fullForm = line.Split(seps);
            aRuleForm = line.Split(seps, blemDictionary.getSize())
```

```
        //Replace ? wildcard with Custom wildcard if ? is not a value
        if (!wildcardExists)
        {
            for (int i = 0; i < aRuleForm.Length; i++)
            {
                aRuleForm[i] = aRuleForm[i].Replace("?", Rule.wildcard);
            }
        }



        //Filter out just the rule form
        String a = aRuleForm[blemDictionary.getSize() - 1];
        aRuleForm[blemDictionary.getSize() - 1] = a.Split(seps)[0];

        Console.WriteLine();
        Console.WriteLine(fullForm[blemDictionary.getSize() + 0]);
        Console.WriteLine(fullForm[blemDictionary.getSize() + 1]);
        Console.WriteLine(fullForm[blemDictionary.getSize() + 2]);
        Console.WriteLine(fullForm[blemDictionary.getSize() + 3]);
        Console.WriteLine(fullForm[blemDictionary.getSize() + 4]);
        Console.WriteLine();

        //Create the rules Uncertainty
        ruleCert = new Certainty(
            Convert.ToDouble(fullForm[blemDictionary.getSize() - 1]),
            Convert.ToDouble(fullForm[blemDictionary.getSize() + 0]),
            Convert.ToDouble(fullForm[blemDictionary.getSize() + 1]),
            Convert.ToDouble(fullForm[blemDictionary.getSize() + 2]),
            Convert.ToDouble(fullForm[blemDictionary.getSize() + 3])
            );

        //Create a Rule
        aRule = new Rule(ruleNumber.ToString(), aRuleForm, blemDictionary.getDecisionPosition(), blemDictionary, ruleCert);
        myRuleSet.add(aRule);

        ruleNumber++;
        line = ruleReader.ReadLine();
    }
    return myRuleSet;
}

public override Dictionary createDictionary(String metaFile)
{
    StreamReader metaReader = new StreamReader(metaFile);
    String line;
    int dictionarySize;

    //Get the dictionary size
    dictionarySize = Int32.Parse(metaReader.ReadLine().Trim());
    Console.WriteLine("Size of Dictionary: " + dictionarySize);

    //Get the position of the key attribute
    int keyAttributePos = Int32.Parse(metaReader.ReadLine().Trim()) - 1;
    Dictionary blemDictionary = new Dictionary(dictionarySize, keyAttributePos);
    Console.WriteLine("Key Attribute: " + keyAttributePos);
```

```
            //Read in the rest of the Dictionary
            String attributeName = "";
            char[] seps = { ' ' };
            String[] values;
            ESCore.Attribute anAttrib;

            for (int i = 0; i < dictionarySize; i++)
            {
                //Get Attribute Name
                line = metaReader.ReadLine().Trim();
                line = line.Replace("  "," ");

                //If I see a blank line interpret as extra line
                if (line == "")
                    break;

                //Get attribute information
                values = line.Split(seps);
                attributeName = values[0].Trim();

                //Get Attribute Values
                line = metaReader.ReadLine();
                line = line.Trim();
                line = line.Replace("  ", " ");
                values = line.Split(seps, Int32.Parse(values[2].Trim()));


                //Create and add Attribute to Dictionary
                anAttrib = new ESCore.Attribute(attributeName, values);
                blemDictionary.add(anAttrib);
            }
            metaReader.Close();
            return blemDictionary;
        }

    }
}
```

APPENDIX E

PMML TO CLIPS TRANSFORMATION

```
using System;
using System.Collections.Generic;
using System.Text;
using ESCore;
using System.IO;
namespace ClipsGenerator
{
  public class ClipsGenerator:ExpertSystemGenerator
  {
    public ClipsGenerator(RuleSet aRuleSet, Dictionary aDictionary, String ID)
      : base(aRuleSet, aDictionary, ID)
    { }

    public ClipsGenerator(String pmmlFile, String ID)
      : base(pmmlFile, ID)
    { }

    public override void createExpertSystem(RuleSet aRuleSet, Dictionary aDictionary, String ID)
    {
      //Create the clips file content
      String code = "";
      int predicatedPos = aDictionary.getDecisionPosition();

      //Define the Rough Fact
      code = "(deftemplate RoughFact \n \t(slot rule (type STRING)) (slot coverage) (slot strength)\n)\n\n";

      //Set fact Duplication to true
      code = code + ";(set-fact-duplication TRUE)\n\n";

      //Create the method for combing similar facts in CLIPS
      code = code + "(defmethod combine-RoughFacts((?c1 NUMBER)(?s1 NUMBER)(?c2 NUMBER)(?s2 NUMBER))\n";
      code = code + "\t(max (* ?c1 ?s1)(* ?c2 ?s2))\n)\n\n";

      //Combine Coverage
      code = code + "(defrule combine-Coverage \n";
      code = code + "\t (declare (auto-focus TRUE))\n";
      code = code + "\t ?RFact1<- (RoughFact (rule ?r)(coverage ?c1)(strength ?s1))\n";
      code = code + "\t ?RFact2<- (RoughFact (rule ?r)(coverage ?c2)(strength ?s2))\n";
      code = code + "\t (test (neq ?RFact1 ?RFact2))\n";
      code = code + "\t =>\n";
      code = code + "\t\t (retract ?RFact1)\n";
      code = code + "\t\t (modify ?RFact2 (coverage (combine-RoughFacts ?c1 ?s1 ?c2 ?s2 )))\n";
      code = code + ")\n\n";
```

```
            //Create the ruleset
            Rule aRule = null;
            String predicate;
            for (int i = 0; i < aRuleSet.getSize(); i++)
            {
               predicate = "";
               aRule = aRuleSet.getRule(i);

               code = code + "(defrule Rule_" + aRule.getName().Trim() + "\n";
               for (int j = 0; j < aRule.getNumAttributes(); j++)
               {
                  if (aRule.getValueAt(j).Trim() != Rule.wildcard && predicatedPos != j)
                     code = code + "\t (RoughFact (rule \"" + aRule.getNameAt(j) +
                        " is " + aRule.getValueAt(j) + "\")(coverage ?c)(strength ?s))\n ";
                  if (predicatedPos == j)
                   predicate = "\t\t (assert (RoughFact (rule \"" + aRule.getNameAt(j) + " is " +
                        aRule.getValueAt(j) + "\")(coverage ?c)(strength ?s)))\n";
               }
               code = code + "\t =>\n";
               code = code + predicate;
               code = code + ")\n\n";
            }

            //create a clips file
            FileStream file = new FileStream(ID, FileMode.OpenOrCreate, FileAccess.Write);
            StreamWriter sw = new StreamWriter(file);
            sw.Write(code);
            sw.Close();
            file.Close();
         }
      }
   }
```