# 5. RME-EP Models

RME-EP is a rule-based expert system shell, a variant of Rete rule engine. It combines expert systems and rule-based model evaluation with mathematical formulas and predictive models. RME-EP provides environment where expert rules are evaluated along with predictive models. There are three components for RME-EP. RME-EP engines operate with these three components;

- **RME-EP models** are prepared from CMSR. They include rules and predictive models. Models are prepared as files and fed into the engine. Models are of class ”cmsmodel.api.CmsApiRmeEp” and loaded into the engine with ”loadRmeEpModel(file)”.

- **Fact managers** manage facts on which rule inferences are performed. User applications are responsible in setting up and initializing facts. Fact managers implement APIs of the Java interface “cmsmodel.api.RmeEpFactManager”. To create a default fact manager, call ”createDefaultFactManager()”.

- **Event handlers** are important mechanism to detect various business events and process them in real time. Event handlers extend the Java class “cmsmodel.api.RmeEpEventHandler”. To create default event handler, call ”createDefaultEventHandler()”. Note that event handlers may be omitted, if not used.

RME-EP models are stored in files. There are two ways you can prepare RME-EP models. One is to use APIs provided either by rosellaweb.webservices. LocalRmeResources or by rosellaweb.managers.GlobalRmeResources. The following APIs can be used;

```
public static cmsmodel.api.CmsApiRmeEp getRmeEp(String name);
public static cmsmodel.api.CmsApiRmeEp getRmeEp(File file);
```

This will return a new RME-EP engine instance with the specified model loaded. It is noted that these APIs are available for web-based applications only. Otherwise, preparation can take a number of manual steps. The required sequences are as follows;

1. Create a RME-EP instance and initialize it. It is noted that there is a class “cmsmodel.api.CmsApiRmeEp”, inside the “rmeengine1.jar” archive file. To create an instance, simply code as follows;

   ```
   CmsApiRmeEp model = CmsApiRmeEp.getRmeEpInstance();
   ```

2. The second step is to load an RME-EP model into the engine using loadRmeEpModel();

   ```
   File model_file = new File(“path/mymodel.rte”);
   rte.loadRmeEpModel(model_file);
   ```

Once RME-EP models are loaded successfully without errors, applications may set up a fact manager and an event handler as follows;

```
RmeEpFactManager fm = model.createDefaultFactManager();
RmeEpEventHandler eh = model.createDefaultEventHandler();
eh.initializeResources();
eh.setEventSeparator("\n");
```

**Fact initialization and Rule evaluation**

Normally, rules are applied to a set of records (or transactions) repetitively. Although each record works independently, some global information may be kept separately through out entire run of records. To this end, facts (=variables) are defined as follows;

- Global facts/variables
- Local facts/variables

In addition, rules are classified into five categories;

- Global initialization rules (or GLOBAL-RESET rules).
- Local initialization rules (or LOCAL-RESET rules).
- (Regular local) rules.
- Local close rules (or LOCAL-CLOSE rules).
- Global close rules (or GLOBAL-CLOSE rules).

Typically, there are two ways how rules are evaluated: Record-driven and Event- driven.

### Record-driven approach

In this approach, records are read from external sources, such as files and database tables, and applied to rules in the following sequence;

1. Initialize both global and local variables to default initial values.
2. Activate all global initialization/reset rules and evaluate fully.
3. For each record (or transaction), perform the followings;
   - 3.1 Initialize all local variables with default initial values.
   - 3.2 Merge local input facts with values of the current record. Applications will replace local facts with the values of the current record.
   - 3.3 Activate all local initialization/reset rules and evaluate fully.
   - 3.4 Activate all (regular local) rules and evaluate fully.
   - 3.5 Activate all local close rules and evaluate fully.
   - 3.6 Output values of output variables.
4. Activate all global close rules and evaluate fully.
5. Output values of global facts.

Note that steps 1 and 2 are global preparation. Normally, it is done to initialize global information. Care should be taken not to make changes on local facts at this stage, since local facts can be changed during local preparation. Local preparation is performed as described in steps 3.1 through 3.3. The main rule evaluations are performed from step 3.4. Local close rules are then applied.

The following codes show the record-driven evaluation method. In this mode, a record or a transaction is read. Then evaluation is performed using the data of the record. The entire sequence is as follows. Blue parts are to be changed according to your requirements;

```
File modelfile = new File("ModelPath.rte");

CmsApiRmeEp model = null;
RmeEpFactManager fm = null;
RmeEpEventHandler eh = null;

try {
   // create model and setup fact manager & event handler
   model = CmsApiRmeEp.getRmeEpInstance();
   model.loadRmeEpModel(modelfile);
   fm = model.createDefaultFactManager();
   eh = model.createDefaultEventHandler();
   eh.initializeResources();
   eh.setEventSeparator("\n");

   /*
    * In Rosella BI server, the following will do
    * all above six lines.
    */
   //model = rosellaweb.webservices.LocalRmeResources.getRmeEp("YourModelPath.rte");

   // global initialization and evaluation;
   model.initialize();
   model.resetGlobally();
   if (model.activateRules(true, false, false, false, false)) {
      model.evaluateAllActivatedRules();
   }

   Vector vinput  = model.getFactEntryListInput();
   Vector voutput = model.getFactEntryListOutput();
   FactData fd;

   // for each record, perform the followings;
   while (true) {
      if (eof_condition) {
         break;
      }

      model.resetLocally();

      // ship input record data;
      for (int i = 0; i < vinput.size(); i++) {
         fd = (FactData)vinput.elementAt(i);
         fd.setValueInString("YourValue");
      }
```

```
      // evalueate;
      if (model.activateRules(false, true, false, false, false)) {
         model.evaluateAllActivatedRules(); // local-reset rules;
      }
      if (model.activateRules(false, false, true, false, false)) {
         model.evaluateAllActivatedRules(); // regular rules;
      }
      if (model.activateRules(false, false, false, true, false)) {
         model.evaluateAllActivatedRules(); // local-close rules;
      }

      // dump output variable data
      for (int i = 0; i < voutput.size(); i++) {
         fd = (FactData)voutput.elementAt(i);
         String str = fd.getValueString();
         Object obj = fd.getValueObject();
      }
   }

   // evaluate global close rules;
   if (model.activateRules(false, false, false, false, true)) {
      model.evaluateAllActivatedRules(); // global-close rules;
   }

   // dump global variables, if any;
   fd = model.getFactData("GlovalVariableName");
   String str = fd.getValueString();
   Object obj = fd.getValueObject();

} catch (Throwable t1) {
   t1.printStackTrace();
}

// release resources for garbage collections;
if (model!=null) {
   // the following frees model database connections
   // and then if BI server, return to model pool
   // or destroy itself.
   model.releaseResources();
   model = null;
}
```

## Event-driven approach

In this approach, events are fed to rules and applied to rules in the following sequence. An event is a change value of a variable.

1. Initialize both global and local variables to default initial values.
2. Activate all global initialization/reset rules and evaluate fully.
3. Initialize all local variables with default initial values.
4. Merge local variables with the current values. Applications will replace local variable with the current values.
5. Activate all local initialization/reset rules and evaluate fully.
6. Activate all (regular local) rules and evaluate fully.
7. For each event variable changing, perform the following;
   - 7.1 Set the changed variable value and activate all the rules affected by the change of the new value of the variable.
   - 7.2 Evaluate all activated rules fully.
   - 7.3 Output values of relevant variables, if needed.
8. Activate all local close rules and evaluate fully.
9. Activate all global close rules and evaluate fully.
10. Output values of relevant global variables, if needed.

Note that this performs global and local preparation at the beginning. Steps 1 and 2 are global preparation. Steps 3 through 5 are local preparation. Once both preparations are completed, all the regular local rules are activated fully based on global and local preparations already performed (step 6).

Once preparation is complete, the rule engine is ready to process events one by one as described in step 7. For example coding, see the next page.

The following codes show the event-driven evaluation method. In this method, facts are updated by external events with API calls to “setFactAndActivateRules()”. Blue parts are to be modified as your requirements.

```
File modelfile = new File("ModelPath.rte");

CmsApiRmeEp model = null;
RmeEpFactManager fm = null;
RmeEpEventHandler eh = null;

try {
   // create model and setup fact manager & event handler
   model = CmsApiRmeEp.getRmeEpInstance();
   model.loadRmeEpModel(modelfile);
   fm = model.createDefaultFactManager();
   eh = model.createDefaultEventHandler();
   eh.initializeResources();
   eh.setEventSeparator("\n");

   /*
    * In Rosella BI server, the following line will do
    * all above six lines.
    */
   // model =
rosellaweb.webservices.LocalRmeResources.getRmeEp("YourModelPath.rte");

   // global initialization and evaluation;
   model.initialize();
   model.resetGlobally();
   if (model.activateRules(true, false, false, false, false)) {
      model.evaluateAllActivatedRules();
   }

   Vector vinput  = model.getFactEntryListInput();
   Vector voutput = model.getFactEntryListOutput();
   FactData fd;

   model.resetLocally();

   // ship initial local/record data, if needed;
   for (int i = 0; i < vinput.size(); i++) {
      fd = (FactData)vinput.elementAt(i);
      fd.setValueInString("YourValue");
   }

   // evalueate;
   if (model.activateRules(false, true, false, false, false)) {
      model.evaluateAllActivatedRules(); // local-reset rules;
   }
   if (model.activateRules(false, false, true, false, false)) {
      model.evaluateAllActivatedRules(); // regular rules;
   }
```

```
   // for each event, perform the followings;
   while (true) {
      if (eof_condition) {
         break;
      }

      // determine which event variable changed
      int k = 1; // changed variable index;
      String st = "New data"; // changed data;
      fd = (FactData)vinput.elementAt(k);
      Object o = fd.convertToDataTypeObject(st);

      if (model.setFactAndActivateRules(fd, o)) {
         model.evaluateAllActivatedRules();
      }

      // dump output variable data
      for (int i = 0; i < voutput.size(); i++) {
         fd = (FactData)voutput.elementAt(i);
         String str = fd.getValueString();
         Object obj = fd.getValueObject();
      }
   }

   // evaluate local close rules, if any;
   if (model.activateRules(false, false, false, true, false)) {
      model.evaluateAllActivatedRules();
   }

   // evaluate global close rules, if any;
   if (model.activateRules(false, false, false, false, true)) {
      model.evaluateAllActivatedRules();
   }

   // dump global variables, if any;
   fd = model.getFactData("GlovalVariableName");
   String str = fd.getValueString();
   Object obj = fd.getValueObject();

} catch (Throwable t1) {
   t1.printStackTrace();
}

// release resources for garbage collections;
if (model!=null) {
   // the following frees model database connections
   // and then if BI server, return to model pool
   // or destroy itself.
   model.releaseResources();
   model = null;
}
```

**Thread and RME-EP API**

RME-EP instances may not be shared between threads. As RME-EP instances keep internal states and used in the subsequent operations, it does not synchronize method calls. It is recommended that threads use own RME-EP instances.

## 5.1 APIs for RME-EP engine

The “rmeengine1.jar” file contains a class “cmsmodel.api.CmsApiRmeEp”. This section describes APIs and related coding conventions. It is important to import the API classes into your Java application programs as follows;

```
import  cmsmodel.api.*;
```

Instantiation of the class can be performed as follows;

```
CmsApiRmeEp  rte = CmsApiRmeEp.getRmeEpInstance();
```

**Loading models**

RME-EP models consist of predictive model definitions and rules. RME-EP models are created from CMSR Rule-based Modeling panel. As RME-EP models contain all the necessary information, RME-EP models can be prepared with the following methods. Note that the files contain RME-EP models.

```
public void loadRmeEpModel(File file) throws Throwable;
public void loadRmeEpModel(String file_name) throws Throwable;
public void loadRmeEpModel(InputStream file_stream) throws Throwable;
```

**Fact managers**

RME-EP engines operate on a set of facts. Facts are managed by fact managers. Fact managers implement the Java interface “cmsmodel.api.RmeEpFactManager”. The following API can be used to create and set a default fact manager.

```
public RmeEpFactManager createDefaultFactManager();
```

**Event handlers**

Event handlers are important part of RME-EP. Rules can generate events. RME-EP engines call event handlers extending the “cmsmodel.api.RmeEpEventHandler” interface class. To create a default event handler, the following API is used;

```
public RmeEpEventHandler createDefaultEventHandler();
```

**Initializing Global/Local Facts**

To initialize global and local fact variables, use the following APIs respectively;

```
public void  resetGlobally();
public void  resetLocally();
```

## Evaluating Models

When model is created, it should be initialized with the following API;

```
public void  initialize();
```

Rules are activated with the following APIs. The first is to activate all rules of specified categories indicated by the API arguments. The second and third APIs are to used to set fact values and fire related rules automatically;

```
public boolean  activateRules(boolean grobalResetRules, boolean localResetRules,
                    boolean regularRules, boolean localCloseRules,
                    boolean globalCloseRules);

public boolean  setFactAndActivateRules(String key, Object value);
public boolean  setFactAndActivateRules(FactData key, Object value);
```

All activated rules are evaluated with the following API;

```
public void evaluatedAllActivatedRules() throws Throwable;
```

## Obtaining Input and Output Fact Variables

Input and output fact variable lists can be obtained with the following APIs. Note that elements of input/output variable list vectors are “cmsmodel.api.FactData”. This class is described in the next section.

```
public Vector getFactEntryListInput();
public Vector getFactEntryListOutput();
```

To get a variable entry, use the following API;

```
public FactData  getFactData(String name);
```

To get the entire variable list, use the following API;

```
public Vector getFactEntryList();
```

## 5.2 cmsmodel.api.FactData

The class definition “cmsmodel.api.FactData” provides all the data about each fact variable. The following APIs can be used to obtain data;

```
public String getName();
public int getDataType();
public Object getValueObject();
public String getValueString();
public Vector getInputValueList();
```

To set data, the following APIs can be used. Note that the first API returns true if there is an error. The second return true if the value is changed with the new value.

```
public boolean setValueInString(String value);
public boolean setObjectAndReturnChangeStatus(Object v);
```

To convert string value to an object of the data type, use the following API;

```
public Object convertToDataTypeObject(String value);
```

In addition, the following variables can be read. Please do not change the values directly. Or it will malfunction!

```
public boolean isInput = false; // true for input variable
public boolean isOutput = false; //true for output variable
public boolean isOptional = false; // optional output
public boolean mandatoryInput = false; // mandatory input
public boolean rangeCheck = false; // input range check
public String  rangeFrom = null; // range from
public String  rangeTo = null; // range to
public String  templateValue = null; // initial input
```

## 5.3 Event Handlers

Event handlers extend the class “cmsmodel.api.RmeEpEventHandler”. The core of the interface is the following methods;

```
public void initializeResources();
public void setEventSeparator(String separator);
```

# 6. Predictive Models

Predictive models are elements of RME-EP models. Predictive models supported include decision trees, neural networks and classifying rules. Generally, predictive models are used in RME-EP modeling as described in the previous chapter. In addition, it is also possible to write directly using predictive modeling APIs described in this chapter. This chapter describes how they can be implemented using predictive modeling APIs. Implementing CMSR predictive models is very simple. The required sequence is as follows;

1. Create a model instance. It is noted that there is a class "cmsmodel.api.CmsApiModel", inside the "starmodel1.jar" archive file. Details of the class are described in the next section. To create an instance, simply write as follows;

```
CmsApiModel model = new CmsApiModel();
```

2. Load a model with a descriptor. As described in the previous section, there are three ways to specify descriptiors. The followings show three different methods;

```
model.loadModel(MyNetwork.getBytes());
model.loadModel(new File(filename));
model.loadModel(socket.getInputStream());
```

3. Obtain input values and feed them into the model. This may be performed through repeated application of "setInputDataObject()" or setting the input object array which can be obtained with "getInputDataArray()". The followings show two different methods;

```
model.setInputDataObject(data0, 0);
model.setInputDataObject(data1, 1);
model.setInputDataObject(data2, 2);

Object[] INPUT = model.getInputDataArray();
INPUT[0] = data0;
INPUT[1] = data1;
INPUT[2] = data2;
```

4. Perform evaluation, that is, call "evaluate()";

```
model.evaluate(false); // false does not generate how and outcome message.
```

5. Get result using the following methods;

```
model.getPredictedIndex();
model.getPredictedLevel();
model.getHowString();
model.getOutcomeString();
```

## 6.1 APIs for Predictive Modeling

The "rmeengine1.jar" file contains a class "cmsmodel.api.CmsApiModel". This section describes APIs and related coding conventions. It is important to import the class into your Java application programs as follows;

```
import  cmsmodel.api.CmsApiModel;
```

Instantiation of the class can be performed with the following default constructor;

```
new CmsApiModel();
```

**Loading Model Information**

```
public void loadModel(byte[] data) throws Exception;
public void loadModel(java.io.File file) throws Exception;
public void loadModel(java.io.InputStream stream) throws Exception;
```

These methods load model information into an instantiation of CmsApiModel. There are three variations to cater different application environments. Note that these methods may throw exceptions.

**Input Field Information**

```
public int getInputFieldCount();
```

This method returns the number of input (or induction) fields.

```
public String getInputFieldName(int k);
```

This method returns the name of the *k*-th input field, where $k = 0, 1, 2\ldots$

```
public String[] getInputFieldValues(int k);
```

This method returns the values of the *k*-th input field. Note that this may not return all the values for non-neural models. It returns only values appearing decision trees or rules. However, neural networks return all the values appearing input data.

**Output Field Information**

public String getOutputFieldName(i);

This method returns the name of the output or target field.

public String[] getOutputFieldValues();

This method returns the values of the output field or target field.

public int getOutputValueCount();

This method returns the number of values of the output or target field.

public int getOutputFieldValueIndex(Object *v*);

This method returns the index number of the value *v* appearing in the output or target field. If *v* is not found amongst values of the output field, this return –1.

**Model Profile Information**

public String getModelProfile();

This returns model profile string. Model profile describes input and out fields and values.

**Input Parameter Information**

public Object[] getInputDataArray();

This returns the array of objects used internally by the model. The array size corresponds to the number of input fields. The array may used to set actual input data. Otherwise, the following method should be called for each input field repeatedly.

public void setInputFieldDataObject(Object v, int k);

This method sets-up the *k*-th input field with the object *v*.

## Evaluation and Result Retrieval

public void evaluate(boolean verbose);

This evaluates the model with input data set priori. Result may be retrieved with the following APIs. Note that verbose indicates whether to generate verbose "how" and "outcome" messages which can be retrieved with corresponding API. For batch evaluation, "false" is recommended.

public int getPredictedIndex();

This method returns the index number of of the predicted value. When error occurred during "evaluate()", this may return –1. For target that returns numerical values, this may return meaningless values, e.g., 0. For targets that predict probability to a specific value, this returns 0 for "false" and 1 for "true" respectively.

public double getPredictedLevel();

For linear target, this returns the actual predicted value. Otherwise, this returns probability of the prediction from 0,0 ~ 1.0.

public double getPredictedProb(String classname);

This returns predicted probability for the class name, ranging values from 0.0 to 1.0. Note that this is for decision trees only!

public String getHowString();

This returns "how" message. If "execute(false);" was called, this may return a null value.

public String getOutcomeString();

This returns "outcome" message. If "execute(false);" was called, this may return a null value.