Building PMML Models - Tutorial Outline

Introduction

This tutorial focuses on <product name> PMML models: how they are constructed, what features can be exploited to represent the problem that the each model is trying to solve, and what is necessary to test and deploy them.

Before diving into the particulars regarding the models, it is important to understand the environment in which the model will execute. A firm understanding of the execution context will aid the model author in factoring his/her model as either one model or possibly as a group of related models where one model depends upon the results of a prior model in the group.

Broadly, the purpose of the <product name> platform is to manage the receipt of event data from some specified source or possibly sources, construct messages from this input, and present these messages to the models in its execution working set that have expressed an interest in such messages. Each model interprets the incoming message, possibly juxtaposing/filtering it with ancillary content that has been prescribed for this model, including a history of prior message events and their results. If appropriate output from that interpretation is prepared and forwarded to consuming applications or even other modesls in the working set.

The <product name> is cluster based. In other words, work is partitioned into distributable units that other nodes in the cluster can execute and solve. This allows the system to scale linearly with the number of nodes participating in the cluster. Furthermore, each node in the cluster can simultaneously execute multiple models in parallel, assuming there are no dependencies between the models executing at that moment.

All of the models that are executing on a given <product name> cluster are known as the *working set*. As previously mentioned, the output message produced by a model can be consumed by other models. This means that other models that are in the working set can be fed the output of a previous model or even the same input plus the output message generated by that prior model. This is particularly useful mechanism to control model complexity,

effectively building up a group of models that work together to analyze the input message content.

As you will see later, nothing special needs to be done to make this happen except to design the input and output data messages properly. See the <product name> <appropriate manual that describes how messages and structured containers can be constructed for use by the models.

The engine uses metadata generated at model compile time to detect the input and output relationships of the related models automatically. This permits the engine to build a partial ordering of the models so that the models have all of their inputs available when they execute. Internally these relationships are represented as a directed acyclic graph (aka DAG). The DAG controls the order of execution. Where dependencies exist, execution of the dependent model is held in abeyance until the model that produces its input message has done so. Unrelated models are free to execute in parallel only constrained by engine resource policies (memory, thread pools and the like).

Before examining the features of <product name> PMML and how to specify and construct models with it, it is important to understand a bit about PMML itself. This is covered in the next section.

What is PMML?

LigaDATA models are expressed with a augmented version of the Predictive Model Markup Language (PMML) standard developed by the Data Mining Consortium. PMML is an XML based language for specifying a wide variety of data mining model types. With the latest release (4.2.1) fifteen different model types are defined. The industry consortium, Data Mining Group, maintains the standard specification at dmg.org. Leading corporations like IBM, SAS, MicroStrategy and SPSS support this effort and are principals in the consortium membership. Please visit dmg.org for more details.

How is LigaDATA PMML different from the DMG standard?

We would be remiss if it was not explained how LigaDATA PMML differs from the DMG

standard. There are both features described in the standard that are currently not supported by <product name> PMML as well as extensions made in the <product name> PMML in order to integrate the model execution into the <product name> platform. What follows, while not exhaustive, are some of the important distinctions. See the FAQ (Tamara... a web link perhaps) that is more complete and will be maintained as new features are added to either the standard or the <product name> PMML. If you feel there are other distinctions that need to be made viz a viz the PMML standard, please contact us. We only want an accurate portrayal of these differences. Briefly,

- LigaDATA is currently focused on the RuleSet model flavor. It is but one of the fifteen model types specified by the Data Mining Group in the PMML XML Schema Definition (XSD). The RuleSet, despite its narrow purpose, can nevertheless be applied to a wide variety of problems in different lines of business.

  Our initial thrust has been to make certain the models running in <product name> are perform reasonably and have the necessary features to in which to integrate the RuleSet models with the <product name> execution environment before horizons are expanded to the other models in the DMG standard. Suffice it to say there are plans to support other model types in the near future in a subsequent release.

- The type system used in LigaDATA PMML is vastly expanded in comparison to the current standard, not limited to the simple types (scalars, boolean, string and time related types). Complex types like messages, containers, arrays, and sets can be used; they can be used in combination if desired. That is, it is possible to have a structure with an array of or set of containers in it as a field.

- For LigaDATA PMML, the MiningSchema section of the model is used principally to define the predicted and supplementary data to be emitted by the model when a result is to be produced. This is in stark contrast to the DMG's approach that describes the Mining Schema this way, “The MiningSchema is the *Gate Keeper* for its model element. All data entering a model must pass through the MiningSchema.”

Instead the data that a model uses is described fully in the metadata manager that is used by the compiler to produce the executable version of the model from the PMML as well as the <product name> platform itself to feed messages to models in its working set.

- Multiple nested models, part of the DMG standard, are not supported at this time. This may be supported in one of the subsequent releases of the <product name>.

- Model verification is not supported. Instead of the techniques suggested in the Model Verfication description on the web site, it is highly desirable that a test suite for the <product name> engine be maintained that exercises logic and features of any model. The distinction here is that the test suite is not model centric. It is <product name> engine centric. Models are effectively just extensions that may be dynamically added to this engine.

- The Output Field specification defined in the current standard is not currently supported.

- The notation used to define external functions and tables in the DMG standard are not used in LigaDATA PMML. The functions and table types used are described instead in the <product name> metadata.

- Finally, MOST <product name> PMML models are NOT valid with respect to the reference XML Schema. The models accepted by the <product name> PMML compiler purposely use new features outside the specification. These new features are designed to describe meaningful computational entities that can be deployed on the <product name> platform.

The distinction to be made here is that IF the model submitted for validation can be expressed with the builtin dataType usage constraints expressed in the current DMG PMML XSD, the <product name> PMML Compiler can be said to produce conforming models. In other words, it can be argued that compiler is “producer conforming” (see http://www.dmg.org/v4-2-1/Conformance.html for the semantics here).

From the “consumer conforming” perspective, our current system will not accept a valid

PMML model produced by another system or vendor at this time. For example, we cannot directly use a Zementis exported model from System R … at least for the time being.

That said, it is time to examine more closely at how models are prepared for execution. This is found in the next section.

PMML Compilation

A <product name> PMML model is not interpreted by the <product name> platform to execute it. Instead it is compiled to a Scala source code representation which in turn is compiled by the Scala compiler to produce a jar file. As part of this compilation process key model information is organized to be cataloged into the <product name> metadata. In addition to the name of the compiled jar, the model name, model version are collected from the PMML Header element. The inputs that the model requires as well as the outputs that it generates are also assembled.

With this information in the metadata cache, the platform engine on a given cluster node knows the appropriate inputs for the model and knows what outputs it may produce that may be required by other down stream models in the working set. This model metadata is the principal input to discovering the model interdependencies in the working set of models. It is the basis of the production of a partial ordering of model execution dependencies (i.e., the DAG) that guarantees the necessary inputs are made available when the node engine calls upon a model with a new message.

It is important to note that for the PMML model compilation to be successful, all data types, messages , containers, and functions used in the model must have been previously added to the <product name> metadata. The metadata fully determines the version of the generic functions to be generated for example.

The compilation process depends on numerous extensions that have been made to the <product name> PMML. These extensions are considered next.

<Product Name> PMML Extensions

Messages

The <product name> engine presents what is known as a message (or messages?) to each model that has been designed to handle them. Messages can be represented by a *fixed* structure (represented as a Scala class instance at runtime) or as a *mapped* message (represented as a Scala Map[String, <some data type>] at runtime).

Fixed messages are used when all fields are available in the incoming data stream and can be presented to the model. The mapped message is used to represent sparse data, where only a few fields out of a domain of fields are available for presentation. All of these distinctions are described in full detail in <reference to the Metadata reference>.

Fields described in messages can be just about any type that can be expressed in a Scala program including the standard scalar types, collections, tuples, and other containers that have their own sets of fields. This allows the formation of fairly complex hierarchical messages.

It should be noted that the only distinction between a container and a message is one of convention. The only significant difference is that messages are cataloged globally in the metadata as top level container structures. The message is the principal vehicle for presenting data to the model.

Container instances instead are used as elements of a message or another structure or an element of a collection.

Messages are indicated (or presented) to a model as a `DataField` element in the PMML model's `DataDictionary` (Note: that the `optype` attribute required for `DataField` is left out for brevity in this example and many others in this tutorial):

```
<DataField name="msg" displayName="msg" dataType="Beneficiary"/>
<DataField name="gCtx" displayName="gCtx" dataType="EnvContext"/>
<DataField name="parameters" displayName="parameters" dataType="container">
       <Value value="gCtx" property="valid"/>
       <Value value="msg" property="valid"/>
</DataField>
```

In the example, the `msg` defines a `dataType` "`Beneficiary`" that describes a message type definition that was added to the Metadata Manager, the source of all type information used in

a <product name> PMML model. Notice too the field `gCtx`, an `EnvContext`, has also been defined. Both appear in the enumerated `DataField` named `parameters` that uses them.

This is a common pattern in all models. There will always be an `EnvContext` followed by one or more messages in the `parameters`. The `EnvContext` is an important element in all <product name> models, providing access to ancillary data. For example lookup tables, filter tables, foreign databases, and things we haven't even contemplated are/can be added as some source/sink of data through the `EnvContext`.

Hierarchical Data

As mentioned in the prior section, data can be hierarchical in nature. To access hierarchical data elements, the FieldRef field attribute uses a conventional '.' deferencing notation found in most programming languages. For example, consider the following DerivedField definition:

```
<DerivedField name="Age" dataType="integer" optype="continuous">
      <Apply function="AgeCalc">
            <FieldRef field="msg.Bene_Birth_Dt"/>
      </Apply>
</DerivedField>
```

The FieldRef field attribute shows how to access a field called Bene_Birth_Dt that is part of the message, `msg`, an instance of a Beneficiary. Note too the use of function AgeCalc. AgeCalc is a user defined function that like Beneficiary, has been previously described in the Metadata Manager. Details about this will be presented shortly.

It should also be said that there is no limit to the nesting that can be used when nesting structures except perhaps coping with the degree of incomprehensibility nesting is taken to the extreme.

Collection Types

As was mentioned in the introduction, most data structures supported in the Scala programming language can be declared in the <product name> metadata and used by the models. This includes much of the concrete classes found in the Scala Collection packages. Not all of them are currently defined in our metadata bootstrap, but it is possible to define them (and the functions that can access them) in the metadata if a given application requires

it. To see how to define these collection types, see <reference to the Metadata reference>.

Here are some examples of how arrays can be referenced and used within a model:

```
<DerivedField name="hl7InfoThisLastYear" dataType="ArrayOfHL7" >
      <Apply function="ToArray">
            <Apply function="ContainerFilter">
                  <FieldRef field="msg.HL7Messages"/>
                  <Constant dataType="fIdent">Between</Constant>
                  <Constant dataType="ident">Clm_Thru_Dt</Constant>
                  <FieldRef field="AYearAgo"/>
                  <FieldRef field="Today"/>
                  <Constant dataType="boolean">true</Constant>
            </Apply>
      </Apply>
</DerivedField>
```

In this example, a `DerivedField` named `hl7InfoThisLastYear` has type `ArrayOfHL7`. `ArrayOfHL7` is an array of `HL7` containers which has been defined in the metadata. Both the HL7 container type and the array of it are defined there. The function, `ContainerFilter`, evaluates an array of `HL7` (the field `msg.HL7Messages`) preserving only those `HL7` containers with field `Clm_Thru_Dt` found within the last year inclusive for the `hl7InfoThisLastYear` value.

For those of you extremely familiar with the PMML standard, you will notice a number of new constructions not possible in the current standard. In addition to the 'dot' notation to dereference structured fields, there are a number of new `dataType` values for the `Constant` elements in the `ContainerFilter` argument list, namely `fIdent` and `ident`. These are hints to the <product name> PMML Compiler as to how to interpret the `Constant's` value.

In this example `ContainerFilter` is what is known as an “Iterable” function. This is intended to suggest what it does. The first argument, `msg.HL7Messages,` is known as the `Iterable`, a collection of some sort that possesses the `Iterable` trait which defines the `filter` behavior for collections. This is a common action performed on collections in Scala.

The subsequent arguments all describe a function that is to be applied to each element in the collection. It's result when true, will retain the array element in the result. In the example, the function `Between` noted with a `Constant` with `dataType fIdent` describes the function name. The `Clm_Thru_Dt Constant` is the name of a field in the `Iterable` (hence its `dataType` being `ident`). The `Between` function's range fields and inclusive boolean are standard PMML

constructs.

Another example:

```
<DerivedField name="inPatientClaimCost" dataType="double">
      <Apply function="Sum">
            <Apply function="ContainerMap">
                  <FieldRef field="inPatientClaims"/>
                  <Constant dataType="fIdent">+</Constant>
                  <Constant dataType="ident">Clm_Pmt_Amt</Constant>
                  <Constant dataType="ident">Nch_Clm_Pd_Amt</Constant>
                  <Constant dataType="ident">Nch_Ddctbl_Amt</Constant>
                  <Constant dataType="ident">Nch_Coinsrnc_Lblty_Am</Constant>
                  <Constant dataType="ident">Nch_Ddctbl_Lblty_Am</Constant>
            </Apply>
      </Apply>
</DerivedField>
```

This is another `Iterable` function example, a thinly veiled way to use the Scala map function on the Interable, `inPatientClaims`. The member function is the PMML standard builtin function '`+`' that adds the values of the remaining `ContainerMap` arguments - all of which are fields in each `inPatientClaims` item. The map function transforms the item type of the input array to the type returned by member function. In this case the member function will return an array of doubles that in turn are supplied to the `Sum` function to produce a summation of all of these cost fields found in all elements of the `inPatientClaims` array.

Again, those familiar with Scala will recognize a map operation being performed on the array. The map function is used to transform some collection to an `Iterable` with another member type or types. For example consider the following:

```
<DerivedField name="conflictMedCdsSet" dataType="ImmutableSetOfString">
      <Apply function="ToSet">
            <Apply function="MakeStrings">
                  <Apply function="ContainerMap">
                        <FieldRef field="conflictMedCds"/>
                        <Constant dataType="ident">Code1</Constant>
                        <Constant dataType="ident">Code2</Constant>
                  </Apply>
                  <Constant dataType="string">,</Constant>
            </Apply>
      </Apply>
</DerivedField>
```

In this example an array of Container structures named `conflictMedCds` whose elements have fields called `Code1` and `Code2` are projected out of the array to form, in this case, an `Iterable` with a `Tuple` that has two elements as its members.

This `Iterable` is given to a function called `MakeStrings` that presumably takes the tuple members from each array element and creates string with these two strings separated by a comma. To assure that there are no duplicate strings present in the `ArrayOfString`, the `ToSet` function is used to eliminate the duplicates. The resulting type and `dataType` for the `DerivedField conflictMedCdsSet` is an `ImmutableSetOfString`. Immutable `Set` types once filled by the assignment expression cannot be changed, a good choice in this instance.

Before moving on, one more example that deals with Sets:

```
<DerivedField name="ContraIndicative" dataType="ArrayOfString">
      <Apply function="ToArray">
            <Apply function="Intersect">
                  <FieldRef field="MedicationIdPairStrings"/>
                  <FieldRef field="conflictMedCdsSet"/>
            </Apply>
      </Apply>
</DerivedField>
```

In this example, we see how the Set, `conflictMedCdsSet`, which was created in the previous example, is put to use in this `DerivedField`, `ContraIndicative`. The `conflictMedCdsSet` describes the valid set of medical conflict code pairs that, were they to be found in the list of `MedicationIdPairStrings` pairs enumerated from the medical records of the individual, would indicate a potential health risk for that individual. The result of the `Intersect` function is an `ImmutableSetOfString`. The model calls for the `Set` to be converted to an `Array`. This is done with the `ToArray` function. Note that the member type of the `Set` and the `Array` are the same. Collection type coercion from Set to Array does not change the member type.

Both `ImmutableSetOfString` and `ArrayOfString` must be defined in the metadata prior to model compilation for these to be understood by the <product name> PMML Compiler.

EnvContext – Access Portal to Persistent Tables, Computation Clusters and More

Arbitrary data content can be made available to any model through the <product name>'s EnvContext singleton object. This is designed as a pluggable component to each instance of the <product name> engine running. It is specified in the <product name>'s configuration file which is supplied as argument when starting the <product name>. Content can both be read as well as written to the EnvContext from a model.

The Scala trait that defines the EnvContext protocol is fairly narrow in scope, offering the following methods:

```
trait EnvContext {
  def initContainers(mgr:MdMgr, dataPath:String,containerNames:Array[String]): Unit
  def initMessages(mgr: MdMgr, dataPath: String, msgNames: Array[String]): Unit
  def getObjects(containerName: String, key: String): Array[BaseContainer]
  def getObject(containerName: String, key: String): BaseContainer
  def setObject(containerName: String, key: String, value: BaseContainer): Unit
  def setObject(containerName: String, elementkey: Any, value: BaseContainer): Unit

  def getMsgObject(containerName: String, key: String): BaseMsg
  def setMsgObject(containerName: String, key: String, value: BaseMsg): Unit

  def contains(containerName : String, key : String) : Boolean
  def containsAny(containerName : String, keys : Array[String]) : Boolean
  def containsAll(containerName : String, keys : Array[String]) : Boolean
}
```

Except for the `contains`, `containsAny`, and `containsAll` methods, the lingua franca of the `EnvContext` protocol is either `BaseContainer` or `BaseMsg` objects. Know that all containers and messages defined by the user will inherit these traits, respectively.

To make their actual content accessible from the model, one needs to “downcast” them to their actual implementation class instance. The following example demonstrates how this can be done:

```
<!--
      Obtain the SmokingCodes BaseContainer array
-->
<DerivedField name="SmokingBaseContainers" dataType="ArrayOfBaseContainer">
      <Apply function="GetArray">
            <FieldRef field="gCtx"/>
            <Constant dataType="string">FilterArrays</Constant>
            <Constant dataType="string">SmokeCodes</Constant>
      </Apply>
</DerivedField>

<!--
      Obtain the SmokingCodes used to form its filter set
-->
<DerivedField name="SmokingCodes" dataType="ArrayOfSmokeCodes"
optype="categorical">
      <Apply function="DownCastArrayMembers">
            <FieldRef field="SmokingBaseContainers"/>
            <Constant dataType="mbrTypename">SmokingCodes</Constant>
      </Apply>
</DerivedField>
```

In the example an array of `SmokeCodes` is retrieved by the function of the first `DerivedField`.

As suggested by its `dataType` it is an `ArrayOfBaseContainer`. The second `DerivedField` is used to downcast the array items to their real container type – namely the member item type of the `DerivedField` "`SmokingCodes`". Its type is "`SmokeCodes`". The `DownCastArrayMembers` uses its second argument to locate the member type to use for the cast.

Broadening EnvContext Semantics

As has been mentioned, the trait that describes the EnvContext is quite narrow, offering some basic services. This begs the question, "Is it possible to expand the sorts of services available in an EnvContext implementation?" The answer is a qualified *yes*. To do this, the developer must accomplish several things. To illustrate, imagine trying to add capabilities for accessing and manipulating Apache Spark Resilient Distributed DataSets (RDDs). See http://spark.apache.org/docs/1.1.0/programming-guide.html#resilient-distributed-datasets-rdds for a more complete list of what RDDs can do. To be able to use RDDs from the models, the following measures would be taken:

- Develop an EnvContext implementation that supports the EnvContext trait protocol *as well as* a mechanism that offers access protocol for retrieving/storing an RDD by name.
- The EnvContext implementation (let's call it RDDEnvContext) must be defined as a FixedContainer in the metadata. It will have no accessible fields, but importantly the physical implementation name used to catalog it should refer to the actual implementation.
- Define an RDD type. Like the EnvContext implementation this type will have no fields, but refer to the Spark RDD full qualified class name for its physical implementation, namely `org.apache.spark.rdd.RDD`.
- Currently we are not supporting invoking object methods directly in the PMML. For the `Iterable` functions (e.g., map, filter, groupBy, et al) on the object, the RDD would be the `Iterable` collection in the Apply function's first argument. The member function specified in the remaining arguments would be for example the map or filter's apply function.
- For other non iterable functions AND certain iterable functions that we currently don't

support, a UDF would be needed to invoke that method. For example, the `union(otherRdd : RDD)` method would require a UDF that accepted both the receiver `RDD` and the other `RDD` to answer the `union` of the two (e.g, `def union(receiver : RDD, otherRdd : RDD)`)

In case you are wondering, this sort of Spark access capability will be offered in a future release of the <product name> in the near future.

EnvContext use will be demonstrated in the examples section, **Putting It All Together,** below. Note that it is also possible to define user defined functions (UDFs) to achieve the same result. UDFs are presented next.

User Defined Functions

Shipped with <product name> platform is a user defined function library that contains some core function behavior of general use to all models. Currently there are approximately 80 functions with unique names in this library (500 + functions if their variants are counted). They include the following:

abs AgeCalc And AnyBetween AsCompressedDate AsSeconds Avg Between ceil CollectionLength CompressedTimeHHMMSSCC2Secs ContainerFilter ContainerMap Contains ContainsAll ContainsAny Count dateDaysSinceYear dateMilliSecondsSinceMidnight dateSecondsSinceMidnight dateSecondsSinceYear Divide endsWith Equal exp First floor Get GetArray GreaterOrEqual GreaterThan If incrementBy Intersect IsIn IsNotIn Last LessOrEqual LessThan ln log10 lowercase MakeOrderedPairs MakePairs MakeStrings Max Median Min Minus Multiply MultiSet Not NotAnyBetween NotEqual Now Or Plus pow Product Put round sqrt startsWith substring Sum threshold Timenow ToArray ToSet trimBlanks Union uppercase YearsAgo

All of the DMG standard built-in functions are implemented as are a number of functions that deal with collection filtering and transformations. The built-in functions described in the standard include the operator notation. These are supported in the PMML and are translated into an equivalent function in the UDF library. For example the operator '+' is translated into one of the variants of the “Plus” UDF.

Writing your own UDF Library

It is also possible to catalog your own library of UDF functions to the <product name> metadata so that your models can use them. Currently only Scala UDFs are supported. To add your own library is quite simple. The functions must be members of of some Scala object that inherits the package com.ligadata.pmml.udfs.UdfBase. For example, if a library called SecretSauce was to be used for the models to be produced, the top part of your UDF source code would look like this:

```
package com.company.secretsauce.udfs

import com.ligadata.pmml.udfs.UdfBase
import <other packages needed by implementation>

/**
      These are the company secret sauce udfs.
 */

object SecretSauce extends com.ligadata.pmml.udfs.UdfBase {
      .
      .
      .
      <SecretSauce UDF implementations>
      .
      .
      .
}
```

Note that the core library UDFs can be made available to your implementation by simply importing `com.ligadata.pmml._`.

Once developed, these are cataloged into the <product name> system via the MetadtataAPI function `AddUDFLib(udfLibJarPath : String)`. All of the functions are cataloged as well as any data types that may be needed and are not currently available in the <product name> metadata. The MetadataAPI uses a metadata introspection tool to examine the UDF jar content and produce the necessary metadata catalog operations.

NOTE: The current mechanism only supports cataloging functions in the SYSTEM namespace at this time. This is going to be changed to allow the UDF lib author to supply a namespace of their choice for the UDF library in a future release.

Macros

Macros can also be defined to perform special code generations that are difficult to implement with a function alone. Macros are used in the same way that functions are used and for the most part are indistinguishable from the user perspective.

We have already seen the use of a macro in one of our prior examples, namely the `DownCastArrayMembers`. This function uses the collection expression in argument one and the member type name (a string) from argument two (a collection with some member type) to substitute an instance of the template associated with the macro, namely `"""%1%.map(itm => itm.asInstanceOf[%2%])"""`. The numerical parameters enclosed in '%' are substituted with the `DownCastArrayMembers'` parameters.

Other uses include the generation of variable update contexts in order to update fields in a container or message type instances.

NOTE: At this writing, new macros can only be introduced by rebuilding the metadata bootstrap. See <programmer developer guide> for more details.

Constants

The `Constant dataType` has been extended to support a number of the new features introduced in the <product name> PMML. In addition to interpreting the `Constant` value to one of the supported DMG data types, they are used to give hints to the compiler as to the nature of the argument in its larger context. Most of these have been introduced in the previous sections, but is worth mentioning one more time what these each `dataType` extension does:

- '`ident`' is used in Iterable functions to indicate that the `Constant`'s value should be interpreted as a field name of the Iterable's item type. This field name is one of the fields found in the Iterable collection specified in the first argument of the function being

used.

- '`fident`' is used in Iterable function specifications to indicate that the `Constant`'s value is the name of the member function to be applied to the Iterable collection found in argument one of the Apply element.
- '`typename`' is used to indicate the `Constant`'s value should be interpreted as the name of the field whose dataType is to be used as the argument value. This usage is typically as part of a macro invocation.
- '`mbrTypename`' is similar to `typename` except the field's `dataType` must be the name of field with a collection as its `dataType`. In this case the collection's item data type is the generated value to be used in the expression. Like `typename`, this is used primarily for macro invocations.

Model Design Considerations

This section takes a step back to look generally at how models are organized, especially the RuleSet model upon which the current version of <product name> is focused.

Simple Models vs. Historical Models

Perhaps one of the most important distinctions to be made is whether the model can process its model constructed with only the information found in the event received in one of the <product name> input queues or will other data be needed. Many models require examining the current input queue message with prior messages of the same type or even other types. These sort of complex data inputs are specified when the messages are defined for the model or group of models. See the <Message Definition> section of the <programmer developer guide> for more details.

Filtering

One of the principal activities of a RuleSet model is to build a system of predicates and filters that will classify the given input as a match or as DMG puts it a *score* that is worthy of subsequent examination.

There are a number of ways to organize the filters to produce these scores. The style used to a large degree depends upon whether the model is a simple one or must consider the current event message with prior related messages (e.g., the medical history of an individual).

Another discriminator as to how to approach the filtering problem is how stable are the discriminating values and whether they might explicitly be used to good effect in the model in order to produce the score. Code usage and model rule “maturity” definitely have a bearing on the style used.

With this in mind, consider the following organizations for your filtering when you start to produce models:

- Explicit Data Value Filtering

  With this approach, the problem is well understood with a standard set of predicates to derive a decision with regard to the model's objective. Code values or other indicators in the input are stable and can be used directly in the model itself to achieve an answer to the problem of study.

- Table Driven Filtering

  When code usage is being improved/modified frequently, it is better to express this variablity in a set of tables defined outside the model. These tables can be modified and used by existing models without the necessity to rebuild and re-catalog the model in the <product name> metadata.

  Typically these tables are made available to the models via the EnvContext. When the <product name> cluster starts, the EnvContext is initialized and the principal accomplishment is the initialization of filter tables used by one or more of the models. These are typically loaded from some external source like a kv store, relational

database table, or fetched from some service.

Two styles are supported:

- Fiilter tables or Sets may be fully retrieved from EnvContext and used inside model. An example of how to do this will be presented in the **Putting it All Together** section.

- The EnvContext trait also defines methods that will perform existence testing of a key or array of keys in a given table held with the EnvContext for this purpose.

Integrating Traceability – Making the Model Results Audit-able

Many businesses which may want to use this software are either heavily regulated or are constrained legally in some fashion (if not both). For them decisions made in a model may need to be revisited if not by the business' internal auditors then by legal opponents seeking claims against the business. For this reason it makes eminent sense to provide enough contextual information as is practical to easily see why a model prediction was made.

As it turns out, this is quite easy to accomplish. The key predicate results and any tabulated information that led to the model score can be added to the model's emitted output with supplementary mining field entries in the model's `MiningSchema` section.

To see how this works, consider a medical model that reviews the prescription drug records for a given insurance member for possible deleterious drug combinations. With a filter table consisting of drug pairs that would produce bad if not toxic effects in the member, records of the members are reviewed for each doctor, lab, or institutional visit over some “lookback” period. From the traceability perspective, it would be prudent that the model not only produced the prediction (there is a problem), but also what drug combination(s) was/were found that determined this finding. The `MiningSchema` might look like this:

```
<MiningSchema>
      <MiningField name="ContraIndicative" usageType="supplementary"/>
      <MiningField name="IsContraIndicative" usageType="predicted"/>
</MiningSchema>
```

where `ContraIndicative` is an array of strings containing the drug combinations that are problematic. It could be derived with a function like this:

```
<DerivedField name="ContraIndicative" dataType="ArrayOfString"
optype="categorical">
      <Apply function="ToArray">
            <Apply function="Intersect">
                  <FieldRef field="MedicationIdPairStrings"/>
                  <FieldRef field="conflictMedCdsSet"/>
            </Apply>
      </Apply>
</DerivedField>
```

The `conflictMedCdsSet` is the filter table and the `MedicationIdPairStrings` is an array that enumerates the unique pairs of drugs found in the member's records.

One final note about this. Especially during model development, it is useful to emit the key decisions and data used to reach a given score result. This will aid in creating a more meaningful set of test data to maintain with the model and can also be used as a pedagogical device when a new or changed model is introduced to the client.

More details regarding testing will be covered in the next section, “Development”.

Development

Introduction

The <product name> platform is designed to operate on one or more Linux based systems that run atop commodity class hardware. With the exception of the <product name> PMML Compiler (and Message Compiler?) the platform will run on Windows as well. Currently the compilation of the generated Scala code from the PMML expects *nix commands to complete the generation of the jar executable.

The current implementation of all components that comprise the platform is written in Scala, a functional style language that generates Java byte codes.

See <appropriate reference manual> for further information regarding the hardware, recommended versions of the operating system, and other details.

Development Environment

See <appropriate reference manual> for complete information regarding the hardware, recommended versions of the operating system, and other details. The key components are:

- Java 1.7.x. The platform uses java.nio and other features that are found only in 1.7 and above.
- Scala 2.10.x. It is recommended that the Scala version installed be the one that the Eclipse Scala plugin uses. It is extremely useful to maintain compatibility with Eclipse and other environments such that the debugging tools work properly. At this writing Scala 2.10.4 is in use by the plugin and defined in our sbt projects.
- The sbt build tool is needed to generate the deployable system. The latest version is recommended. Several sbt plugins are necessary as well. Use the latest versions.
    - com.typesafe.sbteclipse – This plugin will generate eclipse project files that may be imported into your eclipse workspace. The strategy is to make sure any project will build with sbt first, then generate the eclipse project from it.

      Note that other ide plugins are available for Net Beans and IntelliJ. See http://www.scala-lang.org/old/node/91 for additional information.

    - com.orrsella sbt-sublime. If you like the sublime editor, this plugin will help you build sublime projects.

    - com.eed3si9n sbt-assembly. Our scripts will build fat jars that have the kitchen sink in them to ease the deployment of the platform executable and several other tools that are part of the platform. This must be installed.

    - net.virtual-void sbt-dependency-graph – This plugin helps manage the jar madness

endemic with any java jvm based system. It installs a number of graphical viewers including those that create image representations. See https://github.com/jrudolph/sbt-dependency-graph for more details. Note that if you wish to display the graphic forms of these (e.g., a 'dot' file), you need to install the appropriate tools. Graphviz works well and offers a number of display tools and image type coercion services.

Adding Metadata / Compiling Messages, Containers, and Models

All messages, containers, types, functions, and models are added with the <product name> Metadata API. The MetadataAPI is a service implementation that can be utilized by web and desktop applications to catalog metadata elements. A simple command line program is also available.

Most inputs are either expressed with JSON strings or XML strings. The PMML in particular is passed to the API as XML. See the <product name> Programmer Reference Manual (or whatever) for more information with regard to using the Metadata API.

Model Compilation Errors

Even though the use of PMML, an XML model specification language, does simplify the development of models, it doesn't eliminate all problems associated with program development. The compiler can and will produce errors. There are two basic types:

- There are syntactical issues with the input xml such that the compiler could not completely generate a Scala source file for submission to the Scala compiler.
- The Scala source file from the PMML was produced, but the Scala compiler has found problems with the generated source.

Syntax errors are reasonably easy to diagnose and fix. The error messages generated are from the SAX jar shipped with Java.

The second class of errors are more difficult in that they can be caused for any of the

following reasons (and likely others):

- Missing metadata
  - There is a missing data type
  - There is a missing function declaration. For example it is possible to have a version of a function in the metadata but not one that can support the argument types in use in the model.
  - A common issue is attempting to apply a Scala.collection.mutable method to an immutable collection. For this reason, there are two kinds of Sets available: a mutable Set whose type names are of the form SetOf<ItemType> and immutable Set whose type names are of the form ImmutableSetOf<ItemType>/
- Misuse of the Iterable Functions. There are currently some limits on how complex the Iterable function arguments can be. For example, it is not currently possible to use a sub-function (a nested <Apply> element) as an argument to an Iterable function use (e.g., ContainerMap or ContainerFilter). Mitigation of this limitation is scheduled for a future release.

To properly diagnose the problems, regardless of what the cause, both the Scala compiler diagnostic error messages and a copy of the generated Scala source must be examined much as one would diagnose any Java or C++ compile problem. The MetadataAPI will produce the generated Scala source for your inspection when compilation errors are encountered through the service interface to the MetadataAPI.

[Tamara, I have pinged Ramana about this. Basically we need to have a straight story on how the Metadata API will return a copy of the generated Scala with the compiler diagnostics to the user when a model is being compiled and fails with diagnostic errors. Afaik the model author/client is likely to be working through a browser interface on some desktop computer and dealing with the MetadataAPI service.... There will be a fixup/trueup/rewrite here once we settle on what will be done.]

Debugging Models

Once the model is generated, it compiles and its metadata information has been cataloged, it still needs to be tested. The model needs to be added to the <product name> engine's working set and messages sent to it. There are several aspects to this:

- At least to start, work with only a few messages inputs to the engine that will “cover” the logic paths in the model PMML.
- As was suggested previously, expand the supplementary mining field outputs to include all of the relevant derived field values to make it easier to inspect the results for correctness.
- If possible, don't add new UDFs at the same time you are developing a new model. If that is not possible (it often is not), expect to use the debugger to walk into/stop in your UDF functions.
- Once these unit test cases have passed, only then generate additional inputs for a more system level test scenario.

Of course these simple methods and recommendations are difficult to follow. Invariably with a new model, there will be new UDFs and externally prepared filtering tables that need to be used in the logic flow of the model. The engine will produce “ERROR” messages in its log when it catches an exception caused by some issue as these model elements “inter-operate” (not so much).

When this happens, the best solution is to attach the debugger to a single node cluster instance of the <product name> engine.

To do this, build an sbt project for your generated model. It is there that you will set your breakpoints for the diagnosing the ills being experienced. Add the generated scala source to the sbt project in a directory tree that matches the generated package name at the top of the generated source file. From sbt, make sure it compiles. Once satisfied that is the case, generate the eclipse project by running the 'eclipse' task from your sbt command prompt. Leave sbt. Be sure to add the project to the build.sbt file in the root of your <product name> enlistment. Once it is there, run the install script (see <some reference to other section of dev manual> for details). This will compile and install your debug jar and place it in the

appropriate location for debugging. It will also include your compiled project in the target fat jar that is to execute.

All that remains is to start eclipse. The installation script will have generated a new eclipse project for the model (or EnvContext impl or UDF lib) under test. Import the project or projects you have added into your workspace. It is not a bad idea to add at least the <project name> OnLEPManager project there as well so you can see what happens when control is transferred there by the thrown exception.

Note that if it is certain that the problem is in the UDF library you have developed, there is no need necessarily to create an Eclipse project for the generated model. It is useful generally to do so such that you can track both the use UDF use and implementation in the debugger, however.

Now you can start the engine with the following debug options:

java -Xdebug -Xrunjdwp:transport=dt_socket,address=8998,server=y -jar <somepath>/OnLEPManager-1.0 --config /tmp/OnLEPInstall/Engine.cfg

In the example, the jvm listens on port 8998 for notification from the remote debugger that the execution may commence. Until you start your debug session, the jvm will pause.

Set the breakpoints desired in your model or UDF lib that the model uses. You then need to invoke the engine application through the “Remote Java Application” debug instance you have set up. Be sure to use the same port as was used on the command line when you started the engine with the debug options (port 8998 in the example).

With any luck you will stop on your breakpoints as expected. Should you see screens asking for locations of the source files, choose the workspace (if using Eclipse) projects where your breakpoints are set. It may also prove useful to have some of the engine projects built in the IDE as well so that you can for example see how the engine handles the null pointer or class cast exceptions you might be experiencing.

Preparing Filter Tables

As was presented earlier, one of the key ways to organize model filtering is by introducing what are known as filter tables through the <product name> engine's EnvContext plugin. There is a simple tool that ships with the current version of the <product name> platform for constructing MapDb KV stores from comma delimited (csv) files with a header record naming the column names. For example, imagine you needed to filter for smoker diagnoses in some medical application with these keys and values found in file "`smokingCodes.csv`":

```
icd9Code,icd9Descr
3051,"3051 - Tobacco use disorder"
64901,"64901 - Tobacco use pregnancy, antepartum condition"
64902,"64902 - Tobacco use pregnancy, postpartum complication"
64903,"64903 - Tobacco use pregnancy antepartum condition or complication"
64904,"64904 - Tobacco use pregnancy postpartum condition or complication"
```

The KVInit tool can be used as follows to build a table:

```
java -jar $ONLEPLIBPATH/KVInit-1.0
      --kvname com.ligadata.med.SmokeCodes_100
      --kvpath <enginePath>/kvstores/
      --csvpath smokingCodes.csv
      --keyfieldname icd9Code
```

This will cause MapDb files called SmokeCodes.hdb and SmokeCodes.hdb.p to be created. As it happens, this is the name of the Container that was cataloged in the metadata to represent the smoke codes content (a key and description). The EnvContext at startup will be supplied a list of the FixedContainers by the engine to load if it chooses. It could of course delay the load until the first access of the table is made by a model in the working set.

Putting It All Together

In this section a number of annotated sample models from different business domains are presented. These models (without the commentary) can be found in the trunk/SampleApplication folder of your enlistment The xml model text will be presented in

`fixed font`. Commentary regarding design, purpose of derived fields, and general points of interest will be interspersed throughout. The font for these annotations will be with variable width font used for this text ([Tamara, change this comment if you plan to use other fonts or totally reformat this in some way])

Health Care Example

In this model, the in-patient, out-patient, and HL7 history of an individual are presented to the model by the engine, presumably triggered by the arrival of one of these respective records in the <product name> engine's input queue.

```
<PMML xmlns="http://www.dmg.org/PMML-4_1" version="4.1">
```

The Header contains important identification information used to by <product name> to catalog the model and make it available for use. The Application name is used as the name of the generated Scala file. The version is used as the metadata version of the model when cataloged. If the model were to be updated, it is a good idea to change the version to something new. If this is not done, the model metadata catalog operation will fail.

```
  <Header copyright="MedCorp. Copyright 2014" description="COPD Risk Assessment">
      <Application name="COPDRiskAssessment" version="00.01.00"/>
  </Header>
```

The `DataDictionary` is used to describe some of the basic data elements about the model, including the message it expects and its type. The global context reference is also added.. The `parameters` data field below actually tells the PMML Compiler what to place on generated model's class constructor argument list.

Other fields found here like `COPDSeverity` and `Msg_Desynpuf_Id` are updated via use of the `Put` UDF. This will be seen later.

```
  <DataDictionary numberOfFields="8">
      <DataField name="msg" displayName="msg" optype="categorical" dataType="Beneficiary"/>
      <DataField name="gCtx" displayName="globalContext" optype="categorical"
dataType="EnvContext"/>
     <DataField name="parameters" displayName="parameters" dataType="container">
        <Value value="gCtx" property="valid"/>
        <Value value="msg" property="valid"/>
     </DataField>
     <DataField name="COPDSeverity" displayName="COPDSeverity" optype="categorical"
dataType="string"/>
```

```
      <DataField name="Msg_Desynpuf_Id" displayName="Msg_Desynpuf_Id" optype="categorical"
dataType="string"/>


   </DataDictionary>
```

The `TransformationDictionary` is the largest section of the model. It contains all of the DerivedField elements in the model. A `DerivedField` has two parts, a variable used to cache the result of its primary top level function (the second part). If the result is not yet cached in the variable, the function described in the top level `Apply` element is executed. As you will see this function may have arguments that are themselves functions.

```
   <TransformationDictionary>
```

In this model, a number of filter arrays are maintained external to the model and accessed by the model through the EnvContext. The next several functions access these arrays and coerce them to their respective container types. For example, the `DerivedField SputumBaseContainers` retrieves the `Sputum` filter table. Since the EnvContext is an abstract interface, it only knows how to deal with BaseContainers and BaseMsgs. For this reason, it comes across as an ArrayofBaseContainer.

To make this array of elements usable, each element in the Array is downcast by the `DerivedField SputumCodes Apply` function `DownCastArrayMembers`. It reality, this function is actually a macro. Macros look very much like functions except they use one of several templates associated with their metadata to create the string for insertion into the generated code output stream. Of particular note is the `DownCastArrayMembers` second argument `SputumCodes`. Its dataType is `mbrTypename` which is a LigaData extension to the PMML `dataType` permissible values. What `mbrTypename` does is use the Contant's value (`SputumCodes`), look it up in the model dictionaries and answer the member type of the this field's type. To be successful, the type of the named variable must be a collection of some type (in this case it is an Array). The result of the `DowncastArrayMembers` macro is to generate code that will downcast each member of the `SputumBaseContainers` Array of `BaseContainers` to an `Array` of com.ligadata.edifecs.SputumCodes_100, the Scala type associated with `SputumCodes`. Note that a new array is created by this operation. The original array is left untouched.

The same mechanism is used for the other filter arrays for smoking, environmental exposure, coughing, and dyspnea.

```
      <!--
          ********************************************************
          *  EnvContext Access
          ********************************************************
      -->

      <!--
```

```
            Obtain the SputumCodes BaseContainer array
        -->
        <DerivedField name="SputumBaseContainers" dataType="ArrayOfBaseContainer"
optype="categorical">
            <Apply function="GetArray">
                <FieldRef field="gCtx"/>
                <Constant dataType="string">FilterArrays</Constant>
                <Constant dataType="string">SputumCodes</Constant>
            </Apply>
        </DerivedField>

        <!--
            Obtain the SputumCodes used to form its filter set
        -->
        <DerivedField name="SputumCodes" dataType="ArrayOfSputumCodes" optype="categorical">
            <Apply function="DownCastArrayMembers">
                <FieldRef field="SputumBaseContainers"/>
                <Constant dataType="mbrTypename">SputumCodes</Constant>
            </Apply>
        </DerivedField>

        <!--
            Obtain the SmokingCodes BaseContainer array
        -->
        <DerivedField name="SmokingBaseContainers" dataType="ArrayOfBaseContainer"
optype="categorical">
            <Apply function="GetArray">
                <FieldRef field="gCtx"/>
                <Constant dataType="string">FilterArrays</Constant>
                <Constant dataType="string">SmokeCodes</Constant>
            </Apply>
        </DerivedField>

        <!--
            Obtain the SmokingCodes used to form its filter set
        -->
        <DerivedField name="SmokingCodes" dataType="ArrayOfSmokeCodes" optype="categorical">
            <Apply function="DownCastArrayMembers">
                <FieldRef field="SmokingBaseContainers"/>
                <Constant dataType="mbrTypename">SmokingCodes</Constant>
            </Apply>
        </DerivedField>

        <!--
            Obtain the EnvExposureCodes BaseContainer array
        -->
        <DerivedField name="EnvExposureBaseContainers" dataType="ArrayOfBaseContainer"
optype="categorical">
                <Apply function="GetArray">
                    <FieldRef field="gCtx"/>
                    <Constant dataType="string">FilterArrays</Constant>
                    <Constant dataType="string">EnvCodes</Constant>
                </Apply>
        </DerivedField>

        <!--
            Obtain the EnvExposureCodes used to form its filter set
        -->
        <DerivedField name="EnvExposureCodes" dataType="ArrayOfEnvCodes"
optype="categorical">
            <Apply function="DownCastArrayMembers">
                <FieldRef field="EnvExposureBaseContainers"/>
                <Constant dataType="mbrTypename">EnvExposureCodes</Constant>
            </Apply>
        </DerivedField>
```

```
        <!--
            Obtain the CoughCodes BaseContainer array
        -->
        <DerivedField name="CoughBaseContainers" dataType="ArrayOfBaseContainer"
optype="categorical">
                <Apply function="GetArray">
                    <FieldRef field="gCtx"/>
                    <Constant dataType="string">FilterArrays</Constant>
                    <Constant dataType="string">CoughCodes</Constant>
                </Apply>
        </DerivedField>

        <!--
            Obtain the CoughCodes used to form its filter set
        -->
        <DerivedField name="CoughCodes" dataType="ArrayOfCoughCodes" optype="categorical">
            <Apply function="DownCastArrayMembers">
                <FieldRef field="CoughBaseContainers"/>
                <Constant dataType="mbrTypename">CoughCodes</Constant>
            </Apply>
        </DerivedField>

        <!--
            Obtain the CoughCodes BaseContainer array
        -->
        <DerivedField name="DyspnoeaBaseContainers" dataType="ArrayOfBaseContainer"
optype="categorical">
                <Apply function="GetArray">
                    <FieldRef field="gCtx"/>
                    <Constant dataType="string">FilterArrays</Constant>
                    <Constant dataType="string">DyspnoeaCodes</Constant>
                </Apply>
        </DerivedField>

        <!--
            Obtain the CoughCodes used to form its filter set
        -->
        <DerivedField name="DyspnoeaCodes" dataType="ArrayOfDyspnoeaCodes"
optype="categorical">
            <Apply function="DownCastArrayMembers">
                <FieldRef field="DyspnoeaBaseContainers"/>
                <Constant dataType="mbrTypename">DyspnoeaCodes</Constant>
            </Apply>
        </DerivedField>

        <!--
            ********************************************************
            *  Generally used derivations (date range calculation
            *  and filter content extracts)
            ********************************************************
        -->
```

The prior section created arrays of filter codes. Each of them has a string code key and a string code description of the code. The next several functions create `ImmutableSet` instances of the respective keys in each of these arrays. The projection of the key field from the com.ligadata.edifecs.SputumCodes_100 members of `SputumCodes` named `icd9Code` is accomplished with the mapping function, `ContainerMap`, resulting in an Iterable that can be cast to a `Set` by the function `ToSet`. Note that `ContainerMap` when used without a member function will simply return the mentioned field or fields. When more than one is mentioned in the argument list, the appropriate

Scala tuple is returned with the tuple member types the corresponding types of the mentioned fields.

```
       <!-- Project the icd9Code out of the sputum, smoking, envexposure,
           and cough arrays to form respective sets for matching
           icd9 codes found in the inpatient and outpatient histories.-->

       <DerivedField name="SputumCodeSet" dataType="ImmutableSetOfString"
optype="categorical">
           <Apply function="ToSet">
               <Apply function="ContainerMap">
                   <FieldRef field="SputumCodes"/>
                   <Constant dataType="ident">icd9Code</Constant>
               </Apply>
           </Apply>
       </DerivedField>

       <DerivedField name="SmokingCodeSet" dataType="ImmutableSetOfString"
optype="categorical">
           <Apply function="ToSet">
               <Apply function="ContainerMap">
                   <FieldRef field="SmokingCodes"/>
                   <Constant dataType="ident">icd9Code</Constant>
               </Apply>
           </Apply>
       </DerivedField>

       <DerivedField name="EnvExposureCodeSet" dataType="ImmutableSetOfString"
optype="categorical">
           <Apply function="ToSet">
               <Apply function="ContainerMap">
                   <FieldRef field="EnvExposureCodes"/>
                   <Constant dataType="ident">icd9Code</Constant>
               </Apply>
           </Apply>
       </DerivedField>

       <DerivedField name="CoughCodeSet" dataType="ImmutableSetOfString"
optype="categorical">
           <Apply function="ToSet">
               <Apply function="ContainerMap">
                   <FieldRef field="CoughCodes"/>
                   <Constant dataType="ident">icd9Code</Constant>
               </Apply>
           </Apply>
       </DerivedField>

       <DerivedField name="DyspnoeaCodeSet" dataType="ImmutableSetOfString"
optype="categorical">
           <Apply function="ToSet">
               <Apply function="ContainerMap">
                   <FieldRef field="DyspnoeaCodes"/>
                   <Constant dataType="ident">icd9Code</Constant>
               </Apply>
           </Apply>
       </DerivedField>
```

Below are a number of DerivedField elements that use core UDF functions to manipulate dates. The function YearsAgo will substract the supplied integer and subtract it from today's date. For this model, it refers to the earliest date which any record to be considered for the model computation can have. It is worth your while to study what is available in the core UDF library. If it fails to have something you need, you can always add your own library of UDFs to the platform metadata and use them in your

models.

```
<!-- Time now since epoch expressed as ISO8601 date encoded integer -->
<DerivedField name="Today" dataType="integer" optype="categorical">
    <Apply function="AsCompressedDate">
        <Apply function="Now"/>
    </Apply>
</DerivedField>

<!-- Time a year ago since epoch expressed as ISO8601 date encoded integer -->
<DerivedField name="AYearAgo" dataType="integer" optype="categorical">
    <Apply function="AsCompressedDate">
        <Apply function="YearsAgo">
            <Constant dataType="integer">1</Constant>
        </Apply>
    </Apply>
</DerivedField>

<!-- Calculate the member's age -->
<DerivedField name="Age" dataType="integer" optype="continuous">
    <Apply function="AgeCalc">
        <FieldRef field="msg.Bene_Birth_Dt"/>
    </Apply>
</DerivedField>

<!-- 40 years of age or older -->
<DerivedField name="FortyYrsOrOlder" dataType="boolean" optype="categorical">
    <Apply function="greaterOrEqual">
        <FieldRef field="Age"/>
        <Constant dataType="integer">40</Constant>
    </Apply>
</DerivedField>

<!--
    ************************************************************
    *  Filters for smoking, coughing, sputum, etc from various
    *  patient history elements
    ************************************************************
-->
```

The following `DerivedField` shows how to use a date range to constraint or filter the records one should consider. This is done with the `ContainerFilter` function. Its first argument, namely `msg.HL7Messages`, is called the `Iterable`. It is some kind of collection that supports the `Iterable` trait in Scala which is where the `filter` function is declared (what `ContainerFilter` actually generates in the output Scala source for the model). In this case, there is an '`fident`' Constant that specifies what is known as the member function for the `ContainerFilter`. The remaining arguments are arguments to this member function, Between. Notice the other special dataType for the `Constant Clm_Thru_Dt`. The `ident dataType` is a LigaData PMML extension to the permissible PMML dataTypes and are used to indicate that this is a field name of the `Iterable` collection in the first argument. This field is the date that will be compared with the date range supplied to determine if the record is to be considered further in the calculation (returned as an element in the array result).

```
<!-- All hl7 messages from this past year NOTE: Clm_Thru_Dt expressed
         as ISO8601 encoded int since epoch for this method... if it were millisecs,
         things need to be adjusted -->
    <DerivedField name="hl7InfoThisLastYear" dataType="ArrayOfHL7" optype="categorical">
        <Apply function="ToArray">
```

```
            <Apply function="ContainerFilter">
                <FieldRef field="msg.HL7Messages"/>
                <Constant dataType="fIdent">Between</Constant>
                <Constant dataType="ident">Clm_Thru_Dt</Constant>
                <FieldRef field="AYearAgo"/>
                <FieldRef field="Today"/>
                <Constant dataType="boolean">true</Constant>
            </Apply>
        </Apply>
    </DerivedField>
```

This `DerivedField` again uses the `ContainerFilter` on various arrays of containers. The filtering is accomplished on each array and its length is determined and then added together to produce a tally of all smoking related records found for the current beneficiary identified in the incoming message. If that tally is greater than zero, a true is assigned to the `DerivedField`'s variable otherwise a false.

What is of interest here is how functions can be nested to express one logical thought with a `DerivedField`. This `DerivedField` value would be a good candidate for being an output variable as smoking is a key discriminator for the likelihood of COPD.

EnvironmentalExposure, Coughing, et al are managed in the same way in their `DerivedField` elements below.

```
    <!-- Over Smoking coded in icd9 codes in last year  -->
    <DerivedField name="WithSmokingHistory" dataType="boolean" optype="categorical">
        <Apply function="greaterThan">
            <Apply function="+">
                <Apply function="CollectionLength">
                    <Apply function="ContainerFilter">
                        <FieldRef field="inpatientInfoThisLastYear"/>
                        <Constant dataType="fIdent">IsIn</Constant>
                        <Constant dataType="ident">Admtng_Icd9_Dgns_Cd</Constant>
                        <FieldRef field="SmokingCodeSet"/>
                    </Apply>
                </Apply>
                <Apply function="CollectionLength">
                    <Apply function="ContainerFilter">
                        <FieldRef field="inpatientInfoThisLastYear"/>
                        <Constant dataType="fIdent">ContainsAny</Constant>
                        <FieldRef field="SmokingCodeSet"/>
                        <Constant dataType="ident">Icd9_Dgns_Cds</Constant>
                    </Apply>
                </Apply>
                <Apply function="CollectionLength">
                    <Apply function="ContainerFilter">
                        <FieldRef field="outpatientInfoThisLastYear"/>
                        <Constant dataType="fIdent">IsIn</Constant>
                        <Constant dataType="ident">Admtng_Icd9_Dgns_Cd</Constant>
                        <FieldRef field="SmokingCodeSet"/>
                    </Apply>
                </Apply>
                <Apply function="CollectionLength">
                    <Apply function="ContainerFilter">
                        <FieldRef field="outpatientInfoThisLastYear"/>
                        <Constant dataType="fIdent">ContainsAny</Constant>
                        <FieldRef field="SmokingCodeSet"/>
```

```
                        <Constant dataType="ident">Icd9_Dgns_Cds</Constant>
                    </Apply>
                </Apply>
            </Apply>
            <Constant dataType="integer">0</Constant>
        </Apply>
    </DerivedField>
```

Another example of date range filtering. This time the `msg's Inpatient_Claims` array is filtered.

```
    <!-- All inpatient records this past year  NOTE: Clm_Thru_Dt expressed
         as ISO8601 encoded int since epoch for this method... if it were millisecs,
         things need to be adjusted -->
    <DerivedField name="inpatientInfoThisLastYear" dataType="ArrayOfInpatientClaim"
optype="categorical">
        <Apply function="ToArray">
            <Apply function="ContainerFilter">
                <FieldRef field="msg.Inpatient_Claims"/>
                <Constant dataType="fIdent">Between</Constant>
                <Constant dataType="ident">Clm_Thru_Dt</Constant>
                <FieldRef field="AYearAgo"/>
                <FieldRef field="Today"/>
                <Constant dataType="boolean">true</Constant>
            </Apply>
        </Apply>
    </DerivedField>

    <!-- All outpatient records this past year  NOTE: Clm_Thru_Dt expressed
         as ISO8601 encoded int since epoch for this method... if it were millisecs,
         things need to be adjusted -->
    <DerivedField name="outpatientInfoThisLastYear" dataType="ArrayOfOutpatientClaim"
optype="categorical">
        <Apply function="ToArray">
            <Apply function="ContainerFilter">
                <FieldRef field="msg.Outpatient_Claims"/>
                <Constant dataType="fIdent">Between</Constant>
                <Constant dataType="ident">Clm_Thru_Dt</Constant>
                <FieldRef field="AYearAgo"/>
                <FieldRef field="Today"/>
                <Constant dataType="boolean">true</Constant>
            </Apply>
        </Apply>
    </DerivedField>

    <!--
        Tally the reports of Environmental Exposure from the inpatient and
        outpatient records over the past year and answer if that tally
        is greater than 0
      -->
    <DerivedField name="WithEnvironmentalExposures" dataType="boolean"
optype="categorical">
        <Apply function="greaterThan">
            <Apply function="+">
                <Apply function="CollectionLength">
                    <Apply function="ContainerFilter">
                        <FieldRef field="inpatientInfoThisLastYear"/>
                        <Constant dataType="fIdent">IsIn</Constant>
                        <Constant dataType="ident">Admtng_Icd9_Dgns_Cd</Constant>
                        <FieldRef field="EnvExposureCodeSet"/>
                    </Apply>
                </Apply>
                <Apply function="CollectionLength">
                    <Apply function="ContainerFilter">
                        <FieldRef field="inpatientInfoThisLastYear"/>
                        <Constant dataType="fIdent">ContainsAny</Constant>
```

```
                    <FieldRef field="EnvExposureCodeSet"/>
                    <Constant dataType="ident">Icd9_Dgns_Cds</Constant>
                </Apply>
            </Apply>
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="outpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">IsIn</Constant>
                    <Constant dataType="ident">Admtng_Icd9_Dgns_Cd</Constant>
                    <FieldRef field="EnvExposureCodeSet"/>
                </Apply>
            </Apply>
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="outpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">ContainsAny</Constant>
                    <FieldRef field="EnvExposureCodeSet"/>
                    <Constant dataType="ident">Icd9_Dgns_Cds</Constant>
                </Apply>
            </Apply>
        </Apply>
        <Constant dataType="integer">0</Constant>
    </Apply>
</DerivedField>


<!-- Over HL7 AATDeficiency in last year  -->
<DerivedField name="AATDeficiency" dataType="boolean" optype="categorical">
    <Apply function="greaterThan">
        <Apply function="CollectionLength">
            <Apply function="ContainerFilter">
                <FieldRef field="hl7InfoThisLastYear"/>
                <Constant dataType="fIdent">equal</Constant>
                <Constant dataType="ident">AATDeficiency</Constant>
                <Constant dataType="integer">1</Constant>
            </Apply>
        </Apply>
        <Constant dataType="integer">0</Constant>
    </Apply>
</DerivedField>

<!-- Over inp and outp Dyspnoea (Shortness of Breath) in last year  -->
<DerivedField name="Dyspnoea" dataType="boolean" optype="categorical">
    <Apply function="greaterThan">
        <Apply function="+">
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="inpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">IsIn</Constant>
                    <Constant dataType="ident">Admtng_Icd9_Dgns_Cd</Constant>
                    <FieldRef field="DyspnoeaCodeSet"/>
                </Apply>
            </Apply>
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="inpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">ContainsAny</Constant>
                    <FieldRef field="DyspnoeaCodeSet"/>
                    <Constant dataType="ident">Icd9_Dgns_Cds</Constant>
                </Apply>
            </Apply>
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="outpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">IsIn</Constant>
                    <Constant dataType="ident">Admtng_Icd9_Dgns_Cd</Constant>
```

```
                    <FieldRef field="DyspnoeaCodeSet"/>
                </Apply>
            </Apply>
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="outpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">ContainsAny</Constant>
                    <FieldRef field="DyspnoeaCodeSet"/>
                    <Constant dataType="ident">Icd9_Dgns_Cds</Constant>
                </Apply>
            </Apply>
        </Apply>
        <Constant dataType="integer">0</Constant>
    </Apply>
</DerivedField>

<!-- Over inp and outp ChronicCough in last year  -->
<DerivedField name="ChronicCough" dataType="boolean" optype="categorical">
    <Apply function="greaterThan">
        <Apply function="+">
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="inpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">IsIn</Constant>
                    <Constant dataType="ident">Admtng_Icd9_Dgns_Cd</Constant>
                    <FieldRef field="CoughCodeSet"/>
                </Apply>
            </Apply>
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="inpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">ContainsAny</Constant>
                    <FieldRef field="CoughCodeSet"/>
                    <Constant dataType="ident">Icd9_Dgns_Cds</Constant>
                </Apply>
            </Apply>
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="outpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">IsIn</Constant>
                    <Constant dataType="ident">Admtng_Icd9_Dgns_Cd</Constant>
                    <FieldRef field="CoughCodeSet"/>
                </Apply>
            </Apply>
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="outpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">ContainsAny</Constant>
                    <FieldRef field="CoughCodeSet"/>
                    <Constant dataType="ident">Icd9_Dgns_Cds</Constant>
                </Apply>
            </Apply>
        </Apply>
        <Constant dataType="integer">0</Constant>
    </Apply>
</DerivedField>

<!-- Over inp and outp ChronicSputum in last year  -->
<DerivedField name="ChronicSputum" dataType="boolean" optype="categorical">
    <Apply function="greaterThan">
        <Apply function="+">
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="inpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">IsIn</Constant>
                    <Constant dataType="ident">Admtng_Icd9_Dgns_Cd</Constant>
```

```
                    <FieldRef field="SputumCodeSet"/>
                </Apply>
            </Apply>
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="inpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">ContainsAny</Constant>
                    <FieldRef field="SputumCodeSet"/>
                    <Constant dataType="ident">Icd9_Dgns_Cds</Constant>
                </Apply>
            </Apply>
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="outpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">IsIn</Constant>
                    <Constant dataType="ident">Admtng_Icd9_Dgns_Cd</Constant>
                    <FieldRef field="SputumCodeSet"/>
                </Apply>
            </Apply>
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="outpatientInfoThisLastYear"/>
                    <Constant dataType="fIdent">ContainsAny</Constant>
                    <FieldRef field="SputumCodeSet"/>
                    <Constant dataType="ident">Icd9_Dgns_Cds</Constant>
                </Apply>
            </Apply>
        </Apply>
        <Constant dataType="integer">0</Constant>
    </Apply>
</DerivedField>

<DerivedField name="COPDSymptoms" dataType="boolean" optype="categorical">
    <Apply function="or">
        <FieldRef field="Dyspnoea"/>
        <FieldRef field="ChronicCough"/>
        <FieldRef field="ChronicSputum"/>
    </Apply>
</DerivedField>
```

This `DerivedField` shows another example of filtering arrays. This time, however, the one of the PMML standard “built-in” function 'or' is used that collects HL7 records in the last year that have recorded cought, copd, shortness of breath, or sputum diagnoses. Each built-in function mentioned in the standard is implemented in the core UDF lib. In this case 'or' is translated to 'Or'.

What is perhaps not obvious by looking at the `DerivedField` is that these fields in HL7 are not booleans but integers. The CMS data that this model is based encodes booleans as integers. To compensate, a special version of 'or' was written and added to the core UDF library. The core UDF library has 13 different implementations of 'Or' defined in the metadata for the compiler to use to match and verify that the function use is proper. Different argument counts and argument types are declared to handle various common uses. At this writing it is possible for example to 'Or' as many as 7 expressions together where the expression types are all Boolean or all Integer.

```
<!--
    Compare this function that examines multiple characteristics of hl7
    with one pass versus the method in the derivations above that
```

```
    examine one only at a time.  This requires an 'or' udf that
    can accept 'Int' types where '0' is false and 'non 0' is true
-->
<DerivedField name="FamilyHistory" dataType="boolean" optype="categorical">
    <Apply function="or">
        <Apply function="equal">
            <FieldRef field="msg.Sp_Copd"/>
            <Constant dataType="integer">1</Constant>
        </Apply>
        <Apply function="greaterThan">
            <Apply function="CollectionLength">
                <Apply function="ContainerFilter">
                    <FieldRef field="hl7InfoThisLastYear"/>
                    <Constant dataType="fIdent">or</Constant>
                    <Constant dataType="ident">ChronicCough</Constant>
                    <Constant dataType="ident">Sp_Copd</Constant>
                    <Constant dataType="ident">Shortnessofbreath</Constant>
                    <Constant dataType="ident">ChronicSputum</Constant>
                </Apply>
            </Apply>
            <Constant dataType="integer">0</Constant>
        </Apply>
    </Apply>
</DerivedField>
```

The `DerivedField MaterializeOutputs` is used to “materialize” any field that may be held in a container that one might wish to emit as an output value. This measure will be needed until the compiler generates access code to these “outside” non-DataDictionary/non-TransactionDictonary content at output emit time. It updates the `DataDictionary` field `Msg_Desynpuf_Id` with the `msg.Desynpuf_Id` value. In a future release will eliminate this work-around measure.

Notice too that the output data type for this `DerivedField` is a boolean. Unless the system is out of memory, the `Put` function used here *will always* return true. This has some important implications by the `DerivedField` that may use this field. We will see how that is the case in the next three `DerivedField` elements below.

```
<!--
    Materialize these variables from their containers to derived fields to make
    them accessible for output emission
-->

<DerivedField name="MaterializeOutputs" dataType="boolean" optype="categorical">
    <!-- <Apply function="and"> -->
        <Apply function="Put">
            <Constant dataType="string">Msg_Desynpuf_Id</Constant>
            <FieldRef field="msg.Desynpuf_Id"/>
        </Apply>
    <!-- </Apply> -->
</DerivedField>

<!--
    ********************************************************
    *  Principal classification logic... is the patient
    *  a cat2, 1a, 1b, or none of these?
    ********************************************************
-->
```

The `DerivedField CATII_Rule2` is one of the three principal derivations invoked the Rule elements in the RuleSet below. It best shows the high level logic for what it means to be classified Cat2. One must be younger that 40 years old, have at least one of {COPDSymptoms, AATDeficiency, or a FamilyHistory} to be considered in Cat2".

These three derivations all show the use of the "if" function. It has as its predicate the 'and' function and those functions and field reference values referenced inside it. It is followed by the 'true' and 'false' actions for the the if.

Notice that these both refer to invocations of the boolean function '`and`'. Why? Regardless if predicate is true or false, the author wants the `MaterializeOutputs` variable to be derived. If you go back to `MaterialzeOutputs` you will note that it always returns true. In the 'false' action (the second and element), the `MaterializeOuputs` positive boolean result is defeated by `and`'ing it with the a `Constant` false value. This yields the correct return result for the `CATII_Rule2`.

```
<DerivedField name="CATII_Rule2" dataType="boolean" optype="categorical">
          <Apply function="if">
              <Apply function="and">
                  <Apply function="not">
                      <FieldRef field="FortyYrsOrOlder"/>
                  </Apply>
                  <Apply function="or">
                      <FieldRef field="COPDSymptoms"/>
                      <FieldRef field="AATDeficiency"/>
                      <FieldRef field="FamilyHistory"/>
                  </Apply>
              </Apply>
              <!-- Rather than use rule score, you can set variables according
                  to if's predicate above... NOTE: Put always returns true.. so
                  we MUST take specific measure to force CATII_Rule2 value to false -->
              <Apply function="and">
                  <FieldRef field="MaterializeOutputs"/>
                  <Apply function="Put">
                      <Constant dataType="string">COPDSeverity</Constant>
                      <Constant dataType="string">2</Constant>
                  </Apply>
              </Apply>
              <Apply function="and">
                  <FieldRef field="MaterializeOutputs"/>
                  <Apply function="Put">
                      <Constant dataType="string">COPDSeverity</Constant>
                      <Constant dataType="string">NotSet</Constant>
                  </Apply>
                  <Constant dataType="boolean">false</Constant> <!-- force false to be
returned -->
              </Apply>
          </Apply>
      </DerivedField>

      <DerivedField name="CATI_Rule1b" dataType="boolean" optype="categorical">
          <Apply function="if">
              <Apply function="and">
                  <FieldRef field="FortyYrsOrOlder"/>
                  <FieldRef field="WithSmokingHistory"/>
                  <FieldRef field="AATDeficiency"/>
                  <FieldRef field="WithEnvironmentalExposures"/>
```

```
                <FieldRef field="COPDSymptoms"/>
            </Apply>
            <!-- Rather than use rule score, you can set variables according
                to if's predicate above... NOTE: Put always returns true.. so
                we MUST take specific measure to force CATI_Rule1b value to false -->
            <Apply function="and">
                <FieldRef field="MaterializeOutputs"/>
                <Apply function="Put">
                    <Constant dataType="string">COPDSeverity</Constant>
                    <Constant dataType="string">1b</Constant>
                </Apply>
            </Apply>
            <Apply function="and">
                <FieldRef field="MaterializeOutputs"/>
                <Apply function="Put">
                    <Constant dataType="string">COPDSeverity</Constant>
                    <Constant dataType="string">NotSet</Constant>
                </Apply>
                <Constant dataType="boolean">false</Constant>  <!-- force false to be
returned -->
            </Apply>

        </Apply>
    </DerivedField>

    <DerivedField name="CATI_Rule1a" dataType="boolean" optype="categorical">
        <Apply function="if">
            <Apply function="and">
                <FieldRef field="FortyYrsOrOlder"/>
                <FieldRef field="WithSmokingHistory"/>
                <Apply function="or">
                    <FieldRef field="AATDeficiency"/>
                    <FieldRef field="WithEnvironmentalExposures"/>
                    <FieldRef field="COPDSymptoms"/>
                </Apply>
            </Apply>
            <!-- Rather than use rule score, you can set variables according
                to if's predicate above... NOTE: Put always returns true.. so
                we MUST take specific measure to force CATI_Rule1a value to false -->
            <Apply function="and">
                <FieldRef field="MaterializeOutputs"/>
                <Apply function="Put">
                    <Constant dataType="string">COPDSeverity</Constant>
                    <Constant dataType="string">1a</Constant>
                </Apply>
            </Apply>
            <Apply function="and">
                <FieldRef field="MaterializeOutputs"/>
                <Apply function="Put">
                    <Constant dataType="string">COPDSeverity</Constant>
                    <Constant dataType="string">NotSet</Constant>
                </Apply>
                <Constant dataType="boolean">false</Constant>  <!-- force false to be
returned -->
            </Apply>
        </Apply>
    </DerivedField>

  </TransformationDictionary>

  <RuleSetModel modelName="COPDRisk" functionName="classification" algorithmName="RuleSet">
```

The MiningSchema defines the variables in the model that are to be used. There are a number of possible usageType values supported in the DMG standard including `active`, `predicted`, `group`, `order`,

`frequencyWeight`, and `analysisWeight`. Only two of them are of interest in the current <product name> PMML, namely the `predicted` and `supplementary` usageTypes. Others may be mentioned, but will server only as documentation.

The `predicted` value is the key value to be returned by the model as output. The `supplementary` values are ancillary to the prediction, useful for supplying contextually important information regarding the prediction (e.g., why the prediction was made). Other usageTypes may be used but currently are not actively used, serving only to document what a particular variable's purpose is.

```
<MiningSchema>
    <MiningField name="Msg_Desynpuf_Id" usageType="supplementary"/>
    <MiningField name="Age" usageType="supplementary"/>
    <MiningField name="Today" usageType="supplementary"/>
    <MiningField name="AYearAgo" usageType="supplementary"/>

    <MiningField name="COPDSeverity" usageType="predicted"/>

    <MiningField name="WithSmokingHistory" usageType="supplementary"/>
    <MiningField name="AATDeficiency" usageType="supplementary"/>
    <MiningField name="WithEnvironmentalExposures" usageType="supplementary"/>
    <MiningField name="FamilyHistory" usageType="supplementary"/>

    <MiningField name="COPDSymptoms" usageType="supplementary"/>
    <MiningField name="Dyspnoea" usageType="supplementary"/>
    <MiningField name="ChronicCough" usageType="supplementary"/>
    <MiningField name="ChronicSputum" usageType="supplementary"/>

</MiningSchema>
```

The `RuleSet` is where processing starts. It contains the rules that comprise this model and describe the `RuleSelectionMethod` (any of {`firstHit, weightedSum, weightedMax`}. Each `Rule` has a `score` associated with it. Note that it is possible for multiple rules to return the same `score`! When the model is expressed like this, typically all of the rules are processed.

In this example, only the `RuleSelectionMethod firstHit` is specified which causes the model to stop processing subsquent rules in the `RuleSet` if the rule just processed produces a non-default score. For example, if Rule `CATI_Rule1b` were to return true, the score would be set to “1b” and processing would stop. The `firstHit` acts as a short-circuit.

There are other mechanisms that may be specified including `weightedSum` and `weightedMax`. When either or both of these are used, all rules are processed and the best score is returned as the model prediction according to calculations made either on the sum of the weights of the rules that executed or the rule that has the highest weight.

It should also be noted that if no `RuleSelectionMethod` is used, `firstHit` is the default. It is also possible to specify combinations of these methods.

NOTE: Only `firstHit` is currently implemented. It is expected that the `weightedSum` and `weightedMax` algorithms described in the `RuleSet` section of the DMG standard will be implemented before <product name is released to the wild in GitHub.

```
      <RuleSet defaultScore="0">
          <RuleSelectionMethod criterion="firstHit"/>
          <SimpleRule id="CATI_Rule1b" score="1b">
              <SimplePredicate field="CATI_Rule1b" operator="equal" value="true"/>
          </SimpleRule>
          <SimpleRule id="CATI_Rule1a" score="1a">
              <SimplePredicate field="CATI_Rule1a" operator="equal" value="true"/>
          </SimpleRule>
          <SimpleRule id="CATII_Rule2" score="II">
              <SimplePredicate field="CATII_Rule2" operator="equal" value="true"/>
          </SimpleRule>
      </RuleSet>
  </RuleSetModel>
</PMML>
```

Bank Example

Stock Quote Monitor and Limit Sale/Buy Trigger Example

Stock Quote Monitor and Limit Sale/Buy Trigger Example