Enhancing A Production Rule Engine With Predictive Models Using PMML

D. Sottara
DEIS - Department of Computer Science, Electronics and Systems
University of Bologna
Viale Risorgimento 2
40129 Bologna (BO) Italy
dsotty@gmail.com

P. Mello
DEIS - Department of Computer Science, Electronics and Systems
University of Bologna
Viale Risorgimento 2
40129 Bologna (BO) Italy
paola.mello@unibo.it

C. Sartori
DEIS - Department of Computer Science, Electronics and Systems
University of Bologna
Viale Risorgimento 2
40129 Bologna (BO) Italy
claudio.sartori@unibo.it

E. Fry
Department of Modeling and Simulation
Naval Health Research Center
140 Sylvester Road
San Diego, CA 92117, USA
eafry@gmx.com

ABSTRACT
In this paper we describe how the Predictive Model Markup Language (PMML) standard enhances the JBoss Drools production rule engine with native support for using predictive models in business rules. The historic debate between symbolic and connectionist approaches to rule/model orchestra-tion provides numerous examples of hybrid systems combining "hard" and "soft" computing techniques to achieve different levels of integration. Rules are often used to decide when and which model to invoke; model outputs, in turn, can be used to evaluate the preconditions of a rule. In a loosely coupled system, the rule engine calls an external component implementing the predictive model, but this has several disadvantages, most notably the need to setup proper communications and reconcile any difference in the way the components encode the data. We propose instead, a tightly integrated system where predictive models and rules become part of the same reasoning framework. The models, encoded using the PMML 4 standard, are loaded and processed by a compiler implemented using the rule engine itself. The PMML document is transformed into a set of facts that define the model, and a series of rules that formalize the model’s behavior. In addition, most PMML data processing, validation, and transformation procedures are also implemented using auto-generated rules. Finally, in order to integrate model inputs and outputs seamlessly in the inference process, we exploit an extension of the Drools engine which adds native support for uncertainty and/or fuzziness.

Keywords
PMML, Predictive Models, Rule-Based Systems

1. HYBRID SYSTEMS COMBINING RULES AND MODELS : MOTIVATIONS
Organizations often warehouse vast amounts of information, data that can be an invaluable resource if extracted and used in the proper context to support operational processes. Various data mining techniques exist to extract structured information from raw data and this information can be used to create predictive models (PM) that mathematical or statistical approximate the behavior of a complex system. Such approximations are possible even if the domain relationships are only partially unknown, for example, factors determining stock market price, or when problem states are not completely observable as in medical diagnosis.

In addition to models derived using data mining techniques, predictive models can include Business Rules that encode part of an organization’s knowledge about a domain and can be used for tasks such as classification or forecasting. Such rules are usually developed with a top-down approach using appropriate declarative, logic-based languages. While they do not allow a system to “discover” new information from the data they process, rules ensure that the encoded domain knowledge can be applied to other situations needing evaluation. Their inherent declarativeness makes it easier to explain and justify the conclusions they reach. Additionally, they can be easily updated when the decision criteria of the context change. We will use the term “predictive model” in this broader sense and to include models defined and implemented using business rules.

From an artificial intelligence perspective, such an approach can be considered inference performed by an intelligent agent, possibly as a part of a more complex chain. This would allow a developer to create decision support system (DSS)
exploiting not only the analytic and predictive capabilities of predictive models, but that also formalize the policies adopted by an organization as a consequence of the model's predictions.

However, the evaluation of a predictive model per se is not the only reasoning process that can involve the model. Most models can be considered “black-boxes” with limited justification capabilities, so any response provided by the model should be carefully validated. Likewise, the inputs fed into a model should satisfy the implicit preconditions upon which model performance depends, but that the model itself might not be able to evaluate. For example, the performance of a neural network degrades dramatically when the inputs are significantly different from the samples in the training set used to generate it. When the pre- or the post-validation process fails, appropriate actions must be taken to include lowering the confidence in the model’s prediction, deciding that the model must be retrained, or replacing it completely.

Such adjustments are appropriate tasks for a second order decision support system dedicated to managing the predictive models themselves. Given the policy-oriented nature of this meta-DSS, it seems more appropriate to implement it using a rule-based approach rather than mining the appropriate policies from historical examples, which most of the times are not even available.

The goal of our work, however, is to take model and rule integration a further step. Using a tight, if not homogeneous, integration of general rule-based systems and predictive models (rule-based or not), we demonstrate how a production rule engine (PRE) can provide a framework where predictive models can easily be integrated into larger inference systems. Production rule engines have several advantages: first, their reactive nature is well suited to managing events generated in dynamic, evolving environments; second, their pattern-matching capabilities can be used to apply flexible meta-evaluation criteria; third, rules can be used to infer logical consequences and execute side effects allowing a system to both enact policy and control model's behavior at the same time; fourth, production rule engines makes it trivial to include models exploiting native rule chaining and meta-reasoning features offered by the engine itself. Moreover, we will demonstrate how other types of models can be deployed to achieve a comparable, tight level of integration.

We propose a system with the following fundamental architectural characteristics: First, we adopted PMML [2] as the standard for the formalization of predictive models. This language achieves an optimal balance between the definition of a model’s structure and the declarative description of the processes involved in evaluating the models themselves. Such descriptions are useful both for runtime deployment and for the management of the model. Second, we chose the business logic integration platform Drools [9] as our workbench. This open source product is widely adopted in both academic and industry circles. It is object-oriented, written in Java, offers a rich language, and has several features that make implementations cleaner and more robust.

Our Drools-based reference system allowed us to investigate the extent a loose or tight integration of production rules and predictive models can be achieved using PMML. Currently, we have focused on one particular model - neural networks - as a case study, but we are applying the same concepts presented here to other models. Section 2 in particular will discuss basic rule-model integration patterns; section 3 will describe our implementation of the integrated architecture, effectively demonstrating how a production rule engine can become a PMML consumer; section 4 will show how to use the models at runtime within our proposed framework. Section 5 and 6 respectively will present a concrete use case, present our conclusions and outline a roadmap for future work.

2. INTEGRATION PATTERNS

To integrate a predictive model within a rule-based reasoning process, it is necessary to embed the model itself within an inference chain using one of a few basic integration patterns.

Rule-driven models. In the simplest scenario, the model is an independent, stand-alone component invoked by rules acting as "clients" of the model. Such rules are typically used to check preconditions required by a model and for preventing a model from being invoked in a context where its outputs would be invalid. While the concept of validity is abstract and depends on the specific domain, rules can define concrete requirements and perform input and output validation checks as appropriate. As soon as a model’s output is generated, it can become a new fact used to trigger further reasoning, generate new inferences, or trigger additional policies. This pattern is easily implemented in Drools by making a rule evaluation session (i.e., the execution of the rules contained in a knowledge base applied to a specific set of facts) aware of the existence of a model by inserting it in the working memory. The outline of an abstract rule invoking a model is given in listing 1

```
Listing 1: Rule-driven invocation

rule "Invoke"
    when
    $input : Data( <preconditions> )
    // select model(s) from available ones
    $model : ?Model( . . . )
    then
    // will trigger validation rules
    insert( $model.invoke( $input ));
end
```

This is an example of a loose integration pattern: the model is still a black-box component unaware of the rule-based context into which it has been deployed. The rule base needs to know the model’s interface in order to provide an adequate set of inputs and to correctly interpret the model’s outputs. It does not, however, need to know the internal particulars. The immediate advantage of this pattern, other than simplicity, is that even pre-existing legacy models can be easily integrated.

Model-driven rules. In this configuration, models are used to evaluate one or more constraints in a premise pattern. In production rule systems, a premise pattern is a complex
boolean formula, defined by combining atomic constraints with the usual logic connectives (and, or, not). Each constraint is a unary or binary expression that restricts the admissible values of a fact. Only facts matching the defined pattern can activate the rule containing the pattern itself. Common constraints include equality checks and ordinal comparisons, but more complicated constraints are not uncommon, including ones that require a predictive model to be evaluated.

Notice that this pattern is not using rules to compare the model output with a known fact - it is the model itself which performs the comparison. The rule base thus created can be considered a function-replacing hybrid system [4], since the "host" rule-base replaces the implementation of one of its internal components with a more suitable and effective one using different technology. Drosols supports this pattern by allowing the definition of custom, pluggable evaluators.

Listing 2: Model-driven rules

```
when
  Object( field isA target )
then
  ...
end
```

While deceptively simple, this approach greatly increases the expressive power of the rule engine. Rule-based, or more generally, logic-oriented inference engines rely on qualitative, symbolic techniques to reason over a set of available facts. While their nature facilitates the elicitation of the knowledge, the argumentation of the reasoning processes, and the interpretation of the inferred consequences, they are usually less efficient and effective when dealing with more low-level, quantitative informations such as, e.g., sensor inputs. The latter type of information is better handled using a combination of techniques, which are not normally available within symbolic reasoners unless a hybrid architecture such as the one we proposed here is used.

However, our proposed integration pattern cannot be always used in practice unless some important conditions are met. First, it must be possible to interpret the output of a model as a boolean value: hence, only classifier models can be used in this way (notice that a constraint implicitly defines the class of entities which satisfy that constraint and vice versa). This is the reason why the generic custom evaluator in listing 2 has been called isA. Second, even if the output of a classifier model can eventually be considered a boolean value, most models actually return more information. One of the motivations behind the use of predictive models in the first place is that the classification process involves some kind of uncertainty, which the models usually manage by returning an estimated degree of probability, confidence or belief [10]. When the model output is cast to a boolean value, this additional information is lost instead of being propagated along the inference chain. It is self-evident that a fact with an ambiguous classification should not always be treated in the same way as one with a definitely high degree of membership in a class.

Rule/Model chaining. This integration pattern is actually a combination of the previous two. In a typical scenario, a control rule triggers a model and generates an output. This output can either be validated directly using rules or by starting other more complex inference chains. Eventually, one or more of the inferred facts is consumed by an evaluator which in turn triggers another control rule and the process is restarted, as shown in listing 3.

Listing 3: Model and Rule Chaining

```
rule "Trigger"
  when
    $input : Data(<preconditions>)
    $model : ?Model(...)
  then
    insert( new Result($model.invoke($input)) );
end

rule "Eval"
  when
    Result( value pm_eval target )
  then
    insert( new Data(...) );
end
```

Notice that this type of integration can be combined with workflow management techniques such as Drosols/JBPM to orchestrate the execution of multiple models. The workflow would effectively define a business process where the evaluation of the model corresponds to specific tasks to be executed. In such a scenario, models are not just passive entities with respect to flow control logic for they can be used to influence the execution of the business process itself. Despite the tighter level of integration achieved, there is still a degree of separation between the rules and the models, which are based on different implementations (declarative rules vs compiled code) and process different type of data (objects vs primitive datatypes).

3. DEPLOYING PMML MODELS AS KNOWLEDGE RESOURCES

When a complex application requires a combination of rules and predictive models, the models themselves must be deployed within a rule base. Assuming that trained models already exist, each model must be instantiated and connected to the rules according to some integration pattern. A simple approach would be to select the best model currently available, instantiate it with all parameters set correctly and write all the adapter rules to direct data flow from the working memory into the models and vice versa.

This approach is not scalable for such manual integration requires an exact knowledge of the model's interface, making it difficult to keep models and rules aligned going forward. This problem could be solved conveniently using a PMML descriptor together with the model instance. This descriptor would contain the model parameters required to instantiate the model as well as the description of the model interface and which, theoretically, could be used to write the integration rules automatically. Additionally, this solution would add a conflict of responsibility problem, as the PMML standard defines not only the models, but also various pre- and

1 http://www.jboss.org/jbpm
post-processing operations. It would not be clear whether these actions should be executed within the model or within the rule base.

These considerations suggest a more radical approach. Since production rules can easily handle data validation and transformation, it can be argued that rules should also be used to emulate the evaluation of the predictive models themselves, provided, of course, that an appropriate set of facts and rules is present in the working memory. We adopted this approach for an architecture based on hard-computing designed soft computing [7], where the PMML descriptor is parsed and compiled into a logical theory (i.e. a set of rules and facts created automatically at the beginning of the inference process), using a standard mapping that guarantees that the model and the rule engine share an implicit agreement on the model’s interface and internal structure.

The immediate effects of this choice are (i) that a production rule engine such as Drools becomes effectively a PMML consumer, (ii) that PMML descriptors become declarative knowledge resources at the same level of rules and workflows and (iii) that it is possible to achieve a seamless, homogeneous integration of predictive models and (production) rules.

We will now discuss the strategies used to map PMML elements into rules. Section 4 will be dedicated a discussion of how a rule-based model can be appropriately invoked within a rule base.

### 3.1 PMML Compilation

To convert a PMML document into a logical theory, we created a dedicated compiler using a combination of rules and rule templates that improved on an approach we adopted in [1]. The theory generated by the compiler can then be saved for future use, or used directly to create a new Drools session.

A PMML document has a complex structure [5], with dense information content, and several inner references that make it complicated to analyze and transform using XSLT templates or a standard visitor implementation. Consequently, we chose a Drools-based implementation of the visitor pattern [3], using the rule pattern in listing ?? as a guide.

We use JAXB 2 in combination with the PMML 4.0 XSD schema to load PMML documents and convert them directly into a tree structure with typed nodes, i.e. trees where nodes are instances of classes representing the elements defined in the XSD schema. The root of this tree is then inserted into a working memory dedicated to a “compiler” which uses, compiler rules to “visit” the tree’s nodes and generates the target rules. In particular, the visit of a node ensures that any relevant child of that node is inserted in the working memory, and that any context object that might be relevant are also created during the compilation. The blackboard nature of the working memory, in fact, allows for the creation of additional helper objects which can be retrieved later as needed. This avoids forcing the visitor to carry the state information along while visiting the tree.

As the visitor inserts the nodes, reactive compilation rules are triggered. Using appropriate patterns, it is possible to write as many specialized rules as needed to define the criteria used to generate a new rule for the output theory. This implies that there is not a bi-univocal correspondence between the PMML elements and the output rules: in fact, for each element in the PMML file, any number of target rules can be generated.

To this end, we exploited MVEL templates for modularity and manageability reasons: a MVEL (rule) template such as the one in listing 5 is a parametric (rule) prototype which can be instantiated any number of times, replacing all the parameters with concrete values. We chose MVEL due to its balance between simplicity, runtime efficiency and expressiveness, since it natively offers features such as conditional expressions and loops.

### Listing 5: Rule Template Example

```java
@declare '{ruleTemplateN}'
rule *Rule_(@id)*
  when
    @{Type}({ field == @{value} })
  then
    @if { flag } . . . @else . . . @end{}
  end
@end{}
```

All the templates instantiated as a consequence of the compilation rules are added to a global theory: when all the applicable compiler rules have been fired, it is this theory that is either returned or added directly to a knowledge base supporting the actual information processing.

### 3.2 PMML Mapping

In this section, we describe the conceptual mapping we have adopted to convert PMML elements into rules. A detailed description of the rules and the relative data structures is out of the scope of this work, but all the code is freely available for revision and use.

---

2 [http://jaxb.java.net/](http://jaxb.java.net/)
3 [http://mvn.codehaus.org/MVEL+2.0+Templating+Guide](http://mvn.codehaus.org/MVEL+2.0+Templating+Guide)
Header. For every PMML theory, a header is generated to import all necessary classes. The information in the original PMML header is purely descriptive and, other than being used for administrative reasons, has no influence on the evaluation process.

Data definition. DataFields in the PMML DataDictionary and DerivedFields in the TransformationDictionary are used to generate beans to store the various data values. One bean class is generated for each field: its class name is taken from the name attribute of the PMML elements. All data beans (see listing 6) extend a DataField class which provides the common attributes, mirroring some of the properties defined in the PMML schema. The specific attributes— including the value, which has a type different for each field— are instead stored in the subclasses. Output fields further extend the class with attributes which depend on the model which generated them.

Listing 6: DataField definition

```java
declare DataField
@role(event)
  valid : boolean = false
  missing : boolean = false
  context : String
end

declare ConcreteField extends DataField
  value : Double
  displayValue : String = "..."
end
```

Remarkably, fields are marked as events: if the relative feature is enabled in the engine\(^5\), all data are marked with a timestamp and it becomes possible to express temporal constraints on the presence (or absence) of fields of a certain type in the working memory. Notice also that fields have a context attribute. A data field, in fact, can be shared by multiple models, but it can also be processed differently in the context of each model: for example, some validity constraints may depend on the particular model a field is wired to. So, we have chosen to scope and replicate a field instance for each model which is using it, keeping track of this contextualization.

Validation and Manipulation. Newly generated DataField beans have the valid flag set to false, effectively disabling them as most propagation rules check the value of that flag. We use validation rules to make sure that the field’s value is one of the admissible Values or falls within one of the admissible Intervals declared in the PMML document. If this is the case, or if no restriction is imposed on the field’s value, the rules set the valid flag to true. Likewise, manipulation rules detect special values used to model missing or invalid status and set the flags in the DataField objects accordingly.

Transformation. Transformation rules are created visiting DerivedField elements in the TransformationDictionary

Model Input Binding. While the data definition, validation and transformation rules are general, the binding rules operate in the context of a particular predictive model. These rules are generated from the MiningSchema element and, other than contextualizing a DataField, apply a second validation step to check any additional constraint required by the model. If the model inputs require any additional transformation, the transformation rules are instantiated again, this time scoped within the model context.

Model Output Binding. Model outputs are defined as extended DataField with additional model-dependent properties. The final output is actually generated in two steps: first the model direct output is post-processed, using rules which apply reverse transformations (in many cases, when a transformation is its own inverse, it is sufficient to modify the logic generating the template parameters) to obtain the DataFields marked as predicted. Second, the inferred field values are linked to the proper model Outputs and Targets, as defined in the PMML document. While Outputs are updated every time a model is evaluated, the Target fields are evaluated only when the model outputs are valid, and thus retain the last meaningful state. Moreover, we also enhance the Targets, merging any uncertainty output produced by the model in the related DataField facts, as will be discussed in paragraph 4.

Model Implementation: Neural Networks. The first class of models we have implemented using rules are neural networks: given the feed-forward nature of the computation they perform, it is trivial to emulate their behaviour using forward-chaining production rules. To create the model, it was sufficient to define two classes, Neuron and Synapse, to model the network structure and two classes, we arbitrarily called Stym and Charge, to model the propagation of the signals out of and into a neuron respectively.

The class definitions, shown in listing 7, all include a context attribute because all the objects representing a model should belong to the same context. Neurons and Synapses are derived from the Neuron and Con elements in the PMML descriptor: in addition to the links and the synaptic weights, we also keep track of the last activation for each neuron. At runtime, the deployment of a model will simply consist in the creation of an appropriate set of Neuron and Synapse instances. The behaviour of the model, instead, is defined using a set of rules, derived from templates like the one in

\(^5\)i.e. Drools Fusion is used
listing 8 (notice that this is a simplified version of the actual rule due to space limitations).

Listing 7: Neural Network facts

```
declare Neuron
c   context : String
   id : int
   lastOut : double
   dLastOut : double
end

declare Synapse
c   context : String
   source : int = -1
   target : int = -1
   weight : double
end
```

Listing 8: Neural Network evaluation rule

```
rule "Neuron@{nId}"
   salience -100
   when
      Number( $val : doubleValue )
      from accumulate( 
         Charge( context === @ctx ),
            index == @nId, $in : value ),
      sum( $in )
   )
   Neuron( id == @nId )
   then
      double x = 1 / ( 1 + Math.exp(-$val) );
      insertLogical( new Stym(@ctx, @$nId, x, false ) );
end
```

3.3 Auxiliary PMML-generated tools

In addition to creating the model itself, so that it can be evaluated, there are several functionalities which can be built on top of a predictive model, in order to facilitate its use in a concrete application. Some of them are specifically designed for predictive models, others are more general-purpose tools, but all are implemented using rules and typically require a combination of a standard, general theory and some context-dependent rules. The generation of these “wiring” rules is actually a standard operation, once the final context is known: when the context is a predictive model, the presence of the PMML descriptor allows to apply the usual templating technique to avoid the manual generation of those rules. Hence, the models generated and deployed using a PMML descriptor can be configured to include additional features on top of the required pre-, post-processing and evaluation functionalities.

Model Trainer. Implementing the parametric structures of a predictive model using facts in the working memory is also the basis for a rule-based implementation of the model training algorithms. In our neural network case study, we have used the Neuron and Synapse classes to store the activations and weights usually involved in a training procedure. To this end, we have also defined an abstract TargetField class, subclass of DataField. A concrete subclass of TargetField is then created for every output field of the model: the contemporary presence of an output and a target triggers a training step, which is currently implemented using a canonical backpropagation algorithm [6]. The implementation, which again exploits rule templates, also uses working memory facts to store parameters such as the learning rate. This will allow, in a future implementation, to use rule-based criteria to dynamically control the behaviour of the training process, using, for example, declarative heuristics based on a semantical analysis of the context the model is deployed in. Notice that while currently not yet integrated in the project, we have also already proven the feasibility of a rule-based implementation of training algorithms for decision trees, cluster models and self-organizing maps.

Model Informer. Drools Informer is yet another Drools module which creates dynamic, interactive questionnaires to interact with human users in order to collect new facts. A questionnaire is a collection of groups of questions: each question, then, can be answered and the answers can be used for various purposes. Groups and questions can change at runtime, usually depending on the answers provided for some other questions. The management of the questionnaire structure and the validation of the answers is handled using rules, while a web-based GUI interface is used to ask questions and collect answers from the user. The answers can eventually be bound to domain facts, synchronizing the internal state of the model objects with the presentation layer. Using this tool, it is possible to create a questionnaire for each model and bind the questions to the model input DataFields. The DataField type and validation constraints will influence the type of questions proposed to the user and the actual graphical widgets used in the interface. The output fields can optionally be bound to non-interactive graphical elements in the GUI.

Model Validator. While a PMML document supports the definition of DataField validity constraints, we argue that better semantically founded, context-dependent validation rules could be generated to make sure that the model inputs and outputs satisfy certain qualitative and quantitative expectations[]. When expectations are satisfied or, more interestingly, violated, high level Fulfillment and Violation events are generated which, in turn, can trigger appropriate rule-based policies, ranging from invalidating a model prediction to executing a number of training steps in order to update the model, to replacing the model altogether.

4. HOMOGENEOUS MODEL INTEGRATION

Using the PMML compiler, it is possible to load the models directly in a rule session without needing third-party libraries. This rule-based implementation of the predictive models does not alter conceptually the invocation patterns defined in Section 2, but it affects the way they are implemented. In the current implementation, a model is evaluated whenever all its input fields are present in a valid state, so there is no need to collect them explicitly as the template-generated rules will perform that operation automatically when appropriate. Moreover, a model will be re-evaluated whenever any one of the inputs is updated. Notice that by design no two instances of the same field can exist at the same time in the same model context, so a new value effectively overwrites and updates any previously existing value for the same field.
To trigger a model invocation, then, it is sufficient to instantiate the appropriate `DataField` classes and insert them as facts. From a more user-oriented perspective, it is also possible to use the auto-generated Informer questionnaire through the default web-based client or any other compatible client. Finally, as a Drools-specific alternative, named entry points are provided so the user can insert simple, primitive datatypes (strings, integers, floats, etc...) and the rules will generate the `DataFields` automatically.

Given the “open” nature of the model, the user will usually generate the basic `DataFields` defined in the `DataDictionary`, so that any validation or pre-processing rule will be applied to the values. If necessary, however, the invoking rules can bypass this process and generate the `DerivedFields` or even the internal model inputs directly to override the standard model evaluation process. Linking the model outputs back in the inference process, instead, is trivial as the model will automatically insert or update predetermined facts in the working memory which can be matched by the patterns in the premises of other rules.

This data-driven, “forward” invocation of the predictive model becomes seamless since evaluating the model is effectively equivalent to triggering a set of rules in cascade. The reactive, parallel nature of the production rules also facilitates the creation of ensemble systems, loosely integrated hybrid systems where multiple sub-models are invoked in parallel before their outputs are reconciled by a supervisor. Since PMML conveniently supports ensemble models [8] and can be used to define model verification criteria, the creation and evaluation of ensembles will be supported by the framework natively.

Using a predictive model as a constraint evaluator, instead, is no longer possible in the way it was presented in Section 2. The reason is that the model is no longer encapsulated in a standalone component which can be called on demand, but is part of the same working memory where the invoking rule resides. Notice that this is not necessarily a limitation, since it eliminates the need to copy the input values from the caller’s environment (the fields of an object passing through the RETE network as it is being inserted in the working memory) to the model execution environment. This also removes the limitation of having up to two inputs for model evaluation, which usually imposes the use of collections or ad-hoc beans, but, given the query-like nature of the evaluation, the model should better be invoked using a backward-chaining style rather than a forward-chaining one.

In Drools, this is not immediate but still possible using the invocation pattern shown in listing 9. Thus, any constraint of type `Object( this isA "Type" )` - where the generic custom operator `isA` is used to remark the classifier nature of the invoked model - should be rewritten in the form:

```
rule "Main"
when
  $o : Object ( )
then
  isaA( obj == $o, type == "Type")

query isaA( Object $o, String $type )
```

Invoking a query within the premise of a rule triggers a hybrid forward/backward chaining mechanism: the query looks for a particular fact in the working memory, while at the same time a rule is used to intercept the query execution and make sure that the model which is expected to generate that fact has completed its task, using any necessary fact present in the working memory (i.e. sharing the data structures and execution environment of the caller rule).

**Adding uncertainty.** This execution schema allows to deal with the last open issue introduced in paragraph 2. As already mentioned, the output of a classifier model is usually characterized by some degree of uncertainty\(^6\), which should be automatically propagated as a part of the inference process rather than being a static fact, potentially ignored. To this end, we exploit Drools Chance, another experimental module which adds the support for generic “imperfect” reasonings [11]. Using this framework, all the properties of an object (included its type) are not set to a definite value, but either have a degree of uncertainty or a full distribution associated to their value. The constraints in the rules can access these degree and combine them using a generalized version of the logic connectives which are implicitly present in the rules.

When generating the bean classes to store a model’s outputs, we combine the information contained in the PMML elements `Target` and `Output` both to provide any output meta-data as a dedicated fact in the working memory and to create uncertain `DataFields`, such as the generic `isA` in the previous abstract example, which have an implicit associated degree of uncertainty. When the fact is inserted in the working memory, the type check in the pattern `isA(...)` extracts that degree of uncertainty and combines it with any degree resulting from other constraints in the rule or query where that pattern is used.

This guarantees that, while formally the rules are transparent to any degree of uncertainty, they actually can evaluate, combine and propagate even this type of meta-information which the rule programmer can access at any time, usually to influence the enactment of the consequences in the conclusion of one or more rule. This aspect becomes particularly relevant when the rules are part of a decision support system, since even knowing a basic information such as the probability of an outcome or the expected value of a variable can be fundamental in making the final choice.

\(^6\)the discussion on the possible nature of this uncertainty is out of scope for this work
5. CASE STUDY
As a case study, we applied a PMML-encoded neural network in the context of the U.S. Navy SmartVent Project. SmartVent is an advanced closed-loop ventilator management system currently under development that is capable of processing real-time physiologic data, adjusting ventilator parameters as appropriate, and of maintaining intubated subjects within acceptable physiologic parameters under dynamically changing clinical conditions.

The system incorporates a rule-based expert system representing a consensus strategy for the mechanical ventilation of patients with Acute Respiratory Distress Syndrome (ARDS), and a neural network that models optimal physician management when treating the same condition. The expert system is effectively a set of domain-specific data validation rules providing boundary constraints to a Neural Network (NN) core consisting of standard three-layer, feed-forward neural network. At each measurement interval (every 10 mins), arterial blood gas, inspired oxygen concentration, hemoglobin saturation, ventilator settings and arterial blood pressure data are recorded and forwarded to SmartVent for evaluation. The output values generated by the network are the new ventilator settings, e.g. ventilator rate, peak inspiratory pressure, peak end-expiratory pressure, inspiratory time, flow and inspired oxygen concentration. SmartVent can function in a closed-loop mode without physician intervention, or in an advisory capacity mode in which recommends ventilator modifications to the medical team through a bedside display. It is a true, multi-parameter, closed-loop ventilator management system capable of nuanced recommendations. The network itself is a legacy implementation using sigmoidal activation functions in the hidden layer and linear activation functions in the output layer. It was designed and trained using a standard back propagation algorithm using NeuroShell 2 from Ward Systems, Inc. It has excellent operational performance, achieving 92-98% correlation with domain expert recommendations during the preliminary animal model testing (as of yet unpublished data). The only drawback of the system, and one additional reason to support the design of a rule-based training algorithm, is that the NeuroShell 2 tool used to train the network is not a PMML producer, so it was necessary to manually encode the trained model in PMML.

6. CONCLUSIONS AND FUTURE DEVELOPMENTS
This preliminary work shows that it is possible, using a shared data model and a single execution environment, to achieve homogeneous integration of predictive models and rule-based systems. In particular, it has been shown that production rule engines can be suitable PMML consumers and that PMML documents can be considered declarative pieces of knowledge. As the semantics of a PMML document can be instantiated naturally using reactive production rules, a common environment facilitating the combination of the two approaches has been shown to be possible.

From a theoretical perspective, our work shows a point of conjunction between “symbolic” and “connectionist” reasoning techniques. From a more practical viewpoint, it provides a way to combine data-driven, bottom-up techniques with top-down, knowledge-driven methodologies in the creation of knowledge-based decision support systems.

The PMML integration code for the Drool rule engine code is currently released as an optional module, called drools-pmml. Ongoing work is focused on addressing several open issues, most importantly, completing a full implementation of the PMML standard, and in providing better support for the evaluation of models other than neural networks. In future work, we will also consider including additional online training algorithms to make models adaptive in the presence of context change. We would also like to implement support for the serialization of models back to PMML to update/reflect any change made to the model itself. Finally, we plan to enhance the definition of Data Fields with ontologic annotations, so that associations between input fields declared in the PMML and the domain facts present in the working memory can be made using a semantic reasoner rather than a syntactic one.

7. ACKNOWLEDGMENTS
We wish to thank the U.S. Navy Knowledge Management Repository / SmartVent projects, and the DEIS DEPICT Project which co-sponsored this research. The opinions of the authors do not necessarily state or reflect those of their respective employers, the United States Navy, the Department of Defense, or the United States Government, and shall not be used for advertising or product endorsement purposes.

8. REFERENCES
[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley, Massachusetts, 2000.
[10] P. Smets. Imperfect information: Imprecision and
uncertainty, 1996.

Rete-OO Engine for Reasoning with Different Types
of Imperfect Information. *IEEE Trans. Knowl. Data