A Table-Based Representation for Probabilistic Logic: Preliminary Results

Simon Vandevelde,1,3 Victor Verreet,2,3 Luc De Raedt,2,3 Joost Vennekens1,3

1 KU Leuven, De Nayer Campus, Dept. of Computer Science, J.-P.-De Nayerlaan 5, 2860 Sint-Katelijne-Waver, Belgium
2 KU Leuven, Dept. of Computer Science, Celestijnenlaan 200, 3001 Heverlee, Belgium
3 Leuven.AI - KU Leuven Institute for AI, B-3000, Leuven, Belgium
s.vandevelde@kuleuven.be, victor.verreet@kuleuven.be, luc.deraedt@kuleuven.be, joost.vennekens@kuleuven.be

Abstract

We present Probabilistic Decision Model and Notation (pDMN), a probabilistic extension of Decision Model and Notation (DMN). DMN is a modeling notation for deterministic decision logic, which intends to be user-friendly and low in complexity. pDMN extends DMN with probabilistic reasoning, predicates, functions, quantification, and a new hit policy. At the same time, it aims to retain DMN’s user-friendliness to allow its usage by domain experts without the help of IT staff. pDMN models can be unambiguously translated into ProbLog programs to answer user queries. ProbLog is a probabilistic extension of Prolog flexible enough to model and reason over any pDMN model.

Introduction

ProbLog (De Raedt, Kimmig, and Toivonen 2007) is a powerful modeling tool that combines logical reasoning with probabilities. It supports many inference tasks, such as marginal and conditional probability calculations, allowing it to be used for problems such as Bayesian reasoning and inference in social networks (Gutmann, Thon, and De Raedt 2011). However, ProbLog rules are often difficult to interpret for domain experts with no familiarity with Probabilistic Logic Programming (PLP).

The Decision Model and Notation standard (DMN) (Object Modelling Group 2019) is a user-friendly notation for decision logic, published by the Object Management Group (OMG). The goal of DMN is to be readable and usable by all parties involved in the decision process (business people, IT experts, . . .), as well as being executable. Currently, it only allows the modelling of deterministic decision processes.

In this paper, we present a preliminary version of Probabilistic Decision Model and Notation (pDMN): a DMN-like notation for probabilistic logic that aims to combine DMN’s intuitive notation with ProbLog’s powerful probabilistic reasoning capabilities. Our goal is to close the gap between ProbLog experts and domain experts, by lowering the threshold to understand and interpret probabilistic models, and to possibly allow domain experts to create the models themselves. This work is similar in spirit to our earlier work on cDMN (Aerts, Vandevelde, and Vennekens 2020), that extended DMN with constraint programming.

The contributions of this paper are as follows:

1. the pDMN notation for probabilistic programming, which aims to be user-friendly;
2. the translation principles of pDMN into ProbLog;
3. an implementation of a ProbLog-based solver for pDMN.

This paper is structured as follows. First, we go over the DMN and ProbLog concepts necessary for this work. We then introduce our pDMN syntax, and elaborate on how it differs from standard DMN. We also present the translation principles of pDMN into ProbLog, and very briefly go over our implementation of a pDMN solver. Afterwards, we show a full pDMN implementation of a well-known example, to show pDMN in action. Finally, we conclude our preliminary work, and lay out the future work ahead.

Decision Model and Notation

In DMN, all logic is represented by decision tables. An example is shown in Fig. 1. In such a table, the value of the “output” variable(s) (in blue) is defined by the value of the “input” variable(s) (in green). All variables in DMN are either booleans (0-ary predicates), or constants (0-ary functions). Each row of a table represents a decision rule. A row is applicable if its input values match the actual values of the input variables. For example, if $BMI = 22$, the second row of the first table is applicable.

The table’s hit policy, as denoted in the top-left corner, further defines the behavior of the table. In U(nique) tables, only one row may be applicable for a set of input values. In A(ny) tables, multiple rows can be applicable, but they must all agree on the output value(s) they assign. Lastly, in F(irst) tables, the topmost applicable row is always selected. Besides these so-called single hit policies there are also multiple hit policies, which are out of the scope of this paper.

A variable which is an output in one table can be an input in another table. In this way, decisions can be chained together. For instance, starting from $BMI = 22$, we can first decide that $BMI_{Level} = Normal$ and then use the second table to decide that $Healthy = Yes$.

Another component in DMN, besides the decision tables, is the Decision Requirements Diagram (DRD). This is a graph that gives an overview of the structure of a DMN model. However, as this paper focuses in first instance on the decision tables, we will not discuss the DRD further.
Probabilistic Logic Programming

ProbLog is a probabilistic extension of Prolog. A ProbLog program consists of a set of probabilistic facts and a set of Prolog rules. Probabilistic facts are of the form $P_f :: f$, with $P_f \in [0, 1]$ a probability and $f$ an atom. The atom $f$ is true or false with probability $P_f$ and $1 - P_f$ respectively. Rules are written as $h :: b_1, b_2, \ldots, b_n$ where the atom $h$ is called the head and $b_i$ are the body atoms. The head of a rule may never occur in a probabilistic fact. Whenever all the atoms in the body of a rule are true, the head atom is true as well. A rule can also be annotated with a probability, but this is syntactic sugar for adding a unique atom to the body which is true with the annotated probability. Symbolically, the rule

$$P_r :: h :: b_1, b_2, \ldots, b_n$$

is translated into

$$h :: b_1, b_2, \ldots, b_n, f_r \quad \text{and} \quad P_r :: f_r$$

with $f_r$ a newly created atom. ProbLog also allows annotated disjunctions (ADs), written as

$$P_1 :: f_1; P_2 :: f_2; \ldots; P_n :: f_n$$

with $\sum_i P_i \leq 1$. An AD denotes a probabilistic choice where every atom $f_i$ is selected to be true with probability $P_i$, but at most one atom in the AD can be selected. If $\sum_i P_i < 1$ it is possible that none are true.

An interpretation is a truth assignment value to every atom occurring in the program. A model of the program is an interpretation that satisfies every rule and follows the closed world assumption. The closed world assumption states that an atom can only be true in a model whenever it can be derived through at least one rule. The probability of any model $M$ of the program is the product of the probabilities of the facts in the model. The probability of an atom $q$ is the sum of the probabilities of the models in which that atom is true. Hence,

$$P(q) = \sum_{M \models q} \prod_{f \in M} P(f)$$

where the sum runs over all models $M$ in which $q$ is true and the product runs over all the probabilistic facts $f$ in the model $M$. The probability $P(f)$ is the user given value $P_f$ if $f$ is true in the model $M$, and $1 - P_f$ otherwise. An example of a ProbLog program is given in Example 0.1.

Example 0.1. Consider the program

$$0.8 :: a. \quad c := a.$$
$$0.3 :: b(1); 0.5 :: b(2); 0.2 :: b(3). \quad c := b(1).$$

where we are interested in the probability of $c$. This program has 6 models, $\{a, b(1), c\}$, $\{a, b(2), c\}$, $\{a, b(3), c\}$, $\{not(a), b(1), c\}$, $\{not(a), b(2), not(c)\}$ and $\{not(a), b(3), not(c)\}$, where $c$ is only true in the first 4 models. Therefore, the probability of $c$ is

$$P(c) = 0.8 \cdot 0.3 + 0.8 \cdot 0.5 + 0.8 \cdot 0.2 + 0.2 \cdot 0.3 = 0.86$$

pDMN: Syntax

We now elaborate on the syntax of pDMN, our DMN extension for probabilistic logic programming. In pDMN, there are three types of tables: glossary tables, decision tables, and the query table.

Glossary

Variables in pDMN, in contrast to standard DMN, are typed $n$-ary functions and predicates. In order to correctly identify these variables and their arguments, pDMN introduces three glossary tables in which these should be declared: the Type table, the Predicate table and the Function table. These glossary tables contain the required meta-information to correctly interpret the pDMN model. This is analogous to the approach used in cDMN (Aerts, Vandeveld, and Vennekens 2020).

The Type table declares the types used in a pDMN model, together with their domain of elements. For example, the Type table in Fig. 2 declares a type Person, which consists of two elements, ann and bob, and a type Vaccine, which consists of the elements $a$, $b$ and $n(\text{one})$.

The Predicate table declares $n$-ary predicates. There is no fixed naming syntax for predicates; the arguments of a predicate are those types that appear in its description, and the remaining string is considered the predicate’s name. For example, in the glossary of Fig. 2, Person is infected represents a unary predicate is_ infected, which denotes for every Person (i.e., ann and bob) whether they are infected. Similarly, Person contacted Person is a binary predicate contacted that denotes contact between people.

The Function table declares $n$-ary functions. Analogously to predicates, the function’s name contains its arguments. In contrast to predicates, however, functions map their arguments to the type listed in the Type column of the glossary table, instead of to a boolean. For example, vaccine of Person denotes the Vaccine for each Person, i.e., it maps every person (ann and bob) to a vaccine ($a$, $b$, $n$).

Decision Tables

pDMN extends standard DMN decision tables with three new concepts: probabilities, the new Choice hit policy, and quantification. We will briefly touch on each concept, and show an example. Firstly, pDMN allows probabilities in the cells of an output column. For example, the $h1$ and $h2$ tables shown in Fig. 3a respectively define a probability of 0.5 and 0.6 to flip a coin on its head. Note that we use Yes and No to
represent true and false for predicates. In a table containing probabilities, the output values (such as Yes) are not listed in the rules directly, but rather in a separate row above the rules, which contains only output values. If the conditions of a rule are met, the probability of the output variable taking on a specific value is equal to the value that is listed below this output value in that particular row.

The second new concept is the Choice hit policy, which denotes that the output values for the output variable are mutually exclusive (i.e., only one can be assigned to the variable). This is demonstrated in the table in Fig. 3b, which states in its first row that an ordinary die has an equal 1/6 chance for any die value, and in its second row that a biased die has a higher chance of resulting in six. However, because of the Choice hit policy, the die can never e.g. be assigned both "one" and "two" at the same time. If, for instance, the table had the Unique hit policy, it would be possible to have an outcome in which the die has multiple face values at once.

The third and final addition in pDMN is quantification. For example, the Vaccine table shown in Fig. 3c expresses that “For every Person X, there is a chance of 36% that they have received vaccine a, a 63% chance on vaccine b, and a 1% chance of being unvaccinated.” The X here represents a quantification variable of type Person. Similarly, the Infection table expresses that every person X who had contact with an infected person Y could now also be infected, depending on their vaccine’s performance, or lack thereof.

Query

The Query table is the third type of table present in a pDMN model, and is used to denote which symbols’ probability should be calculated. Querying the probability of a predicate is done by adding it to the query table, either with specific elements of a type or with a quantification variable. To query a function, the table should contain a cell of the form fune_name(arg) = val. Here too, it is allowed to write down a specific element of a type or a quantification variable. Examples of query tables are shown in Fig. 4. The table in Fig. 4a verifies the probability of flipping two heads and some heads with coins. Fig. 4b demonstrates querying predicates with a specific variable value (bob), or a quantification variable (X). In the latter case, the probability of the predicate is calculated for every element of the type Person. Lastly, Fig. 4c, in which we want to know the probability that a die lands on a six, shows the querying syntax for functions.

Translating pDMN to ProbLog

To practically use pDMN models, we translate them into ProbLog. We will now go over the general translation principles. Intuitively, every row of a u-table represents a rule in ProbLog, with the input variables forming the body, and the output variable forming the head. If there are multiple output variables present, a rule is created for each of them. For example, the heads table in Fig. 3a translates to the ProbLog rules shown in (7). Note that the rows in which the output was No are not translated, as these do not need to be explicitly formulated in ProbLog due to the closed world assumption.

\[
\begin{align*}
twoHeads & := heads1, heads2. \\
someHeads & := heads1, heads2. \\
someHeads & := heads1, not(heads2). \\
someHeads & := not(heads1), heads2.
\end{align*}
\]  

If the output rows of a table contain probabilities, these are added to their respective ProbLog rules or facts. E.g., the h1 table in Fig. 3a translates to the fact 0.5 :: heads1.

As explained before, DMN also provides the f(first) hit policy. Consider again the heads table in Fig. 3a, except we now consider it as an F-table. To translate the first hit behaviour to ProbLog, for any row in the table we need to add the negation of all the previous rows to the body of the translation. To do this, dummy variables are introduced, representing whether a row has fired or not. The resulting ProbLog translation for this example is shown in (8), where r1, r2 and r3 represent the dummy variables.

\[
\begin{align*}
r1 & := heads1, heads2. \\
r2 & := heads1, not(heads2). \\
r3 & := not(heads1), heads2. \\
twoHeads & := r1. \\
someHeads & := r1. \\
someHeads & := r2, not(r1). \\
someHeads & := r3, not(r1), not(r2).
\end{align*}
\]  

Tables with the newly introduced Choice hit policy are translated into ProbLog’s annotated disjunctions. For example, the table shown in Fig. 3b assigns a value to the 0-ary die value function. In ProbLog, n-ary functions are represented by an \((n+1)\)-ary predicate, resulting in the unary die_value predicate:

\[
\begin{align*}
1/6 :: & \text{die_value(one)}; \ldots; 1/6 :: \text{die_value(six)} \\
& := not(biased). \\
1/5 :: & \text{die_value(one)}; \ldots; 1/2 :: \text{die_value(six)} \\
& := biased.
\end{align*}
\]  

Types declared in the Type table in pDMN are represented by unary predicates in ProbLog, as the latter is not a typed language. Additionally, the contents of the Elements
(a) Example pDMN implementation describing two coinflips.

<table>
<thead>
<tr>
<th>heads</th>
<th>heads1</th>
<th>heads2</th>
<th>twoHeads</th>
<th>someHeads</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I</td>
<td>0.5</td>
<td>I</td>
<td>0.6</td>
<td></td>
</tr>
</tbody>
</table>

(b) Example of a pDMN table with the “Choice” hit policy.

<table>
<thead>
<tr>
<th>Ch</th>
<th>biased</th>
<th>die value</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
<td>1/6</td>
<td>1/6</td>
</tr>
<tr>
<td>two</td>
<td>1/6</td>
<td>1/6</td>
</tr>
<tr>
<td>three</td>
<td>1/6</td>
<td>1/6</td>
</tr>
<tr>
<td>four</td>
<td>1/6</td>
<td>1/6</td>
</tr>
<tr>
<td>five</td>
<td>0.1</td>
<td>0.6</td>
</tr>
<tr>
<td>six</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

(c) Snippet of a pDMN model implementing infections with vaccination.

Figure 3: Snippets of various pDMN examples.

<table>
<thead>
<tr>
<th>Vaccine</th>
<th>Ch</th>
<th>vaccine of X</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.36</td>
<td>0.63</td>
</tr>
<tr>
<td>b</td>
<td>0.63</td>
<td>0.01</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Infection</th>
<th>U</th>
<th>X contacted Y</th>
<th>Y is infected</th>
<th>vaccine of X</th>
<th>X is infected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>n</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Yes</td>
<td>Yes</td>
<td>a</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
<td>b</td>
<td>0.2</td>
<td></td>
</tr>
</tbody>
</table>

The Query table is represented in ProbLog by query statements. For every cell of the table, a new query statement is added. For example, the three query tables shown in Fig. 4 translate to the following ProbLog statements:

\[
\begin{align*}
&\text{person(ann), person(bob).} \\
&\text{vaccine(a), vaccine(b). vaccine(n)} \\
&0.36 :: \text{vaccine}(X, a); 0.63 :: \text{vaccine}(X, b); \\
&0.01 :: \text{vaccine}(X, n) :- \text{person}(X). \\
&0.8 :: \text{injected}(X) :- \text{vaccine}(X, n), \text{injected}(Y), \\
&\text{contacted}(X, Y), \text{person}(X), \text{person}(Y). \\
&0.1 :: \text{injected}(X) :- \text{vaccine}(X, a), \text{injected}(Y), \\
&\text{contacted}(X, Y), \text{person}(X), \text{person}(Y). \\
&0.2 :: \text{injected}(X) :- \text{vaccine}(X, b), \text{injected}(Y), \\
&\text{contacted}(X, Y), \text{person}(X), \text{person}(Y). \\
\end{align*}
\]

The Query table is represented in ProbLog by query statements. For every cell of the table, a new query statement is added. For example, the three query tables shown in Fig. 4 translate to the following ProbLog statements:

\[
\begin{align*}
&\text{query(twoHeads).} \\
&\text{query(someHeads).} \\
&\text{query(vaccine.of(Person)(bob)).} \\
&\text{query(person.is_infected(X)).} \\
&\text{query(die.value(six)).} \\
\end{align*}
\]

The Query table is represented in ProbLog by query statements. For every cell of the table, a new query statement is added. For example, the three query tables shown in Fig. 4 translate to the following ProbLog statements:

\[
\begin{align*}
&\text{query(twoHeads).} \\
&\text{query(someHeads).} \\
&\text{query(vaccine.of(Person)(bob)).} \\
&\text{query(person.is_infected(X)).} \\
&\text{query(die.value(six)).} \\
\end{align*}
\]

The Query table is represented in ProbLog by query statements. For every cell of the table, a new query statement is added. For example, the three query tables shown in Fig. 4 translate to the following ProbLog statements:

\[
\begin{align*}
&\text{query(twoHeads).} \\
&\text{query(someHeads).} \\
&\text{query(vaccine.of(Person)(bob)).} \\
&\text{query(person.is_infected(X)).} \\
&\text{query(die.value(six)).} \\
\end{align*}
\]

The Query table is represented in ProbLog by query statements. For every cell of the table, a new query statement is added. For example, the three query tables shown in Fig. 4 translate to the following ProbLog statements:

\[
\begin{align*}
&\text{query(twoHeads).} \\
&\text{query(someHeads).} \\
&\text{query(vaccine.of(Person)(bob)).} \\
&\text{query(person.is_infected(X)).} \\
&\text{query(die.value(six)).} \\
\end{align*}
\]

The Query table is represented in ProbLog by query statements. For every cell of the table, a new query statement is added. For example, the three query tables shown in Fig. 4 translate to the following ProbLog statements:

\[
\begin{align*}
&\text{query(twoHeads).} \\
&\text{query(someHeads).} \\
&\text{query(vaccine.of(Person)(bob)).} \\
&\text{query(person.is_infected(X)).} \\
&\text{query(die.value(six)).} \\
\end{align*}
\]

The Query table is represented in ProbLog by query statements. For every cell of the table, a new query statement is added. For example, the three query tables shown in Fig. 4 translate to the following ProbLog statements:

\[
\begin{align*}
&\text{query(twoHeads).} \\
&\text{query(someHeads).} \\
&\text{query(vaccine.of(Person)(bob)).} \\
&\text{query(person.is_infected(X)).} \\
&\text{query(die.value(six)).} \\
\end{align*}
\]

The Query table is represented in ProbLog by query statements. For every cell of the table, a new query statement is added. For example, the three query tables shown in Fig. 4 translate to the following ProbLog statements:

\[
\begin{align*}
&\text{query(twoHeads).} \\
&\text{query(someHeads).} \\
&\text{query(vaccine.of(Person)(bob)).} \\
&\text{query(person.is_infected(X)).} \\
&\text{query(die.value(six)).} \\
\end{align*}
\]

The Query table is represented in ProbLog by query statements. For every cell of the table, a new query statement is added. For example, the three query tables shown in Fig. 4 translate to the following ProbLog statements:

\[
\begin{align*}
&\text{query(twoHeads).} \\
&\text{query(someHeads).} \\
&\text{query(vaccine.of(Person)(bob)).} \\
&\text{query(person.is_infected(X)).} \\
&\text{query(die.value(six)).} \\
\end{align*}
\]

The Query table is represented in ProbLog by query statements. For every cell of the table, a new query statement is added. For example, the three query tables shown in Fig. 4 translate to the following ProbLog statements:

\[
\begin{align*}
&\text{query(twoHeads).} \\
&\text{query(someHeads).} \\
&\text{query(vaccine.of(Person)(bob)).} \\
&\text{query(person.is_infected(X)).} \\
&\text{query(die.value(six)).} \\
\end{align*}
\]
If no Query table is present in a pDMN model, it is assumed that the probabilities of all symbols of the model should be queried. In such a case, a ProbLog query rule is generated for every entry in the Predicate and Function glossary tables.

### Implementation

To automatically translate pDMN models to ProbLog and execute them, the translation principles described earlier have been implemented in a solver\(^1\). This solver is largely based on the solver which we created in earlier work for cDMN (Aerts, Vandevelde, and Venekeke 2020), due to the similar nature of the notations. The input for the solver is a pDMN model in the form of a \(\text{xslx}\) spreadsheet. Concretely, the solver works in three steps.

First, it interprets all glossary tables in the spreadsheet, beginning with the Type table. For every entry, the solver creates internal Type objects, necessary to interpret the arguments used in the Predicate and Function tables. The solver then evaluates every decision table one-by-one, using a lex/yacc parser to parse every cell and transform them into a pDMN expression. For example, an expression of the form “vaccine of bob” is translated into “vaccine of person(bob)”.

Next, all decision tables, are converted into ProbLog rules in the manner described earlier. At the same time, the Query table is parsed and converted into ProbLog query statements.

Lastly, the generated specification is executed using ProbLog’s Python API, after which the queried probabilities are shown. In this way, the pDMN execution process consists of a closed pipeline between pDMN modelling and ProbLog execution.

The pDMN solver is available as a Python package, and can be downloaded from its PyPi repository\(^2\).

### Full example

In the previous sections, every example only consisted of limited snippets of pDMN models. To give a view of what a complete pDMN model looks like, this section shows a concrete implementation of the well-known Earthquake example. In this example, a house alarm can be triggered by a burglary, by an earthquake of a certain intensity (heavy, mild or none), or by a combination of the two. Both the burglary and the intensities of the earthquake have a probability associated with them. If the alarm rings, the neighbours John and Mary both could either call the home owner, or they could dismiss the alarm as incorrect and ignore it. We now want to find out the probabilities of either neighbour calling.

The pDMN model for this example is shown in Fig. 5, and consists of the glossary tables, five decision tables and a query table. In the glossary tables, we first introduce two types, Person and Intensity, which respectively represent the neighbours and the earthquake intensities. In the Predicate table, we declare four predicates: the 0-ary predicates burglary, alarm and anycalls, and the unary predicate Person calls. To denote the intensity of the earthquake, we make use of the 0-ary function earthquake, which will thus either be heavy, mild, or none.

Of the five decision tables, two are straightforwardly used to set the probabilities of a burglary and the earthquake intensities. As these concepts do not depend on anything, their decision tables contain no input columns. The Alarm table contains a rule for every possible combination of burglary and earthquake to represent the probability of the alarm triggering. Note that it does not contain a rule in which neither a burglary or an earthquake take place, as the alarm will never trigger in such a situation, thus allowing us to leave out that rule. The fourth decision table, named Calls, expresses that every person \(X\) has a certain probability to call the home owner, depending on whether the alarm rings. Finally, the last decision table defines anycalls = Yes whenever any person \(X\) calls.

To find the probability of each neighbour calling separately, and the probability of either of them calling, the Query table is added to the model in order to finish it. Translating this model to ProbLog using the pDMN solver results in the following code:

```plaintext
% facts
intensity(heavy), intensity(mild), intensity(none).
person(john), person(mary).

% Burglary
0.7 :: burglary.

% Earthquake
0.01 :: earthquake(heavy); 0.19 :: earthquake(mild);
0.8 :: earthquake(none).

% Alarm
0.9 :: alarm :- burglary, earthquake(heavy).
0.85 :: alarm :- burglary, earthquake(mild).
0.8 :: alarm :- burglary, earthquake(none).
0.1 :: alarm :- not(burglary), earthquake(none).
0.3 :: alarm :- not(burglary), earthquake(heavy).

% Calls
0.8 :: person_calls(X) :- alarm, person(X).
0.1 :: person_calls(X) :- not(alarm), person(X).

% anycalls
anycalls :- person_calls(X).
query(person_calls(X)).
query(anycalls).
```

We can also use the pDMN solver to execute the example, by running ProbLog directly. This results in the following output:

```plaintext
>>> pdmn Examples.xlsx -x -n "Earthquake"
{person_calls(mary): 0.501765, person_calls(john): 0.501765, anycalls: 0.6319415}
```
Table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>john, mary</td>
</tr>
<tr>
<td>Intensity</td>
<td>heavy, mild, none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>burglarly</td>
<td></td>
</tr>
<tr>
<td>alarm</td>
<td></td>
</tr>
<tr>
<td>Person calls</td>
<td>anycalls</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>earthquake</td>
<td>Type</td>
</tr>
</tbody>
</table>

Burglary

<table>
<thead>
<tr>
<th>Type</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ch</td>
<td>earthquake</td>
</tr>
<tr>
<td>heavy</td>
<td>mild, none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Calls</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>anycalls</td>
<td>anycalls</td>
</tr>
<tr>
<td>X calls</td>
<td></td>
</tr>
</tbody>
</table>

Alarm

<table>
<thead>
<tr>
<th>Type</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ch</td>
<td>alarm</td>
</tr>
<tr>
<td>anycalls</td>
<td></td>
</tr>
<tr>
<td>X calls</td>
<td></td>
</tr>
</tbody>
</table>

Query

<table>
<thead>
<tr>
<th>Type</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ch</td>
<td>anycalls</td>
</tr>
<tr>
<td>X calls</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Full pDMN model for the *Earthquake* example

**Conclusion**

This paper presents a preliminary version of pDMN, a notation for Probabilistic Logic Programming based on the DMN standard, which aims to combine ProbLog’s expressiveness together with DMN’s readability and user-friendliness. It extends DMN with probabilities, predicates, quantification, and a new hit policy to represent annotated disjunctions. We lay out the general translation principles of converting pDMN into ProbLog code, allowing for the execution of the pDMN models. These principles have also been implemented in an automatic conversion tool, which is available for general use. In future work, we plan on further extending the notation (e.g., with support for more hit policies), formalizing the complete pDMN semantics, extending the DRD to support probabilities and making a user-friendly interface for the system.

**Acknowledgements**

This research received funding from the Flemish Government under the “Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen” programme.

**References**


