Leveraging the Power of IDP with the Flexibility of DMN: a Multifunctional API

Simon Vandevelde*[1,3], Vedavyas Etikala[2,3], Jan Vanthienen[2,3], and Joost Vennekens[1,3]

[1] KU Leuven, De Nayer Campus, Dept. of Computer Science
J.-P- De Nayerlaan 5, 2860 Sint-Katelijne-Waver, Belgium
{s.vandevelde, joost.vennekens}@kuleuven.be
[2] Leuven Institute for Research on Information Systems (LIRIS), KU Leuven
{vedavyas.etikala,jan.vanthienen}@kuleuven.be
[3] Leuven.AI - KU Leuven Institute for AI, B-3000 Leuven, Belgium

Abstract. Decision Model and Notation (DMN) models are user-friendly representations of decision logic. While the knowledge in the model could be used for multiple purposes, current DMN tools typically only support a single form of inference. We present DMN-IDPy, a novel Python API that links DMN as a notation to the IDP system, a powerful reasoning tool, allowing the knowledge in DMN models to be used to its fullest potential. The flexibility of this approach allows us to build intelligent tools based on DMN unlike any other execution engine.

Keywords: Decision Model and Notation · Knowledge Base Paradigm · IDP · API · Python

1 Introduction

The Decision Model and Notation standard [9], designed by the Object Modeling Group (OMG), is a user-friendly, table-based notation for modeling decision logic. Its main goals are to make decision knowledge readable by everyone involved in the decision process (business people, IT experts), and to be executable. Since its start in 2015, DMN has quickly gained popularity in both industry [4,12,8] and academia [5,1].

Typically, DMN is used to automate day-to-day business decisions. Most DMN tools therefore focus on supporting the required functionalities for this specific use.

However, we believe that more ambitious uses of DMN are also possible. In particular, the knowledge that is contained in a DMN model could be used to build knowledge-based AI systems, that can implement various sorts of intelligent behaviour. Consider, for instance, a cobot tasked with assisting an operator in product assembly. It seems likely that the domain knowledge that such a cobot

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

would need can be expressed in DMN, and, moreover, doing so would allow the domain knowledge to be written and maintained directly by the operators themselves, instead of requiring programmers or knowledge engineers as middle men.

To actually implement such a system, the functionality of typical DMN tools does not suffice. For instance, the cobot would need to figure out which sensor input is necessary for specific operations, and such functionality is typically not available. In an effort to allow DMN models to be used in a more flexible way, a translation from DMN to the FO(·) language was presented in [5]. FO(·) is an extension of classical first-order logic, which serves as the input language for the IDP Knowledge Base System [6]. Following the approach of [7], IDP allows the same knowledge base to be (re-)used for different forms of logical inference, facilitating the development of flexible knowledge-based tools. The work of [5] allows the powerful logical inference algorithms of IDP to be applied to DMN models as well. However, to build truly useful intelligent systems, this alone does not suffice: it is also necessary to combine these different inference tasks in a suitable way. Moreover, this should be done using the concepts and terminology from the original DMN model (instead of those from the FO(·) theory that the DMN model is translated to behind the scenes).

To make this possible, we present the DMN-IDPy API: a versatile Python API that combines DMN as a notation with the IDP system as a reasoning engine. It aims to deliver the building blocks required to unlock more powerful and flexible uses for DMN models. In this way, the API facilitates the creation of systems that exhibit intelligent behavior, based on the user-friendly structure and format of DMN models.

This work is similar in spirit to previous work on the PyIDP API [15], which exposes the functionality of IDP to Python programmers, allowing also the knowledge base itself to be represented in a pythonic syntax, rather than the usual syntax of FO(·). The difference to our work is that we now bring DMN into the mix to allow the knowledge to be maintained by domain experts, rather than Python programmers.

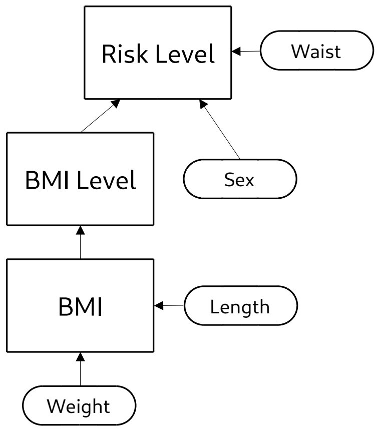
This paper is structured as follows. First, we elaborate on some background in Section 2. Next, we go over all functionalities of our DMN execution API in Section 3. Afterwards, Section 4 showcases a possible application of the API, in the form of a naive chat bot. We also briefly touch on the concrete implementation of the API itself in Section 5. Lastly, we compare our implementation to the current state-of-the-art in Section 6, and we conclude in Section 7.

2 Background

This section elaborates on the DMN standard, the current execution methods supported by the state-of-the-art DMN tools and the IDP system.

2.1 DMN

The Decision Model and Notation (DMN) standard provides a user-friendly notation for (business) decision knowledge. It consists of two main components: the Decision Requirements Diagram (DRD), and decision tables. The DRD is a graph representing the *decision flow* throughout the DMN model. It shows a graphical overview of which decision tables are present, how they connect, which input variables are used, which data sources are needed, and more. Fig. 1a shows an example of a DRD with three decision tables, as represented by the rectangles, and four input variables, as represented by the ovals. The arrows between them represents the flow of information, e.g., the value of *BMI* is defined by the value of the inputs *Weight* and *Length*.



(a) DRD

BMI

U	Weight	Length	BMI
1	—	—	Weight/(Length*Length)

BMI Level

U	BMI	BMILevel
1	< 18.5	Underweight
2	[18.5..25]	Normal
3	(25..30]	Overweight
4	> 30	Obese

Risk Level

U	BMILevel	Sex	Waist	RiskLevel
1	Normal	—	—	Low
2	Underweight	—	—	High
3	Overweight	Male	≤ 102	Increased
4	Overweight	Male	> 102	High
5	Overweight	Female	≤ 88	Increased
6	Overweight	Female	> 88	High
7	Obese	Male	≤ 102	High
8	Obese	Male	> 102	Very High
9	Obese	Female	≤ 88	High
10	Obese	Female	> 88	Very High

(b) Decision Tables

Fig. 1: Decision tables and DRD for the BMI running example.

The second main component consists of decision tables, as shown in Fig. 1b. Every decision table contains one or more input variables and one or more output variables, each corresponding to a column. A decision table defines the value of the output variables in term of the value of the input variables. Each row of the table corresponds to a decision rule. We say that a rule *fires* whenever the actual value of the input variables match the values listed in its cells. The way in which the inputs define the output depends on the *hit policy* of the table. This hit policy can either be single hit (such as *U(nique)*, *F(irst)*, *A(ny)*) or multiple hit (such as *C(ollect)*, *C+* and *C<*). If multiple rows fire at the same time, the hit policy specifies how these rows are combined to determine the value of the output variable.

The example DMN model shown in Fig. 1 consists of three tables in total, defining BMI, BMI Level and Risk Level. For these decisions, it uses four different input parameters: the weight, length, sex and waist size of a person. In this example, all tables have the U hit policy, meaning that only a single row can fire per table. E.g., if the value of *BMI = 23*, the second row in the *BMI Level* table fires, thus assigning *Normal* to the decision variable *BMILevel*. A cell containing "—" signifies that the value of this variable does not matter. For instance, if the BMI Level is underweight, the Risk Level is always high, regardless of sex and waist size.

2.2 Execution Methods

Since the introduction of DMN by OMG, software companies such as Camunda [3], OpenRules [10] and Signavio [11] offer decision modeling software based on this standard. Besides assisting the user in modeling and verifying decisions, some of them also provide execution mechanisms for the models.

Such execution goes back to Decision Table Solvers [14]. Practically, most of these tools all support the same execution method: the *bottom-to-top* approach. This execution method requires the user to input a value for every input variable present in the model, after which the execution engine decides the value of all other variables. For example, supplying a value for *Weight*, *Length*, *Waist* and *Sex* to find the value of *Risk Level*. While this is considered the standard usage of a DMN model, some tools also support additional execution methods.

One such method is *reasoning on sub-decisions*: instead of evaluating every decision table in a model, it is sometimes preferable to evaluate only a specific subset of decisions. If we are only interested in the *BMI Level*, for example, we do not need to evaluate the *Risk Level* table. The advantage of reasoning on sub-decisions is that not all input variables must be known (i.e. *Waist* and *Sex* are irrelevant as long as we do not need to know the *RiskLevel*). Examples of tools capable of this execution method are Camunda and OpenRules, both of which can evaluate a decision table in isolation. By reasoning on a single table at a time, they allow only evaluating the tables necessary for a sub-decision.

Another alternative execution method is the *"wildcard" mode*, such as the one provided by Camunda and Signavio, in which users can evaluate a decision model with partial input values. For example, if the value of *Sex* is unknown, a wildcard value can be used instead, in which case the engine returns a set of all possible output values.

2.3 IDP

The IDP system [6] is a powerful and flexible reasoning engine. As an implementation of the Knowledge Base Paradigm [7], it creates a clear distinction between knowledge and its use. Concretely, knowledge is stored in a so-called knowledge base (KB), written in an extended version of First-Order Logic (FO), called FO(·). As an example, (1) shows a possible FO(·) representation of the *BMI Level* table shown in Fig. 1b in the form of a conjunction of implications, as defined by the semantics of Calvanese et al. [2].

$$\begin{aligned}
&(\mathit{BMI} < 18.5 && \Rightarrow \mathit{BMILevel} = \mathit{Underweight})\\
\wedge&(18.5 \leq \mathit{BMI} \leq 25 && \Rightarrow \mathit{BMILevel} = \mathit{Normal})\\
\wedge&(25 < \mathit{BMI} \leq 30 && \Rightarrow \mathit{BMILevel} = \mathit{Overweight})\\
\wedge&(30 < \mathit{BMI} && \Rightarrow \mathit{BMILevel} = \mathit{Obese})
\end{aligned} \tag{1}$$

In FO(·), we represent each DMN variable by a constant c, with for every such constant a list of possible values $poss(c)$. A total assignment assigns to every constant precisely one value $c^I \in poss(c)$. A partial assignment assigns to every constant a non-empty subset $c^{\mathcal{I}} \subseteq poss(c)$ of its possible values $poss(c)$.

To reason on the knowledge in a KB, the IDP engine supports various inference tasks; three of these are used in this paper. To start, there is the *model expansion* task: given an assignment of values to some of the variables, compute an assignment to the other variables such that the knowledge base is satisfied. If the given variables are precisely the "input" variables of the model, this boils down to the standard "bottom-to-top" execution. However, we can also assign a value to a decision variable and then compute corrsponding values for the input variables. *Propagation* is the second inference task implemented in our API. Here, after assigning values to some variables, the IDP system generates (in-)equalities of the form $c\theta v$ with c a variable, v a value from $poss(c)$, and θ a comparison operator, that are now implied by the KB. E.g., if we add $\mathit{BMI} < 30$ to (1), propagation will automatically derive that $\mathit{BMILevel} \neq \mathit{Obese}$. The final inference task used in this work is *optimization*, which allows us to find a solution with the lowest/highest value for any given term.

Note that by using these inference tasks, we can do more than just bottom-to-top calculation. Indeed, the IDP system has no sense of direction: any variable can be used as "input" by assigning it a value. In this way, we can also use DMN tables "backwards", by going from the output variable to the input variables.

3 API features

This section aims at showcasing the features of the DMN-IDPy API. For every feature, we briefly mention what it is, why it is important and we show a short code snippet to show it in action.

3.1 Bottom-Up Decision Calculation

Our API can be used to provide the same “bottom-to-top” functionality as standard DMN tools. In the example shown in Fig. 1, this corresponds to setting the values for *Weight*, *Length*, *Sex* and *Waist* in order to then calculate the decisions in the following order: *BMI* → *BMILevel* → *RiskLevel*.

```
spec = DMN('bmi.dmn')
spec.set_value('weight', 74)
spec.set_value('length', 1.79)
spec.set_value('sex', 'Male')
spec.set_value('waist', 90)
```

→

```
>>> spec.model_expand(1)
Model 1
==========
riskLevel:={->Low}
waist:={->104}
BMILevel:={->Normal}
bmi:={->23.09540900720951}
sex:={->Male}
weight:={->74}
length:={->1.79}
```

3.2 Reasoning with Incomplete Information

Instead of requiring all input variables to have values assigned to them in order to run the execution, we also allow reasoning on DMN models with incomplete information. This functionality can e.g. be used to calculate the value of one or more sub-decisions without requiring the values of all input variables, thereby reducing the number of necessary operations. For example, if we are merely interested in the value of *BMILevel*, we should be able to perform this decision using only *Weight* and *Length* as inputs.

```
spec.set_value('Weight', 74)
spec.set_value('Length', 1.79)
spec.propagate()
```

→

```
>>> spec.value_of('BMI')
23.09540900720951
```

By supporting reasoning with incomplete information, every DMN model that consists of more than one table can directly and efficiently be used for multiple purposes by reasoning on sub-decision trees.

3.3 Relevance

One of the goals of our API is to allow *generic* tools to be built, by avoiding the need to hard-code which variables must be assigned a value and in which order this should happen. To this end, it allows to query on the fly which variables are relevant for making a certain decision. For example, because *BMI* is defined by *Length* and *Weight*, these latter two variables should both be known in order to decide the value of *BMI*. By implementing this functionality in the API, tools can be built with a more generic nature.

Note that by "inputs" we do not only mean the inputs of a decision table, but rather all *upstream* variables needed for a decision to be made. For example, while the *BMI Level* table only has one input variable, that variable in turn has two input variables. So, in reality, there are three dependencies for *BMI Level*, but at two different levels of the DRD. In the API, we show the number of *node hops* necessary to reach the variable to clearly denote this difference. This information is generated from the DMN file, without making use of the IDP system.

```
>>> spec.dependencies_of('BMILevel')
{'BMI': 0, 'Weight': 1, 'Length': 1}

>>> spec.dependencies_of('BMILevel')
 {'BMILevel': 0, 'BMI': 1, 'Weight': 2, 'Length': 2,
  'Sex': 0, 'Waist': 0}
```

As mentioned in Section 3.2, this can help optimize the required operations needed to decide a variable's value.

3.4 Multidirectional Reasoning

In our goal to get as much use out of a single DMN model as possible, the ability to reason on decisions in any direction is the functionality that results in the most mileage. Instead of only calculating in the direction of the arrows in the DRD (bottom-to-top), we can reason in the other direction as well. Among other things, it then becomes possible to calculate the input variables of the model based on the top-level decision.

To do this, the API supports directly assigning values to the decision variables. For example, if the value for BMI is already known beforehand, we can directly assign that value to the decision variable and use it to derive the value of BMI Level.

```
spec.set_value('BMI', 31)
spec.propagate()
```

→

```
>>> spec.value_of('BMILevel')
Obese
```

For an example of multidirectional reasoning, consider a person who just used the model to calculate that they are overweight, and now wants to query what their weight should be in order to reach a BMI of 25. By entering their length and their desired BMI value, the tool can calculate the weight required to reach their goal.

```
spec.set_value('BMI', '25')
spec.set_value('Length', 1.79)
spec.propagate()
```

→

```
>>> spec.value_of('Weight')
80.1025
```

Here, only a single value for *Weight* remains, because we set both *BMI* and *Length*. However, if we only set *BMI*, multiple values for *Weight* (and *Length*) are still possible, and no equality $Weight = x$ can be propagated. Indeed, instead of a single solution, we now have a *solution space*.

There are multiple ways to traverse this solution space in order to find a single solution. Assigning values to more variables will decrease the size of the space, possibly up until the point where there is only one solution left. If there are no variables left and there are still multiple solutions possible, we can generate solutions via IDP's model expansion inference (as demonstrated in the example in Section 3.1). Alternatively, we can search for the solution with the maximal/minimal value for a specific variable, as further explained in the Section 3.7.

3.5 Known variables

Because of the API's interactive approach, where any variable can be assigned a value at any time, it is important to be able to keep track of which variables are known, i.e., have been assigned a value either by the user or by the reasoning engine via propagation. Consider for instance a case where a user has calculated their BMI level as demonstrated in Section 3.2, by entering their length and weight. If they want to calculate their risk level afterwards, they should only have to enter their sex and waist, as that is the only information that is still missing for this decision.

```
spec.set_value('Length', 1.79)
spec.set_value('Weight', 79)
spec.propagate()
```

$\rightarrow$

```
>>> spec.is_certain('BMI')
True
>>> spec.is_certain('Sex')
False
```

3.6 Variable type and values

Every variable in a DMN model has a data type, such as *Int*, *Float*, *String* or other. Intuitively, these denote the type of data that a variable represents. To avoid errors such as assigning a numerical value to a variable of data type *String*, the API allows querying a variable's type via *type_of*.

String is a special case of data type: where *Int*, *Float*, etc are considered to have infinite ranges, *String* is often limited to a predefined list of possible values. Indeed, it makes sense that only those values that appear in a table can be assigned to a variable. E.g., in the BMI example the variable *Sex* can only be assigned values *Male* or *Female*. To prevent assigning impossible values to a string variable, the API can give a list of all possible values by either returning the variable's predefined list, or, if no list was predefined, by returning a list of all string values which appear at least once for that variable.

```
>>> spec.type_of('Sex')
String
>>> spec.possible_values_of('Sex')
['Male', 'Female']
```

3.7 Optimization

Optimization allows us to find the solution with the highest, or the lowest value for any given numerical variable. Consider a patient that has just entered their weight and length to find out that they have an *Overweight* BMI Level. A logical next question would be: “What should my target weight be if I want to have a normal BMI Level?”. To answer this, they can enter their length and set the value of *BMI Level* to *Normal*. If they then maximize the value of *Weight*, the system will calculate the maximum weight that still results in a normal BMI Level.

```
>>> spec.set_value('Length', 1.79)
>>> spec.set_value('BMILevel', 'Normal')
>>> spec.maximize('Weight')
Model 1
==========
RiskLevel:={->Low}
Waist:={->104}
BMILevel:={->Normal}
BMI:={->25}
Sex:={->Male}
Weight:={->80.1025}
Length:={->1.79}
```

4 Application Example

To truly showcase the power of combining DMN as a modelling tool and IDP as a reasoning engine, this section sketches a possible implementation for a naive chat bot, implemented in less than 25 lines of Python. Its main goal is to allow users to calculate any of the intermediary or top-level variables of a DMN model. In order to achieve this, the bot goes through a few steps. First, it fetches the list of variables and asks the user which variable should be calculated.

```
spec = DMN(sys.argv[1], auto_propagate=True)
vars = spec.get_outputs() + spec.get_intermediary()
req_var = input('Which variable to calculate? {}\n>'.format(variables))
```

Next, the program finds out which input variables should be known in order to make this calculation. Input variables without any effect on the value of the requested variable are not included.

```
deps = spec.dependencies_of(req_var)
missing_vars = [x for x in deps if x in spec.get_inputs()]
print("\nThe following variables are still unknown:")
print(missing_vars)
```

Finally, it loops over every unknown variable and queries the user for its value. Important here is that we ask a different question, based on the data type of the variable. Indeed, the user should be aware of the data type of the variable that is being queried. If the program requests the value of a *String*-based variable, it should also supply the user the list of possible values. Similarly for numerical variables, the user should be notified if the variable is an integer or a float.

```
for var in missing_vars:
    # Ask for the variable's value. Based on var type, ask different question.
    var_type = spec.type_of(var)
    if var_type in ['Real', 'Int']:
        msg = "Value for {} ({}) unknown.\n>".format(var, var_type)
    else:
        pos_vals = spec.possible_values_of(var)
        msg = "Value for {} unknown.\n"\
              "Possible values: [{}]\n>".format(var, pos_vals)
    value = input(msg)
    spec.set_value(var, value)

    if spec.is_known(req_var):
        break

req_var_val = spec.value_of(req_var)
print('Calculated value for {}:\n{}'.format(req_var, req_var_val))
```

Note that at the end of every loop cycle, the program checks whether the variable is known yet. While this might not make much sense at first, because the program specifically fetched the list of necessary inputs for the decision, there are cases where not all inputs might be necessary. Consider for example the decision table for *RiskLevel*. Here, if the values for *Weight* and *Length* are queried first and they lead to a BMI Level that is neither overweight nor obese, then the values of *Sex* and *Waist* will have no impact on this decision.

```
>>> python bot.py bmi.dmn
Which variable to calculate? ['RiskLevel', 'BMILevel', 'BMI']
> Risk Level
The following variables are still unknown:
['Weight', 'Length', 'Sex', 'Waist']
Value for Weight (Real) unknown.
> 79
Value for Length (Real) unknown.
> 1.79
Calculated value for Risk Level:
Low
```

While this implementation uses the BMI example, it is not limited to it. Indeed, by supplying a different DMN model when invoking the program, the chat bot can be used for different purposes. For example, after inserting a DMN model designed to calculate personal taxes, the chat bot is capable of reasoning in that problem field without having to change any code.

5 Implementation

This section briefly elaborates on the implementation of the DMN-IDPy API. To transform DMN models to input for the IDP system, it uses a tool developed in [1,13]. This tool accepts DMN models that are either in XML format (as specified by the DMN standard), or in the form of an Excel spreadsheet.

When using the API, a few internal steps are performed. To begin, as soon as a specification is entered, it is translated internally into the FO($\cdot$) format of the IDP system. This translation is done based on the decision table semantics as defined by Calvanese et al. [2], i.e., every table is represented by a conjunction of material implications. To run the IDP system, we use the *idp-engine*[4] Python package.

Whenever a variable is assigned a value, the underlying IDP specification is updated to represent this change. If the user invokes the *propagation* method, the API immediately runs IDP's propagation inference and updates the values of the other variables accordingly. Similarly, if they invoke the model expansion function, the API triggers IDP's model expansion inference.

The PyDMN-API library is available to download via the Python Package Index[5]. Furthermore, there is also a practical usage guide for the API available online[6]. Note that the API does not (yet) support the full DMN standard. Currently, it is capable of reasoning on tables with the following hit policies: *U*, *F*, *A*, *C+*, *C<* and *C>*. It supports the *Int*, *Float*, *Boolean* and *String* data types, but not e.g. the *Date* type. There is also no support for boxed expressions.

6 Comparison

To the best of our knowledge, there is no other approach that offers such a flexible yet powerful use of DMN models. While there exist tools that support more than exclusively the bottom-to-top calculation, none are capable of performing all features discussed in this work. Table 1 shows a comparison of the functionalities of DMN-IDPy, the OpenRules API and the Camunda API.

As expected, all compared APIs support the bottom-to-top execution. Additionally, they all also support reasoning on incomplete information, but only up to a varying degree. Both OpenRules and Camunda are capable of using incomplete information by reasoning on sub-decisions, as they allow the evaluation of a single decision table isolated from the rest of the DMN model. Thus, it is possible to e.g. use the Risk Level model to only calculate a patient's BMI, as discussed in the example in Section 3.2. However, as the API's only allow reasoning on either the entire model or a single specific table, attempting to reason on a sub-decision consisting of multiple tables (e.g. *BMI* followed by *BMILevel*) requires quite a bit of extra overhead: for each table, we would need to (a) manually supply the inputs, (b) evaluate, and (c) extract the outputs to use as inputs

[4] https://pypi.org/project/idp-engine/
[5] https://pypi.org/project/cdmn/
[6] https://cdmn.readthedocs.io/en/latest/DMN_guide.html

for the next table. In our API, no such workarounds are needed, as it suffices to enter all input values followed by calling the propagation inference. As such, the process of using sub-decisions with DMN-IDPy is much more streamlined.

The wildcard mode, as featured in e.g. Camunda, is possible in DMN-IDPy by leveraging its ability to reason on incomplete information. After entering a partial set of input values, we can generate all remaining solutions using the model expansion inference.

Neither OpenRules nor Camunda support multidirectional reasoning or optimization.

	DMN-IDPy	OpenRules	Camunda
Bottom-to-top	X	X	X
Incomplete Information	X	o	o
Wildcard mode	X		X
Multidirectional Reasoning	X		
Optimization	X		

Table 1: Comparison between functionalities of DMN-IDPy, and state-of-the-art DMN execution engines. (X = full support, o = partial support)

The main downside of our approach is the efficiency of the reasoning engine itself. Where other engines have specific optimized algorithms to perform the bottom-to-top calculation, we use a general purpose reasoning engine. As such, our calculation times will often be a magnitude higher compared to the other state-of-the-art engines. However, we feel that we make up for it with the increased flexibility that the API offers.

7 Conclusion and Future Work

While DMN models are most often used for bottom-to-top calculations, they can be used in many more scenarios. For this to be possible however, DMN needs to be supported by a flexible reasoning tool. In this paper, we present a Python API that enables the IDP reasoning system as an execution engine for DMN. This way, it provides the building blocks necessary to construct intelligent tools based on user-friendly DMN models. The main additions of the API are:

- Support for reasoning in any direction (e.g. going in the other direction of the DRD);
- Support for reasoning on incomplete data (allowing for sub-decision calculations);
- Addition of the optimization of variable values.

In order to showcase DMN-IDPy in action, we created a naive implementation of a chat bot in under 20 lines of Python code. The implementation is generic in the sense that it can be used with any DMN model, without having to change a line of code.

In future work, we will look into extending the API to support more of IDP's inference tasks. Moreover, we will also develop a more extensive, real-life application based on our API to further research its usefulness in a more realistic setting.

References

1. Aerts, B., Vandevelde, S., Vennekens, J.: Tackling the DMN challenges with cDMN: A tight integration of DMN and constraint reasoning. In: Rules and Reasoning, pp. 23–38. Lecture Notes in Computer Science, Springer International Publishing, Cham (2020)
2. Calvanese, D., Dumas, M., Laurson, U., Maggi, F.M., Montali, M., Teinemaa, I.: Semantics, analysis and simplification of DMN decision tables. Information systems (Oxford) **78**, 112–125 (2018)
3. Camunda Services GmbH: Camunda DMN Decision Engine. https://camunda.com/ (2013 – 2021)
4. Car, N.J.: Using decision models to enable better irrigation decision support systems. Computers and Electronics in Agriculture **152**, 290–301 (2018). https://doi.org/https://doi.org/10.1016/j.compag.2018.07.024, http://www.sciencedirect.com/science/article/pii/S0168169917313595
5. Dasseville, I., Janssens, L., Janssens, G., Vanthienen, J., Denecker, M.: Combining DMN and the knowledge base paradigm for flexible decision enactment. Supplementary Proceedings of the RuleML 2016 Challenge, vol. 1620. CEUR-WS.org (2016)
6. De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., Denecker, M.: Predicate logic as a modeling language: The IDP system. In: Declarative Logic Programming: Theory, Systems, and Applications, pp. 279–329. ACM Books (2018), $$Uhttps://doi.org/10.1145/3191315
7. Denecker, M., Vennekens, J.: Building a knowledge base system for an integration of logic programming and classical logic. vol. 5366, pp. 71–76. Garcia de la Banda, Maria, Springer (2008), $$Uhttps://doi.org/10.1007/978-3-540-89982-2
8. Hasic, F., Vanthienen, J.: From decision knowledge to e-government expert systems: the case of income taxation for foreign artists in belgium. Knowledge and information systems **62**(5), 2011–2028 (2020)
9. Object Modelling Group: Decision model and notation (2021), http://www.omg.org/spec/DMN/
10. OpenRules Inc.: OpenRules Decision Manager. https://openrules.com (2003 – 2021)
11. Signavio GmbH: Signavio Process Manager. https://www.signavio.com/ (2009 – 2021)
12. Sooter, L.J., Hasley, S., Lario, R., Rubin, K.S., Hasić, F.: Modeling a clinical pathway for contraception. Applied clinical informatics **10**(5), 935—943 (October 2019). https://doi.org/10.1055/s-0039-3400749
13. Vandevelde, S., Vennekens, J.: A multifunctional, interactive DMN decision modelling tool (2020)
14. Vanthienen, J., Dries, E.: Illustration of a decision table tool for specifying and implementing knowledge based systems. Int. J. Artif. Intell. Tools **3**, 267–288 (1994)

15. Vennekens, J.: Lowering the learning curve for declarative programming: A python API for the IDP system. PRACTICAL ASPECTS OF DECLARATIVE LANGUAGES (PADL 2017), vol. 10137, pp. 86–102. Springer Verlag (2017)