cDMN

Release 1.0.1

Jun 03, 2021

Full Documentation Contents:

1 1. What is cDMN? 1

2 2. How do I use it? 3
- 2.1 Notation . . . 3
- 2.2 Installation . . . 3
- 2.3 Usage . . . 3
- 2.4 Examples . . . 3

3 3. Can I also use it as a tool for standard DMN? 5

4 4. How to contribute 7

5 5. How to reference 9
- 5.1 cDMN Notation . . . 9
  - 5.1.1 1. cDMN vs DMN . . . 10
  - 5.1.2 2. Features . . . 10
  - 5.1.3 3. Tables . . . 11
  - 5.1.4 3.3 Constraint tables . . . 15
  - 5.1.5 3.4 Data tables . . . 15
  - 5.1.6 4. Goal . . . 16
- 5.2 cDMN Solver . . . 16
  - 5.2.1 1. Installation . . . 17
  - 5.2.2 2. Usage . . . 18
  - 5.2.3 3 Installation of the IDP system . . . 19
- 5.3 Examples . . . 20
- 5.4 cDMN implementations of DMCommunity Challenges . . . 20
  - 5.4.1 Other DMCommunity Challenges . . . 21
  - 5.4.2 Full implementations . . . 21
- 5.5 cDRD . . . 52
- 5.6 Python DMN API . . . 54
  - 5.6.1 Start . . . 54
  - 5.6.2 Query Information . . . 54
  - 5.6.3 Interacting with a DMN Model . . . 55
- 5.7 cDMN to FO() conversion . . . 57
  - 5.7.1 1. cDMN to Python conversion . . . 58
  - 5.7.2 2. Python to IDP conversion . . . 59
  - 5.7.3 3. cDMN to IDP example . . . 60

5.8 The cDMN developer reference . . . 62
5.8.1 The cdmn_solver API reference . . . 62
5.8.2 The DMN API reference . . . 62
5.8.3 The glossary API reference . . . 62
5.8.4 The interpret API reference . . . 65
5.8.5 The table API reference . . . 66
5.8.6 The table_operations API reference . . . 67
5.8.7 The idply API reference . . . 71
5.8.8 The parse_xml API reference . . . 72
5.8.9 The post_process API reference . . . 73

Python Module Index **75**

Index **77**

CHAPTER 1

1. What is cDMN?

The Constraint Decision Modeling and Notation language (or cDMN for short) is an extension to the DMN standard, which is managed by the Object Management Group (OMG). An explanation summary of DMN is listed below.

From the DMN website:

DMN is a modeling language and notation for the precise specification of business decisions and business rules. DMN is easily readable by the different types of people involved in decision management. These include: business people who specify the rules and monitor their application; business analysts.

DMN models are created by chaining together **decisions**. Every decision is defined by a **decision table**. For example, the table below shows how to decide the dish to cook based on the season and the amount of guests. Every decision table has input variables, and an output variable. Each row of a table signifies a possible combination of input variables, which define one or more outputs. This way, decisions can be represented in a readable manner. If for example Season is Winter and there are 10 guests, row 2 dictates that the Dish is Pasta.

cDMN extends standard DMN by adding constraint reasoning, variable quantification, more expressive data representations, and more. Because of these additions, cDMN has a couple advantages over DMN.

- Constraints increase expressivity and flexibility.
- Models scale better with input size.
- Models are more compact and straightforward.
- Modelling complex systems becomes possible.

cDMN is part of ongoing research by Bram Aerts, Simon Vandevelde and Joost Vennekens at the EAVISE group. If you are interested in learning more about how cDMN could help you and/or your company, contact us.

For those who want to read more about the theoretical side of cDMN, you can read our paper titled "Tackling the DMN Challenges with cDMN: a Tight Integration of DMN and constraint reasoning", which won the *Best Paper award* at the RuleML+RR2020 conference.

cDMN has also been presented at two conferences: the more theoretical RuleML+RR2020 presentation can be viewed here, and the more practical DecisionCAMP20 presentation can be viewed on their youtube.

CHAPTER 2

2. How do I use it?

2.1 Notation

The *cDMN Notation* page details all cDMN features and how to use them. If you are interested in learning cDMN, it is recommended to read/skim through this page, and then view the concrete cDMN examples to see cDMN in action.

Using the notation is straightforward: models can be created in any spreadsheet editing software you like. If you wish to execute your models, you can use our **cDMN solver**. Although, keep in mind that our solver currently only supports cDMN models in the `.xlsx` format.

2.2 Installation

The cDMN solver consists of a converter written in Python 3, and a knowledge reasoning engine called the IDP system. Information on the installation of the cDMN solver can be found here: *cDMN Solver*. The solver fully works on Windows, Mac and Linux. Furthermore, we are also planning on creating a browser-based tool, which everyone could run, regardless of OS.

2.3 Usage

More information on the usage of the cDMN solver can be found at *2. Usage*.

2.4 Examples

There is a list of examples Furthermore, there's a list of implemented decision modelling challenges, which can serve as a guide: *cDMN implementations of DMCommunity Challenges*.

CHAPTER 3

3. Can I also use it as a tool for standard DMN?

Yes you can! The cDMN solver supports everything in the normal DMN standard, besides boxed expressions and tables with the *C* hit policy. All other hit policies (*U*, *A*, *F*, *C#*, ...) are supported. The DMN can be either be in the form of a spreadsheet, or in the XML format as specified in the standard. Explanation on how to do this is given at the *2. Usage* page.

On top of our cDMN tool, we now bundle a Python DMN API. This API allows for querying the DMN specification for information (such as input names, output names, ...). On top of this, it also allows you to set values, and propagate them throughout the system. A guide on the usage of the API can be found at *Python DMN API*.

We also have a tool which integrates a DMN editor with an IDP-based user-friendly interface, called DMN-IDP. You can try it out for yourself at the online DMN-IDP demo.

CHAPTER 4

4. How to contribute

cDMN's source code is hosted on this GitLab repo under the GNU GPLv3 license. There's also documentation available for developers, over at *The cDMN developer reference*.

CHAPTER 5

5. How to reference

If you used cDMN in a publication or in other works, please reference us as follows:

BibTeX:

```
@incollection{AertsBram2020TtDC,
 series = {Lecture Notes in Computer Science},
 issn = {0302-9743},
 pages = {23--38},
 publisher = {Springer International Publishing},
 booktitle = {Rules and Reasoning},
 isbn = {9783030579760},
 year = {2020},
 title = {Tackling the DMN Challenges with cDMN: A Tight Integration of DMN␣
↪and Constraint Reasoning},
 copyright = {Springer Nature Switzerland AG 2020},
 language = {eng},
 address = {Cham},
 author = {Aerts, Bram and Vandevelde, Simon and Vennekens, Joost},
 }
```

Direct cite:

```
Aerts, Bram, et al. "Tackling the DMN Challenges with CDMN: A Tight Integration of␣
↪DMN and Constraint Reasoning." Rules and Reasoning, Springer International␣
↪Publishing, Cham, 2020, pp. 23-38. Lecture Notes in Computer Science.
```

5.1 cDMN Notation

The cDMN notation is based on the DMN standard, an open standard by the Object Management Group (who also specify the UML standard). If you already know standard DMN, learning cDMN shouldn't be too hard. If you don't know DMN, getting the hang of cDMN might take some practice. However, after making a few examples, you will be able to model your own problems in cDMN without any worries.

5.1.1 1. cDMN vs DMN

cDMN adds four main features:

- constraint reasoning;
- quantification;
- more expressive data;
- data tables.

These additions are further explained in the following subsections.

5.1.2 2. Features

2.1 Constraint reasoning

By allowing to set constraints in the models, more complex problems can be modeled in a readable manner. Moreover, constraint tables are not forced to set a default value when no rule applies. Because of this, rather than define a single solution, a cDMN implementation defines a solution space.

2.2 Quantification

Quantification allows for creating more compact and maintainable tables. Instead of a row only applying for a specific element, it can now apply to every element part of a specific subset.

2.3 More expressive data types

Standard DMN variables are 0-ary functions (known as constants). They require no argument (thus 0-ary) and return a single value. cDMN adds three more data types:

- functions;
- relations;
- booleans.

These allow for a more flexible usage of knowledge and for a better design of tables.

2.4 Data tables

When modelling a problem in cDMN, we divide it in two parts:

- the problem logic;
- the actual problem instance, i.e. what specific problem to solve.

Example

For the map coloring problem, in which we want to color a map so that no two countries share the same color, we have the following division:

- Problem logic: color a map so that no two countries share a color;
- Problem instance: the specific map to color.

In cDMN, the problem instances are represented via data tables. This allows for the application of a single cDMN model to multiple problems, by simply switching out the data tables.

5.1.3 3. Tables

3.1 Glossary

Before writing any business logic, rules or constraints, a specific set of terms need to be defined. We use multiple glossaries for this purpose, with each glossary a list of specific kinds of terms. In total there are 5 different glossaries:

Table	Explanation	Relevant Columns
Type	A type can be seen as a domain of possible elements.	Name, Type, Values
Function	Functions map an amount of types on a single type.	Name, Type
Constant	A constant maps a name on a single type.	Name, Type
Relation	Relations specify a connection between types.	Name
Boolean	Booleans map a type on either True, or False.	Name

More explanation of the 5 different glossaries can be found in the rest of the section, including examples.

3.1.1 Types

A type can be seen as a definition of a domain of possible elements. For example, all integer numbers can be grouped as one type. Or, all the people working at a company could be grouped as the type `Employees`.

cDMN knows 4 basic types, which can be used in the glossary without needing explicit definitions. All other types need to be derived from these basic four types (subtyping).

Type	Explanation
Int	An integer number, ranging from -∞ to +∞.
Float	A number containing a comma, ranging from -∞ to +∞.
String	Normal text.
Datestring	A representation for dates, in the OSI 8601 format. (this is currently experimental)

These basic types are used to create subtypes. This allows a more readable notation, and a reduction of possible values of a type.

For instance, in the following snippet the **type** glossary of the `Reindeer_Ordering` challenge is shown. Here, we define 3 types: `Reindeer`, `Place` and `Relative`.

Subtyping has 2 main advantages:

- The name of the type is easier to interpret (`string` vs `Reindeer`)
- It's possible to set a more narrow set of possible values of a type.

A `string` in itself could be anything: e.g. it could be `a`, `b`, `procrastination`, `yes`, etc. By subtyping `string` into the type `Reindeer`, we can narrow down the possible types to just the names of the reindeer. The same is done for `Place`, which is a subtype of integers. Subtyping here allows us to set a range of integers, which makes sure that there can never be a `Place` outside of the range `[1,9]`.

In the `Name` column, we write the name of our type. This name can be any string, but should be something descriptive of the type it's specifying.

In the `Type` column, we can refer to the supertype we'd like to use. This can be one of the 4 basic types, or another type that is already declared in the glossary.

In the last column, `Values`, we can write down all the elements that are containing in our type. For instance, for the type `Reindeer`, these are the names of all the reindeer. For `Place`, we define a range of integers, from one to nine. Thus, our range is defined as the following values: `1, 2, 3, 4, 5, 6, 7, 8, 9`.

A well-defined range can make the difference between a short and a long solving time.

The concept of subtyping can be pushed even further: it's possible to create subtypes of subtypes of subtypes, ad infinitum.

Note: For string, the column values can be left undefined by inserting a – or by referring to a data table containing those values. In both cases, there should be a data table defining the string. See *3.4 Data tables*.

Important: Subtypes of floats and integers should **always** have a defined range.

3.1.2 Functions

A function in cDMN is a mapping of arguments on a single type. This is especially useful when there's a functional link between two sets of elements. For example, a set of people can be easily mapped on a set of dates of birth using a function. Every person has exactly one date of birth, but a date of birth can have multiple people being born on it.

Note: The term "function" in cDMN is not be confused with "function" in programming. cDMN uses the mathematical meaning.

In the `Name` column, we write down the name for our function. However, this name can not just be anything: there is a specific syntax for functions. The syntax is fairly straightforward: it can be any name as long as the argument types are clearly part of it. For example, `value of Argument` is a function to map an instance of `Argument` on something else. It's possible to list multiple arguments, each of which should be a known type. For instance, in the Reindeer problem our Function glossary could look as follows.

The function `order of Reindeer` will assign a value of the type `Place` to each `Reindeer`. It makes sense to use a function for this purpose, as each reindeer needs a position, and can only have a maximum of one position. There is a direct connection between the two types.

Note: It's currently impossible to use `int` as a super type for a function, because it doesn't have a set range of values. You should always define a subtype of integers, and assign it a correct range.

Warning: Make sure that there are no other types in the name other than the argument types, or the solver will add them to the list of arguments. For example, `place of Reindeer` represents a function with two arguments, namely `Place` and `Reindeer`. In order to only have the `Reindeer` argument, write the `Place` with a lowercase: `place of Reindeer`.

Partial Function

A partial function acts like a normal function, except that not every argument needs to map to a value. Hence the name partial function.

> **Warning:** Take extreme caution when using a partial function. When quantifying over a partial function, make sure you only use the arguments for which a value is set in the partial function.

Constant

The constant is a special type of function, where no argument is used. It only contains one value, instead of a value for each type of argument. This can e.g. be used when defining a cost for a problem, as demonstrated in the following glossary snippet. Note how the keyword `of` isn't used.

3.1.3 Relations

A relation can be seen as a function and produces either `Yes` or `No`.

Unlike the function, the relation has no specific syntax. Any string can form a relation, as long as it contains type names. There can be as many types listed as needed, in every possible order with any possible word in between.

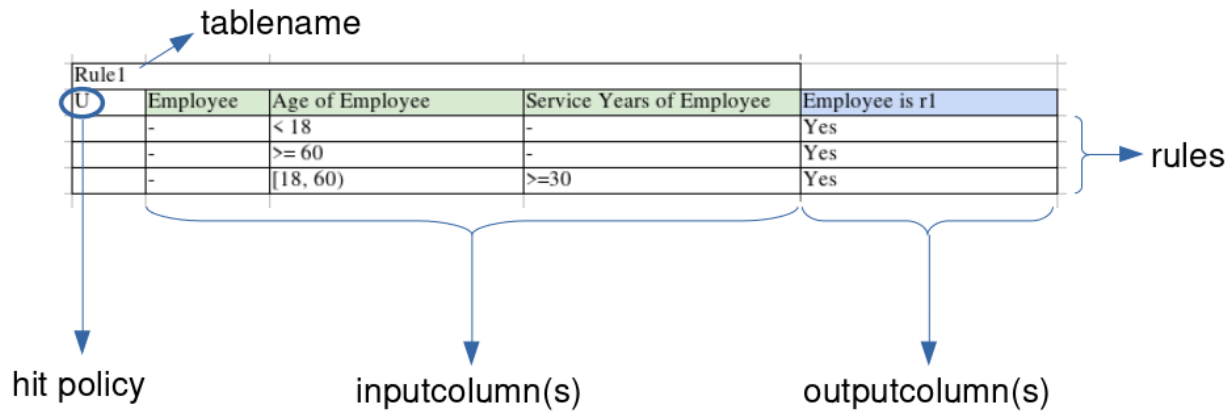
For example, in the `Map_Coloring` challenge, the glossary is as follows.

Boolean

A special type of relation is the boolean. It's a relation which doesn't have any arguments, and it can only be `true` or `false`.

3.2 Decision tables

The decision table is the bread and butter of DMN/cDMN. It allows the user to model a basic decision, in the form of a simple table. An example of an annotated decision table can be found below.



The table decides for every employee whether or not they are eligible for extra vacation days. The text it models is listed in the following example.

Example: vacation days eligibility

All employees under 18 years or over 60 get 5 extra vacation days (they apply for rule1). If an employee is between those ages, but has more or equal to 30 service years, they also get the extra days.

This example is implemented as three rows in the decision table. It checks for every employee (the – translates to a `don't-care`, thus for every employee), every age of employee and every number of service years of employee whether or not they're eligible for the vacation days.

A translation of the table rules to English would be as follows:

1. Every employee, younger than 18, with any amount of service years is eligible.
2. Every employee, older or equal to 60, with any amount of service years is eligible.
3. Every employee, between age 18 (included) and 60 (excluded) with more or equal to 30 years of service years is eligible.

Note: If for a specific employee none of the rules are satisfied, the relation `Employee is r1` is automatically considered as untrue. There is no seperate rule needed to define this.

This specific decision table has the `Unique` hitpolicy, denoted by the `U` in the top-left corner of the table. This means that at most only 1 row can be satisfied, and there is no overlap between the rows. For a table of other hitpolicies, see *3.2.1 Hitpolicies*.

Note: A decision table can have any amount of inputs (even zero), and needs at least one outputcolumn.

3.2.1 Hitpolicies

Decision hitpolicy	Explanation
U	**Unique**: only one rule can be satisfied.
A	**Any**: any of the rules can be satisfied.
F	**First**: only the first rule eligible is satisfied.
C+	**Sum**: sum the values of each satisfied row.
C<	**Minimum**: take the minimum value of each satisfied row.
C>	**Maximum**: take the maximum value of each satisfied row.
C#	**Count**: count the amount of satisfied rows.

For more examples on these hitpolicies, see the *Examples*.

3.2.2 Using the same type twice

In cDMN it's possible to use the same type as many times as you like in a single table. To differentiate between multiple instances of the same type, the syntax `Type called typename` is introduced. The following table shows an example.

In text, this table reads as: "For every Person p1 and Person p2, if p1 likes p2 and p2 likes p1, then p1 matches p2."

Note: Note how we have to explicitly define that p1 does not equal p2.

3.2.3 Syntax in cells

Inside a decision table cell, only a certain syntax is allowed. The below table shows every possible syntax that a cell can have.

Cell	Explanation
-	don't care: the rule goes for every possible value of the inputcolumn.
Yes/No	Can be used when trying to specify the status of a specific relation.
not(type)	(function): Specifies that it can't be the same value.
not(relation)	returns the inverted value of the relation.
val	usefull when you want a rule to only count for one specific value of a type.
<= x; >= x; < x; > x; = x	Used to compare two values.
[x,y]	inclusive range, the value needs to be between x and y or equal to x or y.
(x, y]	the value needs to be between x and y, or equal to y.
[x, y)	the value needs to be between x and y, or equal to x.
(x, y)	the value needs to be between x and y.
#Type	the number of elements in type `Type`.

5.1.4 3.3 Constraint tables

A constraint table is used when a list of constraints needs to be modeled. It can be seen as a decision table in which each row needs to be satisfied. It is denoted by the "Every" hit policy, as in "every rule needs to be satisfied". In tables, it is denoted by `E*`.

For instance, in the Map Coloring problem, we want to define that if two countries share a border, they should have a different color. In text, this is written as "For every country c1, country c2 holds that if c1 and c2 share a border, the color of c1 cannot equal the color of c2".

It's also possible to define multiple constraints, where each one needs to be satisfied. This is shown in the Monkey Business example.

This table translates into the following rules:

1. For each monkey, if the monkey is named Sam, then the place of the monkey is Grass and the fruit is not Banana.
2. For each monkey, if its location is Rock, then the fruit of the monkey is Apple.
3. For each monkey, if its fruit is Pear, the place of the monkey is not the Branch.
4. For each monkey, if the monkey is named Anna, its place is the Stream and it's fruit is not the Pear.
5. For each monkey, if the monkey is named Harriet, its place is not the Branch.
6. for each monkey, if the monkey is named Mike, then its fruit is the Orange.

5.1.5 3.4 Data tables

The data table allows the user to input data that doesn't really fit in a decision table. Although technically you could use the `Unique` hitpolicy to input data, a data table will always work faster. Another advantage is that not every instance of a type needs to be listed in the `values` column of the glossary.

Note: If any of the decision tables or constraint tables holds a specific type value (for instance, the name of the monkeys in the previous table) you still need to explicitly list those values in the glossary.

A simple example of a data table is the one found in Hamburger Challenge. Per ingredient, we input its sodium, fat and calories amount and the cost.

A data table can also do most of the things that works in a normal decision table/constraint table. For instance, it can also quantify over the same type multiple times.

As mentioned earlier, a data table also allows us to leave the `Values` column empty for a type in a data table. This is especially useful when we have a lot of data. For example, in the Balanced Assignment challenge we have a list of 210 people. Each person has a name, department, location, gender and title. Without a data table, each of these types needs to be listed in the `Values` column. Instead of having to type all this data, we can just copy-paste the employee list into a data table!

Caution: When using data tables to set the values of a function, you should make sure that it is complete. In other words, every possible set of inputs for the function should have a defined output.

5.1.6 4. Goal

In standard DMN, there is always only one solution possible for every set of inputs. In cDMN however, we do not define a single solution but rather a solution space. To specify which specific solution we want from this space, the `Goal` table is used. At the moment, there's three possible goals.

Inference	Explanation
Get x models	This will find x amount of solutions.
Minimize val	This will find the solution with the lowest value for val.
Maximize val	This will find the solution with the highest value for val.

An inference method can be specified using an `Goal` table. Some examples are given below.

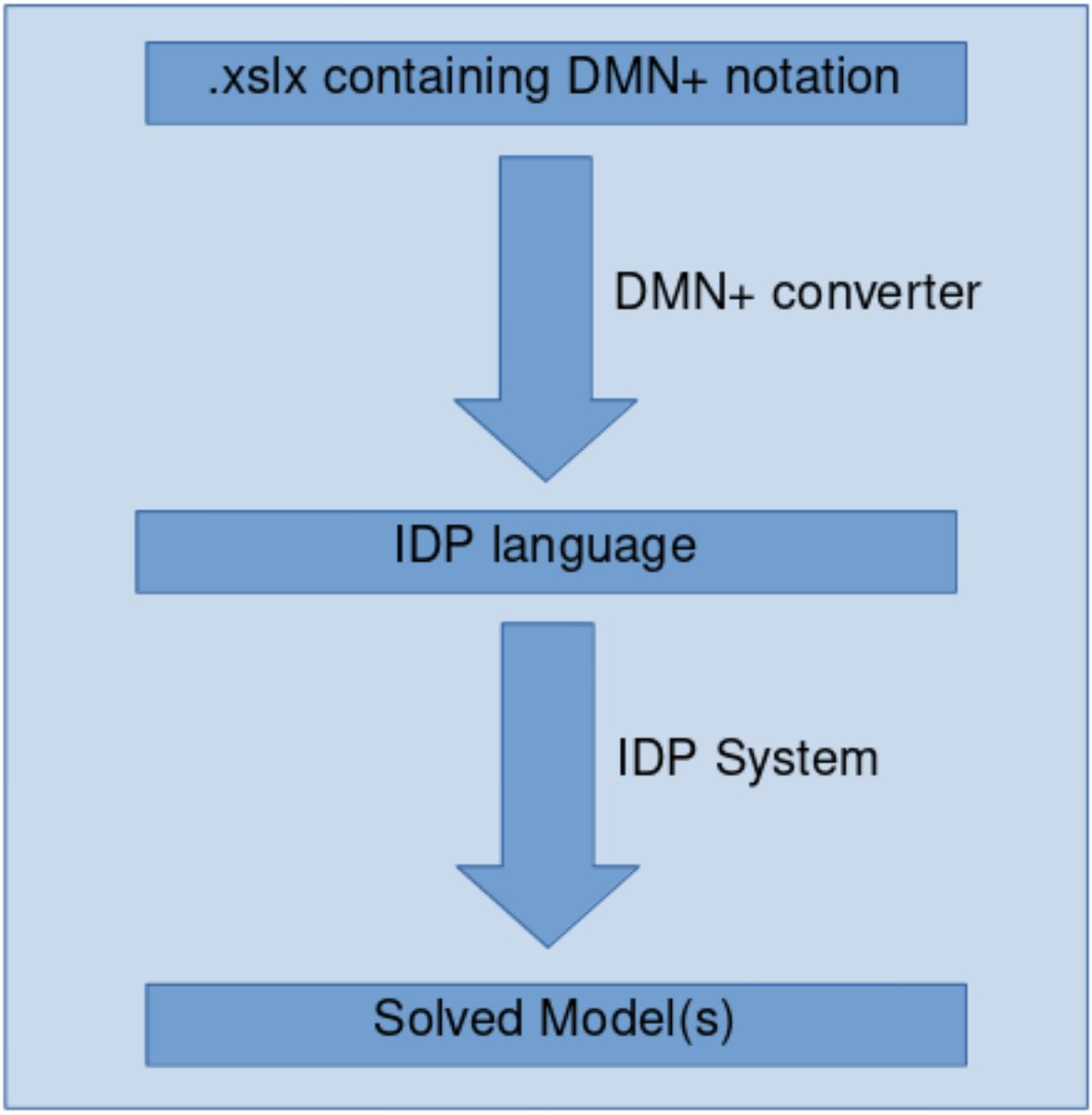
Note: If no `Goal` table is specified, cDMN will default to finding one model.

5.2 cDMN Solver

To be able to execute a spreadsheet, you need to use the cDMN solver. The cDMN solver consists of two main parts:

1. A Python converter from cDMN to IDP language;
2. The IDP system, which is a knowledge reasoning engine (can be run online or downloaded via Pip).

This is roughly shown on the figure below.



5.2.1 1. Installation

The code for the solver is available at our GitLab repository. It has been tested with Python 3.7=<. Other versions will probably work, but there's no guarantee. The list of dependencies is as follows:

- openpyxl
- ply
- numpy
- python-dateutil
- idp-solver

Installing the solver can be done via the Python repositories:

```
$ pip install cdmn
```

This comes with a Python-based version of the IDP system, called IDP-Z3. However, this version of IDP does not yet support all required functionalities (but it should cover most cases already). If you have a cDMN model that requires these, you can install the IDP3 system (see *3 Installation of the IDP system*)

5.2.2 2. Usage

The main usage is as follows:

```
usage: solver.py [-h] [-n name_of_sheet [name_of_sheet ...]] [-o outputfile]
                 [--idp idp] [--interactive-consultant-idp]
                 [--interactive-consultant-idp-z3] [--main]
                 [--errorcheck-overlap overlaptable]
                 [--errorcheck-shadowed shadowedtable]
                 [--errorcheck-rule ERRORCHECK_RULE]
                 [--errorcheck-gap ERRORCHECK_GAP]
                 path_to_file

Run cDMN on DMN tables.

positional arguments:
  path_to_file          the path to the xlsx or xml file

optional arguments:
  -h, --help            show this help message and exit
  -n name_of_sheet [name_of_sheet ...], --name name_of_sheet [name_of_sheet ...]
                        the name(s) of the sheet(s) to execute
  -o outputfile, --outputfile outputfile
                        the name of the outputfile
  --idp idp             the path to the idp executable
  --interactive-consultant-idp
                        generate file specifically for the IDP3 Interactive
                        Consultant
  --interactive-consultant-idp-z3
                        generate file specifically for the IDP-Z3Interactive
                        Consultant
  --main                create a main, to use when generating for the IDP-Z3
                        Interactive Consultant
  --errorcheck-overlap overlaptable
                        the table to check for overlap errors: table is
                        identified by table id
  --errorcheck-shadowed shadowedtable
                        the table to check for shadowed rules: table is
                        identified by table id
  --errorcheck-rule ERRORCHECK_RULE
                        the rule to check for being erronous
  --errorcheck-gap ERRORCHECK_GAP
                        the table to check for input gaps: table is identified
                        by table id
```

For example, to execute the sheet named `Problem1` in the `.xlsx` file located at `Data/DMNImplementation.xlsx`, the following command is used.

```
$ python3 solver.py Data/DMNImplementation.xlsx -n "Problem1" -o "~/idp/Files/"
```

This will convert a sheet called `Problem1` from the `DMNImplementation.xlsx` file and save it in as `~/idp/Files/Problem1.idp`. It's also possible to set the name of the idp name yourself, by changing the output to `~/idp/Files/custom_name.idp`.

If you wish to immediately execute the IDP specification, you can do so by supplying the `--idp-z3` flag. This will automatically run the IDP-Z3 system, and print its output.

```
$ python3 solver.py Data/DMNImplementation.xlsx -n "Problem1" -o "problem.idp" --idp-
↪z3
```

Some cDMN specification can get large, so sometimes it makes sense to create a logical separation in different sheets. It is possible to run multiple sheets together by supplying multiple names after the `-n` flag.

Create a file for the Interactive Consultant interface

The Interactive Consultant interface is an IDP-based, user-friendly interface which allows for straightforward interaction with a cDMN model. In order to create a file that can be used in the interface, the `interactive-consultant-idp` and `interactive-consultant-idp-z3` flags can be used, depending on the IC version.

```
$ python3 solver.py Data/DMNImplementation.xlsx -n "Problem1" -o "problem.idp" --
↪interactive-consultant-idp-z3
```

In the case that a file is being generated for use in the IDP-Z3 webIDE, the `--main` flag should also be supplied.

Use DMN as input

If you wish to execute DMN XML instead of a spreadsheet, you can do so by simply supplying the XML file as input.

Otherwise, it is also possible to create DMN tables inside a spreadsheet. In this case however, it is still necessary to create a glossary for the DMN specification. This can easily be done by filling out the Constant glossary table with all the variable names.

5.2.3 3 Installation of the IDP system

There's two options for the installation of the IDP system:

1. Use the online version, which is limited in input-output and processor usage.
2. Use an offline version, which is limited only by your own hardware (but only available for Linux or Mac).

If you want to only use the online version, you can skip the rest of this section. Most of the time the online version should be fine. Once the problems start requiring more CPU power, or if the server's unreachable, the offline version is the only way to run solve the problems.

To install the offline version, two components are needed: the actual IDP system, and the web interface for the system. A guide on downloading installing these can be found on the IDP website.

Run the IDP system

To execute an IDP file, open the webIDE by running the `webID` file or by surfing to the online IDE. The offline version should show up at `localhost:4004`. If you saved your IDP file in the work folder of the IDE, it should also show up under `Pick File`.

If you're using the online version, simply opening the IDP file in your favorite text editor and copy and pasting the contents to the IDE should be enough.

5.3 Examples

This page lists some examples for cDMN implementations. For now, most examples are based on DMCommunity challenges. In the future, other examples will be added as well.

The following table has a link for every example, together with a short list of the used concepts in the example.

Table 1: cDMN Examples

Example	Concepts
Who Killed Agatha	Quantification, Constraints, Functions
Vacation Days	Quantification, Functions, Relations
Vacation Days Advanced	Quantification, Functions, Relations, Data Table, Optimization
Hamburger Challenge	Quantification, Constraints, Data Table, Functions, Constants, Optimization
Monkey Business	Quantification, Constraints, Functions
Change Making	Quantification, Constants, Optimization
Doctor Planning	Quantification, Constraints, Functions, Relations, Optimization, Data Table
Balanced Assignment	Quantification, Functions
Map Coloring	Quantification, Constraints, Functions, Relations, Data Table
Map Coloring With Violations	Quantification, Constraints, Functions, Relations, Data Table, Optimization
Crack the Code	Quantification, Constraints, Functions, Constants, Complex Maths
Zoo, Buses and Kids	Quantification, Constraints, Functions, Constants, Optimization
Duplicate Product Lines	Quantification, Functions, Relation, Data Table
Calculator with Two Buttons	Quantification, Functions, Constants, Optimization
Virtual Chess Tournament	Quantification, Functions, Constraints, Aggregates
SET	Quantification, Functions, Relations, Data Table, Goal Table, Constraints, Aggregates
Covid Testing	Booleans, Constants, Aggregates
Where is Gold?	Relations, Constants, Aggregates

5.4 cDMN implementations of DMCommunity Challenges

For our publication on cDMN, we looked at DMN challenges posted on the DMCommunity website. In total, we looked at 21 challenges. Their cDMN implementations can be downloaded here. We created full explanations for some of the implementations, which can be found below.

Table 2: DMN Challenges

Challenge	Solved	Link to the description
Who killed Agatha?	Yes	https://dmcommunity.org/challenge/challenge-nov-2014/
Change Making Decision	Yes	https://dmcommunity.org/challenge/challenge-feb-2015/
Make a Good Burger	Yes	https://dmcommunity.org/challenge/make-a-good-burger/
Define Duplicate Product Lines	Yes	https://dmcommunity.org/challenge/challenge-august-2015/
Collection of Cars	Yes	https://dmcommunity.wordpress.com/challenge/challenge-sep-2015/
Monkey Business	Yes	https://dmcommunity.org/challenge/challenge-nov-2015/
Vacation Days	Yes	https://dmcommunity.wordpress.com/challenge/challenge-jan-2016/
Greeting a Customer	Yes	https://dmcommunity.wordpress.com/challenge/challenge-aug-2016/
Loan Approval	No	https://dmcommunity.org/challenge/challenge-dec-2016/
Online Dating Services	Yes	https://dmcommunity.org/challenge/challenge-march-2017/
Classify Department Employees	Yes	https://dmcommunity.org/challenge/challenge-sep-2017/
Soldier Payment Rules	No	https://dmcommunity.org/challenge/challenge-nov-2017/
Reindeer Ordering	Yes	https://dmcommunity.org/challenge/challenge-dec-2017/
Zoo, Buses and Kids	Yes	https://dmcommunity.org/challenge/challenge-july-2018/
Balanced Assignment	Yes	https://dmcommunity.org/challenge/challenge-sep-2018/
Vacation Days Advanced	Yes	https://dmcommunity.org/challenge/challenge-nov-2018/
Map Coloring	Yes	https://dmcommunity.org/challenge/challenge-may-2019/
Map Coloring with Violations	Yes	https://dmcommunity.org/challenge-june-2019/
Crack the Code	Yes	https://dmcommunity.org/challenge/challenge-sep-2019/
Numerical Haiku	Yes	https://dmcommunity.org/challenge/challenge-nov-2019/

5.4.1 Other DMCommunity Challenges

The solution file also contains other DM Community challenges, which weren't discussed in our paper. In the future, more should be added.

Table 3: Other DMN Challenges

Challenge	Link to the description
Nim Rules	https://dmcommunity.org/challenge/challenge-jan-2020/
Doctor Planning	https://dmcommunity.org/challenge/challenge-apr-2020/
Calculator with Two Buttons	https://dmcommunity.org/challenge/challenge-nov-2020/
Virtual Chess Tournament	https://dmcommunity.org/challenge/challenge-dec-2020/
Covid Testing	https://dmcommunity.org/challenge/challenge-may-2021/
Where is Gold	https://dmcommunity.org/challenge/challenge-june-2021/

5.4.2 Full implementations

Monkey Business

This page details the cDMN implementation of the Monkey Business challenge, as listed on https://dmcommunity.org/challenge/challenge-nov-2015/. The challenge is as follows:

Monkey Business

Mrs. Robinson's 4th grade class took a field trip to the local zoo. The day was sunny and warm — a perfect day to spend at the zoo. The kids had a great time and the monkeys were voted the class favorite animal. The zoo had four monkeys — two males, and two females. It was lunchtime for the monkeys and as the kids watched, each one ate a different fruit in their favorite resting place:

1. Sam, who doesn't like bananas, likes sitting on the grass.
2. The monkey who sat on the rock ate the apple. The monkey who ate the pear didn't sit on the tree branch.
3. Anna sat by the stream but she didn't eat the pear.
4. Harriet didn't sit on the tree branch. Mike doesn't like oranges.

Question: Can you determine the name of each monkey, what kind of fruit each monkey ate, and where their favorite resting place was?

The first thing we should do, is create the glossary. We have three types: one for the monkeys, one for the places, and one for the fruits.

Now we need a way to assign a fruit and a place to each monkey. Since every monkey needs exactly one fruit, and exactly one resting place, functions are perfectly suited for this task. A function is a mapping of a set of arguments to a return value. For instance, the function `Place of Monkey` will map each monkey on exactly one place.

That's it! Our glossary is fully done. We can now focus on the logic aspect.

In the challenge description, there were two main pieces of info:

1. The hints of which monkey ate what fruit where.
2. The fact that no monkeys can have the same place or the same fruit.

Both of these are easily turned into constraint tables. For instance, we can state that "For every monkey m1 and m2 (different from m1) must hold that they have a different Place and a different Fruit."

Creating a table for the other piece of info is a bit more work, but is still fairly simple. The full table containing all snippets of information looks like this:

Rule 1 states that Sam sat on the Grass, and didn't eat the Banana. Similarly, rule 2 states that the monkey who sat on the rock, ate the Apple. Every row in this constraint table represents a snippet of information.

And that's it! Using two glossary tables, and two constraint tables, we were able to solve the challenge. Running the cDMN solver gives us the following output:

```
Model 1
=======
structure  : V {
  Fruit = { Anna->Orange; Harriet->Apple; Mike->Banana; Sam->Pear }
  Place = { Anna->Stream; Harriet->Rock; Mike->Branch; Sam->Grass }
}
```

Change Making

This page details the cDMN implementation of the Change Making challenge, as listed on https://dmcommunity.org/challenge/challenge-feb-2015/. The challenge is as follows:

Change Making

Let S be a given sum that we want to achieve with a minimal amount of coins in denominations of x1, x2, ..., xn. Here is a simple example: S is 123 cents, n is 4, x1 is 1 cent, x2 is 10 cents, x3 is 25 cents and xn is 100 cents.

This is a pretty cool example, as it shows the mathematical power of the cDMN solver. It is also a great example to show how cDMN is able to optimize values.

We start by making the glossaries. Firstly, we're going to need a type to represent a range of numbers. We need to do this, because the cDMN solver can't reason with unbounded numbers.

To represent the number of coins, we will use a constant for each denomination. We will also use a constant to keep track of the total amount of coins, and the total amount of money.

Now that our glossary is finished, we can move on to the decision tables. We will be creating three tables which consist exclusively of output columns.

The first one is the total amount of money which we want to find the optimal assignment for. This is easily represented as:

Next up, we want a way to represent the following formula: `TotalMoney = OneCent * 1 + TwoCent * 2 + ...`. Using a decision table with the `C+` hit policy, we can model this as:

Similarly, to count the amount of total coins:

Now all that is left to do, is specify what we want to do with this model. There are multiple solutions possible, but we want the solution with the optimal amount of coins By creating a `Goal` table, we can tell the system to optimize the TotalCoins constant.

And that's it! With just 2 glossary tables, 3 decision tables and a `Goal` table, we were able to model an optimal change making algorithm. If we run this in the cDMN solver, we get the following output:

```
Number of models: 1
Model 1
=======
structure  : V {
  FiftyCent = 0
  FiveCent = 0
  OneCent = 1
  OneEuro = 1
  TenCent = 0
  TotalCoins = 4
  TotalMoney = 123
  TwentyCent = 1
  TwoCent = 1
  TwoEuro = 0
}

Elapsed Time:
0.122001
```

If we want to look for the optimal way to form 567 cents, the cDMN solver outputs the following:

```
Number of models: 1
Model 1
=======
structure  : V {
  FiftyCent = 1
  FiveCent = 1
  OneCent = 0
  OneEuro = 1
  TenCent = 1
  TotalCoins = 7
  TotalMoney = 567
  TwentyCent = 0
```

(continues on next page)

(continued from previous page)

```
  TwoCent = 1
  TwoEuro = 2
}

Elapsed Time:
0.128642
```

Who Killed Agatha

The Who Killed Agatha example can be found at the DMCommunity website.

Who killed Agatha

Someone in Dreadsbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadsbury Mansion, and are the only ones to live there. A killer always hates, and is no richer than his victim. Charles hates noone that Agatha hates. Agatha hates everybody except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone whom Agatha hates. Noone hates everyone. Who killed Agatha?

We can convert the challenge description into the following six clear rules:

1. A killer always hates, and is no richer than it's victim.
2. Charles hates noone that Agatha hates.
3. Agatha hates everybody but the butler.
4. The butler hates everyone not richer than Agatha.
5. The butler hates everyone whom Agatha hates.
6. Noone hates everyone.

The first step when designing a cDMN solution, is creating the glossary. In our glossary, we define all the elements which we'll use in our cDMN implementations.

After having read the description of the challenge, it should be clear that we need at least a `Person` type. This type will serve as the domain containing all the people's names.

We also need a way to declare whether or not a person is hated by another. Since a person can hate multiple people, it is not possible to use a function in this situation. Thus, we introduce a relation instead: `Person hates Person`. The same goes for declaring whether or not a person is richer than another. For this, we introduce the relation `Person richer than Person`.

Lastly, we also need a way to find out who our killer is. A constant lends itself perfectly for this, since there is exactly one killer.

Assuming that our glossary is finished, we can start modeling each of the rules.

Rule 1

A killer always hates, and is no richer than its victim.

Here we need to specify that a killer hates its victim as well as isn't richer than the victim. This can easily be done using a constraint table. We state that the Killer has to hate Agatha, and cannot be richer than her. Note how it is possible to construct a constraint table without input variables. In this case, the outputs are always applicable.

"Killer hates Agatha" and "Killer richer than Agatha" need to be true.

Rule 2

Charles hates noone that Agatha hates.

This rule is also best implemented using a constraint table. We can state that for every person, if Agatha hates them, then Charles doesn't.

For every person for whom "Agatha hates Person" is true, "Charles hates Person" is false.

Rule 3

Agatha hates everybody but the butler.

This can again be modeled using a constraint table. We evaluate each person, and if they are not the butler, Agatha hates them (row 1). If they are the butler, Agatha doesn't hate them (row 2).

Rule 4

The butler hates everyone not richer than Agatha.

We use another constraint table for this rule. We evaluate each person not richer than Agatha, and have the Butler hate that person.

Rule 5

The butler hates everyone whom Agatha hates.

Here we quantify over each person whom Agatha hates, and make "Butler hates Person" true. Coincidentally, we already have a table which has the same input. We can add an extra output column to "Charles hates noone that Agatha hates" and save a table this way.

Rule 6

Noone hates everyone.

For this rule, we need to define a way to know who "everyone" is, and how to model this. A possible solution is to introduce a new function, which will keep track of how many people are hated by one person. For instance, we could extend our `types` glossary with a type called `Number` and then create a `Function` glossary, containing a function to keep track of the number of hated people for each person.

This would then allow us to create two tables: one counting how many persons are hated, and one putting a constraint on that amount. To count how many are hated, we can make use of the `C#` hitpolicy. We need to evaluate every p1, and count the amount of p2 whom p1 hates. Afterwards, we use a `E*` table to make sure nobody hates more than 3 people. In cDMN, the `#Type` operator can be used to count the number of elements in a type. In this case, we use `#Person` to denote the number of people in the problem.

The end

Run the example.

Now that all the rules are encoded in tables, we can run the example and discover the mystery of Dreadsbury mansion! Running the solver on our created model results in the following:

```
Number of models: 1
Model 1
=======
structure  : V {
    Person_hates_Person = { Agatha,Agatha; Agatha,Charles; Butler,Agatha; Butler,
↪Charles; Charles,Butler }
    Person_richer_than_Person = { Butler,Agatha }
    Hated_persons = { Agatha->2; Butler->2; Charles->1 }
    Killer = Agatha
}

Elapsed Time:
0.09528
```

It turns out Agatha hates herself, and was her own killer!

Hamburger Challenge

This example is also taken from dmcommunity.org. It's called Make a Good Burger.

Hamburger Challenge

A burger most include at least one of each item, and no more than five of each item. You must use whole items (for example, no half servings of cheese). The final burger must contain less than 3000 mg of sodium, less than 150 grams of fat, and less than 3000 calories. To maintain certain taste quality standards, you'll need to keep the servings of ketchup and lettuce the same. Also, you'll need to keep the servings of pickles and tomatoes the same. Below is a list of ingredients and their information.

Try to find the most, and least expensive burger possible using the following items.

Item	Sodium (mg)	Fat (g)	Calories	Cost
Beef Patty	50	17	220	0.25
Bun	330	9	260	0.15
Cheese	310	6	70	0.10
Onions	1	2	10	0.09
Pickles	260	0	5	0.03
Lettuce	3	0	4	0.04
Ketchup	160	0	20	0.02
Tomato	3	0	9	0.04

As always, we start by filling out the glossary. From the description of the problem, we immediately know that we will need a type for the items. Also, thinking ahead, we will also need a type for every number used in the problem (such as the number of calories). For this purpose, we create `Item`, and `Nat` (for natural number). Because having to type over every item in the `Values` column is cumbersome, we refer to the data table in which we will be writing down all the food names.

Next up, we need a way to assign each item their nutritional information. Since each item can only have one value for sodium, fat, calories and cost, we can model these best as functions!

We also need a way to find out the total amount of each nutritional value. Since there is only one burger, and only one total amount per attribute, we use constants here.

Now that our glossary is done, we can move on to the next step in modeling the challenge: the data.

Data

Inputting the nutritional values

In our glossary entry for `Item`, we wrote `see DataTable Nutritions`. This tells the cDMN solver to add all the values in the `Item` column of the data table to the list of possible values. Creating the data table is easy. We have one input column, `Item`, and 4 output columns (one for every nutritional value). This greatly shows one of the advantages of data tables: if the list of nutritional values is supplied in table format or as a CSV, all we need to do is copy and paste the values and change the header names.

Rule

The burger must contain at least one of each, and no more than five of each item.

This rule can be modeled as a simple constraint table. We evaluate every item, and specify that their amount is in the range of 1 to 5.

Rule

The amount of lettuce and ketchup is the same, as well as pickles and tomatoes.

Once again, a contraint table suffices. In fact, we can even reuse our previous table by adding two rows. We evaluate each item of lettuce, and make sure that their amount is the same as the amount of ketchup, and repeat this for the pickles and the tomatoes. We could even add these rules to the previous table, since the two rules use the same input and output column.

Rule

The burger must contain less than 3000mg of sodium, less than 150g of fat and less than 3000 calories.

This rule is a bit more complex, because we need to calculate these total values first. For each item, we calculate their number multiplied by their nutritional value and add all of those up to get the total, using the `C+` hit policy. Then we use another `E*` to make sure we don't cross the nutritional thresholds. Note how this second table doesn't have any inputcolumns. Because we use constants, no inputs are needed.

Rule

Find the most expensive burger.

To maximize the total cost, we can add a `Goal` table. In this table, we need to specify the term we want to minimize, which in this case, is `Total Cost`.

We can now run our model using the cDMN solver. This results in the following output:

```
Number of models: 1
Model 1
=======
structure  : V {
  Calories = { Beef_Patty->220; Bun->260; Cheese->70; Ketchup->20; Lettuce->4; Onions-
↪>10; Pickles->5; Tomato->9 }
```

(continues on next page)

(continued from previous page)

```
  Cost = { Beef_Patty->25; Bun->15; Cheese->10; Ketchup->2; Lettuce->4; Onions->9;␣
↪Pickles->3; Tomato->4 }
  Fat = { Beef_Patty->17; Bun->9; Cheese->6; Ketchup->0; Lettuce->0; Onions->2;␣
↪Pickles->0; Tomato->0 }
  Number = { Beef_Patty->1; Bun->1; Cheese->1; Ketchup->1; Lettuce->1; Onions->1;␣
↪Pickles->1; Tomato->1 }
  Sodium = { Beef_Patty->50; Bun->330; Cheese->310; Ketchup->160; Lettuce->3; Onions->
↪1; Pickles->260; Tomato->3 }
  Total_Calories = 598
  Total_Cost = 72
  Total_Fat = 34
  Total_Sodium = 1117
}

Elapsed Time:
0.050393
```

We are able to make a burger which only costs 72, and still meets (meats? :-)) all the requirements.

Vacation Days

This example is another challenge taken from dmcommunity.org, namely Vacation Days Calculation. In this challenge, we need to calculate vacation days for employees, based on their age and years of service. The description is as follows:

Vacation Days

The number of vacation days depends on age and years of service.

Every employee receives at least 22 days. Additional days are provided by the following criteria:

1) Only employees younger than 18 or at least 60 years, or employees with at least 30 years of service will receive 5 extra days.

2) Employees with at least 30 years of service and also employees of age 60 or more, receive 3 extra days, on top of possible additional days already given.

3) If an employee has at least 15, but less than 30 years of service, 2 extra days are given. These 2 days are also provided for employees of age 45 or more. These 2 extra days can not be combined with the 5 extra days.

To solve this challenge, there's two main ways:

1. Convert all these rules to one big C+ table, and count the number of days for each employee.
2. Convert each rule into its own table and relation, and then count the days at the end for each rule satisfied.

Both have been implemented in our available cDMN.xlsx , but only the second one will be explained here. By first checking which rules are satisfied and then calculating the days, we gain readability and traceability in our implementation.

We start modeling the example by building the glossary. First of all, we need one or more types representing the number of vacation days, the age and the service years. We set the range for vacation days between 22 and 32, as these are the theoretical minimum and maximum. For age and service years, we pick between 0 and 100. Remember, choosing a well defined range can be a major improvement in speed of the solver. On the flipside, if your range does not include a value which is needed, errors will show up.

In addition, we will also need a type to represent employees. For the example, we create three dummy employees: Huey, Dewey and Louie.

After creating the types, we can move on to the next sub-glossary. We need a way to map each employee on their information: since every employee can only have exactly one age, amount of service years and amount of vacation days, we opt for functions.

Thirdly, we need a way to express which rules are satisfied for which employee. For this purpose, we introduce a relation for each rule.

Now that our glossary is complete, we can start creating decision and constraint tables. The first tables we should make, are the ones for the extra vacation days. There's three such rules, and thus, three such tables.

Rule 1

Only employees younger than 18 or at least 60 years, or employees with at least 30 years of service will receive 5 extra days.

The easiest way to model this rule (and the following ones) is by using a `U` table. The rows of the table can be translated to the following:

1. Every employee younger than 18 years is eligible.
2. Every employee over the age of 60 is eligible.
3. Every employee between 18 or 60 with at least 30 years of service is eligible.

Rule 2

Employees with at least 30 years of service and also employees of age 60 or more, receive 3 extra days, on top of possible additional days already given.

We opt for another `U` table here. It is similar to the previous table: it uses the same input columns, but a different output column. The rows can be translated to the following:

1. Everyone with more than 30 service years is eligible for rule 2.
2. Everyone older than 60, with less than 30 years of service are also eligible.

Rule

If an employee has at least 15 but less than 30 years of service, 2 extra days are given. These 2 days are also provided for employees of age 45 or more. These 2 extra days can not be combined with the 5 extra days.

Again, the `U` table suits our needs perfectly. We ignore the last sentence of the rule, as we can implement this later. The rows in the following table can be translated as such:

1. Everyone younger than 45 with at least 15 but at maximum 30 service years is eligible.
2. Everyone older than 45 is also eligible.

Now all that we need to do is add all the days of every applicable rule together, and count them in a `C+` table. This table evaluates every row, and sums together the outputs of the satisfied rows. Here we express in row 4 that the 2 days of rule 3 are only counted if the employee is not eligible for rule 1.

To now check the amount of vacation days our employees get, we can insert a data table containing their information, and run the solver.

This gives us the following solution:

```
Number of models: 1
Model 1
=======
structure  : V {
    Age = { Dewey->70; Donald->46; Huey->17; Louie->35 }
    Service_Years = { Dewey->31; Donald->10; Huey->1; Louie->16 }
    Vacation_Days = { Dewey->30; Donald->27; Huey->30; Louie->27 }
}

Elapsed Time:
0.059808
```

Vacation Days Advanced

This example is an advanced version of a previous example, *Vacation Days*. The full specification can be found here: Vacation Days Advanced. In this challenge, we need to calculate vacation days for employees, based on their age and years of service and some more information. In total, there are seven rules:

Vacation Days

1) Every employee receives at least 22 vacation days.
2) Employees younger than 18 or at least 60 years, or employees with at least 30 years of service can receive extra 5 days.
3) Employees with at least 30 years of service and also employees of age 60 or more, can receive extra 3 days, on top of possible additional days already given.
4) If an employee has at least 15 but less than 30 years of service, extra 2 days can be given. These 2 days can also be provided for employees of age 45 or more.
5) A college student is eligible to 1 extra vacation day.
6) If an employee is a veteran, 2 extra days can be given.
7) The total number of vacation days cannot exceed 29.

The seventh rule of the specification is what makes this challenge so interesting. At first, it would seem intuitive to place a soft maximum on the vacation days: if an employee exceeds 29 days, give them 29 days. However, in Jacob Feldman's OpenRules implementation he points out that this would mean we can simply subtract days from some types and only give partial number of eligible days. If we assume that this is not allowed, an employee would not want to make use of every type of vacation days for which they are eligible, but only the ones that get their total closest to 29. In this small example this does not matter too much, but in bigger systems with more mutually exclusive rules this could make a difference.

In this implementation, we will deal with two types of vacation rules: the ones for which an employee is eligible, and the ones that they 'apply', e.g. actually use. In other words, they will not have to use all vacation days for which they are eligible. We will try to pick the used days in such a way that a maximum number is achieved for every employee.

To fill out our glossary, we start by adding types for the vacation days, age, service years, the employees and the different rules. For the example, we create three dummy employees: Huey, Dewey and Louie.

Next, we create functions to map every employee to their age, service years and total vacation days.

In order to express whether an employee is a veteran or student, we add 2 relations. We also add a relation which keeps track of the eligible rules, and the 'applied' rules.

Now that our glossary is complete, we can start creating decision and constraint tables. We start by implementing a table for every rule, to express when an exmployee is eligible.

Now that we can decide which employee is eligible for what rules, we need a way to calculate the number of days. We do this by creating a `C+` table, in which we sum the number of days per rule based on which rules are applied.

Just stating that rules can be applied is of course not enough. We need to make sure that only rules for which an employee is eligible can be applied. In order to do this, we create a constraint table. Note how it states that a rule can only be applied if the employee is eligible, but that it does not state the opposite. In other words, not every eligible rule needs to be applied.

Of course, we also need to add a constraint that no employee can have more than 29 vacation days.

There we have it! We are now able to calculate vacation days for employees. However, just this calculation alone does not do much yet. We need to be able to look for the solution in which the employees have a maximum number of vacation days. To do this, we sum all the vacation days of employees and maximize it. This is a bit unfortunate, but because we added quantification we have to maximize for all employees. If we did not use quantification, we could run the model once for every employee. In this case the employees do not affect each other, so solving the problem for all of them at the same time creates no implications.

To now check the amount of vacation days our employees get, we can insert a data table containing their information, and run the solver.

This gives us the following solution:

```
Number of models: 1
Model 1
=======
structure  : V {
  Employee_applies_Rule = { Dewey,r2; Dewey,r5; Huey,r2; Huey,r6; Louie,r3; Louie,r4 }
  Employee_eligible_for_Rule = { Dewey,r2; Dewey,r3; Dewey,r4; Dewey,r5; Huey,r2;␣
↪Huey,r3; Huey,r6; Louie,r3; Louie,r4 }
  Employee_is_Student = { Dewey }
  Employee_is_Veteran = { Huey }
  Age = { Dewey->70; Huey->17; Louie->35 }
  Service_Years = { Dewey->31; Huey->1; Louie->16 }
  Total_Days = 83
  Vacation_Days = { Dewey->28; Huey->28; Louie->27 }
}

Elapsed Time:
0.111631
```

Doctor Planning

The Doctor Planning example was submitted by us to the dmcommunity.org as a challenge. The full version, together with other submitted solutions, can be found on this page. Its specification is as follows:

Doctor Planning

In a hospital, there has to be a doctor present at all times. In order to make sure this is the case, a planning is made for the next 7 days, starting on a monday. Each day consists of three shifts: an early shift, a late shift and a night shift.

Every shift needs to be assigned to a doctor. In total there are 5 doctors: every doctor has a list of available days, and some have special requirements. In general, the following rules apply:

1. A doctor can only work one shift per day.
2. A doctor should always be available for his shift (see table below)

3. If a doctor has the night shift, they either get the next day off, or the night shift again.
4. A doctor either works both days of the weekend, or none of the days.

A planning should be made in which every requirement is fulfilled.

Name	Available
Fleming	Friday, Saturday, Sunday
Freud	Every day early or late, never night
Heimlich	Every day but neven the night shift on weekends
Eustachi	Every day, every shift
Golgi	Every day, every shift but at max 2 night shifts

To create a cDMN model for this challenge, we start by building our glossary. From reading the description, we can see that we will need the following types:

- Doctor
- Day
- Time (early, late or night)
- Nb_shifts, to represent a number of shifts.

Thus, we create a glossary table as such:

Next, we will need a way to assign doctors to days and shifts. Since for every pair of day and shift (e.g. "Monday" and "Early") we need exactly 1 doctor, we use a function to do so. We also introduce functions to count the number of shifts per doctor and day, and the number of night shifts per doctor. Finally, we create one last function to represent the next day for every day.

The last information to include in our model is the availabilities of every doctor. For this, we implement the relation *Doctor is available on Day and Time*.

Now that our glossary is done, we can move on to implementing the rules. We start by first implementing every rule one by one.

Rule 1

A doctor can only work one shift per day.

This rule can be modeled as a simple constraint table. We evaluate every doctor, on every day, and state that they work at max 1 shift that day.

Rule 2

A doctor should always be available for his shift.

Once again, a single constraint table suffices. We state that for every doctor, day, and time, IF the doctor is assigned to that day and time, THEN he has to be available. In other words, a doctor can only be assigned to a day and time if he is available.

Rule 3

If a doctor has the night shift, they either get the next day off, or the night shift again.

This rule is by far the most difficult one to implement. We go over every doctor who works the night shift, and then state that they cannot be assigned to the early or late shift on the next day.

Rule 4

A doctor either works both days of the weekend, or none of the days.

In order to implement this rule, we create a constraint table that says that every doctor who has a shift on Saturday has to work on Sunday, and vice versa.

Special preference

Golgi only works a maximum of 2 night shifts every week.

This preference nicely shows of the simplicity of constraints in cDMN. We implement it as follows:

Now that every rule has been implemented, we need to create tables for the other concepts which we defined. We start by creating the decision table which defines the next day for every day. Then we create the tables which count the number of shifts per day, and the number of night shifts per week.

Now all that rests is implementing the availablities in a data table. This table is quite long, so only the first few lines are shown.

We can now run our model using the cDMN solver. For example, this is one of the solutions found by the solver:

```
Model 1
=======
structure  : V {
  Doctor_is_available_on_Day_and_Time = { Eustachi,Friday,Early; Eustachi,Friday,Late;
↪ Eustachi,Friday,Night; Eustachi,Monday,Early; Eustachi,Monday,Late; Eustachi,
↪Monday,Night; Eustachi,Saturday,Early; Eustachi,Saturday,Late; Eustachi,Saturday,
↪Night; Eustachi,Sunday,Early; Eustachi,Sunday,Late; Eustachi,Sunday,Night; Eustachi,
↪Thursday,Early; Eustachi,Thursday,Late; Eustachi,Thursday,Night; Eustachi,Tuesday,
↪Early; Eustachi,Tuesday,Late; Eustachi,Tuesday,Night; Eustachi,Wednesday,Early;␣
↪Eustachi,Wednesday,Late; Eustachi,Wednesday,Night; Fleming,Friday,Early; Fleming,
↪Friday,Late; Fleming,Friday,Night; Fleming,Saturday,Early; Fleming,Saturday,Late;␣
↪Fleming,Saturday,Night; Fleming,Sunday,Early; Fleming,Sunday,Late; Fleming,Sunday,
↪Night; Freud,Friday,Early; Freud,Friday,Late; Freud,Monday,Early; Freud,Monday,Late;
↪ Freud,Saturday,Early; Freud,Saturday,Late; Freud,Sunday,Early; Freud,Sunday,Late;␣
↪Freud,Thursday,Early; Freud,Thursday,Late; Freud,Tuesday,Early; Freud,Tuesday,Late;␣
↪Freud,Wednesday,Early; Freud,Wednesday,Late; Golgi,Friday,Early; Golgi,Friday,Late;␣
↪Golgi,Friday,Night; Golgi,Monday,Early; Golgi,Monday,Late; Golgi,Monday,Night;␣
↪Golgi,Saturday,Early; Golgi,Saturday,Late; Golgi,Saturday,Night; Golgi,Sunday,Early;
↪ Golgi,Sunday,Late; Golgi,Sunday,Night; Golgi,Thursday,Early; Golgi,Thursday,Late;␣
↪Golgi,Thursday,Night; Golgi,Tuesday,Early; Golgi,Tuesday,Late; Golgi,Tuesday,Night;␣
↪Golgi,Wednesday,Early; Golgi,Wednesday,Late; Golgi,Wednesday,Night; Heimlich,Friday,
↪Early; Heimlich,Friday,Late; Heimlich,Friday,Night; Heimlich,Monday,Early; Heimlich,
↪Monday,Late; Heimlich,Monday,Night; Heimlich,Saturday,Early; Heimlich,Saturday,Late;
↪ Heimlich,Sunday,Early; Heimlich,Sunday,Late; Heimlich,Thursday,Early; Heimlich,
↪Thursday,Late; Heimlich,Thursday,Night; Heimlich,Tuesday,Early; Heimlich,Tuesday,
↪Late; Heimlich,Tuesday,Night; Heimlich,Wednesday,Early; Heimlich,Wednesday,Late;␣
↪Heimlich,Wednesday,Night }
  Assigned_Doctor = { Friday,Early->Eustachi; Friday,Late->Golgi; Friday,Night->
↪Fleming; Monday,Early->Freud; Monday,Late->Golgi; Monday,Night->Heimlich; Saturday,
↪Early->Heimlich; Saturday,Late->Golgi; Saturday,Night->Eustachi; Sunday,Early->
↪Golgi; Sunday,Late->Heimlich; Sunday,Night->Eustachi; Thursday,Early->Eustachi;␣
↪Thursday,Late->Freud; Thursday,Night->Heimlich; Tuesday,Early->Eustachi; Tuesday,
↪Late->Freud; Tuesday,Night->Heimlich; Wednesday,Early->Freud; Wednesday,Late->Golgi;
↪ Wednesday,Night->Heimlich }
```

(continues on next page)

(continued from previous page)

```
  Nb_Nights = { Eustachi->2; Fleming->1; Freud->0; Golgi->0; Heimlich->4 }
  Nb_Shifts = { Eustachi,Friday->1; Eustachi,Monday->0; Eustachi,Saturday->1;␣
↪Eustachi,Sunday->1; Eustachi,Thursday->1; Eustachi,Tuesday->1; Eustachi,Wednesday->
↪0; Fleming,Friday->1; Fleming,Monday->0; Fleming,Saturday->0; Fleming,Sunday->0;␣
↪Fleming,Thursday->0; Fleming,Tuesday->0; Fleming,Wednesday->0; Freud,Friday->0;␣
↪Freud,Monday->1; Freud,Saturday->0; Freud,Sunday->0; Freud,Thursday->1; Freud,
↪Tuesday->1; Freud,Wednesday->1; Golgi,Friday->1; Golgi,Monday->1; Golgi,Saturday->1;
↪ Golgi,Sunday->1; Golgi,Thursday->0; Golgi,Tuesday->0; Golgi,Wednesday->1; Heimlich,
↪Friday->0; Heimlich,Monday->1; Heimlich,Saturday->1; Heimlich,Sunday->1; Heimlich,
↪Thursday->1; Heimlich,Tuesday->1; Heimlich,Wednesday->1 }
  Next_Day = { Friday->Saturday; Monday->Tuesday; Saturday->Sunday; Sunday->Monday;␣
↪Thursday->Friday; Tuesday->Wednesday; Wednesday->Thursday }
}

Elapsed Time:
0.104035
```

The *Assigned_Doctor* function maps every day and shift on a doctor, conform to all the requirements!

Balanced Assignment

This example is the Balanced Assignment challenge, in which we need to create groups that are as diverse as possible.

Balanced Assignment

Given several employee categories (title, location, department, male/female, ...) and a specified number of groups, assign every employee to a project group so that the groups are the same size, and they are as diverse as possible.

Together with this specification, we are given a list of 210 people and their information.

This challenge is an interesting optimization challenge. As always, we start by filling out the glossary. We create a type to represent the people, and types for every information type we have. Then, we also need a type which will represent a number of persons, and a type which will just represent a large integer.

Note here that because our list has 210 entries, it would be nearly impossible to add all of these into the *Values* column of the glossary. We opt here to not define the values, but instead we will point the glossary to our data table, which will tell the solver that every value in the data table is a possible value.

Next up, we introduce a function for every parameter, which will assign every person to their info. We will also need a function which maps every group on a number, representing the number of people in that group.

To finish our glossary, we introduce a constant *Score* which will keep track of the score of a solution.

With our glossary finished, we can move on to the next step. We need a way to calculate the **Score** value for a solution. To do this, we will check for every two people that if they have something in common, that they are in a different group. If this is the case, then we add 1 to our score.

Now, we want to count the number of people per group. Again, we can do this in a simple table. `Number in group of Person` is a combination of `Number in Group` and `group of Person`. This allows us to count the amount of people per group. Now, because the type `NbPersons` is defined in our glossary as `[16, 19]`, this automatically creates the constrain that every group needs to have between 16 and 19 people.

All that rests now is creating a `Goal` table in order to maximize the score!

And that's it! Sadly, the cDMN solver currently is unable to completely solve this challenge, as it is quite large. However, the cDMN notation is capable of fully modelling the problem, which is the main point.

Map Coloring

The map coloring problem is a common problem. We will show how to solve it using only cDMN. The full DMCommunity challenge, with other submitted solutions can be found here.

Map Coloring Challenge

This challenge deals with map coloring. You need to use no more than 4 colors (blue, red, green, or yellow) to color six European countries: Belgium, Denmark, France, Germany, Luxembourg, and the Netherlands in such a way that no neighboring countries use the same color.

We start by creating our glossary. For this specific problem, the glossary does not need to be complex. We create a type to represent countries and a type to represent the colors.

To assign every country exactly one color, we can use a function. To express that two countries border, we use a relation.

Now that our glossary is done, we can move on to the next step: modeling the problem logic. This logic is very simple: if two countries border, they cannot share the same color. We can represent this straightforwardly using a constraint table. The table is read as: "For every two countries c1, c2, if they border they cannot share the same color."

And that is all the logic we need to solve this problem! We simply need to represent which countries border which using a data table, and we're done.

We can now run our model using the cDMN solver. This results in the following output:

```
Number of models: 1
Model 1
=======
structure  : V {
  Country_borders_Country = { Belgium,France; Belgium,Germany; Belgium,Luxembourg;␣
↪Belgium,Netherlands; France,Luxembourg; Germany,Denmark; Germany,France; Germany,
↪Luxembourg; Netherlands,Germany }
  Color = { Belgium->Yellow; Denmark->Red; France->Green; Germany->Orange; Luxembourg-
↪>Red; Netherlands->Green }
}

Elapsed Time:
0.052835
```

We are now able to color a map without having to worry about bordering countries sharing a color!

Map Coloring With Violations

This version of the map coloring problem is an advanced version. Instead of having four colors, we only have three. In other words, some countries will have to share a color with a bordering country. The full DMCommunity challenge, with other submitted solutions, can be found here.

Map Coloring with Violations

This challenge is an advanced version of the May-2019 Challenge. Previously you had 4 colors (blue, red, green, or yellow) but now you have only 3 colors (blue, red, green). It is not enough to color six European countries: Belgium, Denmark, France, Germany, Luxembourg, and the Netherlands in such a way that no neighboring countries use the same color. So, some neighboring counties may have the same colors but there is a relative cost for such violations:

France – Luxembourg: $257

Luxembourg – Germany: $904

Luxembourg – Belgium: $568

This problem is a bit harder than the previous one, because we can no longer create a constraint that bordering countries can not share a color. Instead, we express this constraint only for all countries that are not Luxembourg.

We start by creating our glossary. We create a type to represent countries and a type to represent the colors. We also add a type *Number* to represent the violation score.

To assign every country exactly one color, we can use a function. We also need a constant to represent the *score* of a solution, which we will call *Violations* To express that two countries border, we use a relation.

Now that our glossary is done, we move on to the next step. We change our constraint table from the previous model into the following: if two countries border of which neither is Luxembourg, they cannot share the same color. This way, Luxembourg is allowed to share a color with its bordering countries.

Now we add a table which will sum the weights of the vialotions, and store it in the *Violations* constant.

And that is all the logic we need to solve this problem! We simply need to represent which countries border which using a data table, and then tell the system to minimize the *Violations* constant.

We can now run our model using the cDMN solver. This results in the following output:

```
Number of models: 1
Model 1
=======
structure  : V {
  Country_borders_Country = { Belgium,France; Belgium,Germany; Belgium,Luxembourg;␣
↪Belgium,Netherlands; France,Luxembourg; Germany,Denmark; Germany,France; Germany,
↪Luxembourg; Netherlands,Germany }
  Color = { Belgium->Red; Denmark->Red; France->Green; Germany->Yellow; Luxembourg->
↪Green; Netherlands->Green }
  Violations = 257
}

Elapsed Time:
0.047935
```

And there we have it! If Luxembourg only shares a color with France, we have the lowest score for violation.

Of course, this is a trivial example: this answer was visible from the start. Instead of only having a violation for Luxembourg, we could create scores for every two bordering countries and then minimize it. This can be done by removing the constraint table, and changing the **C+** table to a table which counts the violation weight for every two countries if they share the same color.

Where *Weight of Country and Country* is a function which maps every pair of countries on a violation weight, which is defined in a data table.

Crack the Code

The *"Crack the Code"* challenge is a challenge proposed by the DMCommunity. It easily be solved by hand, but it is of course more fun to write a program for it. Even though it is not supposed to be implemented in a DMN-like way, we implemented it in cDMN just for fun. :-)

Crack the Code

You need to crack a 3 digit code based on these hints:

- 682 – one number is correct and in the correct position
- 645 – one number is correct but in the wrong position
- 206 – two numbers are correct but in the wrong positions
- 738 – nothing is correct
- 780 – one number is correct but in the wrong position.

At first glance, this is quite a difficult problem to implement in cDMN. We need a way to split a number into digits, which seems like a special string operation. However, by being creative with maths we can perform the splitting, and our problem becomes easier.

We start by creating our glossary. In order to represent a digit, we create a type *Number*. To represent all possible guesses, we create a type *Guess*, which contains all numbers between 000 and 999.

Next, we create functions which will map every guess on their digits. E.g., *units of 345* will be assigned *5*, in the same way that *hundreds of 123* will be assigned *1*. Then we also add functions to keep track of the number of correct digits a guess has, and the number of digits in correct positions.

After defining our functions, we add our constants. *Sol1* through *Sol3* represent the digits of our solution in the following way: *Solution = Sol1 + Sol2*10 + Sol3*100*

Our glossary is now finished. To model the logic of the problem, we start out by adding the formula for the solution, and the logic which will split any number into its digits.

This might seem like a complex table, but it is not impossible to understand once you see how it works.

Next, we need a way to express the known guesses, and their information.

Now we need to define when a number is correct, and when a number is in the right position. For the right position, we achieve this using the following table:

The above table counts the amount of correct positions there are in a guess. For example, for *682* we know that exactly one of the three digits is already in the right position. This means that either the *6*, the *8* or the *2* is correct. The table states that one of the three *SolX* constants, has to be equal to one of the digits of *682*.

For the number of correct guesses, we do the same thing:

This table works in the same way as the previous one, except that a correct digit could still be in the wrong position. So for example for *206*, two of the digits need to be equal to *Sol1*, *Sol2* or *Sol3*.

We can now run our model using the cDMN solver. This results in the following output:

```
Number of models: 1
Model 1
=======
structure  : V {
  Correct_Number = { 0->3; 1->2; 2->3; 3->2; 4->2; 5->3; 6->2; 7->2; 8->2; 9->2; 10->
↪2; 11->1; 12->2; 13->1; 14->1; 15->2; 16->1; 17->1; 18->1; 19->1; 20->3; 21->2; 22->
↪3; 23->2; 24->2; 25->3; 26->2; 27->2; 28->2; 29->2; 30->2; 31->1; 32->2; 33->1; 34->
↪1; 35->2; 36->1; 37->1; 38->1; 39->1; 40->2; 41->1; 42->2; 43->1; 44->1; 45->2; 46->
↪1; 47->1; 48->1; 49->1; 50->3; 51->2; 52->3; 53->2; 54->2; 55->3; 56->2; 57->2; 58->
↪2; 59->2; 60->2; 61->1; 62->2; 63->1; 64->1; 65->2; 66->1; 67->1; 68->1; 69->1; 70->
↪2; 71->1; 72->2; 73->1; 74->1; 75->2; 76->1; 77->1; 78->1; 79->1; 80->2; 81->1; 82->
↪2; 83->1; 84->1; 85->2; 86->1; 87->1; 88->1; 89->1; 90->2; 91->1; 92->2; 93->1; 94->
↪1; 95->2; 96->1; 97->1; 98->1; 99->1; 100->2; 101->1; 102->2; 103->1; 104->1; 105->
↪2; 106->1; 107->1; 108->1; 109->1; 110->1; 111->0; 112->1; 113->0; 114->0; 115->1;␣
↪116->0; 117->0; 118->0; 119->0; 120->2; 121->1; 122->2; 123->1; 124->1; 125->2; 126->
↪1; 127->1; 128->1; 129->1; 130->1; 131->0; 132->1; 133->0; 134->0; 135->1; 136->0;␣
↪137->0; 138->0; 139->0; 140->1; 141->0; 142->1; 143->0; 144->0; 145->1; 146->0; 147->
↪0; 148->0; 149->0; 150->2; 151->1; 152->2; 153->1; 154->1; 155->2; 156->1; 157->1;␣
↪158->1; 159->1; 160->1; 161->0; 162->1; 163->0; 164->0; 165->1; 166->0; 167->0; 168->
↪0; 169->0; 170->1; 171->0; 172->1; 173->0; 174->0; 175->1; 176->0; 177->0; 178->0;␣
↪179->1; 180->0; 181->0; 182->1; 183->0; 184->0; 185->1; 186->0; 187->0; 188->0; 189->
↪0; 190->1; 191->0; 192->1; 193->0; 194->0; 195->1; 196->0; 197->0; 198->0; 199->0;␣
↪200->3; 201->2; 202->3; 203->2; 204->2; 205->3; 206->2; 207->2; 208->2; 209->2; 210->
↪2; 211->1; 212->2; 213->1; 214->1; 215->2; 216->1; 217->1; 218->1; 219->1; 220->3;␣
```

(continues on next page)

(continued from previous page)

```
  Correct_Position = { 0->1; 1->1; 2->2; 3->1; 4->1; 5->1; 6->1; 7->1; 8->1; 9->1; 10-
↪>1; 11->1; 12->2; 13->1; 14->1; 15->1; 16->1; 17->1; 18->1; 19->1; 20->1; 21->1; 22-
↪>2; 23->1; 24->1; 25->1; 26->1; 27->1; 28->1; 29->1; 30->1; 31->1; 32->2; 33->1; 34-
↪>1; 35->1; 36->1; 37->1; 38->1; 39->1; 40->1; 41->1; 42->2; 43->1; 44->1; 45->1; 46-
↪>1; 47->1; 48->1; 49->1; 50->2; 51->2; 52->3; 53->2; 54->2; 55->2; 56->2; 57->2; 58-
↪>2; 59->2; 60->1; 61->1; 62->2; 63->1; 64->1; 65->1; 66->1; 67->1; 68->1; 69->1; 70-
↪>1; 71->1; 72->2; 73->1; 74->1; 75->1; 76->1; 77->1; 78->1; 79->1; 80->1; 81->1; 82-
↪>2; 83->1; 84->1; 85->1; 86->1; 87->1; 88->1; 89->1; 90->1; 91->1; 92->2; 93->1; 94-
↪>1; 95->1; 96->1; 97->1; 98->1; 99->1; 100->0; 101->0; 102->1; 103->0; 104->0; 105->
↪0; 106->0; 107->0; 108->0; 109->0; 110->0; 111->0; 112->1; 113->0; 114->0; 115->0;␣
↪116->0; 117->0; 118->0; 119->0; 120->0; 121->0; 122->1; 123->0; 124->0; 125->0; 126-
↪>0; 127->0; 128->0; 129->0; 130->0; 131->0; 132->1; 133->0; 134->0; 135->0; 136->0;␣
↪137->0; 138->0; 139->0; 140->0; 141->0; 142->1; 143->0; 144->0; 145->0; 146->0; 147-
↪>0; 148->0; 149->0; 150->1; 151->1; 152->2; 153->1; 154->1; 155->1; 156->1; 157->1;␣
↪158->1; 159->1; 160->0; 161->0; 162->1; 163->0; 164->0; 165->0; 166->0; 167->0; 168-
↪>0; 169->0; 170->0; 171->0; 172->1; 173->0; 174->0; 175->0; 176->0; 177->0; 178->0;␣
↪179->0; 180->0; 181->0; 182->1; 183->0; 184->0; 185->0; 186->0; 187->0; 188->0; 189-
↪>0; 190->0; 191->0; 192->1; 193->0; 194->0; 195->0; 196->0; 197->0; 198->0; 199->0;␣
↪200->0; 201->0; 202->1; 203->0; 204->0; 205->0; 206->0; 207->0; 208->0; 209->0; 210-
↪>0; 211->0; 212->1; 213->0; 214->0; 215->0; 216->0; 217->0; 218->0; 219->0; 220->0;␣
↪221->0; 222->1; 223->0; 224->0; 225->0; 226->0; 227->0; 228->0; 229->0; 230->0; 231-
↪>0; 232->1; 233->0; 234->0; 235->0; 236->0; 237->0; 238->0; 239->0; 240->0; 241->0;␣
↪242->1; 243->0; 244->0; 245->0; 246->0; 247->0; 248->0; 249->0; 250->1; 251->1; 252-
↪>2; 253->1; 254->1; 255->1; 256->1; 257->1; 258->1; 259->1; 260->0; 261->0; 262->1;␣
↪263->0; 264->0; 265->0; 266->0; 267->0; 268->0; 269->0; 270->0; 271->0; 272->1; 273-
↪>0; 274->0; 275->0; 276->0; 277->0; 278->0; 279->0; 280->0; 281->0; 282->1; 283->0;␣
↪284->0; 285->0; 286->0; 287->0; 288->0; 289->0; 290->0; 291->0; 292->1; 293->0; 294-
↪>0; 295->0; 296->0; 297->0; 298->0; 299->0; 300->0; 301->0; 302->1; 303->0; 304->0;␣
↪305->0; 306->0; 307->0; 308->0; 309->0; 310->0; 311->0; 312->1; 313->0; 314->0; 315-
↪>0; 316->0; 317->0; 318->0; 319->0; 320->0; 321->0; 322->1; 323->0; 324->0; 325->0;␣
↪326->0; 327->0; 328->0; 329->0; 330->0; 331->0; 332->1; 333->0; 334->0; 335->0; 336-
↪>0; 337->0; 338->0; 339->0; 340->0; 341->0; 342->1; 343->0; 344->0; 345->0; 346->0;␣
↪347->0; 348->0; 349->0; 350->1; 351->1; 352->2; 353->1; 354->1; 355->1; 356->1; 357-
↪>1; 358->1; 359->1; 360->0; 361->0; 362->1; 363->0; 364->0; 365->0; 366->0; 367->0;␣
↪368->0; 369->0; 370->0; 371->0; 372->1; 373->0; 374->0; 375->0; 376->0; 377->0; 378-
↪>0; 379->0; 380->0; 381->0; 382->1; 383->0; 384->0; 385->0; 386->0; 387->0; 388->0;␣
↪389->0; 390->0; 391->0; 392->1; 393->0; 394->0; 395->0; 396->0; 397->0; 398->0; 399-
↪>0; 400->0; 401->0; 402->1; 403->0; 404->0; 405->0; 406->0; 407->0; 408->0; 409->0;␣
↪410->0; 411->0; 412->1; 413->0; 414->0; 415->0; 416->0; 417->0; 418->0; 419->0; 420-
↪>0; 421->0; 422->1; 423->0; 424->0; 425->0; 426->0; 427->0; 428->0; 429->0; 430->0;␣
↪431->0; 432->1; 433->0; 434->0; 435->0; 436->0; 437->0; 438->0; 439->0; 440->0; 441-
↪>0; 442->1; 443->0; 444->0; 445->0; 446->0; 447->0; 448->0; 449->0; 450->1; 451->1;␣
↪452->2; 453->1; 454->1; 455->1; 456->1; 457->1; 458->1; 459->1; 460->0; 461->0; 462-
↪>1; 463->0; 464->0; 465->0; 466->0; 467->0; 468->0; 469->0; 470->0; 471->0; 472->1;␣
↪473->0; 474->0; 475->0; 476->0; 477->0; 478->0; 479->0; 480->0; 481->0; 482->1; 483-
↪>0; 484->0; 485->0; 486->0; 487->0; 488->0; 489->0; 490->0; 491->0; 492->1; 493->0;␣
↪494->0; 495->0; 496->0; 497->0; 498->0; 499->0; 500->0; 501->0; 502->1; 503->0; 504-
↪>0; 505->0; 506->0; 507->0; 508->0; 509->0; 510->0; 511->0; 512->1; 513->0; 514->0;␣
↪515->0; 516->0; 517->0; 518->0; 519->0; 520->0; 521->0; 522->1; 523->0; 524->0; 525-
↪>0; 526->0; 527->0; 528->0; 529->0; 530->0; 531->0; 532->1; 533->0; 534->0; 535->0;␣
↪536->0; 537->0; 538->0; 539->0; 540->0; 541->0; 542->1; 543->0; 544->0; 545->0; 546-
↪>0; 547->0; 548->0; 549->0; 550->1; 551->1; 552->2; 553->1; 554->1; 555->1; 556->1;␣
↪557->1; 558->1; 559->1; 560->0; 561->0; 562->1; 563->0; 564->0; 565->0; 566->0; 567-
↪>0; 568->0; 569->0; 570->0; 571->0; 572->1; 573->0; 574->0; 575->0; 576->0; 577->0;␣
↪578->0; 579->0; 580->0; 581->0; 582->1; 583->0; 584->0; 585->0; 586->0; 587->0; 588-
↪>0; 589->0; 590->0; 591->0; 592->1; 593->0; 594->0; 595->0; 596->0; 597->0; 598->0;␣
↪599->0; 600->0; 601->0; 602->1; 603->0; 604->0; 605->0; 606->0; 607->0; 608->0; 609-
↪>0; 610->0; 611->0; 612->1; 613->0; 614->0; 615->0; 616->0; 617->0; 618->0; 619->0;␣
↪620->0; 621->0; 622->1; 623->0; 624->0; 625->0; 626->0; 627->0; 628->0; 629->0; 630-
↪>0; 631->0; 632->1; 633->0; 634->0; 635->0; 636->0; 637->0; 638->0; 639->0; 640->0;␣
↪641->0; 642->1; 643->0; 644->0; 645->0; 646->0; 647->0; 648->0; 649->0; 650->1; 651
↪>1; 652->2; 653->1; 654->1; 655->1; 656->1; 657->1; 658->1; 659->1; 660->0; 661->0;␣
↪662->1; 663->0; 664->0; 665->0; 666->0; 667->0; 668->0; 669->0; 670->0; 671->0; 672-
↪>1; 673->0; 674->0; 675->0; 676->0; 677->0; 678->0; 679->0; 680->0; 681->0; 682->1;␣
```

(continues on next page)

(continued from previous page)

```
  Hundreds = { 0->0; 1->0; 2->0; 3->0; 4->0; 5->0; 6->0; 7->0; 8->0; 9->0; 10->0; 11->
↪0; 12->0; 13->0; 14->0; 15->0; 16->0; 17->0; 18->0; 19->0; 20->0; 21->0; 22->0; 23->
↪0; 24->0; 25->0; 26->0; 27->0; 28->0; 29->0; 30->0; 31->0; 32->0; 33->0; 34->0; 35->
↪0; 36->0; 37->0; 38->0; 39->0; 40->0; 41->0; 42->0; 43->0; 44->0; 45->0; 46->0; 47->
↪0; 48->0; 49->0; 50->0; 51->0; 52->0; 53->0; 54->0; 55->0; 56->0; 57->0; 58->0; 59->
↪0; 60->0; 61->0; 62->0; 63->0; 64->0; 65->0; 66->0; 67->0; 68->0; 69->0; 70->0; 71->
↪0; 72->0; 73->0; 74->0; 75->0; 76->0; 77->0; 78->0; 79->0; 80->0; 81->0; 82->0; 83->
↪0; 84->0; 85->0; 86->0; 87->0; 88->0; 89->0; 90->0; 91->0; 92->0; 93->0; 94->0; 95->
↪0; 96->0; 97->0; 98->0; 99->0; 100->1; 101->1; 102->1; 103->1; 104->1; 105->1; 106->
↪1; 107->1; 108->1; 109->1; 110->1; 111->1; 112->1; 113->1; 114->1; 115->1; 116->1;␣
↪117->1; 118->1; 119->1; 120->1; 121->1; 122->1; 123->1; 124->1; 125->1; 126->1; 127->
↪1; 128->1; 129->1; 130->1; 131->1; 132->1; 133->1; 134->1; 135->1; 136->1; 137->1;␣
↪138->1; 139->1; 140->1; 141->1; 142->1; 143->1; 144->1; 145->1; 146->1; 147->1; 148->
↪1; 149->1; 150->1; 151->1; 152->1; 153->1; 154->1; 155->1; 156->1; 157->1; 158->1;␣
↪159->1; 160->1; 161->1; 162->1; 163->1; 164->1; 165->1; 166->1; 167->1; 168->1; 169->
↪1; 170->1; 171->1; 172->1; 173->1; 174->1; 175->1; 176->1; 177->1; 178->1; 179->1;␣
↪180->1; 181->1; 182->1; 183->1; 184->1; 185->1; 186->1; 187->1; 188->1; 189->1; 190->
↪1; 191->1; 192->1; 193->1; 194->1; 195->1; 196->1; 197->1; 198->1; 199->1; 200->2;␣
↪201->2; 202->2; 203->2; 204->2; 205->2; 206->2; 207->2; 208->2; 209->2; 210->2; 211->
↪2; 212->2; 213->2; 214->2; 215->2; 216->2; 217->2; 218->2; 219->2; 220->2; 221->2;␣
↪222->2; 223->2; 224->2; 225->2; 226->2; 227->2; 228->2; 229->2; 230->2; 231->2; 232->
↪2; 233->2; 234->2; 235->2; 236->2; 237->2; 238->2; 239->2; 240->2; 241->2; 242->2;␣
↪243->2; 244->2; 245->2; 246->2; 247->2; 248->2; 249->2; 250->2; 251->2; 252->2; 253->
↪2; 254->2; 255->2; 256->2; 257->2; 258->2; 259->2; 260->2; 261->2; 262->2; 263->2;␣
↪264->2; 265->2; 266->2; 267->2; 268->2; 269->2; 270->2; 271->2; 272->2; 273->2; 274->
↪2; 275->2; 276->2; 277->2; 278->2; 279->2; 280->2; 281->2; 282->2; 283->2; 284->2;␣
↪285->2; 286->2; 287->2; 288->2; 289->2; 290->2; 291->2; 292->2; 293->2; 294->2; 295->
↪2; 296->2; 297->2; 298->2; 299->2; 300->3; 301->3; 302->3; 303->3; 304->3; 305->3;␣
↪306->3; 307->3; 308->3; 309->3; 310->3; 311->3; 312->3; 313->3; 314->3; 315->3; 316->
↪3; 317->3; 318->3; 319->3; 320->3; 321->3; 322->3; 323->3; 324->3; 325->3; 326->3;␣
↪327->3; 328->3; 329->3; 330->3; 331->3; 332->3; 333->3; 334->3; 335->3; 336->3; 337->
↪3; 338->3; 339->3; 340->3; 341->3; 342->3; 343->3; 344->3; 345->3; 346->3; 347->3;␣
↪348->3; 349->3; 350->3; 351->3; 352->3; 353->3; 354->3; 355->3; 356->3; 357->3; 358->
↪3; 359->3; 360->3; 361->3; 362->3; 363->3; 364->3; 365->3; 366->3; 367->3; 368->3;␣
↪369->3; 370->3; 371->3; 372->3; 373->3; 374->3; 375->3; 376->3; 377->3; 378->3; 379->
↪3; 380->3; 381->3; 382->3; 383->3; 384->3; 385->3; 386->3; 387->3; 388->3; 389->3;␣
↪390->3; 391->3; 392->3; 393->3; 394->3; 395->3; 396->3; 397->3; 398->3; 399->3; 400->
↪4; 401->4; 402->4; 403->4; 404->4; 405->4; 406->4; 407->4; 408->4; 409->4; 410->4;␣
↪411->4; 412->4; 413->4; 414->4; 415->4; 416->4; 417->4; 418->4; 419->4; 420->4; 421->
↪4; 422->4; 423->4; 424->4; 425->4; 426->4; 427->4; 428->4; 429->4; 430->4; 431->4;␣
↪432->4; 433->4; 434->4; 435->4; 436->4; 437->4; 438->4; 439->4; 440->4; 441->4; 442->
↪4; 443->4; 444->4; 445->4; 446->4; 447->4; 448->4; 449->4; 450->4; 451->4; 452->4;␣
↪453->4; 454->4; 455->4; 456->4; 457->4; 458->4; 459->4; 460->4; 461->4; 462->4; 463->
↪4; 464->4; 465->4; 466->4; 467->4; 468->4; 469->4; 470->4; 471->4; 472->4; 473->4;␣
↪474->4; 475->4; 476->4; 477->4; 478->4; 479->4; 480->4; 481->4; 482->4; 483->4; 484->
↪4; 485->4; 486->4; 487->4; 488->4; 489->4; 490->4; 491->4; 492->4; 493->4; 494->4;␣
↪495->4; 496->4; 497->4; 498->4; 499->4; 500->5; 501->5; 502->5; 503->5; 504->5; 505->
↪5; 506->5; 507->5; 508->5; 509->5; 510->5; 511->5; 512->5; 513->5; 514->5; 515->5;␣
↪516->5; 517->5; 518->5; 519->5; 520->5; 521->5; 522->5; 523->5; 524->5; 525->5; 526->
↪5; 527->5; 528->5; 529->5; 530->5; 531->5; 532->5; 533->5; 534->5; 535->5; 536->5;␣
↪537->5; 538->5; 539->5; 540->5; 541->5; 542->5; 543->5; 544->5; 545->5; 546->5; 547->
↪5; 548->5; 549->5; 550->5; 551->5; 552->5; 553->5; 554->5; 555->5; 556->5; 557->5;␣
↪558->5; 559->5; 560->5; 561->5; 562->5; 563->5; 564->5; 565->5; 566->5; 567->5; 568->
↪5; 569->5; 570->5; 571->5; 572->5; 573->5; 574->5; 575->5; 576->5; 577->5; 578->5;␣
↪579->5; 580->5; 581->5; 582->5; 583->5; 584->5; 585->5; 586->5; 587->5; 588->5; 589->
↪5; 590->5; 591->5; 592->5; 593->5; 594->5; 595->5; 596->5; 597->5; 598->5; 599->5;␣
↪600->6; 601->6; 602->6; 603->6; 604->6; 605->6; 606->6; 607->6; 608->6; 609->6; 610->
↪6; 611->6; 612->6; 613->6; 614->6; 615->6; 616->6; 617->6; 618->6; 619->6; 620->6;␣
↪621->6; 622->6; 623->6; 624->6; 625->6; 626->6; 627->6; 628->6; 629->6; 630->6; 631->
↪6; 632->6; 633->6; 634->6; 635->6; 636->6; 637->6; 638->6; 639->6; 640->6; 641->6;␣
↪642->6; 643->6; 644->6; 645->6; 646->6; 647->6; 648->6; 649->6; 650->6; 651->6; 652->
↪6; 653->6; 654->6; 655->6; 656->6; 657->6; 658->6; 659->6; 660->6; 661->6; 662->6;␣
↪663->6; 664->6; 665->6; 666->6; 667->6; 668->6; 669->6; 670->6; 671->6; 672->6; 673->
↪6; 674->6; 675->6; 676->6; 677->6; 678->6; 679->6; 680->6; 681->6; 682->6; 683->6;␣
```

(continues on next page)

(continued from previous page)

```
  Sol1 = 2
  Sol2 = 5
  Sol3 = 0
  Solution = 52
  Tens = { 0->0; 1->0; 2->0; 3->0; 4->0; 5->0; 6->0; 7->0; 8->0; 9->0; 10->1; 11->1;␣
↪12->1; 13->1; 14->1; 15->1; 16->1; 17->1; 18->1; 19->1; 20->2; 21->2; 22->2; 23->2;␣
↪24->2; 25->2; 26->2; 27->2; 28->2; 29->2; 30->3; 31->3; 32->3; 33->3; 34->3; 35->3;␣
↪36->3; 37->3; 38->3; 39->3; 40->4; 41->4; 42->4; 43->4; 44->4; 45->4; 46->4; 47->4;␣
↪48->4; 49->4; 50->5; 51->5; 52->5; 53->5; 54->5; 55->5; 56->5; 57->5; 58->5; 59->5;␣
↪60->6; 61->6; 62->6; 63->6; 64->6; 65->6; 66->6; 67->6; 68->6; 69->6; 70->7; 71->7;␣
↪72->7; 73->7; 74->7; 75->7; 76->7; 77->7; 78->7; 79->7; 80->8; 81->8; 82->8; 83->8;␣
↪84->8; 85->8; 86->8; 87->8; 88->8; 89->8; 90->9; 91->9; 92->9; 93->9; 94->9; 95->9;␣
↪96->9; 97->9; 98->9; 99->9; 100->0; 101->0; 102->0; 103->0; 104->0; 105->0; 106->0;␣
↪107->0; 108->0; 109->0; 110->1; 111->1; 112->1; 113->1; 114->1; 115->1; 116->1; 117-
↪>1; 118->1; 119->1; 120->2; 121->2; 122->2; 123->2; 124->2; 125->2; 126->2; 127->2;␣
↪128->2; 129->2; 130->3; 131->3; 132->3; 133->3; 134->3; 135->3; 136->3; 137->3; 138-
↪>3; 139->3; 140->4; 141->4; 142->4; 143->4; 144->4; 145->4; 146->4; 147->4; 148->4;␣
↪149->4; 150->5; 151->5; 152->5; 153->5; 154->5; 155->5; 156->5; 157->5; 158->5; 159-
↪>5; 160->6; 161->6; 162->6; 163->6; 164->6; 165->6; 166->6; 167->6; 168->6; 169->6;␣
↪170->7; 171->7; 172->7; 173->7; 174->7; 175->7; 176->7; 177->7; 178->7; 179->7; 180-
↪>8; 181->8; 182->8; 183->8; 184->8; 185->8; 186->8; 187->8; 188->8; 189->8; 190->9;␣
↪191->9; 192->9; 193->9; 194->9; 195->9; 196->9; 197->9; 198->9; 199->9; 200->0; 201-
↪>0; 202->0; 203->0; 204->0; 205->0; 206->0; 207->0; 208->0; 209->0; 210->1; 211->1;␣
↪212->1; 213->1; 214->1; 215->1; 216->1; 217->1; 218->1; 219->1; 220->2; 221->2; 222-
↪>2; 223->2; 224->2; 225->2; 226->2; 227->2; 228->2; 229->2; 230->3; 231->3; 232->3;␣
↪233->3; 234->3; 235->3; 236->3; 237->3; 238->3; 239->3; 240->4; 241->4; 242->4; 243-
↪>4; 244->4; 245->4; 246->4; 247->4; 248->4; 249->4; 250->5; 251->5; 252->5; 253->5;␣
↪254->5; 255->5; 256->5; 257->5; 258->5; 259->5; 260->6; 261->6; 262->6; 263->6; 264-
↪>6; 265->6; 266->6; 267->6; 268->6; 269->6; 270->7; 271->7; 272->7; 273->7; 274->7;␣
↪275->7; 276->7; 277->7; 278->7; 279->7; 280->8; 281->8; 282->8; 283->8; 284->8; 285-
↪>8; 286->8; 287->8; 288->8; 289->8; 290->9; 291->9; 292->9; 293->9; 294->9; 295->9;␣
↪296->9; 297->9; 298->9; 299->9; 300->0; 301->0; 302->0; 303->0; 304->0; 305->0; 306-
↪>0; 307->0; 308->0; 309->0; 310->1; 311->1; 312->1; 313->1; 314->1; 315->1; 316->1;␣
↪317->1; 318->1; 319->1; 320->2; 321->2; 322->2; 323->2; 324->2; 325->2; 326->2; 327-
↪>2; 328->2; 329->2; 330->3; 331->3; 332->3; 333->3; 334->3; 335->3; 336->3; 337->3;␣
↪338->3; 339->3; 340->4; 341->4; 342->4; 343->4; 344->4; 345->4; 346->4; 347->4; 348-
↪>4; 349->4; 350->5; 351->5; 352->5; 353->5; 354->5; 355->5; 356->5; 357->5; 358->5;␣
↪359->5; 360->6; 361->6; 362->6; 363->6; 364->6; 365->6; 366->6; 367->6; 368->6; 369-
↪>6; 370->7; 371->7; 372->7; 373->7; 374->7; 375->7; 376->7; 377->7; 378->7; 379->7;␣
↪380->8; 381->8; 382->8; 383->8; 384->8; 385->8; 386->8; 387->8; 388->8; 389->8; 390-
↪>9; 391->9; 392->9; 393->9; 394->9; 395->9; 396->9; 397->9; 398->9; 399->9; 400->0;␣
↪401->0; 402->0; 403->0; 404->0; 405->0; 406->0; 407->0; 408->0; 409->0; 410->1; 411-
↪>1; 412->1; 413->1; 414->1; 415->1; 416->1; 417->1; 418->1; 419->1; 420->2; 421->2;␣
↪422->2; 423->2; 424->2; 425->2; 426->2; 427->2; 428->2; 429->2; 430->3; 431->3; 432-
↪>3; 433->3; 434->3; 435->3; 436->3; 437->3; 438->3; 439->3; 440->4; 441->4; 442->4;␣
↪443->4; 444->4; 445->4; 446->4; 447->4; 448->4; 449->4; 450->5; 451->5; 452->5; 453-
↪>5; 454->5; 455->5; 456->5; 457->5; 458->5; 459->5; 460->6; 461->6; 462->6; 463->6;␣
↪464->6; 465->6; 466->6; 467->6; 468->6; 469->6; 470->7; 471->7; 472->7; 473->7; 474-
↪>7; 475->7; 476->7; 477->7; 478->7; 479->7; 480->8; 481->8; 482->8; 483->8; 484->8;␣
↪485->8; 486->8; 487->8; 488->8; 489->8; 490->9; 491->9; 492->9; 493->9; 494->9; 495-
↪>9; 496->9; 497->9; 498->9; 499->9; 500->0; 501->0; 502->0; 503->0; 504->0; 505->0;␣
↪506->0; 507->0; 508->0; 509->0; 510->1; 511->1; 512->1; 513->1; 514->1; 515->1; 516-
↪>1; 517->1; 518->1; 519->1; 520->2; 521->2; 522->2; 523->2; 524->2; 525->2; 526->2;␣
↪527->2; 528->2; 529->2; 530->3; 531->3; 532->3; 533->3; 534->3; 535->3; 536->3; 537-
↪>3; 538->3; 539->3; 540->4; 541->4; 542->4; 543->4; 544->4; 545->4; 546->4; 547->4;␣
↪548->4; 549->4; 550->5; 551->5; 552->5; 553->5; 554->5; 555->5; 556->5; 557->5; 558-
↪>5; 559->5; 560->6; 561->6; 562->6; 563->6; 564->6; 565->6; 566->6; 567->6; 568->6;␣
↪569->6; 570->7; 571->7; 572->7; 573->7; 574->7; 575->7; 576->7; 577->7; 578->7; 579-
↪>7; 580->8; 581->8; 582->8; 583->8; 584->8; 585->8; 586->8; 587->8; 588->8; 589->8;␣
↪590->9; 591->9; 592->9; 593->9; 594->9; 595->9; 596->9; 597->9; 598->9; 599->9; 600-
↪>0; 601->0; 602->0; 603->0; 604->0; 605->0; 606->0; 607->0; 608->0; 609->0; 610->1;␣
↪611->1; 612->1; 613->1; 614->1; 615->1; 616->1; 617->1; 618->1; 619->1; 620->2; 621-
↪>2; 622->2; 623->2; 624->2; 625->2; 626->2; 627->2; 628->2; 629->2; 630->3; 631->3;␣
↪632->3; 633->3; 634->3; 635->3; 636->3; 637->3; 638->3; 639->3; 640->4; 641->4; 642-
```

(continues on next page)

(continued from previous page)

```
  Units = { 0->0; 1->1; 2->2; 3->3; 4->4; 5->5; 6->6; 7->7; 8->8; 9->9; 10->0; 11->1;␣
↪12->2; 13->3; 14->4; 15->5; 16->6; 17->7; 18->8; 19->9; 20->0; 21->1; 22->2; 23->3;␣
↪24->4; 25->5; 26->6; 27->7; 28->8; 29->9; 30->0; 31->1; 32->2; 33->3; 34->4; 35->5;␣
↪36->6; 37->7; 38->8; 39->9; 40->0; 41->1; 42->2; 43->3; 44->4; 45->5; 46->6; 47->7;␣
↪48->8; 49->9; 50->0; 51->1; 52->2; 53->3; 54->4; 55->5; 56->6; 57->7; 58->8; 59->9;␣
↪60->0; 61->1; 62->2; 63->3; 64->4; 65->5; 66->6; 67->7; 68->8; 69->9; 70->0; 71->1;␣
↪72->2; 73->3; 74->4; 75->5; 76->6; 77->7; 78->8; 79->9; 80->0; 81->1; 82->2; 83->3;␣
↪84->4; 85->5; 86->6; 87->7; 88->8; 89->9; 90->0; 91->1; 92->2; 93->3; 94->4; 95->5;␣
↪96->6; 97->7; 98->8; 99->9; 100->0; 101->1; 102->2; 103->3; 104->4; 105->5; 106->6;␣
↪107->7; 108->8; 109->9; 110->0; 111->1; 112->2; 113->3; 114->4; 115->5; 116->6; 117-
↪>7; 118->8; 119->9; 120->0; 121->1; 122->2; 123->3; 124->4; 125->5; 126->6; 127->7;␣
↪128->8; 129->9; 130->0; 131->1; 132->2; 133->3; 134->4; 135->5; 136->6; 137->7; 138-
↪>8; 139->9; 140->0; 141->1; 142->2; 143->3; 144->4; 145->5; 146->6; 147->7; 148->8;␣
↪149->9; 150->0; 151->1; 152->2; 153->3; 154->4; 155->5; 156->6; 157->7; 158->8; 159-
↪>9; 160->0; 161->1; 162->2; 163->3; 164->4; 165->5; 166->6; 167->7; 168->8; 169->9;␣
↪170->0; 171->1; 172->2; 173->3; 174->4; 175->5; 176->6; 177->7; 178->8; 179->9; 180-
↪>0; 181->1; 182->2; 183->3; 184->4; 185->5; 186->6; 187->7; 188->8; 189->9; 190->0;␣
↪191->1; 192->2; 193->3; 194->4; 195->5; 196->6; 197->7; 198->8; 199->9; 200->0; 201-
↪>1; 202->2; 203->3; 204->4; 205->5; 206->6; 207->7; 208->8; 209->9; 210->0; 211->1;␣
↪212->2; 213->3; 214->4; 215->5; 216->6; 217->7; 218->8; 219->9; 220->0; 221->1; 222-
↪>2; 223->3; 224->4; 225->5; 226->6; 227->7; 228->8; 229->9; 230->0; 231->1; 232->2;␣
↪233->3; 234->4; 235->5; 236->6; 237->7; 238->8; 239->9; 240->0; 241->1; 242->2; 243-
↪>3; 244->4; 245->5; 246->6; 247->7; 248->8; 249->9; 250->0; 251->1; 252->2; 253->3;␣
↪254->4; 255->5; 256->6; 257->7; 258->8; 259->9; 260->0; 261->1; 262->2; 263->3; 264-
↪>4; 265->5; 266->6; 267->7; 268->8; 269->9; 270->0; 271->1; 272->2; 273->3; 274->4;␣
↪275->5; 276->6; 277->7; 278->8; 279->9; 280->0; 281->1; 282->2; 283->3; 284->4; 285-
↪>5; 286->6; 287->7; 288->8; 289->9; 290->0; 291->1; 292->2; 293->3; 294->4; 295->5;␣
↪296->6; 297->7; 298->8; 299->9; 300->0; 301->1; 302->2; 303->3; 304->4; 305->5; 306-
↪>6; 307->7; 308->8; 309->9; 310->0; 311->1; 312->2; 313->3; 314->4; 315->5; 316->6;␣
↪317->7; 318->8; 319->9; 320->0; 321->1; 322->2; 323->3; 324->4; 325->5; 326->6; 327-
↪>7; 328->8; 329->9; 330->0; 331->1; 332->2; 333->3; 334->4; 335->5; 336->6; 337->7;␣
↪338->8; 339->9; 340->0; 341->1; 342->2; 343->3; 344->4; 345->5; 346->6; 347->7; 348-
↪>8; 349->9; 350->0; 351->1; 352->2; 353->3; 354->4; 355->5; 356->6; 357->7; 358->8;␣
↪359->9; 360->0; 361->1; 362->2; 363->3; 364->4; 365->5; 366->6; 367->7; 368->8; 369-
↪>9; 370->0; 371->1; 372->2; 373->3; 374->4; 375->5; 376->6; 377->7; 378->8; 379->9;␣
↪380->0; 381->1; 382->2; 383->3; 384->4; 385->5; 386->6; 387->7; 388->8; 389->9; 390-
↪>0; 391->1; 392->2; 393->3; 394->4; 395->5; 396->6; 397->7; 398->8; 399->9; 400->0;␣
↪401->1; 402->2; 403->3; 404->4; 405->5; 406->6; 407->7; 408->8; 409->9; 410->0; 411-
↪>1; 412->2; 413->3; 414->4; 415->5; 416->6; 417->7; 418->8; 419->9; 420->0; 421->1;␣
↪422->2; 423->3; 424->4; 425->5; 426->6; 427->7; 428->8; 429->9; 430->0; 431->1; 432-
↪>2; 433->3; 434->4; 435->5; 436->6; 437->7; 438->8; 439->9; 440->0; 441->1; 442->2;␣
↪443->3; 444->4; 445->5; 446->6; 447->7; 448->8; 449->9; 450->0; 451->1; 452->2; 453-
↪>3; 454->4; 455->5; 456->6; 457->7; 458->8; 459->9; 460->0; 461->1; 462->2; 463->3;␣
↪464->4; 465->5; 466->6; 467->7; 468->8; 469->9; 470->0; 471->1; 472->2; 473->3; 474-
↪>4; 475->5; 476->6; 477->7; 478->8; 479->9; 480->0; 481->1; 482->2; 483->3; 484->4;␣
↪485->5; 486->6; 487->7; 488->8; 489->9; 490->0; 491->1; 492->2; 493->3; 494->4; 495-
↪>5; 496->6; 497->7; 498->8; 499->9; 500->0; 501->1; 502->2; 503->3; 504->4; 505->5;␣
↪506->6; 507->7; 508->8; 509->9; 510->0; 511->1; 512->2; 513->3; 514->4; 515->5; 516-
↪>6; 517->7; 518->8; 519->9; 520->0; 521->1; 522->2; 523->3; 524->4; 525->5; 526->6;␣
↪527->7; 528->8; 529->9; 530->0; 531->1; 532->2; 533->3; 534->4; 535->5; 536->6; 537-
↪>7; 538->8; 539->9; 540->0; 541->1; 542->2; 543->3; 544->4; 545->5; 546->6; 547->7;␣
↪548->8; 549->9; 550->0; 551->1; 552->2; 553->3; 554->4; 555->5; 556->6; 557->7; 558-
↪>8; 559->9; 560->0; 561->1; 562->2; 563->3; 564->4; 565->5; 566->6; 567->7; 568->8;␣
↪569->9; 570->0; 571->1; 572->2; 573->3; 574->4; 575->5; 576->6; 577->7; 578->8; 579-
↪>9; 580->0; 581->1; 582->2; 583->3; 584->4; 585->5; 586->6; 587->7; 588->8; 589->9;␣
↪590->0; 591->1; 592->2; 593->3; 594->4; 595->5; 596->6; 597->7; 598->8; 599->9; 600-
↪>0; 601->1; 602->2; 603->3; 604->4; 605->5; 606->6; 607->7; 608->8; 609->9; 610->0;␣
↪611->1; 612->2; 613->3; 614->4; 615->5; 616->6; 617->7; 618->8; 619->9; 620->0; 621-
↪>1; 622->2; 623->3; 624->4; 625->5; 626->6; 627->7; 628->8; 629->9; 630->0; 631->1;␣
↪632->2; 633->3; 634->4; 635->5; 636->6; 637->7; 638->8; 639->9; 640->0; 641->1; 642-
↪>2; 643->3; 644->4; 645->5; 646->0; 647->7; 648->8; 649->9; 650->0; 651->1; 652->2;␣
↪653->3; 654->4; 655->5; 656->6; 657->7; 658->8; 659->9; 660->0; 661->1; 662->2; 663-
↪>3; 664->4; 665->5; 666->6; 667->7; 668->8; 669->9; 670->0; 671->1; 672->2; 673->3;␣
↪674->4; 675->5; 676->6; 677->7; 678->8; 679->9; 680->0; 681->1; 682->2; 683->3; 684-
```

(continues on next page)

(continued from previous page)

```
}

Elapsed Time:
10.521423
```

The solution to Crack The Code is 052!

Zoo, Buses and Kids

Zoo, Buses and Kids is a small optimization problem taken from the DMCommunity challenges.

Zoo, Buses and Kids

300 kids need to travel to the London zoo. The school may rent 40 seats and 30 seats buses for 500 and 400 £. How many buses of each to minimize cost?

You can probably solve this problem without the use of a computer, but it's still a nice showcase of optimization within cDMN.

We start by creating a glossary. We need a type to represent buses, sizes, price ranges, number of kids and the number of buses.

For every bus, we need to express exactly one price, and exactly one size. Furthermore, we need a way to express how many buses of each type we will have, and how many children will be on each bus type. Since all of these concepts map one type on exactly one type, we use functions for this purpose.

To represent our total number of kids which need seating and the total price of all the buses, we use constants.

The next step is inputting the data in our model. We know for each bus type their size and their price, and we represent this in a data table.

We also already know the total number of kids.

We now express two things in order to complete the logic:

1. Every kid needs a bus.
2. Buses cannot have a number of children exceeding their size.

This first rule is state by the following table. It checks that the sum of the number of kids assigned to each bus type equals the total number of kids.

The second rule is expressed by a constraint. For every type of bus, there cannot be more children than the size of the type multiplied by the number of the type.

After implementing these two rules, we still need to add the formula for calculating the total price, and then tell the solver to minimize it by adding a `Goal` table.

We can now run our model using the cDMN solver. This results in the following output:

```
Number of models: 1
Model 1
=======
structure  : V {
  Kids = { Big->240; Small->60 }
  Number = { Big->6; Small->2 }
  Price = { Big->500; Small->400 }
  Size = { Big->40; Small->30 }
```

(continues on next page)

(continued from previous page)

```
  TotalKids = 300
  TotalPrice = 3800
}

Elapsed Time:
0.091493
```

The optimal solution is to get 6 big buses, and 2 small ones for a total cost of 3800 pounds.

Duplicate Product Lines

This page details our solution to the Define Duplicate Product Lines challenge from dmcommunity.org

---

Define Duplicate Product Lines

Let's assume that your organization handles sales orders similar to the one below:

Product SKU	Price	Quantity
SKU-1000	10	10
SKU-1	20	3
SKU-2	30	1
SKU-1	20	4
SKU-3	40	6
SKU-3	42	8
SKU-4	15	2

> Create a decision model that is capable to decide which product lines inside such orders are duplicate. The duplicate product lines can be recognized by user-defined similarity rules, e.g. two product lines are considered "duplicate" if they have the same "sku" and "price".

To solve this challenge, we simply need to check whether two orders have the same properties. To be able to model these properties, we first define them in our glossary. We create a type for every product (one to seven), a type containing the possible SKU's, a type containing the possible prices, and a type representing the quantities.

Because every product has exactly one SKU, Price and Quantity, we model these properties using functions.

We also need a way to express that two products are duplicates of each other. For this purpose, we introduce a relation.

Now our glossary is complete, and we can start add logic. For this specific challenge, the logic is simple: if two products share the same SKU and Price, they are duplicates. We express this in a table as follows.

In other words, the table above states "For every Product p1 and p2 (not equal to p1), if they have the same SKU and the same Price, they are duplicates."

Now, all that remains is add in our data table containing the product information.

Running the solver using this model gives us the following solution.

```
Model 1
=======
structure  : V {
  Product_is_duplicate_of_Product = { 2,4; 4,2 }
  Price = { 1->10; 2->20; 3->30; 4->20; 5->40; 6->42; 7->15 }
  Quantity = { 1->10; 2->3; 3->1; 4->4; 5->6; 6->8; 7->2 }
```

(continues on next page)

(continued from previous page)

```
  SKU = { 1->SKU1000; 2->SKU1; 3->SKU2; 4->SKU1; 5->SKU3; 6->SKU3; 7->SKU4 }
}

Elapsed Time:
0.087904
```

Calculator with Two Buttons

The following problem was posed as the DMCommunity November 2020 challenge.

Calculator with Two Buttons

A calculator, initially displaying 0, has only two buttons:

- "+" adds 1 to the number on the display;
- "x" multiplies the number on the display by 10.

What is the least number of button presses needed to show 5034?

This problem can easily be solved by hand, but it is always fun to create a cDMN specification for these types of problems.

As always, we start by creating a glossary. We need a type to represent presses, the two types of buttons and the value of the number.

For every press, we need to represent what button was pressed and what value the display showed beforehand. To do this, we create two functions, *button of Press* and *value of Press*. In this way, if *button of 5 = mult*, this would mean that the sixth button pressed (starting at zero) was a multiplication. Similarly for *value of 5 = 4*, then the value before the sixth button was pressed is 4.

To represent the number of presses needed to reach our goal of 5034, we introduce a constant called *NbPresses*.

With our glossary done, we can start constructing tables. The first table we add is a constraint table in order to specify that our starting number is zero, by fixing the value of the first press.

After doing so, we add the constraints stating how the value changes when a button is pressed. The first rule of the below table states that when the *add* button is pressed, the next value is the previous value plus one. The second rule states that when the *mult* button is pressed, the next value is the previous value times ten.

Now we define what that the value of *NbPresses* is the press when the value is 5034.

Now that we have all the required logic in place, we simply need to tell the system that we want to minimize the number of presses via the `Goal` table.

We can now run our model using the cDMN solver. This results in the following output:

```
Number of models: 1
Model 1
=======
structure  : V {
  Button = { 0->add; 1->add; 2->add; 3->add; 4->add; 5->mult; 6->mult; 7->add; 8->add;
↪ 9->add; 10->mult; 11->add; 12->add; 13->add; 14->add; 15->add; 16->add; 17->add;␣
↪18->add; 19->add; 20->add }
  NbPresses = 15
  Value = { 0->0; 1->1; 2->2; 3->3; 4->4; 5->5; 6->50; 7->500; 8->501; 9->502; 10->
↪503; 11->5030; 12->5031; 13->5032; 14->5033; 15->5034; 16->5035; 17->5036; 18->5037;
↪ 19->5038; 20->5039 }
```

(continues on next page)

(continued from previous page)

```
}

Elapsed Time:
0.312465
```

The number 5034 can be reached in 15 presses!

Virtual Chess Tournament

This example is another challenge posted by the dmcommunity.org as a challenge. It is the December 2020 challenge. Its specification is as follows:

Virtual Chess Tournament

Three world champions Fischer, Kasparov, and Karpov played in a virtual chess tournament. Each player played 7 games against two other opponents. Each player received 2 points for a victory, 1 for a draw, and 0 for a loss. We know that Kasparov, known as the most aggressive player, won the most games. Karpov, known as the best defensive player, lost the least games. And Fischer, of course, won the tournament.

Questions

1. Is it really possible? Does this problem have a solution?
2. If yes, what is the final score?
3. Are there other possible solutions?

This is quite a cool problem to model in cDMN. As always, we start by creating the glossary. We first need to create types to represent the following concepts:

- Person (Fischer, Kasparov, Karpov)
- Game (1 through 7)
- Score
- Result (win, loss, draw)

Thus, we create a glossary table as such:

Now we need to think about what other concepts we need to express. First of all, we need a way to keep track of the result of games between two players. This could be represented by a function, as every game has exactly one result. To keep track of the final scores of a player, we will also use a function, for the same reason (every player has exactly one final score). Lastly, we will want to count the total number of wins and losses of every player, and again we can perfectly use functions here.

And that's all we need for this specific challenge. Now that our glossary is done, we can move on to implementing the logic. We start by implementing the table to count the score. To do this, we "loop over" every game played, we check the result and we set the score accordingly. The first rule in the table table below can be read as "For every person p1 and person p2 (that is not p1) and game, if the result of the game was a Win, then we add 2 to the score of the first person.

One important caveat is that we must not forget to express the asymmetry within the *result of Game between p1 and p2* function. Indeed, if p1 has won from p2, then p2 must always have lost from p1. To do this, we can add a constraint table to make sure that this is always the case.

Next, we still need a way to count the number of wins and losses for every player. This can easily be done by introducing count tables. It works similarly to the table to count the scores: we loop over every game, and add 1 to the number of games won if the result was a win.

Now all that remains is creating the score, wins and losses constraints. To express that "Fischer has won", we want to express that he always has the highest score. We express "Kasparov wins the most games" and "Karpov loses the fewest games" similarly, all in one constraint table (although it could as easily be expressed in three different ones).

We can now run our model using the cDMN solver. For example, this is one of the solutions found by the solver:

```
Number of models: 1
Model 1
=======
structure  : V {
  games_lost_of_Person = { Fischer->3; Karpov->2; Kasparov->5 }
  games_won_of_Person = { Fischer->4; Karpov->1; Kasparov->5 }
  result_of_Game_between_Person_and_Person = { 1,Fischer,Fischer->Loss; 1,Fischer,
↪Karpov->Draw; 1,Fischer,Kasparov->Win; 1,Karpov,Fischer->Draw; 1,Karpov,Karpov->
↪Draw; 1,Karpov,Kasparov->Draw; 1,Kasparov,Fischer->Loss; 1,Kasparov,Karpov->Draw; 1,
↪Kasparov,Kasparov->Loss; 2,Fischer,Fischer->Win; 2,Fischer,Karpov->Draw; 2,Fischer,
↪Kasparov->Win; 2,Karpov,Fischer->Draw; 2,Karpov,Karpov->Draw; 2,Karpov,Kasparov->
↪Win; 2,Kasparov,Fischer->Loss; 2,Kasparov,Karpov->Loss; 2,Kasparov,Kasparov->Loss;␣
↪3,Fischer,Fischer->Loss; 3,Fischer,Karpov->Draw; 3,Fischer,Kasparov->Loss; 3,Karpov,
↪Fischer->Draw; 3,Karpov,Karpov->Loss; 3,Karpov,Kasparov->Loss; 3,Kasparov,Fischer->
↪Win; 3,Kasparov,Karpov->Win; 3,Kasparov,Kasparov->Win; 4,Fischer,Fischer->Loss; 4,
↪Fischer,Karpov->Draw; 4,Fischer,Kasparov->Loss; 4,Karpov,Fischer->Draw; 4,Karpov,
↪Karpov->Win; 4,Karpov,Kasparov->Draw; 4,Kasparov,Fischer->Win; 4,Kasparov,Karpov->
↪Draw; 4,Kasparov,Kasparov->Draw; 5,Fischer,Fischer->Win; 5,Fischer,Karpov->Draw; 5,
↪Fischer,Kasparov->Loss; 5,Karpov,Fischer->Draw; 5,Karpov,Karpov->Win; 5,Karpov,
↪Kasparov->Draw; 5,Kasparov,Fischer->Win; 5,Kasparov,Karpov->Draw; 5,Kasparov,
↪Kasparov->Draw; 6,Fischer,Fischer->Win; 6,Fischer,Karpov->Draw; 6,Fischer,Kasparov->
↪Win; 6,Karpov,Fischer->Draw; 6,Karpov,Karpov->Loss; 6,Karpov,Kasparov->Draw; 6,
↪Kasparov,Fischer->Loss; 6,Kasparov,Karpov->Draw; 6,Kasparov,Kasparov->Draw; 7,
↪Fischer,Fischer->Win; 7,Fischer,Karpov->Draw; 7,Fischer,Kasparov->Win; 7,Karpov,
↪Fischer->Draw; 7,Karpov,Karpov->Loss; 7,Karpov,Kasparov->Loss; 7,Kasparov,Fischer->
↪Loss; 7,Kasparov,Karpov->Win; 7,Kasparov,Kasparov->Draw }
  score_of_Person = { Fischer->15; Karpov->13; Kasparov->14 }
}

Elapsed Time:
6.214157
```

And there is our answer to question one and two.

1. It is really possible
2. The final score is Fischer: 15, Karpov: 13 and Kasparov: 14.

The answer for the third question is a bit more difficult. Because the search space is rather big, the internal solver has difficulties finding all the solutions. Therefore, it is unable to conclude in reasonable time that there exist solutions in which there is a different result for the final scores.

SET

SET is a board game published by Set Enterprises, Inc, in which the goal is to find a *set* of cards. Concretely, the rules are as follows:

SET

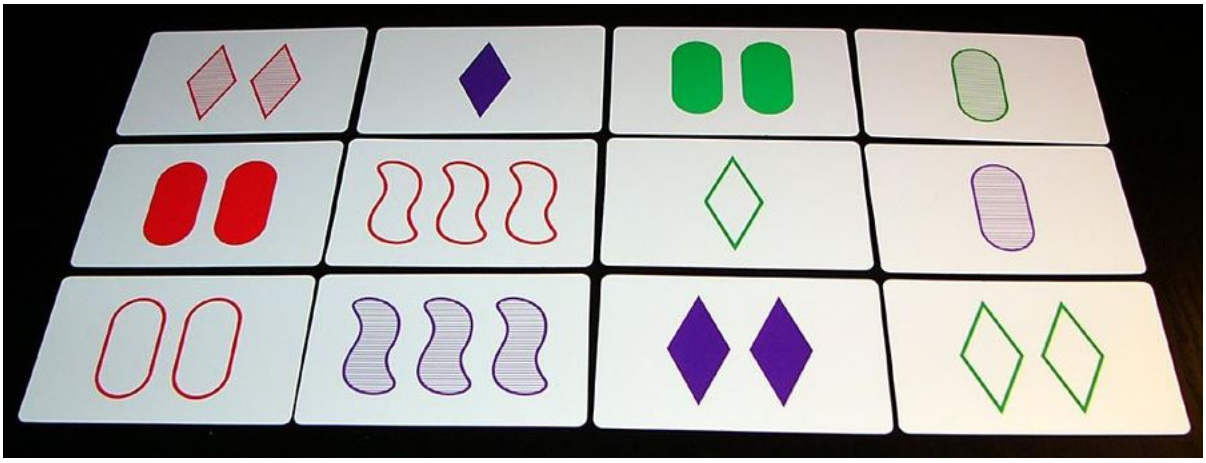
12 cards are placed on the table, each with four attributes:

1. The symbol on the card (diamonds, squiggles, rectangles)
2. The number of symbols (one, two, three)
3. The color of the symbols (red, green or purple)
4. The fill of the symbols (striped, filled, empty)

The goal is now to find a set of **three** cards, that either have the same value or a different value for every attribute.

For example, a possible set in the image below would be the most top-left card together with the two bottom-right cards, as they each have:

- The same symbol (diamonds)
- The same number (2)
- Different colors
- Different fills



First, we start building the glossary of the model. There are a few types that should be declared:

- Card (we will use indices 1 through 12)
- Color
- Number
- Shape
- Fill

Next, we want to add a way to link card indices to card attributes. For this, functions are perfect: every card has exactly 1 value for every attribute. The concrete values for these functions will be filled out later in a data table.

We also need a way to keep track of what cards are selected to be in our set, and if they have the same/different color, same/different number, ... We will use a relation for the former, and booleans for the latter (as these are either the same, or different).

Finally, to keep track of how many cards have been selected by the system, we introduce a constant.

Now that our glossary is done, we can move on to implementing the rules. These are actually quite straight forward. For every attribute, we want to say the following:

Rule

For every two cards that are selected, if SameAttribute is true, their attribute values should be the same. Similarly, if SameAttribute is false, their attribute values should be different.

Concretely, this is implemented in the following constraint tables:

The *shape constaint* table, for example, reads as follows: "For every Card c1 and Card c2 (that isn't c1), if both c1 and c2 are selected and SameShape is true, then their shape should be the same. If SameShape is false, their shape should be different."

To finish up, we want to make sure that a set always consists of three selected cards. For that, we add a table which counts the number of selected cards, followed by a constraint on that number.

Having entered all the rules, all that rests is filling out our data table. We use the example shown in the image above here.

We can now run our model using the cDMN solver. In total, we are able to find 5 sets!

```
Model 1
==========
color_of_Card:={(1)->red; (2)->purple; (3)->green; (4)->green; (5)->red; (6)->red;␣
↪(7)->red; (8)->purple; (9)->red; (10)->purple; (11)->purple; (12)->green}
number_of_Card:={(1)->two; (2)->one; (3)->two; (4)->one; (5)->two; (6)->three; (7)->
↪one; (8)->one; (9)->two; (10)->three; (11)->two; (12)->two}
fill_of_Card:={(1)->striped; (2)->filled; (3)->filled; (4)->striped; (5)->filled; (6)-
↪>empty; (7)->empty; (8)->striped; (9)->empty; (10)->striped; (11)->filled; (12)->
↪empty}
shape_of_Card:={(1)->diamonds; (2)->diamonds; (3)->rectangles; (4)->rectangles; (5)->
↪rectangles; (6)->squiggles; (7)->diamonds; (8)->rectangles; (9)->rectangles; (10)->
↪squiggles; (11)->diamonds; (12)->diamonds}
NbCards:={->3}
selected_Card:={(1)->true; (2)->false; (3)->false; (4)->false; (5)->false; (6)->false;
↪ (7)->false; (8)->false; (9)->false; (10)->false; (11)->true; (12)->true}
SameColor:={->false}
SameNumber:={->true}
SameShape:={->true}
SameFill:={->false}

Model 2
==========
color_of_Card:={(1)->red; (2)->purple; (3)->green; (4)->green; (5)->red; (6)->red;␣
↪(7)->red; (8)->purple; (9)->red; (10)->purple; (11)->purple; (12)->green}
number_of_Card:={(1)->two; (2)->one; (3)->two; (4)->one; (5)->two; (6)->three; (7)->
↪one; (8)->one; (9)->two; (10)->three; (11)->two; (12)->two}
fill_of_Card:={(1)->striped; (2)->filled; (3)->filled; (4)->striped; (5)->filled; (6)-
↪>empty; (7)->empty; (8)->striped; (9)->empty; (10)->striped; (11)->filled; (12)->
↪empty}
shape_of_Card:={(1)->diamonds; (2)->diamonds; (3)->rectangles; (4)->rectangles; (5)->
↪rectangles; (6)->squiggles; (7)->diamonds; (8)->rectangles; (9)->rectangles; (10)->
↪squiggles; (11)->diamonds; (12)->diamonds}
NbCards:={->3}
selected_Card:={(1)->false; (2)->false; (3)->false; (4)->true; (5)->false; (6)->true;␣
↪(7)->false; (8)->false; (9)->false; (10)->false; (11)->true; (12)->false}
SameColor:={->false}
SameNumber:={->false}
SameShape:={->false}
SameFill:={->false}

Model 3
```

(continues on next page)

(continued from previous page)

```
==========
color_of_Card:={(1)->red; (2)->purple; (3)->green; (4)->green; (5)->red; (6)->red;␣
↪(7)->red; (8)->purple; (9)->red; (10)->purple; (11)->purple; (12)->green}
number_of_Card:={(1)->two; (2)->one; (3)->two; (4)->one; (5)->two; (6)->three; (7)->
↪one; (8)->one; (9)->two; (10)->three; (11)->two; (12)->two}
fill_of_Card:={(1)->striped; (2)->filled; (3)->filled; (4)->striped; (5)->filled; (6)-
↪>empty; (7)->empty; (8)->striped; (9)->empty; (10)->striped; (11)->filled; (12)->
↪empty}
shape_of_Card:={(1)->diamonds; (2)->diamonds; (3)->rectangles; (4)->rectangles; (5)->
↪rectangles; (6)->squiggles; (7)->diamonds; (8)->rectangles; (9)->rectangles; (10)->
↪squiggles; (11)->diamonds; (12)->diamonds}
NbCards:={->3}
selected_Card:={(1)->false; (2)->false; (3)->false; (4)->false; (5)->false; (6)->true;
↪ (7)->true; (8)->false; (9)->true; (10)->false; (11)->false; (12)->false}
SameColor:={->true}
SameNumber:={->false}
SameShape:={->false}
SameFill:={->true}

Model 4
==========
color_of_Card:={(1)->red; (2)->purple; (3)->green; (4)->green; (5)->red; (6)->red;␣
↪(7)->red; (8)->purple; (9)->red; (10)->purple; (11)->purple; (12)->green}
number_of_Card:={(1)->two; (2)->one; (3)->two; (4)->one; (5)->two; (6)->three; (7)->
↪one; (8)->one; (9)->two; (10)->three; (11)->two; (12)->two}
fill_of_Card:={(1)->striped; (2)->filled; (3)->filled; (4)->striped; (5)->filled; (6)-
↪>empty; (7)->empty; (8)->striped; (9)->empty; (10)->striped; (11)->filled; (12)->
↪empty}
shape_of_Card:={(1)->diamonds; (2)->diamonds; (3)->rectangles; (4)->rectangles; (5)->
↪rectangles; (6)->squiggles; (7)->diamonds; (8)->rectangles; (9)->rectangles; (10)->
↪squiggles; (11)->diamonds; (12)->diamonds}
NbCards:={->3}
selected_Card:={(1)->true; (2)->false; (3)->false; (4)->true; (5)->false; (6)->false;␣
↪(7)->false; (8)->false; (9)->false; (10)->true; (11)->false; (12)->false}
SameColor:={->false}
SameNumber:={->false}
SameShape:={->false}
SameFill:={->true}

Model 5
==========
color_of_Card:={(1)->red; (2)->purple; (3)->green; (4)->green; (5)->red; (6)->red;␣
↪(7)->red; (8)->purple; (9)->red; (10)->purple; (11)->purple; (12)->green}
number_of_Card:={(1)->two; (2)->one; (3)->two; (4)->one; (5)->two; (6)->three; (7)->
↪one; (8)->one; (9)->two; (10)->three; (11)->two; (12)->two}
fill_of_Card:={(1)->striped; (2)->filled; (3)->filled; (4)->striped; (5)->filled; (6)-
↪>empty; (7)->empty; (8)->striped; (9)->empty; (10)->striped; (11)->filled; (12)->
↪empty}
shape_of_Card:={(1)->diamonds; (2)->diamonds; (3)->rectangles; (4)->rectangles; (5)->
↪rectangles; (6)->squiggles; (7)->diamonds; (8)->rectangles; (9)->rectangles; (10)->
↪squiggles; (11)->diamonds; (12)->diamonds}
NbCards:={->3}
selected_Card:={(1)->false; (2)->false; (3)->true; (4)->false; (5)->false; (6)->false;
↪ (7)->true; (8)->false; (9)->false; (10)->true; (11)->false; (12)->false}
SameColor:={->false}
SameNumber:={->false}
SameShape:={->false}
```

(continues on next page)

(continued from previous page)

```
SameFill:={->false}

No more models.
```

Covid Testing

"Covid Testing" is the May 2021 DMCommunity challenge, and is based on a Covid triage procedure that was at one point used by the Belgian government.

Covid Testing

We ask to submit a knowledge specification, through DM tables or otherwise, that describes the following triage procedure that determines for (a group of) patients who should undergo subsequent testing:

1. Patients present sneezing and/or coughing as symptoms; this is determined by what the patient him or herself details (i.e., anamnesis).
2. A patient's temperature is measured. The default fever threshold is 38°C; due to different methods of measurement for young children and their associated accuracy, for patients younger than 10 the fever threshold is considered to be 37.2°C.

Based on the first two steps, the number of presented symptoms is determined. If a patient presents at least two (2) of these symptoms, subsequent testing is performed, otherwise standard quarantine practices are advised.

In this knowledge specification, pay special attention to the nature of the pandemic as an ongoing crisis with expertise being gained continuously. Specifically, take into account how the following scenarios impact your specification:

The possible emergence of additional symptoms such as loss of smell or loss of taste. A different threshold of symptoms deemed present before referral to subsequent testing.

This challenge does actually not require any of the new concepts that cDMN brings to the table – as such, the implementation will consist purely of standard DMN.

First, we start by building the glossary. Just from reading the challenge description, it is clear that we need ways to represent the following things:

- Age
- Temperature
- The number of total symptoms
- Whether a person suffers from coughing
- Whether a person suffers from sneezing
- Whether a person suffers from fever
- Whether a PCR test is advisable

The first three concepts can easily be represented using constants, while the latter four concepts are perfectly fit to be represented by boolean variables. Note that no type glossary is needed, because all types are either *int* or *real*.

Using this small glossary, we will be able to model the challenge. First, we should decide if a patient has a fever.

Rule

The default fever threshold is 38°C; due to different methods of measurement for young children and their associated accuracy, for patients younger than 10 the fever threshold is considered to be 37.2°C.

Using an F-table, we can elegantly model this rule. Note that even though a U-table could also have been used, we chose for an F-table because it better fits the *default reasoning* used in the rule. By default reasoning, we mean logic that consists of a set of IF-ELSE sentences, followed by a final ELSE statement. E.g., we can rewrite the above statement to: "If a patient is younger than 10 and has a temperature higher than 37.2, they have a fever. Else if the patient is older than 10 and has a temperature higher than 38, they also have a fever. Otherwise, they have no fever."

Next, we introduce a table to count the number of symptoms. Instead of using a count table, we instead opt for a sum table, as this will allow us to use weighted sums in the case some symptoms become more important than others. Another advantage of using this table structure is that adding new symptoms in the future will be easy. Indeed, only 1 row and 1 column need to be added, consisting mostly of cells containing "-".

And finally, we add a table to define if a PCR Test is necessary.

By now also adding a data table, we can enter patient data and use the model to find out if they require further PCR testing.

We can now run our model using the cDMN solver.

```
Model 1
==========
Temperature:={->38.2}
Age:={->18}
NbSymptoms:={->2}
Sneezing:={->false}
Coughing:={->true}
Fever:={->true}
PCRTest:={->true}
Quarantine:={->true}
```

Where is Gold?

This is an implementation for the June 2021 DMCommunity challenge. The challenge is as follows:

Where is Gold?

There are three boxes, but only one of them has gold inside. Additionally, each box has a message printed on it. One of these messages is true and the other two are lies.

- The first box says, "Gold is in this box".
- The second box says, "Gold is not in this box".
- The third box says, "Gold is not in Box 1".

Which box contains the gold?

As always with the DMCommunity challenges, this is quite a fun one to implement. We start by building our glossary. It is clear that we need a way to reason on boxes, so for we first introduce a type *Box*.

Next, we need a way to express what boxes "speak the truth", and what boxes "lie". Because the boolean nature of truth/lie, we can use a relation (representing *true/false* for every box's statement).

To express what box contains the gold, we can make perfect use of a constant, *GoldBox*. Similarly, to express the number of boxes speaking the truth, we introduce a constant *NbTruth*.

With our glossary finished, we continue to our decision/constraint tables. We start by defining when a box speaks the truth.

The rules of the table can be read as follows:

- For box 1: if it contains the gold, it speaks the truth.
- For box 2: if it does not contain the gold, it speaks the truth.
- For box 3: if box 1 does not contain the gold, it speaks the truth.

We now count how many boxes speak the truth, in order to then create a constraint on it.

The first table expresses that "For every box *b*, if it speaks the truth, add 1 to *NbTruth*". The second table then expresses a constraint that *NbTruth* should always be 1, i.e., only one box may speak the truth.

With those two tables created, we are done! We can now run our model using the cDMN solver, to find out which box contains the gold.

```
Model 1
==========
GoldBox:={->2}
NbTruth:={->1}
Box_speaks_truth:={1->false; 2->false; 3->true}

No more models.
Elapsed Time:
0.679s
```

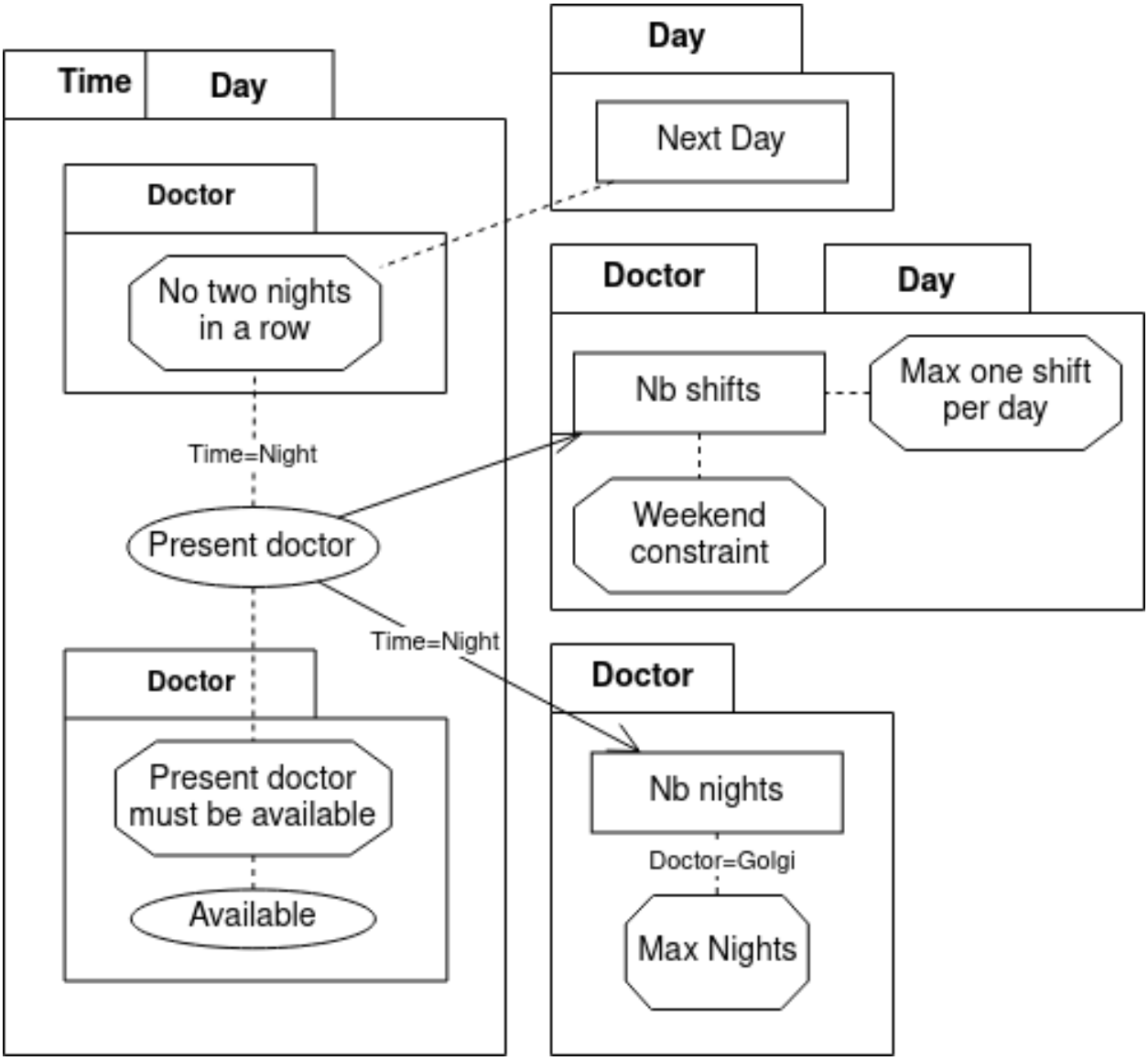
5.5 cDRD

In standard DMN, it is possible to draw the structure of a DMN model in a so-called Decision Requirements Diagram. This DRD is a graph which expresses the structure by representing the connections between inputs, decisions, knowledge sources and more. Creating a DRD has two main advantages:

1. It improves interpretability of models by end-users.
2. It enhances the traceability of decisions, as it becomes easier to check why a certain decision was made by manually following the decisions in the graph from the inputs to the top decision.

In cDMN, we introduce the cDRD. Similar to the DRD, it shows the structure of a cDMN model: more specifically, it shows the declarative relations between variables. However, there are a few key differences between the two.

We will now explain these differences using a running example based on the *Doctor Planning* case. The cDRD for this case is as follows:



Defining Variables As in regular DRDs, a decision table is represented by drawing an arrow to the variable that is defined by the table from each of the other variables that appear in the table. However, in cDMN this does not mean that the table can only be used to compute the defined variable. It does not imply a computing order, but rather an order of defining.

Quantification Another difference is that while DMN tables always define a single variable, this is no longer the case in cDMN. For example, *Next day for Day* represents a variable for every value of `Day`, i.e. `Next day for Mon`, `Next Day for Tue`, ... The decision table which defines the function contains logic that is applied multiple times, once for every day. To denote this, we use the so-called "Plate Notation". Variables that contain a type are drawn with a type plate around them, to denote that all nodes inside the plate represent a variable for every instance of the type. To increase readability, all nodes inside the plate can leave out the type name (`Next day` instead of `Next day for day`).

Constraint tables are drawn by a rectangle with clipped corners. They do not define any variables, but merely express a constraint over the values of variables. All variables should be connected to the node by a dotted line, with optional arrowhead.

Specific values: In cDMN it is possible to write a function/relation using a specific value for a type in the table header. For example, the table below creates a constraint specifically for `Golgi`. To represent this in the cDRD, we add

`Doctor=Golgi` as a small annotation to the `Doctor` arrow.

5.6 Python DMN API

The cDMN Python package now also comes with a DMN API, that can be used from inside a Python program. It has the following features:

- Query information about the DMN
- Retrieve list of variables
- Retrieve dependencies of variables
- Find meta-information about variables
- Set the values of certain variables
- Propagate the values throughout the model
- and more

5.6.1 Start

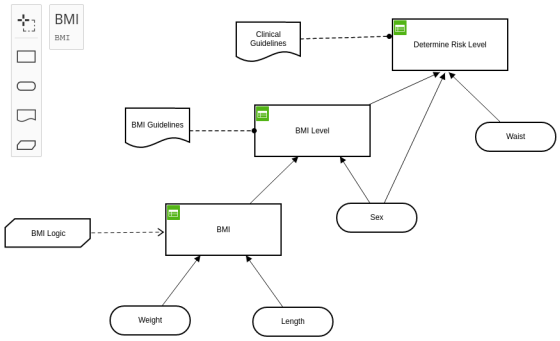
In order to use the DMN API, start by importing the module.

```
from cdmn.API import DMN
```

The next step is to load a DMN specification. This specification can either be given as a string (using the `str` argument) or as a path to a `.dmn` file. We will demonstrate this using the *BMILevel.dmn* example.

```
specification = DMN('./path/to/bmi.dmn')
```

If you would like to try out this example for yourself, you may download the DMN file. For context, this is what the example model looks like:



5.6.2 Query Information

Now that our DMN model has been loaded, we can query information such as inputs, outputs and other variables using `get_inputs()`, `get_outputs()` and `get_intermediary()`.

```
print("inputs:", spec.get_inputs())
print("outputs:", spec.get_outputs())
print("other:", spec.get_intermediary())
```

results in:

```
inputs: ['weight', 'sex', 'length', 'waist']
outputs: ['riskLevel']
other: ['BMILevel', 'bmi']
```

The API also allows us to query dependencies of tables. For example, if we want to find out what variables need to be set in order to calculate the BMI, we can use the `dependencies_of` method. This returns every dependency, together with their "distance" to the variable.

```
print('Dependencies of BMI:', spec.dependencies_of('bmi'))
print('Dependencies of riskLevel:\n', spec.dependencies_of('riskLevel')
```

results in:

```
Dependencies of BMI: {'weight': 0, 'length': 0}
Dependencies of riskLevel: {'BMILevel': 0, 'bmi': 1, 'weight': 2, 'length': 2, 'sex':␣
↪0, 'waist': 0}
```

We can also query the possible values of a type.

```
print(spec.possible_values_of('riskLevel'))
```

```
High, Increased, Very High, Low, Extremely_High
```

5.6.3 Interacting with a DMN Model

Besides viewing information, we are also able to enter our own data, and actually put the DMN model to use. Setting the value of a variable can be done using the `set_value` method.

```
spec.set_value('weight', 74)
spec.set_value('length', 1.79)
```

After setting values, it makes sense to infer information with them. Currently, the DMN API allows for two "inference tasks". These methods translate the DMN model into an IDP specification, which is then executed.

- `model_expand`: find a set of assignments to the variables that satisfies the DMN specification.
- `propagation`: using the information that we currently have, find out if we can derive the values of other variables.

Model Expansion

Model expansion can be used as such:

```
print(spec.model_expand().getvalue())
```

This will print a max of 10 solutions ("models") in the following form:

```
Model 1
==========
BMILevel:={->Underweight}
bmi:={->23.09540900720951}
weight:={->74}
sex:={->Male}
riskLevel:={->Low}
length:={->1.79
```

Propagation

Using the propagation inference, we attempt to infer the value of other variables. In other words: "using our current assignment of variables, are there any other variables that have a known value (e.g. can only have 1 value)?" After executing the propagation, we can use the `is_certain()` method to query whether a variable is certainly known.

For example:

```
spec.propagate()
if spec.is_certain('bmi'):
    print('BMI:', spec.value_of('bmi'))

 if spec.is_certain('riskLevel'):
        print('riskLevel:', spec.value_of('riskLevel'))

 if spec.is_certain('BMILevel'):
     print('BMILevel:', spec.value_of('BMILevel'))
 else:
     print('BMILevel is still unknown.')
```

results in the following output:

```
BMI: 23.09540900720951
riskLevel: Low
BMILevel is still unknown.
```

Because the length and weight were already known, the system was capable of deriving that `BMI` is also known (table 1). Furthermore, the value of `riskLevel` was also derived (table 2). The `BMILevel` on the other hand, can still have multiple values.

Autopropagation

It is also possible to have the system automatically apply the propagation inference whenever a new variable is entered. When loading in a DMN file, supply the `auto_propagate` flag as follows:

```
spec = DMN(path='./path/to/bmi.dmn', auto_propagate=True)

spec.set_value('weight', 74)
spec.set_value('length', 1.79)
if spec.is_certain('bmi'):
    print('Your BMI:', spec.value_of('bmi'))
```

```
Your BMI: 23.095
```

Multidirectionality

A feature of our DMN API is that it has no sense of directionality. I.e., it is not necessary to use the DMN specification from inputs to outputs. In the following example, we calculate the BMI as before. However, if the BMI is too high, we also calculate what weight the patient should have in order to be healthy. In this way, the propagation can work 'backwards', using the same logic!

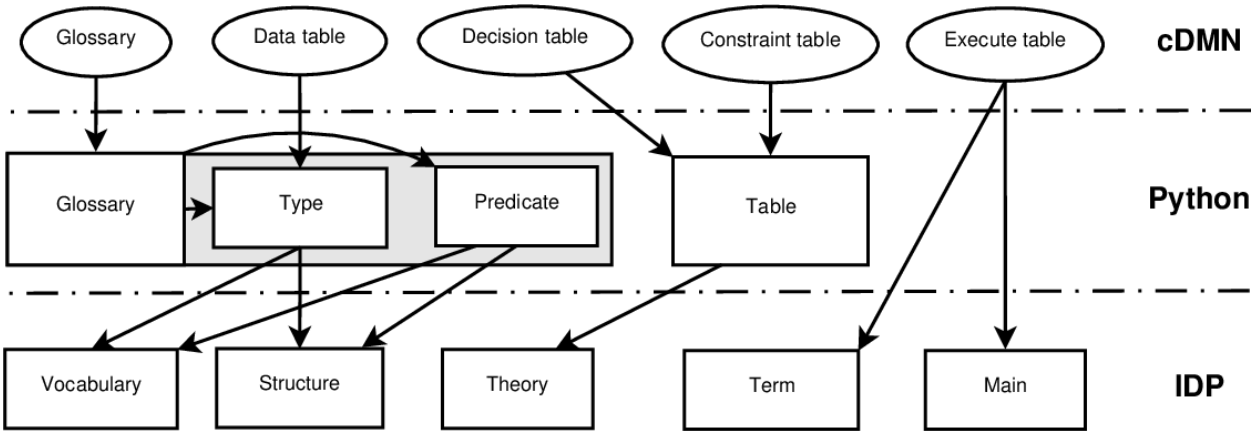
```
spec.clear()  # clears all the previously set values

spec.set_value('weight', 97)
spec.set_value('length', 1.79)
print("Your bmi is:", spec.value_of('bmi'))

if spec.is_certain('bmi') and float(spec.value_of('bmi')) > 25:
    print('Your BMI is too high!')
    # Get current weight.
    cur_weight = spec.value_of('weight')

    # Calculate ideal weight.
    spec.set_value('weight', None)
    spec.set_value('bmi', 25)
    ideal_weight = float(spec.value_of('weight'))
    print("For a healthy bmi, you should weigh {} kg.".format(ideal_weight))
    print("This is a difference of {} kg.".format(round(cur_weight - ideal_weight,␣
↪2)))
```

```
Your bmi is: 30.273711806747606
Your BMI is too high!
For a healthy bmi, you should weigh 80.1025 kg.
This is a difference of 16.9 kg
```

5.7 cDMN to FO() conversion

This page details the full cDMN to FO() conversion. This conversion is done in two steps; first from cDMN to Python objects, then from these objects to the IDP system. In *1. cDMN to Python conversion*, an explanation is given for the translation of cDMN to Python objects. In *2. Python to IDP conversion*, the conversion of Python to FO() is explained. An example of what the result of the conversion looks like is given in *3. cDMN to IDP example*.

The basic conversion can be seen in the following image.



5.7.1 1. cDMN to Python conversion

The first step in the conversion to Python objects, is for Python to open the `.xlsx` file. This is done using the OpenPyXL package.

The solver reads all the tables, organises them by type and then parses them in the following order:

1. Sub-glossary tables
2. Data tables
3. Decision tables and Constraint tables
4. The goal table

1.1 Glossarries

The glossaries are parsed in the following order:

1. Type
2. Function
3. Constant
4. Relation
5. Boolean

The order of the latter four doesn't really matter, but it is necessary to always parse the Type sub-glossary first. Parsing this sub-glossary creates Python `type` objects. These objects contain a name, their cDMN supertype and the possible values for that type.

After creating a `type` object for every row in the Type sub-glossary, the other sub-glossaries are parsed next. Functions and relations need to be able to know what types they contain in their name, hence the need to parse the types first.

1.2 Data tables

The values in data tables are stored in dictionaries, to make it easier to format them later on for the IDP system. When interpreting the values for a type in a data table, there exist two possiblities. Either the `Values` column of the type was filled out in the sub-glossary with values, or it contained a reference to a data table such as `see Data Table x`. In the former case, each value in the data table is checked on validity. If for instance value `4` is found for a type which defined its range as `[5..10]`, the solver throws an error. In the latter case, every value found in the data table is added to the type's list of possible values.

1.3 Decision tables and Constraint tables

Decision and constraint tables are represented in a simple way. They are converted into `Table` objects, which contain a list of inputs, a list of outputs, the list of rows, their name and their hit policy. No more information is necessary in order to represent these tables in Python.

1.4 Goal table

The goal table is the easiest of all: when it's read, it simply creates a dict which contains the inference method (either model expand, minimize or maximize) and the amount of requested models.

5.7.2 2. Python to IDP conversion

Converting the Python objects to IDP is done in a certain order. For the IDP system, five different blocks of code need to be generated:

1. Vocabulary
2. Structure
3. Theory
4. Term
5. Main

The order of this code does not really matter, as long as the main is at last code block.

2.1 Vocabulary

The vocabulary block contains the definitions for the types, functions and predicates used in the IDP model. To create this block, every relevant Python object has a `to_idp_voc` method to generate the relevant code. For instance, for `Type` objects, this returns the type name followed by `isa` and its subtype. Relations (and booleans) are formatted as `name(arg0, arg1, ...)`. Functions (and constants) are formatted as `name(arg0, arg1, ...): returnval`.

2.2 Structure

The structure defines all the possible values for the known types, as well as the satisfied predicates and fully known functions. For this, each `Type` and `Predicate` object has a `to_idp_struct` method. For types, this generates `typename = values`.

For relations, this is of the form `relation_name = { }`, with between the brackets each set of argument values seperated by semicolons. The arguments inside the set are seperated by a comma.

For functions, this is of the form `function_name = { }`, with between the brackets each set of argument values seperated by semicolons. The arguments inside a set are seperated by a comma. The arguments and the returnvalue are seperated by a `->`.

2.3 Theory

The theory contains the different rules. To create the theory, each `Table` object is translated based on their hit policy.

Hit Policy	IDP concept
U	Definitions
F	Definitions
A	Definitions
C+, C#, ..	Aggregates
E*	Constraints

2.4 Term and Main

Both the term and the main block are fully created out of the dictionary formed by the goal table. The term only exists for optimization problems: no term is created for model expansion.

The main is mostly the same every time, except for two parameters: the inference task (model expansion or a form of optimization) and the amount of models.

5.7.3 3. cDMN to IDP example

As an example, we show the conversion of the Who Killed Agatha challenge. First we start off with the translation of the glossary:

The glossary tables translate into `Vocabulary` block of the IDP system. The above tables would result in the following vocabulary:

```
Vocabulary V {
    type Person constructed from { Agatha, Butler, Charles }
    type Number = { 0..100 } isa int

    Hated_persons(Person): Number
    _Hated_persons(Person): Number

    Killer: Person

    Person_hates_Person(Person, Person)
    Person_richer_than_Person(Person, Person)
}
```

We can see that every row in the glossary results into a row in the vocabulary. The only exception to this are functions which are used in aggregates such as `C#`: they also need an auxiliary function to deal with a bug in the IDP system. This auxiliary function has the same name, but start with an underscore (as can be seen here by `Hated_persons` and `_Hated_persons`).

All the other glossary elements underwent obvious transformations:

- String types are now types which use `constructed from`.
- Int types use `= {x..y} isa int` as their definition.
- Functions are translated into `Func_name(Arg1, Arg2, ...): Return_Value`.
- Constants are translated into `Const_name: Return_Value`.
- Relations are translated into `Rel_name(Arg1, Arg2, ...)`.
- Booleans are translated into `Boolean_name`.

Constraint tables are also translated very clearly. As the table in *2.3 Theory* shows, they are translated into implications. The above table translates to the following in the IDP system:

```
//A killer always hates, and is no richer than his victim
true => Person_hates_Person(Killer, Agatha) & ~(Person_richer_than_Person(Killer,␣
↪Agatha)).
```

This is a very straightforward translation. Note that, because there is no input column, the implication is always true.

This is also a very straightforward table to translate. It translates into the following:

```
//Agatha hates everybody but the butler
!Person[Person]: Person ~= Butler => Person_hates_Person(Agatha, Person).

!Person[Person]: Person = Butler => ~(Person_hates_Person(Agatha, Person)).
```

The above table is a bit harder to translate than the constraint tables. Because of a bug in aggregates in the IDP system, we have to work using an auxiliary function. The actual count is done in the first line.

```
//Count enemies
!p1[Person]:  _Hated_persons(p1) = sum(#{  p2[Person] : Person_hates_Person(p1, p2) &␣
↪p1 = p1 & true }).
{
    !p1[Person]: Hated_persons(p1) = _Hated_persons(p1).
}
```

The full translation of the entire cDMN implementation is shown next. Note how the `Structure` block is empty, because no data table was used.

```
Vocabulary V {
    type Person constructed from { Agatha, Butler, Charles }
    type Number = { 0..100 } isa int

    Hated_persons(Person): Number
    _Hated_persons(Person): Number

    Killer: Person

    Person_hates_Person(Person, Person)
    Person_richer_than_Person(Person, Person)
}

Structure S:V {
}
Theory T:V {
    //A killer always hates, and is no richer than his victim
    true => Person_hates_Person(Killer(), Agatha) & ~(Person_richer_than_
↪Person(Killer(), Agatha)).

    //Agatha hates everybody but the butler
    !Person[Person]: Person ~= Butler => Person_hates_Person(Agatha, Person).

    !Person[Person]: Person = Butler => ~(Person_hates_Person(Agatha, Person)).

    //Charles hates no one that Agatha hates
    !Person[Person]: Person_hates_Person(Agatha, Person) => ~(Person_hates_
↪Person(Charles, Person)) & Person_hates_Person(Butler, Person).

    //Butler hates everyone not richer than Agatha
    !Person[Person]: ~(Person_richer_than_Person(Person, Agatha)) => Person_hates_
↪Person(Butler, Person).

    //Count enemies
    !p1[Person]:  _Hated_persons(p1) = sum(#{  p2[Person] : Person_hates_Person(p1,␣
↪p2) & p1 = p1 & true }).
    {
        !p1[Person]: Hated_persons(p1) = _Hated_persons(p1).
    }
    //Noone hates all
    !Person[Person]: true => Hated_persons(Person) < 03.

}
```

(continues on next page)

(continued from previous page)

```
procedure main(){
stdoptions.nbmodels = 1
printmodels(modelexpand(T,S))
}
```

5.8 The cDMN developer reference

This page details the reference for the cDMN solver.

In a nutshell, the solver consists of 5 elements:

- An Excel to numpy converter (*The cdmn_solver API reference*).
- *The glossary API reference*, which contains all types and predicates.
- *The interpret API reference*.
- *The table API reference*, which represents a DMN+ table.
- *The idply API reference*, which translates DMN notation inside cells to IDP notation.
- *The parse_xml API reference*, a class to convert DMN XML into cDMN tables.
- *The post_process API reference*, functions used for post processing generated configurations.

5.8.1 The cdmn_solver API reference

Copyright 2020 Simon Vandevelde, Bram Aerts, Joost Vennekens This code is licensed under GNU GPLv3 license (see LICENSE) for more information. This file is part of the cDMN solver.

`solver.`**`main`**`()`
The main function for the cDMN solver.

5.8.2 The DMN API reference

5.8.3 The glossary API reference

Copyright 2020 Simon Vandevelde, Bram Aerts, Joost Vennekens This code is licensed under GNU GPLv3 license (see LICENSE) for more information. This file is part of the cDMN solver.

`class` `glossary.`**`Glossary`**(*glossary_dict: dict*)
The Glossary object contains all types, functions and predicates. During initialisation, it reads and interprets all the types, functions, constants, relations and booleans it can find and it reports any errors. Once the Glossary is created and initialized without errors, it's possible to print out the predicates in their IDP version.

`add_aux_var`(*aux*)
Some variables need to use auxiliary variables, for instance those found in the outputcolumns of C# tables. This method allows the creation of those variables. No aux var are created when makin an IDP file for the autoconfig interfaces.

:arg List<str> a list containing strings of the variables.

`contains`(*typestr*)
Checks whether or not a type was already added to the glossary.

Returns bool True if the type has been added already.

find_type(*t*)
Looks for types in the glossary.

Returns List<Type> the types found.

lookup(*string: str*)
TODO REWORK ENTIRE METHOD.

read_datatables(*datatables*, *parser*)
Reads and interprets the datatables. Also checks if the values in the datatables appear in the possible values column of the glossary.

Firstly it checks which columns are input, and which are output. A column is an inputcolumn if it contains the table title in the first cell, and an output if it contains *None* in the first cell.

Iterates over every outputcolumn, deciphers which predicate the outputcolumn represents, and sets its "struct_args" to a combination of the input arguments and the output column's arguments. The predicate uses this struct_args to format its struct string.

Parameters

- **List<np.array>** – datatables, containing all the datatables.
- **parser** –

Returns None

to_idp_voc(*target_lang: str = 'idp'*)
Function which turns every object in a glossary into their vocabulary definitions.

Parameters **target_lang** – the format for the output. Either *idp* or *idpz3*.

Returns voc string

to_json_dicts()
Creates a dict entry for every predicate and function, which is later turned into json.

Returns dict

class glossary.**Predicate**(*name: str, args: List[glossary.Type], super_type: glossary.Type, partial=False, full_name=None, zero_arity=False*)
Class which represents both predicates and functions. This double meaning is a relic of the past, and is to be fixed. In the future, a separate Function class should be created.

static from_string(*full_name: str*, *predicate: bool*, *super_type: glossary.Type*, *glossary: glossary.Glossary*, *partial=False*, *zero_arity=False*)
Static method to create a predicate from string.

Parameters

- **str** – full_name, the full name.
- **bool** – predicate, true if predicate, false if function.
- **Type** – super_type, the super type of the predicate.
- **Glossary** – glossary, the glossary.
- **bool** – partial, whether or not it's a partial function.
- **zero_arity** – bool which should be True when the predicate is a 0-arity predicate (constants and booleans).

Returns Predicate

interpret_name()
Method to interpret the name. This method forms a generic name representation, by replacing the arguments by dummies. In this way, it creates a skeleton structure for the name.

Thus, it returns the name, without the arguments. For instance, *Country borders Country* becomes *(?P<arg0>.+) borders (?P<arg1>.+)*. This way, arg0 and arg1 can be found easily later on.

is_function()
Method to check whether the predicate is a function. Since only functions have super types, we use that as a check. Note that constants are a special case of functions.

Returns boolean

is_relation()
Method to check whether the predicate is a relation. A predicate is either a relation or a function, so we use that as a check. Note that booleans are a special case of relations.

Returns boolean

lookup(*string: str*)
Method to compare a string to this predicate, to see if the predicate appears in the string in any form. TODO: make this more clear.

to_idp_struct(*target_lang: str = 'idp'*)
If a function or predicate receives a value in a datatable, we need to set it's values in the structure. When parsing the datatable in "read_datatables", we set the "struct_args" of the predicates/functions that get a value. struct_args could look like: {key1|key2:value}. However, it's possible to input multiple keys per cell to save space.

For instance: "Jim|Skydiving, Soccer" needs to be formatted as "Jim, Skydiving; Jim, Soccer". The same goes for functions.

To achieve this, we split the keys on their seperator, and then we split each key on a comma. This way, we have an array of keys in which each item is an array of subkeys. We need to form every possible combination of these keys, and to do this we use itertools.product.

Parameters **target_lang** – the target language format. Either *idp* or *idpz3*.

Returns str

to_idp_voc(*target_lang: str*)
Convert the predicate/function to a string for the IDP vocabulary.

Parameters **target_lang** – the format for the output. Either *idp* or *idpz3*.

Returns str the predicate/function in vocabulary format.

class glossary.**Type**(*name: str*, *super_type*, *posvals='-'*)
TODO

basetype
The basetype represents one of the ancestor types, such as int or str.

Returns type the basetype.

check_values(*values*, *tablename*)
Method to check if the values listed in a datatable match with the values listed in the possible values column(if a datatable was used).

If the possible values column is left empty, then it assumes all the values are correct and it fills the possible values automatically. This is needed so that the type can input these values into constructed from.

If the possible values column contains values, then every value used in a datatable needs to match a value in the possible values.

Returns boolean True if all the values match.

Throws ValueError if a value appears in the datatable but not in posvals.

to_idp_struct(*target_lang: str = 'idp'*)
Converts all the information of the Type into a string for the IDP structure. Normal types don't need a structure, as their possible values are listed as "constructed from" in the voc. This is here for future's sake.

Returns str the string for the structure.

to_idp_voc(*target_lang: str = 'idp'*)
Converts all the information of the Type into a string for the IDP vocabulary.

Parameters **target_lang** – the format for the output. Either *idp* or *idpz3*.

Returns str the vocabulary form of the type.

to_theory()
TODO

5.8.4 The interpret API reference

Copyright 2020 Simon Vandevelde, Bram Aerts, Joost Vennekens This code is licensed under GNU GPLv3 license (see LICENSE) for more information. This file is part of the cDMN solver.

class interpret.**PredicateInterpretation**(*pred: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1d2b750>, arguments, inter: interpret.VariableInterpreter, variables*)
TODO

type
Method to get the type of the predicate.

Returns Type supertrype of the predicate.

value
TODO

class interpret.**Value**(*value: str, valuetype, variables*)
An object to represent a value.

check(*variables*)
TODO

class interpret.**VariableInterpreter**(*glossary: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1fb4450>*)
TODO

interpret_value(*value: str, variables: Dict[KT, VT] = {}, expected_type=None*)
Method to interpret a value of a cDMN notation. Here be dragons.

Parameters

- **value** – string which needs to be interpreted.
- **variables** – list containing the variables.
- **expected_type** –

Returns str the interpretation of the notation.

5.8.5 The table API reference

class table.**Table**(*array: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e3156390>*, *parser: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e3156990>*, *aux_var_needed=True*)

The table object represents decision and constraint tables.

Attr name str

Attr hit_policy str

Attr inputs List[np.array]

Attr outputs List[np.array]

Attr rules List[np.array]

export(*target_lang: str = 'idp'*)

Export tries to find the hit policy for a table, and then returns the method needed to transfer the table to idp form. These hit policies are currently:

- A, U, F -> translate to definitions;
- E* -> translate to implications;
- C+, C<, C>, C# -> translate to aggregates.

Every hit policy has it's own method.

Parameters

- **target_lang** – the target output format. Either *idp* or *idpz3*.
- **check_error** – a type of error to check a table for.
- **target_error** – the name of the output for a table to check.

Returns method the output of export method for the table.

find_auxiliary() → List[str]

Every output in a C# table needs to use an auxiliary variable to work correctly. This method makes a list of those output variables, so that the auxiliary versions can be created.

Returns List[str]

fstring

Method to decide the string format for a certain hit policy.

Returns str the format string.

variables_iq_oq(*repres=None*)

Method to generate the needed variables and quantifiers :returns variables, iquantors, oquantors: TODO

table.**variname**(*string: str*) → str

Function to return the variable of a header in the form of "Type called var"

Parameters **string** – the headerstring of which the variable name needs to be found.

Returns str the variable name.

5.8.6 The table_operations API reference

Copyright 2019 Simon Vandevelde, Bram Aerts, Joost Vennekens This code is licensed under GNU GPLv3 license (see LICENSE) for more information. This file is part of the cDMN solver.

table_operations.**create_dependency_graph**(*tables: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cc5450>], parser: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cad750>*) → Dict[KT, VT]

Function to create dependency graph of variables. Basically, we check for every output of a decision table what the inputs are. The outputs depend on these inputs. By doing this for every table, we can build a graph of what knowledge depends on what other knowledge.

Parameters

- **tables** – the list of arrays, each containing a table.
- **parser** – the parser.

Returns str the theory for the IDP file.

table_operations.**create_display**(*goal_var: List[str]*) → str

Format the display block for a DMN specification.

table_operations.**create_json**(*glossary*) → dict

Function to form the json file.

Parameters **glossary** – the glossary

Returns None

table_operations.**create_main**(*inf: Dict[str, object], parser: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cad750>, target_lang: str = 'idp'*) → str

Function which creates the main for the IDP file.

Parameters

- **inf** – a dictionary containing the inference method and it's applicable variables (e.g. what the term looks like, or how many models should be generated)
- **parser** – the parser, needed to parse the optimization term.
- **target_lang** – the output format to create. Either idp or idpz3.

Returns str the main for the IDP file.

table_operations.**create_struct**(*tables: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cc5c90>], parser: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cad750>, glossary: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cad7d0>, target_lang: str = 'idp'*) → str

The structure consists of data supplied by data tables.

The data inside data tables is interpreted and set as "struct_args". "struct_args" is then used in the to_idp_struct() method to form the structure.

Parameters

- **tables** – the list of arrays, each containing a table.
- **parser** – the parser to read the headers of data tables.
- **glossary** – the glossary

- **target_lang** – the output format to create. Either idp or idpz3.

Returns str the structure for the IDP file.

table_operations.**create_theory**(*tables: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cc5710>], parser: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cad750>, aux_var_needed: bool, target_lang: str = 'idp'*) → str

Function to create the theory in the IDP format.

Parameters

- **tables** – the list of arrays, each containing a table.
- **parser** – the parser.
- **aux_var_needed** – a booloan signifying whether auxiliary variables are needed. This is the case for C# aggregates, when no interface is used.
- **target_lang** – the output format for the file. Either *idp* or *idpz3*

Returns str the theory for the IDP file.

table_operations.**create_voc**(*glossary*, *target_lang: str = 'idp'*)

Function which creates the vocabulary for the IDP file.

Parameters

- **glossary** –
- **target_lang** – the target output format. Either *idp* or *idpz3*.

Returns str the vocabulary for the IDP file.

table_operations.**explore**(*sheet: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cad1d0>], boundaries: List[int]*)

Tries to find the ranges for each table.

Parameters

- **sheet** – the sheet containing all the tables.
- **bounaries** – a list containing the theoretic boundaries.

Returns None

table_operations.**fill_in_merged**(*file_name: str*, *sheet_names: List[str] = None*) → List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cada90>]

Loads up a specific sheet and returns it. A sheet is comprised of a list of tables.

Parameters

- **file_name** – path to the xlsx file.
- **sheet** – name of the sheet to load in.

Returns List<np.array> a list of all the tables in a sheet.

table_operations.**find_auxiliary_variables**(*tables: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cc5910>], parser: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cad750>*) → List[str]

Due to a bug in the IDP systems, the variables in a C# table each need to use an auxiliary variable to function properly. This has the same name as the variable, but preceded with an underscore.

This function gets the auxiliary variables for every table.

table_operations.**find_datatables**(*tables: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e3424910>]*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e34249d0>

Locates the Datatable, which contains data that needs to be expressed in the structure or our idp file.

Parameters **tables** – the list of arrays, each containing a table.

Returns np.array containing the datatable

table_operations.**find_execute_method**(*tables: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e34247d0>]*) → Dict[str, object]

Locates the Goal table, which contains the inference method. There are four possible inference methods:

- "get x models", where x is the number of required models;
- "get all models", in order to get all models;
- "minimize x", where x is a term to minimize;
- "maximize x", where x is a term to maximize.

Parameters **tables** – the list of arrays, each containing a table.

Returns Dict[str, str] contains the inference method and its applicable variables.

table_operations.**find_first**(*sheet: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cad790>*) → tuple

TODO

Parameters **sheet** – the sheet

table_operations.**find_glossary**(*tables: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e313cb50>]*) → Dict[str, <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e313c5d0>]

Locates the glossarytables, and places them in a dictionary for each type. The five tables it looks for are:

- Type
- Function
- Constant
- Relation
- Boolean

Returns Dict[str, np.array] containing the glossary for type, function,

constant, relation and boolean.

table_operations.**find_glossary_table**(*tables: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e313cd90>], name: str, critical: bool = False*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e3424cd0>

Looks for a specific glossary table with name "name".

If the critical boolean is set, an error is returned when no glossary is found. For example, there should always be a type glossary. Non-critical glossaries only print warnings when none are found.

Parameters

- **tables** – the list of arrays, each containing a table.
- **name** – the name of the table to find.
- **critical** – True if a table needs to be found. E.g. if the table is not found, an error is thrown.

Returns np.array the glossary table, if found.

table_operations.**find_tables**(*tables: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e305a050>]*) → List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e305ac90>]

Looks for decision tables and constraint tables.

Parameters tables – the list of arrays, each containing a table.

Returns List[np.array] a list containing only the decision and constraint tables.

table_operations.**find_tables_by_name**(*tables*, *name*)

Looks for tables based on a regex expression representing their name.

Parameters

- **np.array** – the tables.
- **str** – the name, in regex.

Returns List<np.array> the tables found.

table_operations.**get_dependencies**(*variable*, *dependency_graph*, *level=0*, *downstream=False*) → Dict[KT, VT]

If downstream is False, then the function returns all the upstream dependencies, i.e., the variables that need to have values before the variable's value can be set.

If downstream is set to True, it does the opposite: it returns all the upstream dependencies, i.e. the variables that depend on this variable's value.

table_operations.**identify_tables**(*sheets: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1cadfd0>]*) → List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e3128f10>]

Function which looks for all the tables in a given sheet. Creates a list of boundaries for the tables.

Parameters sheets – a list containing a numpy array representing the sheet.

Returns List[np.array] containing boundaries of all the tables.

table_operations.**replace_with_error_check**(*tables: List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e305a210>], check_error: str, target_error, rule_error=None, deps=[]*) → List[<sphinx.ext.autodoc.importer._MockObject object at 0x7f38e305a490>]

Replaces a model by one for error checking. There are three types of errors which can be checked for errors:

- Overlap in the inputs (multiple rules fire for same input).
- Gaps in the inputs (input for which no rules applies).
- Shadowed rules (rules which can never fire).

Parameters

- **tables** – the list of arrays, each containing a table.
- **target_error** – the table which we want to find.

- **check_error** – the type of error check to do.
- **rule_error** – the rule to check for errors. Only for shadowed.

Returns List[np.array] a list containing only the decision and constraint

5.8.7 The idply API reference

Copyright 2020 Simon Vandevelde, Bram Aerts, Joost Vennekens This code is licensed under GNU GPLv3 license (see LICENSE) for more information. This file is part of the cDMN solver.

class idply.**Parser**(*i: <sphinx.ext.autodoc.importer._MockObject object at 0x7f38e1fb7690>*, *target_lang: str*)

The parser is used to parse cDMN strings and find out their underlying secrets. This is a complex part of the solver, and works on magic and hope.

p_compare(*t*)
statement : expression EQUALS expression | expression COMPARE expression

p_domain_size(*t*)
expression : HASHTAG expression

p_dontcare(*t*)
statement : expression EQUALS MINUS | expression EQUALS

p_dot(*t*)
expression : NUMBER DOT NUMBER

p_eqlist(*t*)
statement : expression EQUALS list

p_eqnot(*t*)
statement : expression EQUALS NOT LPAREN list RPAREN

p_eqnotlist(*t*)
statement : expression EQUALS NOT LPAREN expression RPAREN

p_expression_abs(*t*)
expression : ABS expression

p_expression_binop(*t*)

expression [expression PLUS expression]

expression MINUS expression
expression TIMES expression
expression DIVIDE DIVIDE expression
expression DIVIDE expression
expression PERCENT expression

p_expression_group(*t*)
expression : LPAREN expression RPAREN

p_expression_name(*t*)
expression : ID

p_expression_number(*t*)
expression : NUMBER

p_expression_uminus(*t*)
expression : MINUS expression %prec UMINUS

p_lbound (*t*)
lbound : LPAREN | LBRACK | RBRACK

p_list (*t*)
list : expression COMMA expression | list COMMA expression

p_no (*t*)
statement : expression EQUALS NO

p_nrange (*t*)
statement : expression EQUALS NOT LPAREN lbound expression COMMA expression rbound RPAREN

p_prange (*t*)
statement : expression EQUALS LPAREN expression UNTIL expression RPAREN

p_quotes (*t*)

expression [LQUOTE expression RQUOTE]

STRAIGHTQUOTE expression STRAIGHTQUOTE

p_range (*t*)

statement [expression EQUALS lbound expression UNTIL expression rbound]

expression EQUALS lbound expression COMMA expression rbound

p_rbound (*t*)
rbound : RPAREN | RBRACK | LBRACK

p_returnval (*t*)
return : statement

p_rrange (*t*)
statement : expression EQUALS LPAREN expression UNTIL expression RBRACK

p_yes (*t*)
statement : expression EQUALS YES

parse_val (*column_header: str*, *value: str*, *variables*) → str
Parse a cDMN notation inside a cell.

Parameters

- **column_header** – the column header of the cell
- **value** – the value inside the cell
- **variables** – no idea

Returns str the parsed notation.

t_ID (*t*)
[a-zA-Z_][a-zA-Z_0-9s]*

t_NUMBER (*t*)
d+

t_newline (*t*)
n+

5.8.8 The parse_xml API reference

Copyright 2020 Simon Vandevelde, Bram Aerts, Joost Vennekens This code is licensed under GNU GPLv3 license (see LICENSE) for more information. This file is part of the cDMN solver.

class parse_xml.**XMLparser**(*xml_str: str*)

The parser class responsible for parsing the xml file, and turning it into cDMN tables. This means a conversion from XML into the Type and Constant glossary, and into standard Decision Tables.

get_goal_variables()

Find the variables which are defined by the last decision table. This decision table always contains the "final decision" of the specification, and thus, its outputs contain the goals.

Returns goal_variables list containing the names of the goal var.

get_table_dependencies(*table_name*, *dependencies=[]*)

Returns a list of all the tables which depend on the table with *table_name*. This includes indirect dependencies as well.

Returns dependencies list of all dependencies.

get_tables()

Method to return all the tables as one big array. This is the internal cDMN representation.

Returns tables a list containing all decision and glossary tables.

5.8.9 The post_process API reference

 This file is part of the cDMN solver.

post_process.**merge_definitions**(*idp: str*) → str

In cDMN, it is possible to create different tables that each define the same concept. In cases where relations are defined, this can be problematic. E.g., in the *Vacation_Days_Advanced_old* example, we define the *Employee eligible for Rule* relation in 5 different tables, each for a different value of *Rule*. In IDP, a definition is expected to be complete: all possible values for the defined concept should be defined. All other values are implicitly impossible. Because the cDMN implementation creates 5 different definitions, each defining a different value, this results in an unsat in IDP.

The solution to this problem is to merge all tables defining the same concept.

Python Module Index

g

glossary, 62

i

idply, 71
interpret, 65

p

parse_xml, 72
post_process, 73

s

solver, 62

t

table, 66
table_operations, 67

Index

A

add_aux_var() (*glossary.Glossary method*), 62

B

basetype (*glossary.Type attribute*), 64

C

check() (*interpret.Value method*), 65
check_values() (*glossary.Type method*), 64
contains() (*glossary.Glossary method*), 62
create_dependency_graph() (*in module table_operations*), 67
create_display() (*in module table_operations*), 67
create_json() (*in module table_operations*), 67
create_main() (*in module table_operations*), 67
create_struct() (*in module table_operations*), 67
create_theory() (*in module table_operations*), 68
create_voc() (*in module table_operations*), 68

E

explore() (*in module table_operations*), 68
export() (*table.Table method*), 66

F

fill_in_merged() (*in module table_operations*), 68
find_auxiliary() (*table.Table method*), 66
find_auxiliary_variables() (*in module table_operations*), 68
find_datatables() (*in module table_operations*), 69
find_execute_method() (*in module table_operations*), 69
find_first() (*in module table_operations*), 69
find_glossary() (*in module table_operations*), 69
find_glossary_table() (*in module table_operations*), 69
find_tables() (*in module table_operations*), 70
find_tables_by_name() (*in module table_operations*), 70
find_type() (*glossary.Glossary method*), 62
from_string() (*glossary.Predicate static method*), 63
fstring (*table.Table attribute*), 66

G

get_dependencies() (*in module table_operations*), 70
get_goal_variables() (*parse_xml.XMLparser method*), 73
get_table_dependencies() (*parse_xml.XMLparser method*), 73
get_tables() (*parse_xml.XMLparser method*), 73
Glossary (*class in glossary*), 62
glossary (*module*), 62

I

identify_tables() (*in module table_operations*), 70
idply (*module*), 71
interpret (*module*), 65
interpret_name() (*glossary.Predicate method*), 63
interpret_value() (*interpret.VariableInterpreter method*), 65
is_function() (*glossary.Predicate method*), 64
is_relation() (*glossary.Predicate method*), 64

L

lookup() (*glossary.Glossary method*), 63
lookup() (*glossary.Predicate method*), 64

M

main() (*in module solver*), 62
merge_definitions() (*in module post_process*), 73

P

p_compare() (*idply.Parser method*), 71
p_domain_size() (*idply.Parser method*), 71
p_dontcare() (*idply.Parser method*), 71

p_dot() (*idply.Parser method*), 71
p_eqlist() (*idply.Parser method*), 71
p_eqnot() (*idply.Parser method*), 71
p_eqnotlist() (*idply.Parser method*), 71
p_expression_abs() (*idply.Parser method*), 71
p_expression_binop() (*idply.Parser method*), 71
p_expression_group() (*idply.Parser method*), 71
p_expression_name() (*idply.Parser method*), 71
p_expression_number() (*idply.Parser method*), 71
p_expression_uminus() (*idply.Parser method*), 71
p_lbound() (*idply.Parser method*), 71
p_list() (*idply.Parser method*), 72
p_no() (*idply.Parser method*), 72
p_nrange() (*idply.Parser method*), 72
p_prange() (*idply.Parser method*), 72
p_quotes() (*idply.Parser method*), 72
p_range() (*idply.Parser method*), 72
p_rbound() (*idply.Parser method*), 72
p_returnval() (*idply.Parser method*), 72
p_rrange() (*idply.Parser method*), 72
p_yes() (*idply.Parser method*), 72
parse_val() (*idply.Parser method*), 72
parse_xml (*module*), 72
Parser (*class in idply*), 71
post_process (*module*), 73
Predicate (*class in glossary*), 63
PredicateInterpretation (*class in interpret*), 65

R

read_datatables() (*glossary.Glossary method*), 63
replace_with_error_check() (*in module table_operations*), 70

S

solver (*module*), 62

T

t_ID() (*idply.Parser method*), 72
t_newline() (*idply.Parser method*), 72
t_NUMBER() (*idply.Parser method*), 72
Table (*class in table*), 66
table (*module*), 66
table_operations (*module*), 67
to_idp_struct() (*glossary.Predicate method*), 64
to_idp_struct() (*glossary.Type method*), 65
to_idp_voc() (*glossary.Glossary method*), 63
to_idp_voc() (*glossary.Predicate method*), 64
to_idp_voc() (*glossary.Type method*), 65
to_json_dicts() (*glossary.Glossary method*), 63
to_theory() (*glossary.Type method*), 65
Type (*class in glossary*), 64
type (*interpret.PredicateInterpretation attribute*), 65

V

Value (*class in interpret*), 65
value (*interpret.PredicateInterpretation attribute*), 65
VariableInterpreter (*class in interpret*), 65
variables_iq_oq() (*table.Table method*), 66
variname() (*in module table*), 66

X

XMLparser (*class in parse_xml*), 72