UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Ülari Laurson

Verification and Simplification of DMN Decision Tables

Master's Thesis (30 ECTS)

Supervisor: Marlon Dumas, PhD
Supervisor: Irene Teinemaa, MSc

Tartu 2016

Verification and Simplification of DMN Decision Tables

Abstract: The Decision Model and Notation (DMN) is a standard notation to specify decision logic in business applications. A central construct in DMN is a decision table. The rising use of DMN decision tables to capture and to automate everyday business decisions raises the need to support analysis tasks on decision tables. This thesis provides scalable algorithms to tackle three analysis tasks: detection of overlapping rules, detection of missing rules and simplification of decision tables via rule merging. All proposed algorithms have been implemented in an open-source DMN editor and are tested on large decision tables derived from a credit lending data-set.

Keywords: Decision Model and Notation, Decision Table, Sweep Algorithm, Overlapping Rules, Missing Rules, Rule Merginng

CERCS: P175, Informatics, systems theory

DMNi otsustabelite verifitseerimine ja lihtsustamine

Lühikokkuvõte: Decision Model and Notation (DMN) on standardne notatsioon, mida kasutatakse ärirakendustes otsuste loogika kirjeldamiseks. Otsustabelid on DMNi üks peamisi osi. DMNi otsustabelite suurenev kasutatavus igapäevaste äriotsuste ülesmärkimiseks ja automatiseerimiseks on tõstatanud vajadust analüüsida otsustabeleid. See lõputöö annab ülevaate DMN otsustabelist ja kirjeldab kolme skaleeruvat algoritmi, mis on mõeldud leidmaks kattuvaid reegleid ja puuduvaid reegleid ning lihtsustada otsustabeleid kasutades reeglite ühendamist. Kõik välja pakutud algoritmid on implementeeritud avatud lähtekoodiga DMN redaktorisse ja katsetatud suurte otsustabelite peal, mis pärinevad krediidiandmise andmebaasist.

Võtmesõnad: Decision Model and Notation, Otsustabelid, Sweep Algoritm, Kattuvad Reeglid, Puuduvad Reeglid, Reeglite Ühendamine

CERCS: P175, Informaatika, süsteemiteooria

Contents

1 Introduction **4**
1.1 Decision tables 4
1.2 Goal and problem statement 6

2 Background **9**
2.1 Decision table 9
2.2 Line sweeping 12
2.3 Rule merging 14
2.3.1 Polygon rectangulation 14
2.3.2 Classical approach 15
2.4 DMN tools 17
2.4.1 Signavio 17
2.4.2 Prologa 18

3 Contribution **19**
3.1 Finding overlapping rules 20
3.2 Finding missing rules 23
3.3 Decision table simplification 26

4 Tool implementation **33**
4.1 Syntactic check 35
4.2 Overlapping rules 36
4.3 Missing rules 37
4.4 Table simplification 38

5 Evaluation **41**
5.1 Verification 41
5.2 Simplification 44

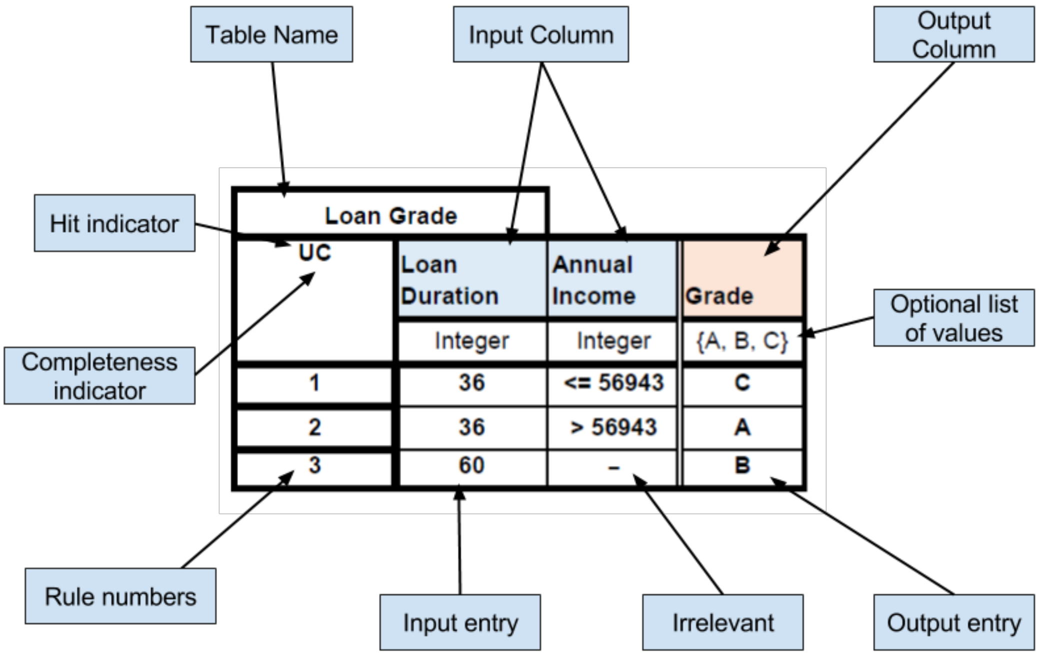
6 Conclusion **46**

1 Introduction

In 2014, Object Management Group (OMG) published Decision Model and Notation (DMN) standard [11]. Decision Model and Notation primary goal is to provide a common notation that is readily understandable for all business users including business managers, analysts, and developers. DMN consists two levels: a "decision requirements level" and a "decision logic level". The decision requirements level defines the input data we need in order to make a decision. The decision logic level describes how each decision is made. Decisions are usually expressed as decision tables. In decision tables, the columns representing the inputs and outputs of a decision, and rows represent rules. Columns typically are typed so that they have an associated domain. A rule is a conjunction of basic expressions (one basic expression per cell). Basic expressions are captured using a language known as S-Feel (Simplified Friendly Enough Expression Language).

1.1 Decision tables

Before we describe this thesis goal and analyze the problem, let us give an overview of decision tables and their elements.



Loan Grade

UC	Loan Duration	Annual Income	Grade
	Integer	Integer	{A, B, C}
1	36	<= 56943	C
2	36	> 56943	A
3	60	-	B

Figure 1: Sample decision table with its elements

In Figure 1 we see one example of the decision table, where rules are as rows. There are two more possible ways how to construct these tables, rules as columns (see Table 2) and rules as cross-table (see Table 4). For simplicity we are using only rules as rows and all other examples are following the same structure. Let us describe each element a little bit closer:

1. **Table name**: Name of the table. In this example table name is "Loan Grade."

2. **Input and output column**: Concrete conditions of a rule. In this example, we see two input column "Loan Duration" and "Annual Income", and one output column "Grade."

3. **Input and output entry**: One input or output value in input or output column. Output column has three output entries "C", "A" and "B".

4. **Irrelevant**: If input entry has the value "-", then it means that every input satisfies the input entry, and this clause is irrelevant for the specified rule. In Figure 1 we see that input "Annual Income" has a "-" that means every value satisfies this input.

5. **Rule numbers**: Each rule has its rule number. Rule numbers are counted from top to bottom. This example has three rules.

6. **Hit indicator**: Indicates, if only one rule can satisfy or multiple rules are allowed too. In our examples, we use "U" (Unique) hit indicator. It means that only one rule can match. There are other hit indicators in DMN standard, but we are using only "Unique", because if decision table has "Unique" hit policy, then the table cannot contain any overlapping rules.

7. **Completeness indicator**: Optional attribute. We are using "C" (Complete) in our work because then all possible input combinations must be covered in the decision table.

8. **List of values**: Optional attribute. Can specify what values a column can have. In our work, we do not use this optional attribute.

The rules in the decision table in Figure 1 should be interpreted as follows:

1. If the customer asks for a loan duration of 36 and his annual income is less than 56943, then he gets a grade "C."

2. If the customer asks for a loan duration of 36 and his annual income is more than 56943, then he gets a grade "A."

3. If the customer asks for a loan duration of 60, then she gets a grade "B."

1.2 Goal and problem statement

Using DMN decision tables for making critical business decisions raises the question of ensuring the correctness and simplicity of these tables. Detecting errors in DMN tables may prevent costly defects. Correct tables have to be consistent and complete.

The main contribution of this thesis is scalable algorithms for two basic correctness checking tasks over DMN tables and one algorithm for simplification task. Hoover and Chen describe their work how to verify decision tables, but they only deal with the categorical values [9]. These two correctness checking tasks are the detection of overlapping rules and detection of missing rules. Detection of missing rules allows to check for completeness of the table, while detection of overlapping rules allows one to check for two possible issues: (i) inconsistent rules, meaning two rules that overlap and that are associated with two different outputs; and (ii) redundant rules, which occurs when two rules overlap and they have the same output.

Let us now explain, with examples, the possible cases of overlapping and missing rules. In the first case, all inputs and outputs of two rules overlap, then these rules are *redundant*. In Table 1 we see rule (Age = Adult, Marital Status = Married, Parental Status Kids, Discount = 30%) is covered in rules 2 and 3, and they are *redundant*. The second kind of overlap occurs when all inputs are the same and outputs are different, these rules are *inconsistent*. Rule 1 and 3 are *inconsistent* in rule (Age = Adult, Marital Status = Single, Parental Status = Kids). In Table 1 we see that rule (Age = Child, Marital Status = Single, Parental Status = No Kids) is not covered. It means that table has at least one missing rule.

UC	Inputs			Outputs
	Age	**Marital Status**	**Parental Status**	**Discount**
1	Adult	Single	-	10%
2	-	Married	-	20%
3	-	-	Kids	30%

Table 1: Decision table with missing and overlapping rules

In addition to correctness checking, another useful analysis task on DMN decision tables is to detect and merge adjacent rules with the same output. By doing so, we can obtain a simpler decision table (fewer rules), which potentially can improve readability. Specifically, two rules can be merged if they have the same output, and they only differ in one input column or if an input has a numeric type, then two rules have to be contiguous in one input column, and the rest input columns have to be the same. In Table 2, we can merge rules 1 and 2, we get new rule (Age = [20..50], Marital Status = Married, Discount = 20%). We can also merge rules 4 and 5 into one rule and get a new merged rule (Age = [60..90], Marital Status = "-" Discount = 30%). The merged table (simplified table) can be seen in Table 3.

UC	Inputs		Outputs
	Age	**Marital Status**	**Discount**
1	[20..30]	Married	20%
2	(30..50]	Married	20%
3	(30..50]	Single	30%
4	(60..90]	Married	30%
5	(60..90]	Single	30%

Table 2: Decision table with overlapping rules

UC	Inputs		Outputs
	Age	**Marital Status**	**Discount**
1	[20..50]	Married	20%
2	(30..50]	Single	30%
3	(60..90]	-	30%

Table 3: Decision table after rule merging

The proposed scalable algorithms are based on a novel geometric interpretation of DMN tables and inspired by sweep-based spatial join algorithms [1]. Each rule in a decision table is mapped to an iso-oriented hyper-rectangle in an N-dimensional space (N represents the number of columns). Accordingly, the problem of detecting overlapping rules is mapped to that of detecting overlapping hyper-rectangles. Meanwhile, the problem of detecting missing rules is mapped to that of differencing the N-dimensional space defined by the N columns of a DMN table, and the collection of hyper-rectangles induced by its rules. The problem of simplification is mapped to that of splitting hyper-rectangles into smaller hyper-rectangles that do not overlap. In the end, if possible these hyper-rectangles are merged back into bigger hyper-rectangles so that no hyper-rectangles overlap each other.

Chapter 2 describes in more detail background of decision tables, discusses related work and gives an overview of different DMN tools. Furthermore, it describes other approaches and methods that can be used to partially solve verification and simplification problems. Chapter 3 presents the proposed algorithms for verification and simplification, and explains them in detail. Chapter 4 introduces a DMN toolkit dmn-js and explains how our proposed algorithms work there. Chapter 5 represents the empirical evaluation of these algorithms. Chapter 6 summarizes the contributions and outlines future work.

2 Background

In this chapter, we describe DMN in more detail and give an overview of DMN background. We present the algorithm and method that can be used to solve decision table verification and simplification problems. Also, in this chapter we describe two DMN tool and explain, how they verify and simplify decision tables.

2.1 Decision table

Before decision tables, flowcharts were used to express decisions. The flowchart has flaws such as hard to draw, difficult to comprehend, hard to control if the chart is complete or not, to all the same series of actions give the same result. Decision table was first introduced in 1963 [12]. Decision tables are very understandable and can be written in almost any language. Another way to represent decisions is to use decision trees or regression trees. A simple decision table and the corresponding decision tree can be seen in Figure 3.

Formerly, decision tables were often presented using the vertical "rules as columns" layout [12, 14] (see Figure 2). In Figure 2 we see the main parts of decision tables. Black arrows indicate how to read rules. The horizontal double line separates input and output entries, and the vertical double line separates expressions from entries. Nowadays, it is more common to display rules as rows (see Table 1), Camunda and Signavio are displaying this way rules. Rules as rows is more feasible if the number of rules is larger than the number of input and output attributes. Another possible way to represent decision table is as cross-table (see Table 4). In cross-table, there are no rule numbers and hit policies.

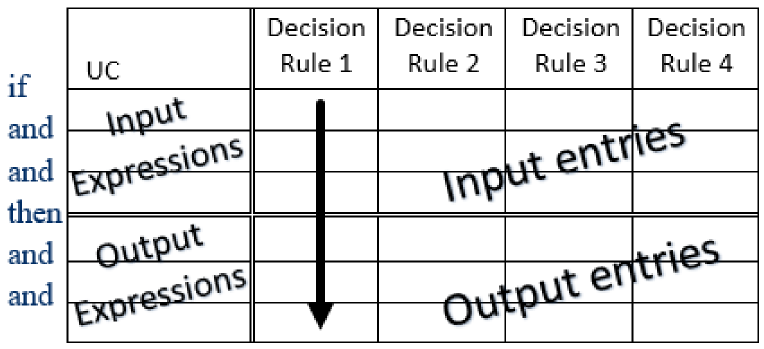


Figure 2: Decision table with rules as columns

Traditionally, decision tables are required to be unique, i.e. only one rule

Discount			
		Marital Status	
Discount		Married	Single
Age	[20..30]	20%	0%
	(30..50]	0%	30%
	(50..90]	30%	30%

Table 4: Rules as cross-table

matches a given set of inputs, and complete, i.e. all possible input combinations are covered. The DMN standard alleviates this requirement by introducing the concept of *hit policy*, which determines the outcome of a decision table in case of multiple matches. DMN hit policies consist of two parts: a hit indicator and a completeness indicator. There are two groups of hit indicators single and multiple hit policy [11]. Hit indicators are represented by a single letter in DMN decision tables (marked in bold below).

Single hit policies are the following:

1. **U**nique: no overlap between rules and only one rule can match. Default indicator.

2. **A**ny: overlap between rules is possible, but overlapping rules have to have the same output. Otherwise, the result is undefined.

3. **P**riority: rules may overlap, with different outputs. Returned is the rule that has the highest output priority. An ordered list of output values represents output priorities.

4. **F**irst: rules may overlap, with different outputs. The first rule that matches is returned.

Multiple hit policies are the following:

1. **O**utput order: all hits are returned in decreasing order of output priorities. An ordered list of output values represents output priorities.

2. **R**ule order: all hits are returned in rule order.

3. **C**ollect: returns all hits in the arbitrary order. Operators, such as sum, min, max and count, can be added to the outputs.

Completeness indicator is an optional attribute. Complete means that all possible input combinations are covered in the decision table. Incomplete indicates that tables may have some missing input combinations. In our work, all our decision tables are Unique and Complete so that the problem of finding overlapping and missing rules become relevant.

UC	Inputs			Outputs
	Income	**Credit History**	**Dept**	**Rating**
1	[0..15000]	-	-	High
2	[15000..35000]	Unknown	High	High
3	[15000..35000]	Unknown	Low	Moderate
4	[15000..35000]	Bad	-	High
5	[15000..35000]	Good	-	Moderate
6	$>$ 35000	Unknown	-	Low
7	$>$ 35000	Bad	-	Moderate
8	$>$ 35000	Good	-	Low

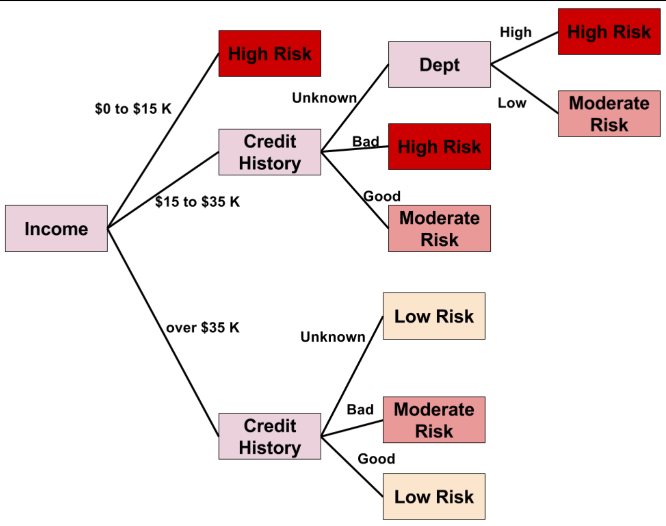


Figure 3: Decision table and the corresponding decision tree

So far, we have mentioned that DMN uses decision tables, but the full scope of DMN is much wider. For example, DMN covers the construct of decision requirements diagrams (DRD), see Figure 4. In DRD, we can represent complex decisions and show the dependencies of each decision. DRD consists of three main parts:

1. **decisions**: are represented as decision tables.
2. **input data**: the data that you give to your decision table in order to get output.
3. **relation between decisions**: arrows which connect different DRD parts.

DMN covers some additional DRD symbols and parts, but these three are the most important and commonly used.

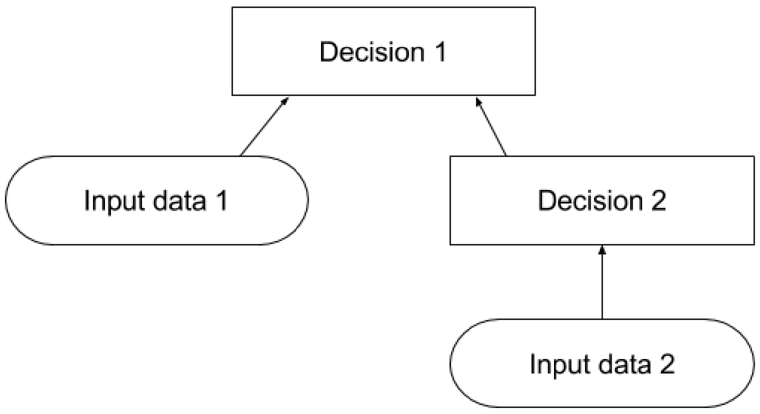


Figure 4: A simple Decision Requirements Diagram

In Figure 4 we see that two decision tables can be connected. First, we calculate Decision 2 and its output is the input to Decision 1. This way we can connect several decision tables so that each decision table provides input to the next decision table. This way we can build more complex decisions. In our current work only use one decision table at a time, i.e. our tables do not give any output to another table, and our tables do not need any outputs from other decision tables.

There are several tools, such as Signavio[1], Prologa[2] and Bizagi Studio[3], that supports DMN, enabling one to construct DRD diagrams and create decision tables.

2.2 Line sweeping

A sweep line algorithm (plane sweep algorithm) [1, 15] is used to find overlap or collision between lines (planes). Let us explain, how this algorithm works and detects overlap in a simple case.

Our task is to find all the intersection between N horizontal and vertical line segments. We have lines as in the left side of Figure 5 and we now want to find the intersection between these lines. In sweep line algorithm a vertical line just sweeps the data from left to right. Each time a vertical line hits a line segment's end points we need to perform some actions. The set keeps track of y-coordinates. Another

[1]http://www.signavio.com
[2]http://www.http://feb.kuleuven.be/prologa/
[3]http://www.bizagi.com/en/products/bpm-suite/studio

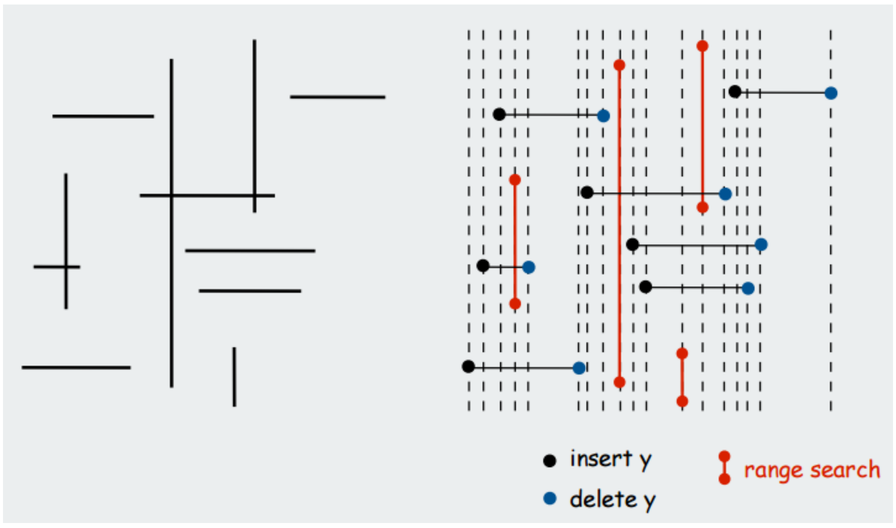


Figure 5: Line sweeping algorithm in action [16]

event happens when sweeping algorithm hits a right end (represented by blue dots in Figure 5) of a horizontal line segment then, this line segment or y-coordinate is removed from the set because this line is processed completely. The third kind of event happens when algorithm hits a vertical line (represented by red lines in Figure 5). Now algorithm has to perform a 1D-range search for y-coordinates that are in the set, and if some y-coordinates are between these vertical line endpoints, then we have an overlap between two lines.

For storing x-coordinates we have a couple of options. We can use a priority queue or sort x-coordinates into ascending order. It is important that x-coordinates are sorted otherwise line sweeping algorithm does not work properly. Sweep line algorithm takes $NlogN+R$ time to find all the R intersections between N orthogonal line segments. Sorting x-coordinates, inserting and deleting y-coordinates will take $NlogN$ time, and range search will take $NlogN + R$ time [15].

Spatial overlap. We described how line sweeping algorithm works with two-dimensional data, but what about three-, four-, or hundred-dimensional data? How can we find in this data an overlap? In the case of multidimensional data, an overlap occurs only if there are overlap in all dimensions. In Figure 6 we see that shapes A and C overlap (C is inside A), the second overlap is between A and B (partial overlap). Shape D does not overlap with any other shape. Using sweeping line algorithm, we can find these overlaps. We start by sweeping the first dimension, if overlap has been found, then we sweep the second dimension (it is necessary only to sweep these data that overlapped in the previous dimension), if we still have an overlap, then we scan the next dimension and so on. If we found

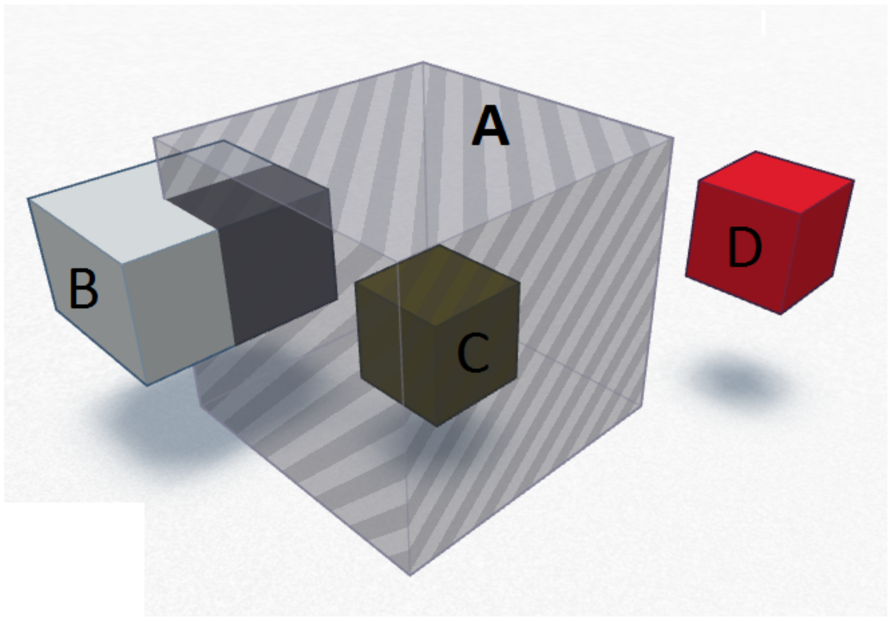


Figure 6: Spatial overlap

that two shapes or data overlap in all the dimension, then these two elements overlap [1]. In this way, we can locate the overlap between finite-dimensional data. Overlap size between A and C is C itself because C is inside A. Overlap between A and B is the dark gray shape that we can see in Figure 6. Overlap does not have to be between exactly two elements, it can occur between two or more elements.

2.3 Rule merging

2.3.1 Polygon rectangulation

A possible way to simplify a decision table is to convert all the rules into polygons. Then, the problem reduces to the problem of polygon rectangulation [4, 10]. In polygon rectangulation problem we give an orthogonal polygon (all interior angles are 90 or 270 degrees) as input and the polygon will be decomposed into adjacent, non-overlapping rectangles that will fully cover the input polygon. One can solve the problem by finding a maximum independent set [19] in bipartite intersection graph [22] of axis-parallel diagonals. The diagonals connect pairs of concave vertices (at the point of a 270-degree interior angle). We see concave vertices in Figure 7 the figure on the right. In Figure 7, the left figure represents the input polygon and center figure represents the result of polygon partition into rectangles.

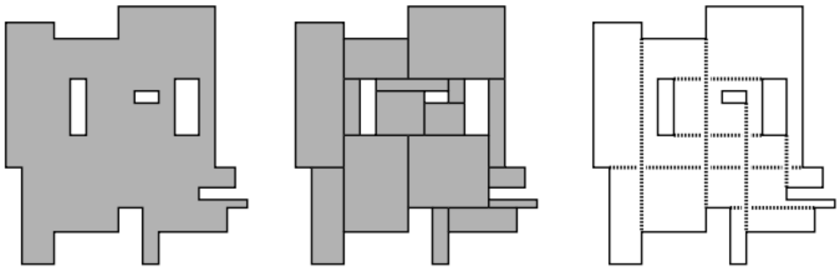


Figure 7: left: orthogonal polygon, center: minimum number of rectangles, right: diagonals that connect pairs of concave vertices [4]

Using this approach, we always get the minimum number of rectangles, which can be calculated as $n/2 + h - g - 1$ [5, 18]. The n is the vertex count of the current polygon; h represents the holes and g is the maximum size of a set of disjoint diagonals (connecting two concave vertices of the polygon). If we take Figure 7, then the minimum number of rectangles is $38/2 + 3 - 4 - 1 = 16$. The idea of converting rules into polygons and then solving polygon rectangulation problem can be used to simplify decision tables that contain only two dimensions, and it always gives us the optimal solution. The problem is that it only works with two-dimensional cases. Polygon rectangulation problem in three-dimensional version is NP-complete [3].

2.3.2 Classical approach

Pollack proposed a simple approach for combining decision tables, i.e. two rules that have the same output and differ only in one condition can be combined together [13, 17]. Also, Hewett and Leuchner [8] used very similar approach to simplify decision tables. In this approach, we first have to calculate all combinations of conditions. If we have conditions like in Table 5, then we can construct a decision table that can be seen in Table 6.

Conditions	**Values**
Annual Income > 10,000	Yes(Y), No(N)
Loan Size > 20,000	Y, N
Loan Duration > 36	Y, N

Table 5: Conditions and values

Next, we take the first rule (N, N, N) and try to combine these rule with other

		Possible Combinations							
Conditions	Annual Income > 10,000	N	N	N	N	Y	Y	Y	Y
	Loan Size > 20,000	N	N	Y	Y	N	N	Y	Y
	Loan Duration > 36	N	Y	N	Y	N	Y	N	Y
Actions	Grade A					X	X		X
	Grade B	X	X	X	X			X	

Table 6: Decision table with all combinations and actions

rules. The second rule (N, N, Y) can be merged together, and the new rule would be: N, N, "-". If we combine two rules together, then we start at the beginning again, this means we take the new rule (N, N, "-") and try to combine it with the other rules, but no merge is possible. Next, we take second rule (N, Y, N) and try to join to all the rules that are on the right side of the rule because we already have attempted to combine it with the rules that are on the left. The rule (N, Y, N) can be combined with the rule (N, Y, Y) and the new rule would be (N, Y, "-"). We again, start from the first rule (N, N, "-") and we see that it can be combined with the rule (N, Y, "-"). The new rule is then N, "-", "-". Again, start with the first rule (N, "-", "-") and try to combine with other rules that are on right side of the rule. We see that no merging is possible and take the next rule (Y, N, N), and it can combine with the rule (Y, N, Y), the new rule (Y, N, "-"). So again, we start with the first rule and try to combine it with other rules. No other rules can merge. The final decision table can be seen in Table 7.

		Combinations			
Conditions	Annual Income > 10,000	N	Y	Y	Y
	Loan Size > 20,000	-	N	Y	Y
	Loan Duration > 36	-	-	N	Y
Actions	Grade A		X		X
	Grade B	X		X	

Table 7: Decision table after merging

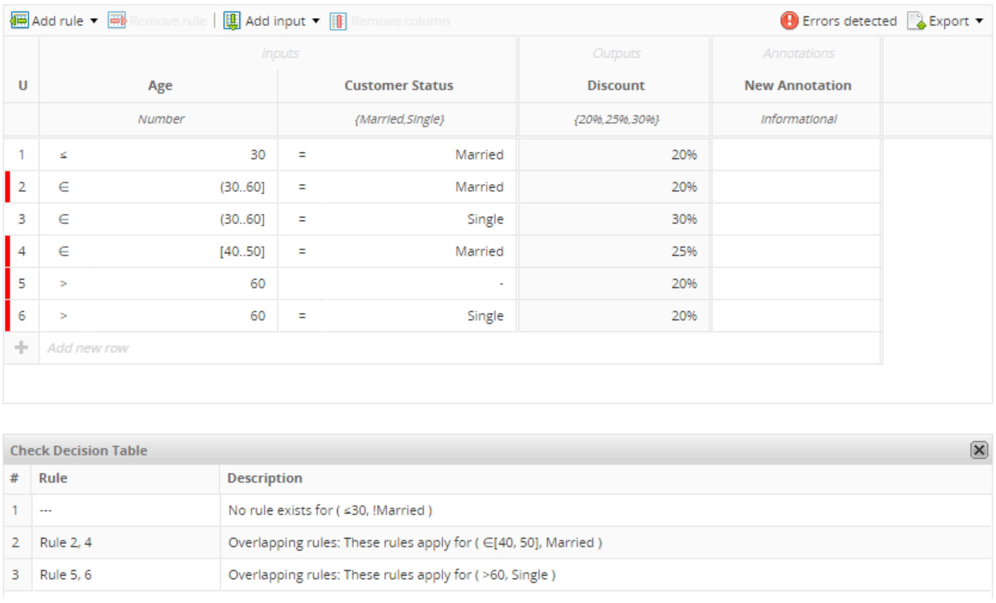
This approach can be used with numerical domains as well. Then, two rules can be combined, if they have the same result and only diverge in one condition, and the condition where they differ the values are contiguous otherwise two rules cannot merge.

2.4 DMN tools

Tools that allow one to construct business rules, such as Signavio and Prologa, provide some level of verification or simplification, but the algorithms used by these tools are proprietary and the information is undisclosed.

2.4.1 Signavio

Signavio is a software vendor and provides three tools: Process Editor for process modeling, Decision Manager for decision management and Effektif for creating work-flows. We are most interested in Decision Manager because with this tool one can create decision tables and verify them.



	Inputs				Outputs	Annotations
U	Age		Customer Status		Discount	New Annotation
	Number		{Married,Single}		{20%,25%,30%}	Informational
1	≤	30	=	Married	20%	
2	∈	(30..60]	=	Married	20%	
3	∈	(30..60]	=	Single	30%	
4	∈	[40..50]	=	Married	25%	
5	>	60		-	20%	
6	>	60	=	Single	20%	
+	Add new row					

Check Decision Table

#	Rule	Description
1	---	No rule exists for (≤30, !Married)
2	Rule 2, 4	Overlapping rules: These rules apply for (∈[40, 50], Married)
3	Rule 5, 6	Overlapping rules: These rules apply for (>60, Single)

Figure 8: Signavio Decision Manager decision table example

In Figure 8 it can be seen how Signavio Decision Manager displays decision tables and how the tool shows the errors in tables. How is this different from our work? The first difference is that Signavio does not tell us if the overlap also occurs in outputs. For example, rules 5 and 6 have the same output and input, the overlap between rules 2 and 4 occurs only in inputs. At this time, we cannot say anything about the overlap in outputs, based on the error table provided by Signavio.

The second difference is how Signavio highlights overlapping rules in the decision table. Signavio adds a vertical red line to rule numbers where the overlap occurs, but it colors all the overlaps. In this example, we have two different overlaps among the various rules. If the decision table is much bigger and has more overlaps in it, then it is much harder to find the overlap groups. In our implementation, you can push a button, and it will highlight these rules where the overlap occurs. Our implementation can be seen in Figure 18.

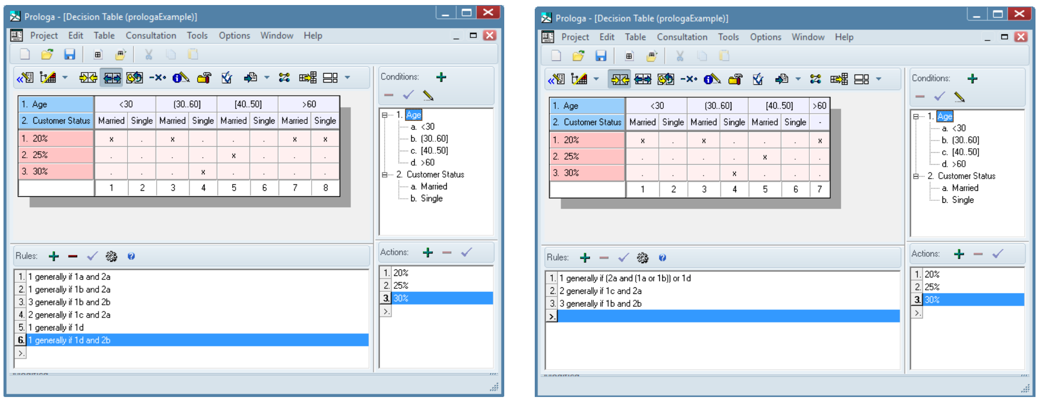
Another difference is with missing rules. How the missing rule is showed is very similar to our work, but it does not have the possibility of adding missing rules automatically. Our implementation has this functionality. We implemented a button for each missing rule, if the button is clicked, then it would add a missing rule automatically into decision table without typing it ourselves, this can be seen in Figure 19.

The diagnosis of overlapping and missing rules produced by Signavio is unnecessarily large: it often reports the same rule overlap multiple times. This behavior will be further explained in Chapter 5. Last difference is simplification option. Signavio does not allow any decision table simplification. In Figure 8, our simplification algorithm would have merged rules 5 and 6 into one rule.

2.4.2 Prologa

Prologa [20, 21], developed by Jan Vanthienen, allows modeling business decision knowledge, business rules, decision models, complex procedures in the form of decision tables. In Prologa, one can construct the decision tables in a way that prevents overlapping and missing rules. The tool also supports the simplification of a decision table via rule merging: two rules are merged when all but one of their input entries are the same, and their output entries are identical as well. Prologa has a limitation; it requires columns to have boolean or categorical domains. It means that numerical domains need to be discretized into intervals when constructing a decision table [20].

In Figure 9 in the left can be seen Prologa tool and it has the same values as in Figure 8. In the right can be seen how Prologa has simplified the decision table. Also, Prologa tool can minimize the inputted rules (rules can be seen in Rules section below the decision table in Figure 9); we can see that it minimized the inputted rules and right figure in Figure 9 has only three rules (original six). Minimization will combine multiple rules together so that combined rule will give the same output or information as the not combined rules. In the left figure in Figure 9 rules 1, 2, 5 and 6 can be combined to one (see rule 1 in the right figure in Figure 9).


Prologa - [Decision Table (prologaExample)]
Project Edit Table Consultation Tools Options Window Help
Conditions:
Actions:
Rules:

1. Age	<30		(30..60]		[40..50]		>60	
2. Customer Status	Married	Single	Married	Single	Married	Single	Married	Single
1. 20%	x	.	x	.	.	.	x	x
2. 25%	.	.	.	.	x	.	.	.
3. 30%	.	.	.	x	.	.	.	.
	1	2	3	4	5	6	7	8

1. 1 generally if 1a and 2a
2. 1 generally if 1b and 2a
3. 3 generally if 1b and 2b
4. 2 generally if 1c and 2a
5. 1 generally if 1d
6. 1 generally if 1d and 2b

1. Age	<30		(30..60]		[40..50]		>60
2. Customer Status	Married	Single	Married	Single	Married	Single	-
1. 20%	x	.	x	.	.	.	x
2. 25%	.	.	.	.	x	.	.
3. 30%	.	.	.	x	.	.	.
	1	2	3	4	5	6	7

1. 1 generally if (2a and (1a or 1b)) or 1d
2. 2 generally if 1c and 2a
3. 3 generally if 1b and 2b

1. Age
   a. <30
   b. (30..60]
   c. [40..50]
   d. >60
2. Customer Status
   a. Married
   b. Single

1. 20%
2. 25%
3. 30%

Figure 9: Prologa tool before and after simplification

3 Contribution

We describe in this chapter three algorithms: (i) detecting overlapping rules, (ii) detecting (in)completeness, and (iii) simplify decision table via rule merging. The presented algorithms rely on a geometric interpretation of a DMN decision table. Every rule in a table is presented as an iso-oriented hyper-rectangle in an N-dimensional space (N represents the number of columns). So, an input entry in a rule can be seen a constraint over one of the columns (i.e. dimensions).

In the case of a numerical dimension, an input entry is an interval (potentially with an infinite upper or lower bound) and thus it defines a segment or line over the dimension corresponding to that column. In the case of a categorical column, we can map to a disjoint interval each value of the column's domain – e.g. "Refinancing" to [0..1), "Card payoff" to [1..2), "Car leasing" to [2..3), etc. – and we can see an input entry under this column as defining a segment (or set of segments) over the dimension corresponding to the column in question. The conjunction of the entries of a row hence specify a hyper-rectangle, or in the case of a multi-valued categorical input entry (e.g. {"Refinancing", "Car leasing"}) multiple hyper-rectangles. The hyper-rectangles are iso-oriented because in S-FEEL only constraints of the form "attribute operator literal" are allowed and those constraints define iso-oriented lines or segments.

For example, decision table and the geometric interpretation of the table 1 are shown in Figure 10. The two dimensions, x and y, represent the two input columns (*Annual income* and *Loan size*) respectively. The table contains four rules: A, B, C, and D. Some of them are overlapping. For example, rule A overlaps with rule C. Their intersection is the rectangle $[500, 1000] \times [500, 1000]$. The table also contains missing values. For example, vector $\langle 200, 2000\rangle$ does not match any rule in Table 10.

3.1 Finding overlapping rules

Algorithm 1 finds overlapping rules in a DMN table. This algorithm is an extension of the line-sweep algorithm for two-dimensional spatial joins proposed in [1]. The idea of this latter algorithm is to pick one dimension (e.g. x-axis), project all objects into this dimension, and then sweep an imaginary line orthogonal to this axis (i.e. parallel to the y-axis). The line stops at every point in the x-axis where either an object starts or ends. When the line makes a "stop", we gather all objects that intersect the line (the *active list*). These objects overlap along their x-axis projection. In [1], it is then checked if the objects also overlap in the y-axis, and if so they are added to the result set (i.e. the objects overlap). Algorithm 1 extends this idea to N dimensions.

The algorithm takes as input:

1. **ruleList**, containing all rules of the input DMN table;
2. **i**, containing the index of the column under scrutiny;
3. **N**, representing the total number of columns;
4. **OverlappingRuleList**, storing the rules that overlap.

UC	Inputs		Outputs
	Annual Income	**Loan Size**	**Grade**
A	[0..1000]	[0..1000]	VG
B	[250..750]	[4000..5000]	G
C	[500..1500]	[500..3000]	F
D	[2000..2500]	[0..2000]	P

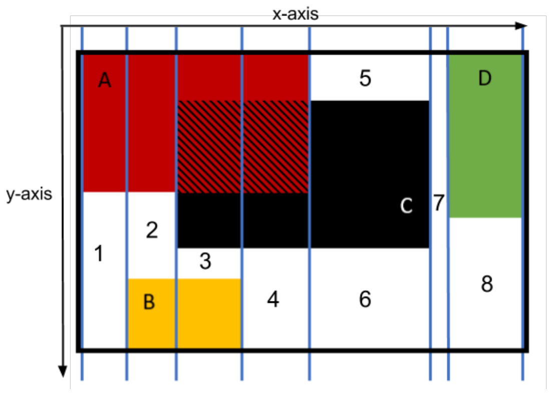


Figure 10: DMN decision table and its geometric representation

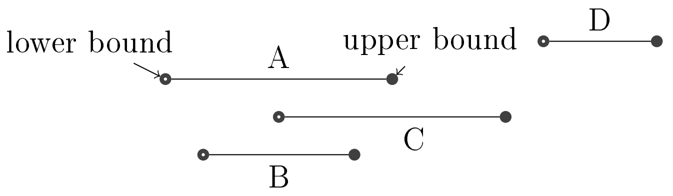
The algorithm starts analyzing the first column of the table (axis x). All rules are projected over this column. Note that the projection of a rule on a column is an interval. We indicate the projection of rule K over axes x and y with I_K^x and I_K^y respectively. All the intervals are represented in terms of upper and lower bounds. The bounds are sorted in ascending order (line 7), if the column has numerical values.

For categorical columns, we first find for every categorical value the rules that contain this categorical value. Next, we control each categorical value to each other and check if some categorical values have the same rules. If we have rules as in Table 8, then we find for each categorical value the rules where it is present. We get map were $\{A : \{1, 2, 3\}, B : \{1, 2, 3\}, C : \{1, 3\}, D : \{3, 4\}\}$. In this map, we can merge A and B because they have the same rules in them. We can say then that rules 1, 2 and 3 overlap in $\{A, B\}$, rules 1 and 3 overlap in $\{C\}$ and rules 3 and 4 overlap in $\{D\}$.

UC	Inputs	Outputs
	Value	**Result**
1	A, B, C	-
2	A, B	-
3	A, B, C, D	-
4	D	-

Table 8: Decision table with categorical values

The algorithm iterates over the list of sorted bounds (line 8). In the case of Figure 10, the rules projected over the x-axis correspond are:



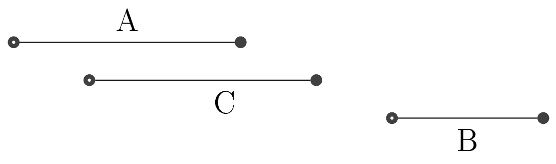
Considering the rules above, the algorithm first analyzes the lower bound of I_A^x. Therefore, I_A^x is added to an active list of intervals for the first column x, $\mathcal{L}_x$, since the bound processed is a lower bound (line 13). Next, the algorithm processes the lower bound of I_B^x and I_B^x is added to $\mathcal{L}_x$. Then, the lower bound of I_C^x is processed and I_C^x is added to $\mathcal{L}_x$. Finally, the algorithm processes the upper bound of I_B^x.

Algorithm 1: Procedure findOverlappingRules.

```
Input: ruleList; i; N; overlappingRuleList.
1  if $i == N$ then
2      define current overlap currentOverlapRules; /* it contains the list of
         rules that overlap up to the current point */ ;
3      if !overlappingRuleList.includes(currentOverlapRules) then
4          overlappingRuleList.put(currentOverlapRules);
5  else
6      define the current list of bounds $\mathcal{L}_{x_i}$;
7      sortedListAllBounds = ruleList.sort($i$);
8      foreach currentBound $\in$ sortedListAllBoundaries do
9          if !currentBound.isLower() then
10             findOverlappingRules($\mathcal{L}_{x_i}$, $i$ +1, $N$, overlappingRuleList); /*
                 recursive call */
11             $\mathcal{L}_{x_i}$.delete(currentBound);
12         else
13             $\mathcal{L}_{x_i}$.put(currentBound);
14         lastBound = currentBound;
15 return overlappingRuleList;
```

Every time an upper bound of an interval is processed (line 9), the following column of the table is analyzed (in this case y) by invoking *findOverlappingRules* recursively (line 10). The recursive call is invoked only if the current bound is an upper bound because lower bound does not give us any extra knowledge that we do not get from only processing the upper bound and no overlapping rules will be missed if we do not call the recursion. This would make the algorithm only slower because of the extra checking. That is the reason why we only invoke recursive call with upper bounds.

All the intervals projections on y of the rules corresponding to intervals contained in $\mathcal{L}_x$ (in our example A, B, and C) are represented in terms of upper bounds and lower bounds:



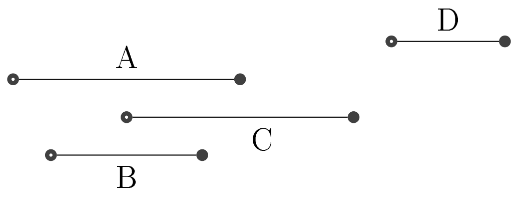
The bounds are sorted in ascending order. The algorithm iterates over the list of sorted bounds. Considering the intervals above, the algorithm first encounters the lower bound of I_A^y. Therefore, I_A^y is added to the active list of intervals for the second column y, $\mathcal{L}_y$. Next, the algorithm processes the lower bound of I_C^y and adds I_C^y to $\mathcal{L}_y$. Then, the upper bound of I_C^y is processed. Since there is no other column in the table, this means that all the rules corresponding to the intervals in $\mathcal{L}_y$ overlap. At the end of each recursion, the interval corresponding to the current bound is removed from the current active list (line 11). In addition, when the last column of the table is processed (line 1), the algorithm checks whether the identified set of overlapping rules is contained in one of the other sets produced in a previous recursion (lines 3). This check is important because our algorithm is not smart enough to identify only the maximal sets of overlapping rules with a non-empty intersection when sweeping the rules, but it will also return subsets of the maximal sets. This is the reason why we need to check if identified set is contained in one of the other sets produced in a previous recursion. If this identified set is not contained in previous sets, then the new set of overlapping rules is added to the output list overlappingRuleList (line 4). In this way, the procedure outputs maximal sets of overlapping rules having a non-empty intersection stored in overlappingRuleList (line 16).

3.2 Finding missing rules

Algorithm 2 describes the procedure for finding missing rules, which is also based on the line-sweep principle. The algorithm takes as inputs five parameters:

1. **ruleList**, containing all rules of the input DMN table;
2. **missingIntervals**, storing the current missing intervals;
3. **i**, containing the index of the column under scrutiny;
4. **N**, representing the total number of columns;
5. **MissingRuleList**, storing the missing rules.

The algorithm starts analyzing the first column of the table (axis x). Consider again the projection of the table in Figure 10 on x:



Lower and upper bounds of each interval are sorted in ascending order (line 3), if the current column has numerical values. For categorical column, we just check all the categorical values that are represented in the current rules to all possible values that can be in this column. If some categorical values are not represented in current rules, then these values are the missing categorical values and we are adding it into *missingIntervals*. The algorithm iterates over the list of sorted bounds (line 4).

Considering the rules above, the algorithm first analyzes the lower bound of I_A^x. Therefore, I_A^x is added to an active list of intervals for the first column x, $\mathcal{L}_x$. An interval is added to the active list only if its lower bound is processed (line 15). If the upper bound of an interval is processed, the interval is removed from the list (line 17). Next, the algorithm processes the lower bound of I_B^x. Since $\mathcal{L}_x$ is not empty, I_B^x is not added to $\mathcal{L}_x$ yet (line 11). Starting from the interval $\mathcal{I}_{\mathcal{A},\mathcal{B}}$ (line 12) having the lower bound of I_A^x as lower bound and the lower bound of I_B^x as upper bound, the following column of the table is analyzed (in this case y) by invoking *findMissingRules* recursively (line 13).

All the interval projections on y of the rules corresponding to intervals contained in $\mathcal{L}_x$ (in our example only A) are represented in terms of upper and lower bounds, obtaining in this case the following simple situation:

The bounds are sorted in ascending order. The algorithm iterates over the list of sorted bounds. The first bound taken into consideration is the lower bound of I_A^y so that I_A^y is added to $\mathcal{L}_y$ (since $\mathcal{L}_y$ is empty). Since this bound corresponds to the minimum possible value for y, there are no missing values between the minimum possible value for y and the lower bound of I_A^y (line 5). Next, the algorithm processes the second bound in $\mathcal{L}_y$ that is the upper bound of I_A^y. Considering that the upper bound of I_A^y is the last one in $\mathcal{L}_y$, the algorithm checks if this value corresponds to the maximum possible value for y (line 5). Since this is not the case, this means that there are missing values in the area between the upper bound of I_A^y and the next bound over the same column (in this case area 1). The algorithm checks if the identified area is contiguous to an area of missing values previously found (line 7), we need to check this because our sweeping algorithm may split the missing rules into several pieces (see Figure 10 areas 4 and 6). If this is the case, the two areas are merged (line 8). Doing so our algorithm will produce less missing rules and is more comprehensible by humans. If this is not the case, the area is added to a list of missing value areas (line 10). In our case, area 1 is added to a list of missing value areas. Note that the algorithm merges two areas of missing values only when the intervals corresponding to one column are contiguous and the ones corresponding to all the other columns are the same.

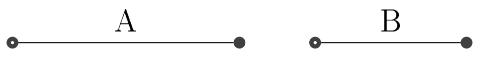
Algorithm 2: Procedure findMissingRules.

```
Input: $ruleList$; $missingIntervals$; $i$; $N$; $missingRuleList$.
1  if $i > N$ then
2      define the current list of boundaries $\mathcal{L}_{x_i}$;
3      $sortedListAllBoundaries = ruleList$.sort($i$);
4      foreach $currentBound \in sortedListAllBoundaries$ do
5          if $!areContiguous(lastBound, currentBound)$ then
6              $missingIntervals[i]$ = constructInterval($lastBound$,
                $currentBound$);
7              if $missingRuleList.canBeMerged(missingIntervals);$ then
8                  $missingRuleList$.merge($missingIntervals$);
9              else
10                 $missingRuleList$.add($missingIntervals$);
11         if $(!\mathcal{L}_{x_i}.isEmpty())$ then
12             $missingIntervals$ $[i]$ = constructInterval($lastBound$,
                $currentBound$);
13             findMissingRules($\mathcal{L}_{x_i}$,$missingIntervals$,$i$ +1, $N$,
                $missingRuleList$); /* recursive invocation */
14         if $currentBound.isLower()$ then
15             $\mathcal{L}_{x_i}$.put($currentBound$);
16         else
17             $\mathcal{L}_{x_i}$.delete($currentBound$);
18         $lastBound = currentBound$;
19 return $missingRuleList$;
```

In the example in Figure 10, areas 4 and 6 are merged.

At this point, the recursion ends and the algorithm proceeds analyzing the intervals in the projection along the x-axis. The last bound processed was the lower bound of I_B^x so that I_B^x is added to $\mathcal{L}_x$. Next, the algorithm processes the lower bound of I_C^x (since $\mathcal{L}_x$ is not empty, I_C^x is not added to $\mathcal{L}_x$ yet). Starting from the interval $\mathcal{I}_{\mathcal{B},\mathcal{C}}$ having the lower bound of I_B^x as lower bound and the lower bound of I_C^x as upper bound, the following column of the table is analyzed (in this case y) again through recursion.

All intervals projections on y of the rules corresponding to intervals contained in $\mathcal{L}_x$ (in this case A and B) are represented in terms of upper and lower bounds:



The bounds are sorted in ascending order. The algorithm iterates over the list of sorted bounds. Considering the rules above, the algorithm first processes the lower bound of I_A^y so that I_A^y is added to $\mathcal{L}_y$ ($\mathcal{L}_y$ is empty). Then, the upper bound of I_A^y is processed. When the algorithm reaches the upper bound of an interval in a certain column, the interval is removed from the corresponding active list. Therefore, I_A^y is removed from $\mathcal{L}_y$. Next, the lower bound of I_B^y is processed. Since $\mathcal{L}_y$ is empty, the algorithm checks if the previously processed bound is contiguous with the current one (line 5). Since this is not the case, this means that there are missing values in the area between the upper bound of I_A^y and the next bound over the same column (in this case area 2). The algorithm checks if the identified area is contiguous to an area of missing values previously found. If this is the case, the two areas are merged. If this is not the case, the area is added to a list of missing value areas (in our case area 2 is added to a list of missing value areas). The list of missing areas stored in missingRuleList is returned by the algorithm (line 19).

3.3 Decision table simplification

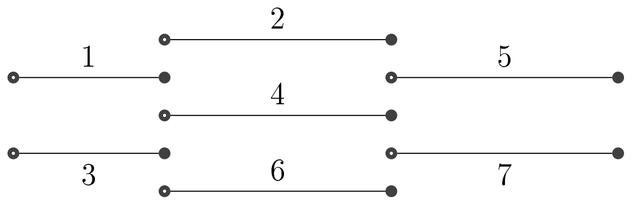
Before we describe rule merging algorithm, we have to describe procedures that we are doing before we can merge rules. First, we have to group all the rules that have the same output into one group. In Figure 10 all the rules would be in the separate group and no rule merging would be possible. Let us pretend that all the rules have the same output, then we have one group where are rules: A, B, C and D. Next, we make a connected graph [6] where rules are represented as nodes and nodes are connected with an edge, if they overlap or contiguous in one dimension. To find connected graph we can use sweep line technique as well. We first sweep the first dimension, we sweep the line until we reach some cap (previous interval and next interval are not contiguous and they do not overlap) and all the rules that were before the cap we take them to next dimension and again do the line sweep with these rules. Doing this, we end up with the connected graph. Separating rules into connected graphs will make the algorithm faster because we only have split and merge rules that are in the connected graph that we are processing. So the splitting part will not produce so many rules as it would be if we would apply our splitting algorithm to all the rules. Also, in merging part, the algorithm has to check fewer rules, if they can be merged into one rule. Applying this to rules A, B, C and D, we get three connected graphs: (A and C), (B) and (D). Next, we split each connected graph separately. The rules are sliced into smaller rules where there is no overlap between rules, this can be seen in group $G1$ in Figure 11. Doing all previously explained procedures, we end up with three groups of rules $G1$, $G2$ and $G3$ (see Figure 11).

Next, we will find the order of the input dimensions. Numerical dimensions are before categorical dimension in the order. The numerical dimension columns are sorted into descending order based on their cuts count (different bounds count). The categorical columns are sorted into ascending based on their different categorical value count. The numerical columns are processed by descending order because then there is more possibility that several rules can merge into one than columns where there are fewer cuts. It should produce fewer rules in the end. The categorical columns order to not affect the result so much. In our algorithm, they are sorted in ascending order, but we can sort these in descending order. We process the numerical columns first because it gave us better result mostly than another way around. These columns order will not always give the best result, it heavily depends on the current decision table. Further analysis is necessary. Finally, we are applying our rule merging algorithm to these groups. In our example, only rules that belong to group $G1$ the merging algorithm is applied because other groups have only one rule in them.

Algorithm 3 describes the procedure for finding missing rules, which is also based on the line-sweep principle. The algorithm takes as inputs three parameters:

1. **ruleList**, containing all rules of the input DMN table;
2. **N**, representing the total number of columns;
3. **columnOrder**, containing the input column order.

The algorithm first takes the input column from *columnOrder* that is analyzed first (line 3). In our case, it would be axis x. In the case of Figure 11, the rules in group $G1$ are projected over the x-axis are:



Upper and lower bounds of each interval are sorted in ascending order (line 4), if the column has numerical values. For categorical columns, we compare all rules *ruleList* to each other, if two rules have the same value in all the columns, except in current column (*dimensionIndex*). If rules have the same values, then the two rules are merged, and the categorical values are also merged that were in current column, so the merged rule has both rule categorical values. The algorithm iterates over the list of sorted bounds (line 5).

Considering the rules above, the algorithm first analyzes the lower bound of I_1^x. Then, it checks the if-clauses, but all the clauses are false, and it saves the bound I_1^x into *lastBound* (line 16). Next, the algorithm processes the lower bound

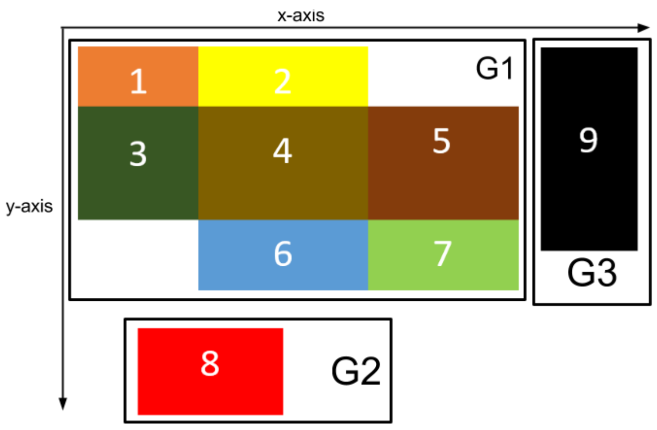


Figure 11: Sliced rules and their groups

of I_3^x and it assigns the current bound to *lastBound* (line 16). Next, the algorithm processes the upper bound of I_1^x. The current bound is added into *backList* (line 8) because it is the upper bound and *frontList* is empty (line 7). Next bound is the upper bound of I_3^x and it is added into *backList* (line 8). The algorithm processes the next bound that is the lower bound of I_2^x. The bound I_2^x is added into *frontList* (line 10) because it is the lower bound, *backList* is not empty and it is contiguous with the previous bond (line 9). Next, bound is the upper bound of I_4^x. The current bound is added into *frontList* (line 10) because it is the lower bound, *backList* is not empty and it has the same bound value as previous bound (line 9). Next bound I_6^x is also added into *frontList*. Next bound is the upper bound of I_2^x. We now apply the *mergeRules* function (line 12) because the others if-clauses are not true and *backList* is not empty and *frontList* is not empty (line 10) two of the function parameters are current *backList* and *frontList*. All the rules in *backList* are contiguous with *frontList* in *dimensionIndex*.

First, *mergeRules* takes one rule in *backList* and compares to all the other rules that are in *frontList* one by one. The comparison of two rules will tell us if two rules have the same input in all the input columns except one (*dimensionIndex*). We already know that two rules are contiguous in *dimensionIndex*. If they are the same, then two rules are merged into one rule. In our example, if *backList* contains $\{I_1^x, I_3^x\}$ and *frontList* contains $\{I_2^x, I_4^x, I_6^x\}$ then rules I_1^x and I_2^x, and rules I_3^x and I_4^x are merged into one rule. In *ruleList*, we have now $\{I_2^x, I_4^x, I_6^x, I_5^x, I_7^x\}$, where rules I_2^x and I_3^x have new boundaries (these boundaries are also updated in *backList*). Next, our *mergeRules* function tries to merge adjacent rules, we do this kind of merge because it will give us less rule in the end than without adjacent merging. If we have rules like Figure 12 and merge rules without merging adjacent rules, then we end up with three rules: {1, 2}, {3, 4, 7, 8} and

{5, 6} (the rules in set are merged), but if we do adjacent merging, then we end up with two rules: {1, 2, 3, 4, 5, 6} and {7, 8}. We see that merging neighbor rules give us fewer rules than without it. For adjacent merging, the function takes the *backList* array and compares each rule against each other. Two rules are adjacent if in one dimension they are contiguous and in all other dimension, they have the same value. In current *backList*, two rules are adjacent: I_2^x and I_4^x. These two rules are merged and new *ruleList* contains four rules $\{I_2^x, I_6^x, I_5^x, I_7^x\}$. Finally, *mergeRules* returns the *ruleList*.

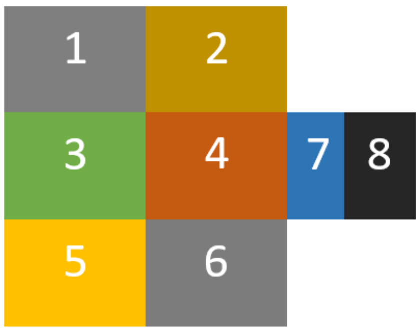


Figure 12: Rules geometric representation

After *mergeRules* function, the algorithm has updated *ruleList* and will make the *backList* and *frontList* empty (lines 13, 14). Newly merged rules can be seen in Figure 13. Last we were processing the upper bound of I_2^x and now we are adding this into *backList* (line 15). Next, the algorithm processes the upper bound of I_4^x, but we see that this rule has merged with I_2^x and the rule is not anymore in *ruleList* and the algorithm ignores that rule (line 6). Next bound is the upper bound of I_6^x and it is added to *backList* (line 8) because the rule is still in *ruleList* list (line 6). The algorithm next processes the lower bound of I_5^x and will add this into *frontList* (line 10) because it is the lower bound and is contiguous with last bound (line 9). Next bound is the lower bound of I_7^x and this also will be added into *frontList* (line 10). Next, the algorithm processes the upper bound of I_5^x and *mergeRules* function will be called (line 12) because *backList* and *frontList* are not empty, and previous if-clauses were not true (line 11).

First, *mergeRules* takes one rule in *backList* and compares to all other rules that are in *frontList* one by one. The comparison of two rules will tell us if two rules have the same input in all the input columns except one, where they are contiguous (x-axis). If they are the same, then two rules are merged into one rule. In our example, if *backList* contains $\{I_2^x, I_6^x\}$ and *frontList* contains $\{I_5^x, I_7^x\}$ then rules I_6^x and I_7^x are merged into one rule. So, in *ruleList* we have now $\{I_2^x, I_5^x,$

Algorithm 3: Procedure hyperplaneSweep.

```
Input: ruleList; N; columnOrder.
1  for i in range(N) do
2      define backList; /* it contains the list of rules that end in same point */
         define frontList; /* it contains the list of rules that start in same
         point and meet with backList rules */ ;
3      dimensionIndex = columnOrder[i];
4      sortedListAllBounds = ruleList.sort(dimensionIndex);
5      foreach currentBound ∈ sortedListAllBoundaries do
6          if currentBound.isIn(ruleList) then
7              if (currentBound.isUpper() && frontList.isEmpty()) then
8                  backList.put(currentBound);
9              else if (currentBound.isLower() && !backList.isEmpty()
                 && (areContiguous(lastBound, currentBound) ||
                 (lastBound.value == currentBound.value)) then
10                 frontList.put(currentBound);
11             else if (!backList.isEmpty() && !frontList.isEmpty()) then
12                 ruleList = mergeRules(ruleList, backList, frontList,
                     dimensionIndex);
13                 backList = {};
14                 frontList = {};
15                 backList.put(currentBound);
16             lastBound = currentBound;
17     ruleList = mergeRules(ruleList, backList, frontList,
         dimensionIndex);
18 return ruleList;
```

$I_7^x\}$, where the rule I_7^x has new boundaries (these boundaries are also updated in *backList*). Next, our *mergeRules* function tries to merge adjacent rules. For this, the function takes the *backList* array and compares each rule against each other. Two rules are adjacent if in one dimension they are contiguous and in all other dimensions, they have the same value. In current *backList*, there are not adjacent rules. So, *mergeRules* returns the *ruleList* that contains three rules $\{I_2^x, I_5^x, I_7^x\}$.

After *mergeRules* function, the algorithm has updated *ruleList* and will make the *backList* and *frontList* empty (lines 13, 14). Newly merged rules can be seen in Figure 14. Last we were processing the upper bound of I_5^x and now the algorithm is adding this into *backList* (line 15). Next, the algorithm processes the upper bound of I_7^x and is added into *backList* (line 8). We have no bounds to

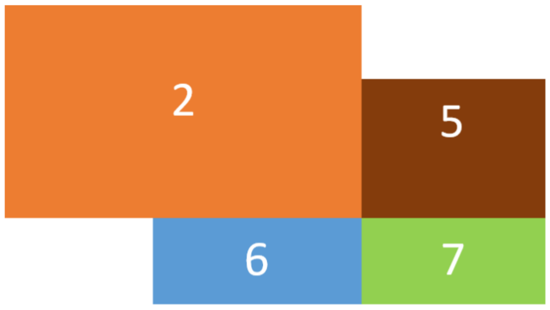
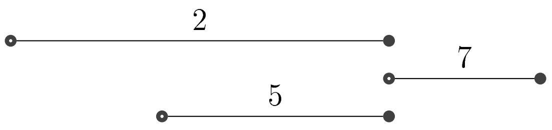


Figure 13: Rules after the first merge

process and we try to merge these rules that are in $backList$ and $frontList$ (line 17). The algorithm cannot merge any rule so the algorithm will process the next dimension.

The algorithm takes the second input column from $columnOrder$ (line 3). All interval projections on y of the rules corresponding to rules in Figure 14 are represented in terms of upper and lower bounds:



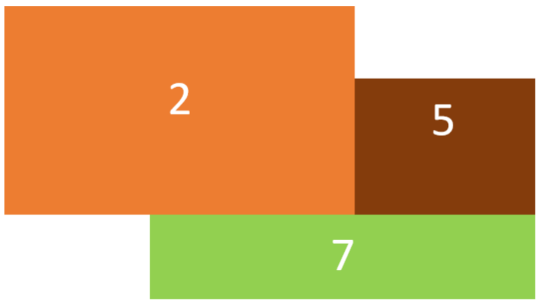


Figure 14: Rules after the second merge

Lower and upper bounds of each interval are sorted in ascending order (line 4). The algorithm iterates over the list of sorted bounds (line 5). The algorithm will process the bounds the same way as it did with the x-axis. The algorithm does not merge any rules because there are no rules that can be merged. Algorithm then stops and returns the $ruleList$ (line 18).

After all groups are analyzed and swiped over, we end up with five rules that can be seen in Figure 15.

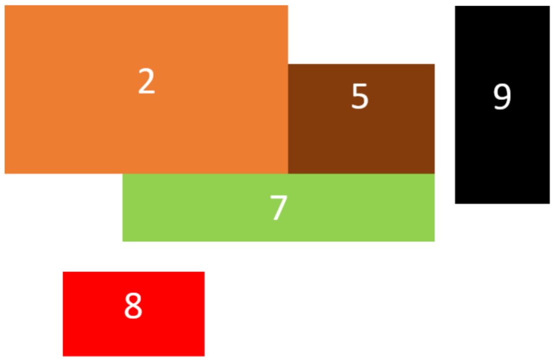


Figure 15: Rules after merging

4 Tool implementation

We implemented the algorithms on top of dmn-js: the open-source rendering and editing toolkit of Camunda DMN[4]. In it current version, dmn-js does not support correctness verification or simplification. Our dmn-js extension with verification and simplification features can be found at `https://github.com/ulaurson/dmn-js` and a deployed version is available for testing at `http://dmn.cs.ut.ee/`.

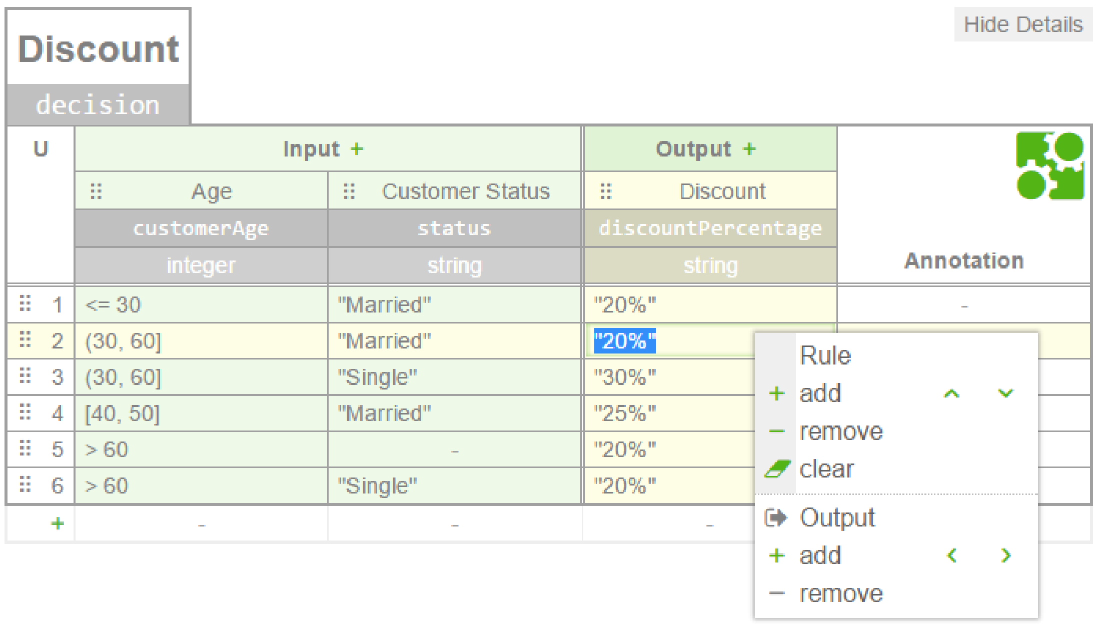
Camunda it is an open source platform and can be used for workflow and business process management. Camunda allows you to create BPMN diagrams and decision tables, but it does not provide any decision table verification or simplification. It is possible to download Camunda DMN tool into your computer and start writing business rules yourself. All the sources are available on GitHub. Camunda allows all people to contribute to dmn-js. Also, people can fix issues and implement new features into DMN.

It uses Node.js, which is an open-source, runtime environment for developing a server-side web application. Also, you have to install npm and grunt to Node.js in order to build the project, if you download it from GitHub. npm is a package manager for Node.js and grunt is a JavaScript task runner. The tool itself is also developed in JavaScript. The tool is a library-style application that makes adding new functionality very simple. We made a new folder into library folder and added our scripts into there. We did not have to change any core dmn-js scripts. To use our new created functions we have to change index.html and index.js. Into index.html we added our verification and simplification buttons, into index.js we modify so that its modeler reads our scripts and add functionalities behind these added buttons, so when buttons are clicked our functions are executed.

In Figure 16, we see that there are two additional gray cells that the regular decision table does not have. These two rows describe technical details. These details are necessary for decision engine in order to execute the decision. The first gray row contains variable names. In this example, we see three variable names, *customerAge*, *status*, *discountPercentage*. Second rows are telling to decision engine the type of the expression, in this example we see two types: integer and string. There are more types such as Boolean, double and date (our algorithm does not work with date type). dmn-js allows us to hide these technical details by clicking the button "Hide Details" and it will hide these technical detail cells (see Figure 18).

In dmn-js one can change hit policy, the user just has to click on the hit policy and a list will pop-up, where one can select different hit policies. The tool has functions that allow us to add and delete input and outputs. Adding an input and output is very simple the user just has to click green plus sign that is behind

[4]`https://camunda.org/`


Discount

decision

Hide Details

U	Input +		Output +	Annotation
	Age	Customer Status	Discount	
	customerAge	status	discountPercentage	
	integer	string	string	
1	<= 30	"Married"	"20%"	-
2	(30, 60]	"Married"	"20%"	
3	(30, 60]	"Single"	"30%"	
4	[40, 50]	"Married"	"25%"	
5	> 60	-	"20%"	
6	> 60	"Single"	"20%"	
+	-	-	-	

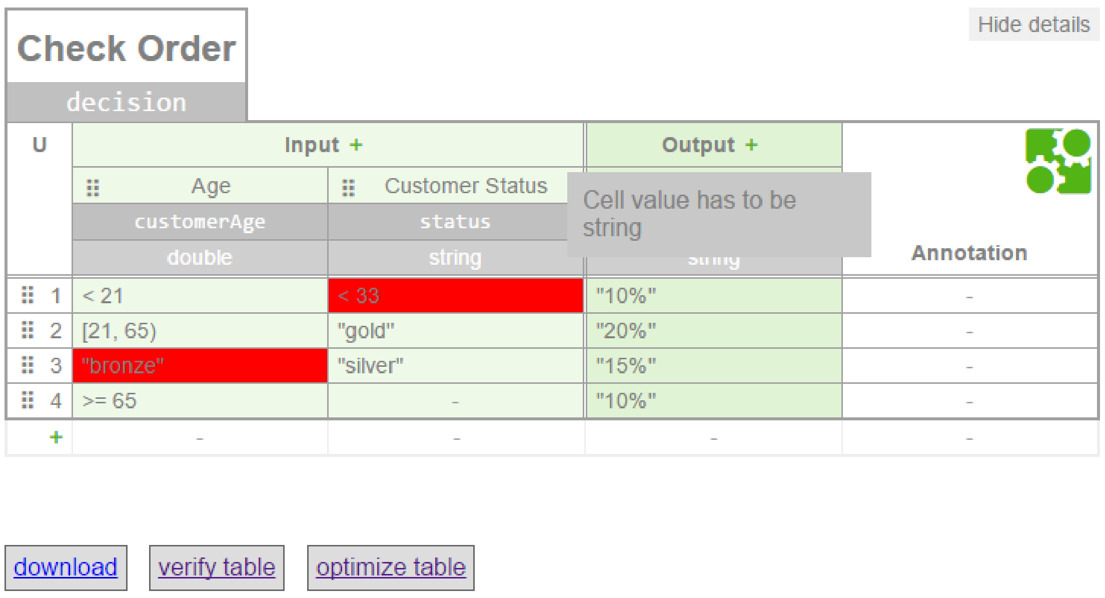
Figure 16: dmn-js decision table example

Input and Output, see Figure 16. If the user performs a mouse right-click on one cell then a new panel opens. The panel has two sections: Rule and Output or Input. Active rule and column cells backgrounds are light yellow. In Rule section, we can add new rules below or above of our active rule, delete our current active rule, and clear the active cell content. In Output section, we can add new column to our active column left or right side, and remove our current active column. dmn-js has functionality that allows us to change column and rule orders, one has to click on these six dots and hold the mouse button down and just move where user want it and release the mouse button. The dmn-js tool also has functionality for reading decision tables from XML and writing decision tables into XML. These functionalities allow us to save decision tables and later reading it again, the table is saved into a .dmn format. The button that downloads decision table into your computer can be seen in Figure 17, the button is called "download".

Before we go to detail what we implemented into dmn-js, let us explain how these decision tables are represented in computer readable manner. All the rules, inputs and outputs are saved as objects. These objects contain IDs (rule, column and cell), a reference to the previous or next rule, a reference to the previous or to the next column, cell value, column type and much more that are irrelevant. We take these objects and make our new objects, these objects contain only information that we need such as rule ID, column ID, cell ID, rule number, cell value and column type. Then, we create a new object that represents one rule, its value is an array. Into the array, we add all previously constructed objects that belong to the current rule. The array first element is the object that represents the first column value and the second element is the another object that represents the second column value and so on.

4.1 Syntactic check

We implemented a syntactic check that checks if each cell has the right type of content. Our algorithm just controls each cell and checks if its content matches its column type. If not, then we color this cell red and add a tooltip that is hidden at first. To add the tooltip, we first make the error text and make the div element where we put the error text. Next, we associate the created div with the cell where the syntactic error is. Also, we add two mouse events to the cell. One event is onmouseover that will show the div element with the error text, the div position is calculated based on the cursor position. onmouseout will hide the div element. Pointing a mouse on error cell will execute onmouseover event and will reveal tooltip and its content. We are making the div element for every error cell separately and also add these two mouse events into each error cell. In Figure 17 we see two syntactic errors.



U	Input +		Output +	Annotation
	Age	Customer Status		
	customerAge	status		
	double	string	string	
1	< 21	🟥 < 33	"10%"	-
2	[21, 65)	"gold"	"20%"	-
3	🟥 "bronze"	"silver"	"15%"	-
4	>= 65	-	"10%"	-
+	-	-	-	-

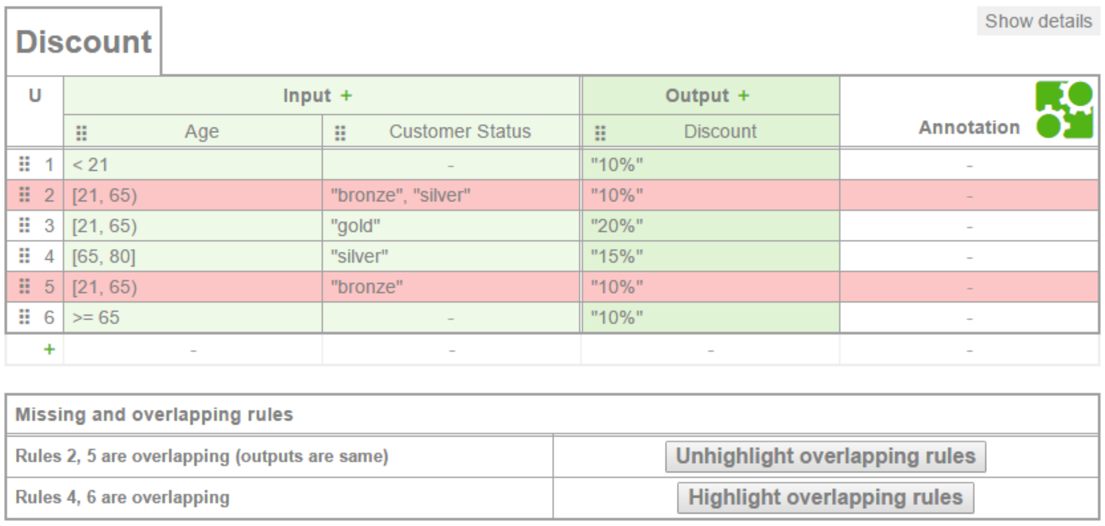
Figure 17: Decision table with syntactic errors

In third rule and first column we see an error. We have a syntactic error because all the cells in this column have to be double, but this cell has a string value. The second syntactic error is in first rule and second column, it cell content has to be a string, but right now it content is an integer. We mentioned that we implemented a tooltip for every error cell, we can see one of the tooltips in Figure 17. Our cursor is pointed to a cell which value is "$<$ 33" and a gray tooltip shows what is wrong with this cell, if we drag mouse out of the cell, then tooltip will be hidden again.

4.2 Overlapping rules

For verification, a user just has to click a verifier button and our algorithm will verify the decision table. Verification button is below decision table in the left corner, see Figure 17. When clicked this button it will find all overlaps in the decision table. Our algorithms will find two kinds of overlaps, overlaps where inputs and outputs are the same and overlaps where inputs are the same and at least one output is different. In Figure 18 we see that our algorithm has found two overlaps. Rules 2 and 5, and rules 4 and 6 have overlaps. Let us explain in more detail about these overlapping rules that we can see in Figure 18. The overlapping rules can be seen in "Missing and overlapping rules" table. It produces maximal sets of overlapping rules with a non-empty intersection. If at the end of rule numbers is "(outputs are the same)" then these rules have same inputs and outputs. We see that rules 4 and 6 are also overlapping, but they have different outputs. We implemented a function that will highlight the overlapping rules when clicked. The overlapping rules are highlighted in red. In Figure 18 we see that highlighted are rules 2 and 5. Clicking the second time same button will unhighlight overlapping rules.

Our algorithms take input these objects that we constructed from dmn-js objects we described earlier (contain objects where each object represents one rule). Our implemented findOverlappingRules finds returns an array where each array element represent an overlap. The overlaps are saved as objects and it contains the row IDs (keys) that overlap each other. Next, we are adding output values and rule number to each row IDs. We get new an array that has the information that we need in order to check the overlap type. Next, we check if inside an overlap all the rules have the same output or not, if not then we separate rules into a smaller group where each group has the same output values. Next, we take each overlap object and get the rule numbers and overlap type. Using that information we construct new table row element (tr), it will be added to error table. Into the row we store the overlapping rule IDs of current overlap, we need this information for highlighting rules, this information is hidden from the user. Row first column has the output string, it tells us which rules overlap and overlap type. Row second column contains the highlighting button where we add a click event listener. This event listener will add a new CSS class to all the rows that overlap or remove the class from the rows. The class will color the overlapping rows to light red. For highlighting, we use the stored overlapping rule IDs. This way we add every overlap into the error table.



Discount

Show details

U	Input +		Output +	Annotation
	Age	Customer Status	Discount	
1	< 21	-	"10%"	-
2	[21, 65)	"bronze", "silver"	"10%"	-
3	[21, 65)	"gold"	"20%"	-
4	[65, 80]	"silver"	"15%"	-
5	[21, 65)	"bronze"	"10%"	-
6	>= 65	-	"10%"	-
+	-	-	-	-

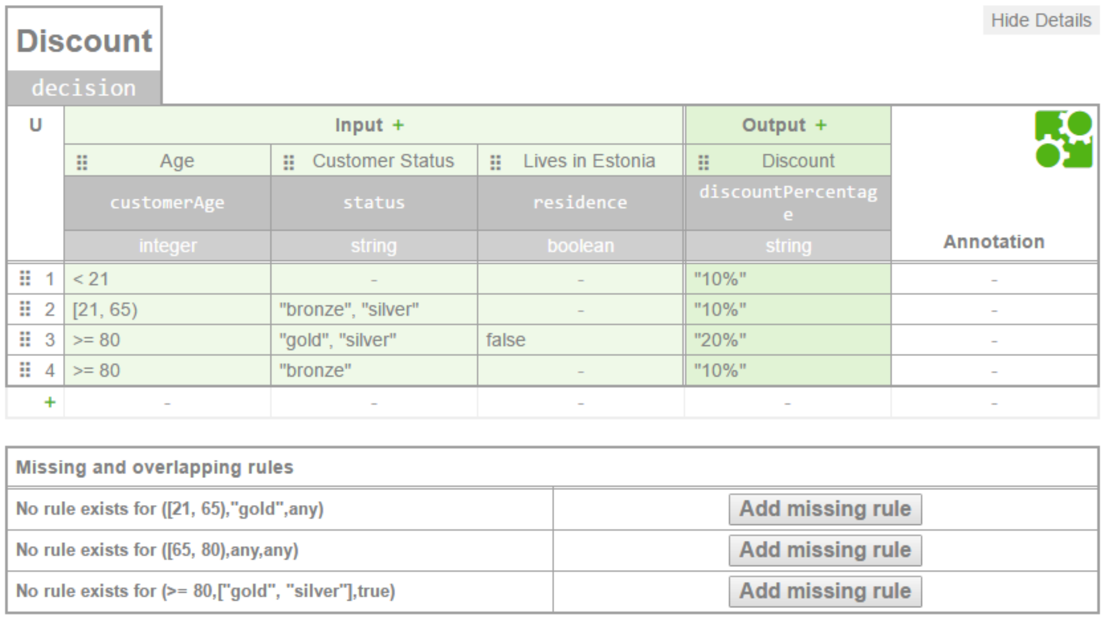
Missing and overlapping rules	
Rules 2, 5 are overlapping (outputs are same)	Unhighlight overlapping rules
Rules 4, 6 are overlapping	Highlight overlapping rules

Figure 18: dmn-js decision table with overlapping rules

4.3 Missing rules

The same verification button that finds the overlapping rules also finds the missing rules. In Figure 19 decision table has three missing rules. These missing rule cases are very easy to read. In each missing rule description, there are values inside brackets. The first value inside brackets shows, what is missing in the first input column, the next value shows, what is missing in the second input column and so on. For example, missing rule ([21, 65), "gold", any) has following missing rule: Age = [21, 65), Customer Status = "gold", Lives in Estonia = "-". “Any” means irrelevant value ("-") and every input will satisfy this clause.

In Figure 19 we can say that there are three kinds of missing rule cases. First, missing rule case is missing a categorical value. With interval [21, 65) there is no rule with "Customer Status" equals "gold". The second case is missing an interval value. In decision table, there is no rule where "Age" equals [65, 80). The third case has a missing Boolean value. With "Age" equals >= 80 and "Customer Status" equals "gold" or "silver" there is no rule with "Lives in Estonia" equals true.



U	Input +			Output +	
	Age	Customer Status	Lives in Estonia	Discount	
	customerAge	status	residence	discountPercentage	
	integer	string	boolean	string	Annotation
1	< 21	-	-	"10%"	-
2	[21, 65)	"bronze", "silver"	-	"10%"	-
3	>= 80	"gold", "silver"	false	"20%"	-
4	>= 80	"bronze"	-	"10%"	-
+	-	-	-	-	-

Missing and overlapping rules	
No rule exists for ([21, 65),"gold",any)	Add missing rule
No rule exists for ([65, 80),any,any)	Add missing rule
No rule exists for (>= 80,["gold", "silver"],true)	Add missing rule

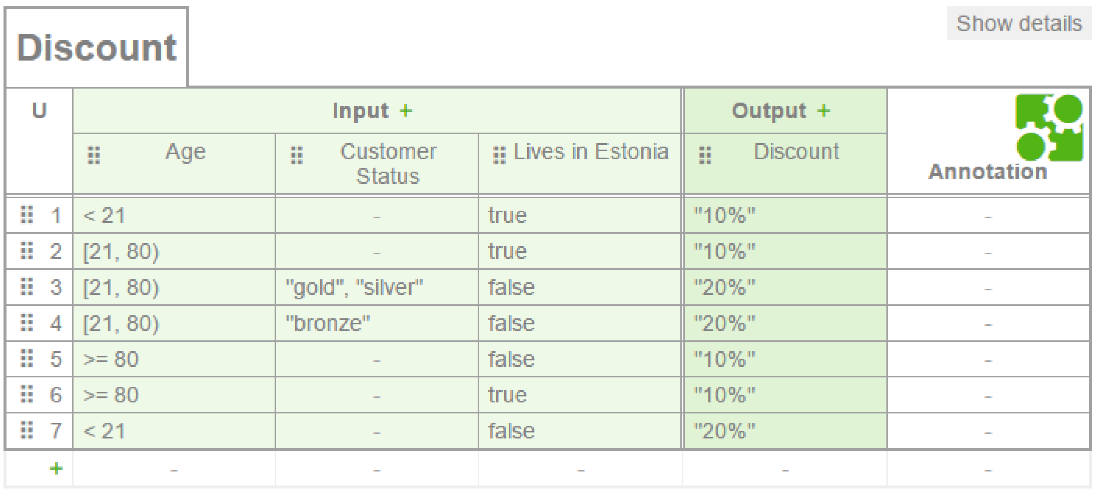
Figure 19: dmn-js decision table with missing rules

Our missing rule algorithms will output an object that contains arrays. Each array represents a missing rule, array elements contain the missing intervals. For every missing rule, we make a new table row element (tr) using array information, it will be added into the error table. Into the row we store the array that contains the missing intervals, this information is hidden from the user. The row first column contains the information of the missing rule. The row second column contains the button for adding missing rules into the table to this button we add a click event listener. This event listener will add a new row into decision table and use the hidden data to fill the added row cells with the intervals and the clicked row from error table will be removed. To output columns, the implementation will add a "-".

4.4 Table simplification

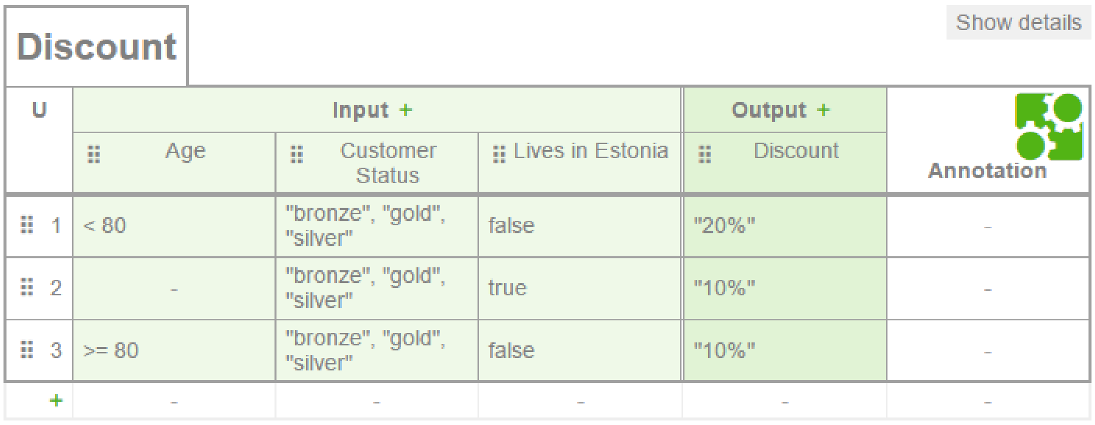
Last functionality that we added into dmn-js was decision table simplification via rule merging. The user can just press the simplification button and the algorithm will simplify the decision table, see Figure 17 where simplification button is located. First, our algorithm finds the new simplified rules. Next, the algorithm will delete the old table and then will add each rule one by one into the decision table.

Simplification will make the decision table more readable and understandable.


Discount

Show details

U	Input +			Output +	Annotation
	Age	Customer Status	Lives in Estonia	Discount	
1	< 21	-	true	"10%"	-
2	[21, 80)	-	true	"10%"	-
3	[21, 80)	"gold", "silver"	false	"20%"	-
4	[21, 80)	"bronze"	false	"20%"	-
5	>= 80	-	false	"10%"	-
6	>= 80	-	true	"10%"	-
7	< 21	-	false	"20%"	-
+	-	-	-	-	-


Discount

Show details

U	Input +			Output +	Annotation
	Age	Customer Status	Lives in Estonia	Discount	
1	< 80	"bronze", "gold", "silver"	false	"20%"	-
2	-	"bronze", "gold", "silver"	true	"10%"	-
3	>= 80	"bronze", "gold", "silver"	false	"10%"	-
+	-	-	-	-	-

Figure 20: Decision table before and after simplification

In Figure 20, the above decision table is a regular decision table that has no overlapping rules and no missing rules. Lower decision table represents the simplified table, it has fewer rows and it is more readable. Our algorithm merged four times two rules into one in this example, it also can merge three or more rules into one rule. Let us explain more how and why we can merge rules into one. Figure 20 decision table allows our algorithm to merge rules 1 and 2 because two rules only differ in one input clause and output clause is the same. These two intervals can merge into one because two intervals are contiguous. In our example, "$<$ 21" and "[21, 80)" we can merge into "$<$ 80." New merged rule can be merged with rule 6 (see rule 3 lower table in Figure 20). Rules 3 and 4 we can merge because they differ only in "Customer Status". After merging cell has values: "gold", "silver", "bronze." New merged rule can merge with rule 7 (see rule 1 lower table in Figure

20). We see that rules 1 and 3 in lower table are also contiguous in one column and in other columns they have same values, but we cannot merge these rules because the output clauses are different.

5 Evaluation

For the evaluation, we created decision tables from a loan dataset of LendingClub – a peer-to-peer lending marketplace[5]. The employed dataset contains data about all loans issued in 2013-2014 (23 5629 loans). For each loan, there are attributes of the loan itself (e.g., amount, purpose), of the lender (e.g., income, family status, property ownership), and a credit grade (A, B, C, D, E, F, G).

Using Weka [7], we trained decision trees to classify the grade of each loan from a subset of the loan attributes. Then each trained decision tree into a DMN table by mapping each path from the root to a leaf of the tree into a rule. For this, we used algorithms that were implemented by Irene Teinemaa. Then we implemented an algorithm that interprets a DMN table into .dmn file. The .dmn file then we imported into dmn-js. For verification and simplification, we used different attributes and pruning parameters in decision tree discovery.

5.1 Verification

Using different attributes and pruning parameters in the decision tree discovery, we generated DMN tables containing approx. 500, 1000 and 1500 rules and 3, 5 and 7 columns (nine tables in total). The 3-dimensional (i.e. 3-column) tables have one categorical and two numerical input columns; the 5-dimensional tables have two categorical and three numerical input columns, and the 7-dimensional tables have two categorical and five numerical input columns.

By construction, the generated tables do not contain overlapping or missing rules. To introduce overlapping rules in a table, we selected 10% of the rules. For each of them, we then randomly selected one column, and we injected noise into the input entry in the cell in the selected column by decreasing its lower bound and increasing its upper bound in the case of a numerical domain (e.g. interval [3..6] becomes [2..7]) and by adding one value in the case of a categorical domain (e.g. { Refinancing, CreditCardPayoff } becomes { Refinancing, CreditCardPayoff, Leasing }). These modifications make it that the rule will overlap others. Conversely, to introduce missing rule errors, we selected 10% of the rules, picked a random column for each row and "shrank" the corresponding input entry.

We checked each generated table both for missing and incomplete rules and measured execution times averaged over 5 runs on a single core of a 64-bit 2.2 GHz Intel Core i5-5200U processor with 16GB of RAM. The results are shown in Table 9. Execution times for missing rules detection are under 2 seconds, except for the 7-columns tables with 1000-1500 rules. The detection of overlapping rules leads to higher execution times, due to the need to detect sets of overlapping

[5]Dataset available at `https://www.lendingclub.com/info/download-data.action`

rules and ensure maximality. The execution times for overlapping rules detection on the 3-columns tables is higher than on the 5-columns tables because the 5-columns tables have fewer rule overlaps. This is because there are proportionally fewer categorical columns in the 5-columns tables than in the 3-columns ones, and the modifications made to categorical columns create more overlaps.

In addition to implementing our algorithms, we implemented algorithms designed to produce the same output as Signavio. In Signavio, if multiple rules have a joint intersection (e.g. rules {r1, r2, r3}) the output contains an overlap entry for the triplet {r1, r2, r3} but also for the pairs {r1, r2}, {r2, r3} and {r1, r3} (i.e. subsets of the overlapping set). Furthermore, the overlap of pair {r1, r2} may be reported multiple times if r3 breaks $r1 \cap r2$ into multiple hyper-rectangles (and same for {r2, r3} and {r1, r3}). Meanwhile, our approach produces only maximal sets of overlapping rules with a non-empty intersection. In Table 12 can be seen what Signavio outputs and what our approach is outputting. Also, Signavio will output more missing rules because Signavio does not merge the missing rules. Differences between our approach and Signavio approach can be seen in Table 14.

Table 10 shows the number of sets of overlapping rules and the number of missing rules identified by our approach vs. Signavio's one. In all runs, both the number of overlapping and missing rules is drastically lower in our approach.

	3 columns			5 columns			7 columns		
#rules	499	998	1 492	505	1 000	1 506	502	1 019	1 496
overlapping time	432ms	7 975ms	29 201ms	240ms	2 158ms	7 079ms	8 318ms	8 953ms	84 927ms
missing time	160ms	611ms	1 672ms	163ms	820ms	1 942ms	2 173ms	7 029ms	18 263ms

Table 9: Execution times (in milliseconds)

		3 columns			5 columns			7 columns		
#rules		499	998	1 492	505	1 000	1 506	502	1 019	1 496
#overlapping rule sets	our approach	239	725	1204	172	363	632	300	494	854
	Signavio	1 226	10 920	23 115	679	3 692	8 921	23 175	22 002	118 920
#missing rules	our approach	117	330	726	136	254	462	134	322	518
	Signavio	668	2 655	5 386	563	2 022	4 832	5 201	18 076	43 552

Table 10: Number of reported errors of type "overlapping rules" & "missing rule"

UC	Inputs		Outputs
	Age	**Points**	**Discount**
1	<= 100	-	-
2	[50..75]	[0..100]	-
2	>= 50	-	-

Table 11: Decision table with overlapping rules

Approach	**Rules**	**Description**
Signavio#1	Rule 1, 3	Overlapping rules: These rules apply for ([50, 75], <0)
Signavio#2	Rule 1, 2, 3	Overlapping rules: These rules apply for ([50, 75], [0, 100])
Signavio#3	Rule 1, 3	Overlapping rules: These rules apply for ([50, 75], >100)
Signavio#4	Rule 1, 3	Overlapping rules: These rules apply for ((75, 100], any)
Our approach#1	Rule 1, 2, 3	Outputs are same

Table 12: Outputted overlapping rules by Signavio and our approach. Based on Table 13

UC	Inputs		Outputs
	Age	**Points**	**Discount**
1	< 100	[0..100]	-
2	[20..50]	> 200	-
2	>= 100	-	-

Table 13: Decision table with missing rules

Approach	**Description**
Signavio#1	No rule exists for (<20, <0)
Signavio#2	No rule exists for (<20, >100)
Signavio#3	No rule exists for ([20, 50], <0)
Signavio#4	No rule exists for ([20, 50], (100, 200])
Signavio#5	No rule exists for ((50, 100), <0)
Signavio#6	No rule exists for ((50, 100), >100)
Our approach#1	No rule exists for (<100, <0)
Our approach#2	No rule exists for (<20, >100)
Our approach#3	No rule exists for (<[20, 50], (100, 200])
Our approach#4	No rule exists for ((50, 100), >100)

Table 14: Outputted missing rules by Signavio and our approach. Based on Table 13

5.2 Simplification

Using different attributes and pruning parameters in the decision tree discovery, we generated DMN tables containing approx. 100 rules, 3 and 5 columns and increased 1, 2 and 3 columns bounds (six tables in total). The 3-dimensional (i.e. 3-column) tables have one categorical and two numerical input columns; the 5-dimensional tables have two categorical and three numerical input columns. These generated tables do not contain overlapping rules. We inject noise into the table the same way as we do in overlapping rules case (see the previous subsection). Only different is that we do not select one column each time, but in some cases, we select two columns or three columns and inject noise into these selected input columns.

We simplified each generated table and measured execution times averaged over 5 runs on a single core of a 64-bit 2.6 GHz Intel Core i5-3230M processor with 4GB of RAM. The results are shown in Table 15. In this table, the execution time includes rule splitting and rule merging times. Execution times for our algorithm are under 6 seconds, except for the 7-columns table with 3-columns increased. The execution times for our approach is better than Pollack approach in every experiment. The execution times with our algorithm are all below one minute than Pollack approach execution times are all more than one minute. Our algorithm is faster because our algorithm will find connected rules and because of that we have fewer rules to split and fewer rules to check if they can merge. Also, our merge algorithm is faster. Pollack algorithm will produce a huge amount of rules after splitting and merging algorithm has to compare a lot of rules. We can also optimize Pollack algorithm with first finding the connected rules and then applying splitting and merging rules to these rules, this will make the algorithm much faster. The last row of Table 15 represents the execution time of simplification where all possible rule column order was used to merge rules together and the best solution was outputted. For the merging part, we used our implemented merging algorithm. This approach is slower than our single column order approach, but it is faster than Pollack approach. This approach heavily depends on the number of dimensions, the more dimensions the slower the approach is. This approach does not suit for DMN tables where are many dimensions.

	3 COLUMNS			5 COLUMNS		
#Columns modified	1	2	3	1	2	3
our approach	526ms	1 128ms	5 720ms	345ms	550ms	30 073ms
Pollack approach	14min	12min	15min	58min	51min	565min
All combination	1 234ms	2 086ms	10 262ms	8 527ms	13 515ms	689 323ms

Table 15: Execution times

Table 16 shows the number of rules after simplification by our approach vs.

Pollack approach [13]. Our algorithm produces fewer rules than Pollack approach. Our algorithm will output more than 50 rules less than Pollack approach. We see that trying every possible dimension order gives the best solution, the lowest number of rules. We see that our approach with single dimension order is not so far behind from all dimension combination, the differences are less than 10 rules.

	3 COLUMNS			5 COLUMNS		
#Columns modified	1	2	3	1	2	3
our approach	104	107	96	105	104	119
Pollack approach	178	178	158	165	163	206
All combination	97	101	90	102	100	110

Table 16: Number of rules after optimization

6 Conclusion

To summarize, the contributions of this thesis are:

1. An approach to interpret a DMN decision table (with N input columns) as a set of hyper-rectangles in an N-dimensional space.
2. Sweep-line algorithms for detecting overlapping and missing rules in DMN decision tables.
3. A sweep-line algorithm to simplify a decision table by merging adjacent rules.

The proposed algorithms have been implemented on top of the dmn-js DMN open-source toolkit. A deployed instance of the tool implementation is available at `http://dmn.cs.ut.ee` and the source code can be found at `http://github.com/ulaurson/dmn-js`

Based on the tool implementation, we have conducted empirical evaluations to compare the proposed algorithms with respect to existing approaches (in particular the one implemented in Signavio) in terms of scalability, conciseness of the feedback provided to users (in the case of overlapping and missing rules) and compactness of the simplified tables. The results show that the proposed approach consistently outperforms existing ones at the expense of some performance overhead.

A paper describing the algorithms for detection of missing and overlapping rules and their implementation has been accepted at the BPM conference [2].

The contributions of this thesis could be further extended in the following directions: One direction for future work is to encompass other aspects of the DMN standard, such as the concept of Decision Requirements Graphs (DRGs), this allows multiple decision tables to be linked together in various ways. The current proposal focuses on analyzing DMN tables where the expressions are written in S-FEEL (i.e., expressions of the form attribute-operator-literal), future works will focus on supporting more complex types of expressions. Also, to make the splitting part of simplification smarter, so it will have to merge less rule in merge part. Have to analyze more rule merging part, particularly when it better to merge adjacent rules and when not. So, we get even fewer rules than with our current simplification algorithm. Finally, analyze more deeply the splitting and merging approach and look if there is a possibility to combine splitting and merging part together, so both are performed in a single pass.

References

[1] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Scalable sweeping-based spatial join. In *VLDB*, 1998.

[2] Diego Calvanese, Marlon Dumas, Ülari Laurson, Fabrizio Maria Maggi, Marco Montali, and Irene Teinemaa. Semantics and analysis of DMN decision tables. *CoRR*, abs/1603.07466, 2016.

[3] Victor J. Dielissen and Anne Kaldewaij. Rectangular partition is polynomial in two dimensions but np-complete in three. *Inf. Process. Lett.*, 38(1):1–6, 1991.

[4] David Eppstein. Graph-theoretic solutions to computational geometry problems. *CoRR*, abs/0908.3916, 2009.

[5] Leonard A. Ferrari, P. V. Sankar, and Jack Sklansky. Minimal rectangular partitions of digitized blobs. *Computer Vision, Graphics, and Image Processing*, 28(1):58–71, 1984.

[6] Christopher D. Godsil and Gordon F. Royle. *Algebraic Graph Theory*. Graduate texts in mathematics. Springer, 2001.

[7] Mark A. Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[8] Rattikorn Hewett and John H. Leuchner. Restructuring decision tables for elucidation of knowledge. *Data Knowl. Eng.*, 46(3):271–290, 2003.

[9] Douglas N Hoover and Zewei Chen. Tablewise, a decision table tool. In *Computer Assurance, 1995. COMPASS'95. Systems Integrity, Software Safety and Process Security. Proceedings of the Tenth Annual Conference on*, pages 97–108. IEEE, 1995.

[10] J Mark Keil. Polygon decomposition. *Handbook of Computational Geometry*, 2:491–518, 2000.

[11] Object Management Group. Decision Model and Notation (DMN) 1.0, 2015.

[12] Solomon L. Pollack. Analysis of the decision rules in decision tables. Memorandum RM-3669-PR, RAND Corporation, 1963.

[13] Solomon L Pollack, Harry T Hicks, and William J Harrison. Decision tables theory and practice. 1971.

[14] Udo W. Pooch. Translation of decision tables. *Comp. Surv.*, 6(2):125–151, 1974.

[15] Robert Sedgewick. *Algorithms in C.* Addison-Wesley, 1990.

[16] Robert Sedgewick and Kevin Wayne. Geometric algorithms. `https://www.cs.princeton.edu/~rs/AlgsDS07/17GeometricSearch.pdf`, 2007.

[17] Keith Shwayder. Combining decision rules in a decision table. *Commun. ACM*, 18(8):476–480, 1975.

[18] Valeriu Soltan and Alexei Gorpinevich. Minimum dissection of rectilinear polygon with arbitrary holes into rectangles. In *Proceedings of the Eighth Annual Symposium on Computational Geometry, Berlin, Germany, June 10-12, 1992*, pages 296–302, 1992.

[19] Robert Endre Tarjan and Anthony E. Trojanowski. Finding a maximum independent set. *SIAM J. Comput.*, 6(3):537–546, 1977.

[20] Jan Vanthienen and Elke Dries. Illustration of a decision table tool for specifying and implementing knowledge based systems. *International Journal on Artificial Intelligence Tools*, 3(2):267–288, 1994.

[21] Jan Vanthienen, Christophe Mues, and Ann Aerts. An illustration of verification and validation in the modelling phase of KBS development. *Data Knowl. Eng.*, 27(3):337–352, 1998.

[22] Hongyuan Zha, Xiaofeng He, Chris H. Q. Ding, Ming Gu, and Horst D. Simon. Bipartite graph partitioning and data clustering. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management, Atlanta, Georgia, USA, November 5-10, 2001*, pages 25–32, 2001.