Semantics and Analysis of DMN Decision Tables

Diego Calvanese[1], Marlon Dumas[2], Ülari Laurson[2],
Fabrizio M. Maggi[2], Marco Montali[1], and Irene Teinemaa[2]

[1]Free University of Bozen-Bolzano, Italy
[2]University of Tartu, Estonia

Abstract. The Decision Model and Notation (DMN) is a standard notation to capture decision logic in business applications in general and business processes in particular. A central construct in DMN is that of a decision table. The increasing use of DMN decision tables to capture critical business knowledge raises the need to support analysis tasks on these tables such as correctness and completeness checking. This paper provides a formal semantics for DMN tables, a formal definition of key analysis tasks and scalable algorithms to tackle two such tasks, i.e., detection of overlapping rules and of missing rules. The algorithms are based on a geometric interpretation of decision tables that can be used to support other analysis tasks by tapping into geometric algorithms. The algorithms have been implemented in an open-source DMN editor and tested on large decision tables derived from a credit lending dataset.

1 Introduction

Business process models often incorporate decision logic of varying complexity, typically via conditional expressions attached either to outgoing flows of decision gateways or to conditional events. The need to separate this decision logic from the control-flow logic [2] and to capture it at a higher level of abstraction has motivated the emergence of the Decision Model and Notation (DMN) [8]. A central construct of DMN is that of a decision table, which stems from the notion of decision table proposed in the context of program decision logic specification in the 1960s [10]. A DMN decision table consists of columns representing the inputs and outputs of a decision, and rows denoting rules. Each rule is a conjunction of basic expressions captured in an expression language known as S-FEEL (Simplified Friendly Enough Expression Language).

The use of DMN decision tables as a specification vehicle for critical business decisions raises the question of ensuring the correctness of these tables, in particular the detection of inconsistent or incomplete DMN decision tables. Indeed, detecting errors in DMN tables at specification time may prevent costly defects down the road during business process implementation and execution.

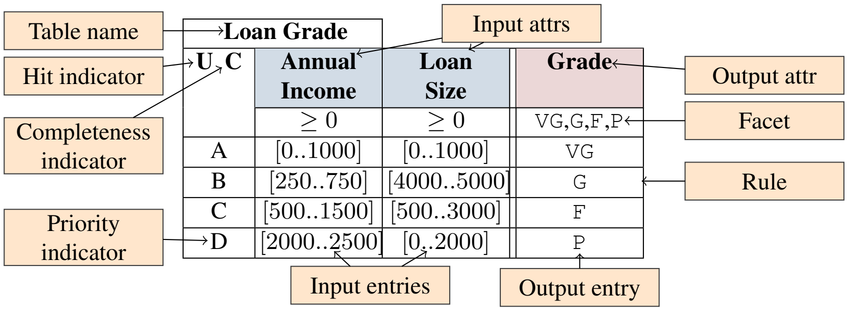
This paper provides a foundation for analyzing the correctness of DMN tables. The contributions of the paper are: (i) a formal semantics of DMN tables; (ii) a formalization of correctness criteria for DMN tables; and (iii) scalable algorithms for two basic correctness checking tasks over DMN tables, i.e., detection of overlapping rules and detection of missing rules (i.e., incompleteness). The latter algorithms are based on a novel geometric interpretation of DMN tables, wherein each rule in a table is mapped to an iso-oriented hyper-rectangle in an N-dimensional space (where N is 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 computing the difference between the N-dimensional universe defined by the domains of the N columns of a DMN table, and the set of hyper-rectangles induced by its rules. Based on this geometric interpretation and inspired by sweep-based spatial join algorithms [1], the paper presents scalable algorithms for these two analysis tasks. The algorithms have been implemented atop the dmn-js editor and evaluated over decision tables of varying sizes derived from a credit lending dataset.

The rest of the paper is structured as follows. Section 2 introduces DMN and discusses related work. Section 3 presents the formalization of DMN tables and their associated correctness criteria. Section 4 presents the algorithms for correctness analysis while Section 5 discusses their empirical evaluation. Finally, Section 6 summarizes the contributions and outlines future work directions.

2 Background and Related Work

2.1 Overview of DMN Decision Tables

A DMN table consists of columns corresponding to input or output attributes, and rows corresponding to rules. Each column has a type (e.g., a string, a number, or a date), and optionally to a more specific domain of possible values, which we hereby call a *facet*. Each row has an identifier, one expression for each input column (a.k.a. the *input entries*), and one specific value for each output column (the *output entries*). For example, Table 1 shows a DMN table with two input columns, one output column and four rules.



Loan Grade

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

Table name
Hit indicator
Completeness indicator
Priority indicator
Input attrs
Output attr
Facet
Rule
Input entries
Output entry

Table 1: Sample decision table with its constitutive elements

Given an input configuration consisting of a vector of values (one entry per column), if every input entry of a row holds true for this input vector, then the vector *matches* the row and the output entries of the row are evaluated. For example, vector $\langle 500, 4230\rangle$ matches rule B in Table 1, thus yielding G in the output configuration. To specify how output configurations are computed from input ones, a DMN table has a *hit indicator* and a *completeness indicator*. The hit indicator specifies whether only one or multiple rows of the table may match a given input, and if multiple rules match an input, how should the output be computed. The completeness indicator specifies whether every input must match at least one rule or potentially none. If an input configuration matches multiple rules, this may contradict the hit policy. Similarly, if no rule matches an input configuration, this may contradict the completeness indicator. The former contradiction is called *overlapping rules* while the latter is called *missing rule*.

2.2 Analysis of DMN Decision Tables

The need to analyze decision tables from the perspective of completeness (i.e., detecting missing rules) as well as consistency and non-redundancy (i.e., detecting overlapping rules) is widely recognized [3]. These two analysis tasks have been tackled using rough sets [9]. However, this approach requires that the domains of the input attributes are boolean or categorical. Numerical attributes need to be previously discretized into intervals and in such a way that no two intervals over any column overlap. For example, approaches based on rough sets cannot handle situations where multiple overlapping intervals appear along the same attribute (e.g., [151..300] and [200..250]). Instead, the table needs to be expanded so that these intervals do not overlap (e.g., intervals [151..300] and [200..250] need to be broken down into [151..200], [201..250] and [251..300]) and this expansion can in the worst case increase the size of the table exponentially.

Prologa [11,12] is a tool for modeling and executing classical decision tables. It supports the construction of decision tables in a way that prevents overlapping or missing rules. It also supports the simplification of decision tables via rule merging: two rules are merged when all but one of their input entries are identical, and their output entries are also identical. However, Prologa has the same intrinsic limitation as the rough set approach: it requires columns to have categorical domains. Numerical domains need to be broken down into elementary non-overlapping intervals as explained above. The same limitations hold in other techniques for detecting overlapping and missing rules [7,13] and algorithms for simplifying decision tables [6]. In other words, while the verification and simplification of decision tables with discrete or discretized domains has received much attention, the case where the columns have both discrete domains and numeric domains with arbitrary interval expressions has not been considered in the literature.

Signavio's DMN editor[1] detects overlapping and missing rules without imposing discretization of numeric domains. However, the employed techniques are undisclosed and no empirical evaluation thereof has been reported. Also, 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 Section 5.

OpenRules[2] uses constraint satisfaction techniques to analyze business rules, in particular rules encoded in decision tables. While using a general solver to analyze decision tables is an option (e.g., an SMT solver such as Z3 [4]), this approach leads to a boolean output (is the set of rules satisfiable?), and cannot natively highlight specific sets of rules that need to be added to a table (missing rules), nor specific overlaps between pairs of rules that need to be resolved.

[1] http://www.signavio.com

[2] http://openrules.com/

3 Formalization

In this section, we provide a formalization of DMN decision tables, unambiguously defining their input/output semantics, and at the same time introducing several analysis tasks focused on correctness checking. As a concrete specification language for input entries, we consider the S-FEEL language introduced in the DMN standard itself.

Our formalization is based on classical predicate logic extended with data types, which are needed to capture conditions that employ domain-specific predicates such as comparisons interpreted over the total order of natural numbers. Such formalization is important per sè, as it defines a clear, unambiguous semantics of decision tables, and also as an interlingua supporting the comparison of different analysis techniques.

3.1 Data Types and S-FEEL Conditions

We first introduce the building blocks of decision tables, i.e., the types of the modeled attributes, and conditions over such types expressed using the S-FEEL language. A data type $\mathcal{T}$ is a tuple $\langle \Delta_\mathcal{T}, \Sigma_\mathcal{T} \rangle$, where $\Delta_\mathcal{T}$ is an *object domain*, and $\Sigma_\mathcal{T} = \Sigma^P_\mathcal{T} \uplus \Sigma^F_\mathcal{T}$ is a *signature*, constituted by a set $\Sigma^P_\mathcal{T}$ of *predicate symbols*, and a set $\Sigma^F_\mathcal{T}$ of *function symbols* (disjoint from $\Sigma^P_\mathcal{T}$). Each predicate symbol $R \in \Sigma^P_\mathcal{T}$ comes with its own arity n, and with an n-ary predicate $R^\mathcal{T} \subseteq \Delta^n_\mathcal{T}$ that rigidly defines its semantics. Each function symbol $f \in \Sigma^F_\mathcal{T}$ comes with its own arity m, and with a function $\Delta^m_\mathcal{T} \to \Delta_\mathcal{T}$ that defines its semantics. To make the arity explicit in predicate and function symbols, we use the standard notation R/n and f/m. As usual, we assume that every data type is equipped *equality* as a predefined, binary predicate interpreted as the identity on the underlying domain. Hence, we will not explicitly mention equality in the signatures of data types. In the following, we show some of the S-FEEL data types[3]:

- $\mathcal{T}_\mathbb{S} = \langle \mathbb{S}, \emptyset, \emptyset \rangle$ – strings.
- $\mathcal{T}_\mathbb{B} = \langle \{\mathsf{true}, \mathsf{false}\}, \emptyset, \emptyset \rangle$ – boolean attributes.
- $\mathcal{T}_\mathbb{Z} = \langle \mathbb{Z}, \{\mathbf{0}/0, \mathbf{1}/0, </2, >/2\}, \{+/2, -/2, \cdot/2, \div/2\} \rangle$ – integer numbers equipped with the usual comparison predicates and binary operations;
- $\mathcal{T}_\mathbb{R}$ (defined as $\mathcal{T}_\mathbb{Z}$ by replacing the domain $\mathbb{Z}$ with $\mathbb{R}$, and by reinterpreting all predicates and functions accordingly) – real numbers equipped with the usual comparison predicates and binary operations.

The set of all such types is denoted by $\mathfrak{T}$. Since decision tables do not support conditions that combine multiple data types, we can assume that the *object domains of all types in* $\mathfrak{T}$ *are pairwise disjoint*.

S-FEEL allows one to formulate conditions over types. These conditions constitute the basic building blocks for facets and rules, which in turn are the core of decision tables. The syntax of an *(S-FEEL) condition* $\mathcal{Q}$ over type is:

$$\begin{aligned}
\mathcal{Q} &::= \text{“}-\text{”} \mid \mathit{Term} \mid \text{“}\texttt{not}(\text{” } \mathit{Term} \text{ “})\text{”} \mid \mathit{Comparison} \mid \mathit{Interval} \mid \mathcal{Q}_1,\mathcal{Q}_2 \\
\mathit{Comparison} &::= \mathit{COp}\ \mathit{Term} \\
\mathit{COp} &::= \text{“}=\text{”} \mid \text{“}<\text{”} \mid \text{“}>\text{”} \mid \text{“}\leq\text{”} \mid \text{“}\geq\text{”} \\
\mathit{Interval} &::= (\text{“(”} \mid \text{“[”})\ \mathit{Term}_1 \text{ “..” } \mathit{Term}_2\ (\text{“)”} \mid \text{“]”}) \\
\mathit{Term} &::= v \mid f(\mathit{Term}_1, \ldots, \mathit{Term}_m)
\end{aligned}$$

where v is an object and f is an m-ary function.

[3] Date/time data types are also supported but can be considered as simple numeric attributes.

S-FEEL supports the following conditions on a given data type $\mathcal{T} = \langle \Delta_\mathcal{T}, \Sigma_\mathcal{T} \rangle$: *(i)* "$-$" indicates *any value*, i.e., it holds for every object in $\Delta_\mathcal{T}$. *(ii)* "$= \ Term$" indicates a *matching expression*, which holds for the object in $\Delta_\mathcal{T}$ that corresponds to the result denoted by term $Term$. A term, in turn, corresponds either to a specific object in $\Delta_\mathcal{T}$, or to the recursive application of an m-ary function in $\Sigma_\mathcal{T}$ to m terms. It is worth noting that, in the actual S-FEEL standard, the symbol " $=$ " is usually omitted, that is, when resolving the scope symbol $\mathcal{Q}$, $Term$ is interpreted as a shortcut notation for "$= \ Term$". *(iii) $Comparison$* is only applicable when $\mathcal{T}$ is a numeric data type, and indicates a *comparison condition*, which holds for all objects that are related via the employed comparison predicate to the object resulting from expression $Term$. *(iv) $Interval$* is only applicable when $\mathcal{T}$ is numeric, and allows the modeler to capture membership conditions that tests whether an input object belongs to the modeled interval. *(v)* "$\mathcal{Q}_1$,$\mathcal{Q}_2$" indicates an *alternative condition*, which holds whenever one of the two conditions $\mathcal{Q}_1$ and $\mathcal{Q}_2$ holds.

Example 1. The fact that a risk category is either high, medium or low can be expressed by the following condition over $\mathcal{T}_\mathbb{S}$: "`high`,`medium`,`low`". By using $\mathcal{T}_\mathbb{Z}$ to denote the age of persons (in years), the group of people that are underage or old (i.e., having at least 70 years) is captured by condition "[0..18], ≥ 70". ■

3.2 Decision Tables

We are now in the position of defining DMN decision tables. See Table 1 for a reference example. A *decision table $\mathcal{D}$* is a tuple $\langle T, I, O, \mathsf{Type}, \mathsf{Facet}, R, \mathsf{Priority}, C, H \rangle$, where:

- T is the *table name*.
- I and O are disjoint, finite sets of *input* and *output attributes* (represented as strings).[4]
- $\mathsf{Type} : I \uplus O \rightarrow \mathfrak{T}$ is a *typing function* that associates each input/output attribute to its corresponding data type.
- Facet is a *facet function* that associates each input/output attribute $\mathbf{a} \in I \uplus O$ to a condition over $\mathsf{Type}(\mathbf{a})$, defining the *acceptable objects* for that attribute. Facet functions are depicted as "optional lists of values" in Table 1.
- R is a finite set of *rules* $\{r_1, \ldots, r_p\}$. Each rule r_k is a pair $\langle \mathsf{If}_k, \mathsf{Then}_k \rangle$, where If_k is an *input entry function* that associates each input attribute $\mathbf{a^{in}} \in I$ to a condition over $\mathsf{Type}(\mathbf{a^{in}})$, and Then_k is an *output entry function* that associates each output attribute $\mathbf{a^{out}} \in O$ an object in $\mathsf{Type}(\mathbf{a^{out}})$.
- $\mathsf{Priority} : R \rightarrow \{1, \ldots, |R|\}$ is a *priority function* injectively mapping rules in R to a corresponding rule number defining its priority. If no priority is explicitly given, in accordance with the standard we assume that the priority is implicitly defined by the graphical ordering in which rule entries appear inside the decision table.
- $C \in \{\mathtt{c}, \mathtt{i}\}$ is the *completeness indicator*, where $\mathtt{c}$ is the default value and stands for *complete* table, while $\mathtt{i}$ stands for *incomplete* table.

[4] These are called "expressions" in the DMN standard, but we prefer the term "attribute" as it is less ambiguous.

- $H \in \{\mathtt{u}, \mathtt{a}, \mathtt{p}, \mathtt{f}\}$ is the *(single) hit indicator* defining the policy for the rule application, where: *(i)* u is the default value and stands for *unique hit policy*, *(ii)* $H = \mathtt{a}$ stands for *any hit policy*, *(iii)* $H = \mathtt{p}$ stands for *priority hit policy*, and *(iv)* $H = \mathtt{f}$ stands for *first hit policy*.

We now informally review the intuitive semantics of rules and of completeness/hit indicators in DMN, moving to the formalization in Section 3.3.

Rule semantics. Intuitively, rules follow the standard "if-then" interpretation. Rules are matched against *input configurations*, which map the input attributes to objects in such a way that each object *(i)* belongs to the type of the corresponding input attribute, and *(ii)* satisfies the corresponding facet. If, for every input attribute, the assigned object satisfies the condition imposed by the rule on that type, then the rule *triggers*, and bounds the output attributes to the actual objects mentioned by the rule.

Example 2. Consider the decision table in Table 1. The input configuration where $\mathbf{Income}$ is 500 and $\mathbf{Loan}$ is 4230, triggers rule B. ■

Completeness indicator. When the table is declared to be complete, the intention is that every possible input configuration must trigger at least one rule. Incomplete tables, instead, have input configurations with no matching rule.

Hit policies. Hit policies specify how to handle the case where multiple rules are triggered by an input configuration. In particular:

- "Unique hit" indicates that at most one rule can be triggered by a given input configuration, thus avoiding the need of handling how to compute the output objects in the case of multiple triggered rules.
- "Any hit" indicates that when multiple rules are triggered, they must agree on the output objects, thus guaranteeing that the output is unambiguous.
- "Priority hit" indicates that whenever multiple rules trigger, then the output is unambiguously computed by only considering the contribution of the triggered rule that has highest priority.
- "First hit" can be understood as a variant of the priority hit, in which priority is implicitly obtained from the ordering in which rules appear in the decision table. Hence, this case is subsumed by that of priority hit.
- "Collect" implies that multiple rules can match an input configuration and when this is the case, all matching rules are fired the resulting output configurations are aggregated. Aggregation is orthogonal to correctness checking, and thus we leave the "Collect" policy outside the scope of the formalization below.

3.3 Formalization of Rule Semantics and of Analysis Tasks

We first define how conditions map to corresponding formulae. Since each condition is applied to a single input attribute, the corresponding formula has a single free variable corresponding to that attribute. Given a condition $\mathcal{Q}$ over type $\mathcal{T} = \langle \Delta_{\mathcal{T}}, \Sigma_{\mathcal{T}} \rangle$, the *condition formula for* $\mathcal{Q}$, written $\Phi_{\mathcal{Q}}$, is a formula using predicates/functions in $\Sigma_{\mathcal{T}}$ and objects from $\Delta_{\mathcal{T}}$, and possibly mentioning a single free variable, constructed as follows:

$$
\mathit{\Phi}_{\mathcal{Q}} \triangleq \begin{cases}
\mathit{true} & \text{if } \mathcal{Q} = \text{``}-\text{''} \\
\neg \mathit{\Phi}_{\mathit{Term}} & \text{if } \mathcal{Q} = \text{``}\texttt{not}(\mathit{Term})\text{''} \\
x = \mathit{Term} & \text{if } \mathcal{Q} = \mathit{Term} \\
x\ \mathit{COp}\ \mathit{Term} & \text{if } \mathcal{Q} = \text{``}\mathit{COp}\ \mathit{Term}\text{''and } \mathit{COp} \in \{<,>,\leq,\geq\} \\
x > \mathit{\Phi}_{\mathit{Term_1}} \land x < \mathit{\Phi}_{\mathit{Term_2}} & \text{if } \mathcal{Q} = \text{``}(\mathit{Term}_1..\mathit{Term}_2)\text{''} \\
x > \mathit{\Phi}_{\mathit{Term_1}} \land x \leq \mathit{\Phi}_{\mathit{Term_2}} & \text{if } \mathcal{Q} = \text{``}(\mathit{Term}_1..\mathit{Term}_2]\text{''} \\
x \geq \mathit{\Phi}_{\mathit{Term_1}} \land x < \mathit{\Phi}_{\mathit{Term_2}} & \text{if } \mathcal{Q} = \text{``}[\mathit{Term}_1..\mathit{Term}_2)\text{''} \\
x \geq \mathit{\Phi}_{\mathit{Term_1}} \land x \leq \mathit{\Phi}_{\mathit{Term_2}} & \text{if } \mathcal{Q} = \text{``}[\mathit{Term}_1..\mathit{Term}_2]\text{''} \\
\mathit{\Phi}_{\mathcal{Q}_1} x \lor \mathit{\Phi}_{\mathcal{Q}_2} x & \text{if } \mathcal{Q} = \text{``}\mathcal{Q}_1,\mathcal{Q}_2\text{''}
\end{cases}
$$

As usual, we also use notation $\mathit{\Phi}_{\mathcal{Q}}(x)$ to explicitly mention the free variable of the condition formula.

Example 3. Consider the S-FEEL conditions in Example 1. The condition over the risk category is $\mathit{Risk} = \texttt{high} \lor \mathit{Risk} = \texttt{medium} \lor \mathit{Risk} = \texttt{low}$. The condition formula person ages is instead: $(\mathit{Age} \geq 0 \land \mathit{Age} \leq 18) \lor \mathit{Age} \geq 70$. ■

With this notion at hand, we now formalize the notions of correctness of rule specifications, semantics of rules, and semantics of completeness and hit indicators. These notions are building blocks for an overall notion of *table correctness*.

Let $\mathcal{D} = \langle T, I, O, \mathsf{Type}, \mathsf{Facet}, R, \mathsf{Priority}, C, H\rangle$ be a decision table with m input attributes $I = \{\mathbf{a_1}, \ldots, \mathbf{a_m}\}$, n output attributes $O = \{\mathbf{b_1}, \ldots, \mathbf{b_n}\}$, and p rules $R = \{r_1, \ldots, r_p\}$. We use variables $x_1, \ldots, x_m$ for objects matching the input attributes, and variables $y_1, \ldots, y_n$ for those matching the output attributes.

Facet correctness. We first consider the *Facet correctness* of $\mathcal{D}$, which intuitively amounts to check whether all the mentioned input conditions and output objects are compatible with their corresponding attribute facets. Given an attribute $\mathbf{a} \in I \cup O$ and a corresponding input variable x, we can identify whether *a condition* $\mathcal{Q}$ *over* $\mathbf{a}$ *is compatible with* $\mathbf{a}$, i.e., whether the condition is specified in such a way that can potentially trigger, or is instead contradictory with the facet attached to $\mathbf{a}$:

$$\mathit{Compatible}_{\mathbf{a}}^{\mathcal{Q}} \triangleq \exists x.\mathit{\Phi}_{\mathsf{Facet}(\mathbf{a})}(x) \land \mathit{\Phi}_{\mathcal{Q}}(x)$$

Rule semantics. A rule $r = \langle \mathsf{If}, \mathsf{Then}\rangle \in R$ is *triggered by* a configuration $x_1, \ldots, x_m$ of input objects whenever each such object matches with the corresponding input condition:

$$\mathit{TriggeredBy}_r(x_1, \ldots, x_m) \triangleq \bigwedge_{i \in \{1,\ldots,m\}} \mathit{Matches}_{\mathbf{a_i}}^{\mathsf{If}(\mathbf{a_i})}(x_i)$$

Two configurations $\vec{x}$ and $y_1, \ldots, y_n$ of input and output objects are *input-output related* by a rule $r = \langle \mathsf{If}, \mathsf{Then}\rangle \in R$ if the rule is triggered by the input configuration, and binds the output as specified by the output configuration:

$$\mathit{IORel}_r(\vec{x}, y_1, \ldots, y_n) \triangleq \mathit{TriggeredBy}_r(\vec{x}) \land \bigwedge_{j \in \{1,\ldots,n\}} \mathit{Matches}_{\mathbf{b_j}}^{\mathsf{Then}(\mathbf{b_j})}(y_j)$$

Completeness. When declaring that a table is (in)complete, there is no guarantee that the specified rules guarantee this property. To check whether this is indeed the case, we introduce a formula that holds whenever each possible input configuration triggers at least one rule:

$$\mathit{Complete}_{\mathcal{D}} \triangleq \forall x_1, \ldots, x_m. \bigvee_{k \in \{1,\ldots,p\}} \mathit{TriggeredBy}_{r_k}(x_1, \ldots, x_m)$$

Hit policies. We start with the unique hit policy, which requires that each input configuration triggers at most one rule. This can be formalized as follows:

$$\mathit{Unique}_{\mathcal{D}} \triangleq \forall \vec{x}. \bigwedge_{i \in \{1,\ldots,p\}} \left(\mathit{TriggeredBy}_{r_i}(\vec{x}) \rightarrow \bigwedge_{j \in \{1,\ldots,p\} \setminus \{i\}} \neg \mathit{TriggeredBy}_{r_j}(\vec{x}) \right)$$

We then continue with the any hit policy. Here multiple rules may be triggered by the same input configuration, but if so, then they must agree on the output. This can be formalized as follows:

$$\begin{aligned} \mathit{AgreesOnOutput}_{\mathcal{D}} \triangleq \bigwedge_{i,j \in \{1,\ldots,p\}, i \neq j} \big(\forall \vec{x} \forall \vec{y}.\ & \mathit{TriggeredBy}_{r_i}(\vec{x}) \\ & \wedge \mathit{TriggeredBy}_{r_j}(\vec{x}) \rightarrow \mathit{IORel}_{r_i}(\vec{x}, \vec{y}) \\ & \qquad\qquad\qquad\qquad \wedge \mathit{IORel}_{r_j}(\vec{x}, \vec{y})\big) \end{aligned}$$

We now consider the case of priority hit policy. This requires to reformulate the rule semantics, so as to consider the whole decision table and the priority of the rules. In particular, with this hit policy a rule $r \in R$ is *triggered with priority* by an input configuration $\vec{x}$ if it is triggered by $\vec{x}$ in the sense specified above, and no rule of higher priority is triggered by the same input $\vec{x}$:

$$\mathit{TriggeredWithPriorityBy}_r(\vec{x}) \triangleq \mathit{TriggeredBy}_r(\vec{x}) \wedge \bigwedge_{r_h \in \{r' \mid r' \in R \text{ and } \mathsf{Priority}(r') > \mathsf{Priority}(r)\}} \neg \mathit{TriggeredBy}_{r'}(\vec{x})$$

Finally, we observe that the priority hit policy may create a situation in which some rules are never triggered. This happens when other rules of higher priority have more general input conditions. We formalize this notion by introducing a formula dedicated to check when a rule $r_1 \in R$ *is masked* by another rule $r_2 \in R$:

$$\mathit{MaskedBy}_{r_1}^{r_2} \triangleq \mathsf{Priority}(r_2) > \mathsf{Priority}(r_1) \wedge \forall \vec{x}. \mathit{TriggeredBy}_{r_1}(\vec{x}) \rightarrow \mathit{TriggeredBy}_{r_2}(\vec{x})$$

Correctness formula. We now combine the previously defined formulae into a single formula that captures the overall correctness of a decision table.

We say that $\mathcal{D}$ is *correct* if the following conditions hold:

1. Every table cell, i.e., every input condition or output object, is legal for the corresponding attribute (considering the attribute type and facet).
2. The completeness indicator corresponds to c iff the table is indeed complete.
3. The rules are compatible with the hit policy indicator:
   (a) if the hit policy is u, each input configuration triggers at most one rule;
   (b) if the hit policy is a, all overlapping rules (i.e., rules that could simultaneously trigger) have the same output;

(c) if the hit policy is p, all rules are "useful", i.e., no rule is masked by a rule with higher priority.

Based on the previously introduced formulae, we formalize correctness as:

$$\begin{aligned} \mathit{Correct}_{\mathcal{D}} \triangleq & \bigwedge_{\langle \mathsf{If},\mathsf{Then}\rangle \in R} \left(\bigwedge_{\mathbf{a}\in I} \mathit{Compatible}_{\mathbf{a}}^{\mathsf{If}(\mathbf{a})} \wedge \bigwedge_{\mathbf{b}\in O} \mathit{Compatible}_{\mathbf{b}}^{\mathsf{Then}(\mathbf{b})} \right) \\ & \wedge \big((C = \mathtt{c}) \leftrightarrow \mathit{Complete}_{\mathcal{D}}\big) \\ & \wedge \big((H = \mathtt{u}) \rightarrow \mathit{Unique}_{\mathcal{D}}\big) \\ & \wedge \big((H = \mathtt{a}) \rightarrow \mathit{AgreesOnOutput}_{\mathcal{D}}\big) \\ & \wedge \Big((H = \mathtt{p}) \rightarrow \bigwedge_{r_1,r_2\in R} \neg \mathit{MaskedBy}_{r_1}^{r_2}\Big) \end{aligned}$$

Global input-output formula. We combine the previously defined formulae into a single formula that captures the overall input-output relation induced by $\mathcal{D}$. This is done by exploiting the notion of input-output related configurations by a rule, so as to cover the entire table. Specifically we say that an input configuration $\vec{x}$ and an output configuration $\vec{y}$ are *input-output related* by $\mathcal{D}$ if:

1. the hit policy is either u or a, and there exists a rule that relates $\vec{x}$ to $\vec{y}$ (in the case of any hit policy, there could be many, but they establish the same input-output relation, so it is sufficient to pick one of them);
2. the hit policy is p, and there exists a rule relating $\vec{x}$ to $\vec{y}$ without any other rule of higher priority that is triggered by $\vec{x}$ (if such a rule exists, then it is such rule that has to be selected to relate input-output).

This is formalized as follows:

$$\begin{aligned} \mathit{IORel}_{\mathcal{D}}(\vec{x},\vec{y}) \triangleq & \Big((H = \mathtt{u} \vee H = \mathtt{a}) \rightarrow \bigvee_{r\in R} \mathit{IORel}_r(\vec{x},\vec{y})\Big) \\ & \wedge \left(\begin{aligned} (H = \mathtt{p}) \rightarrow \textstyle\bigvee_{r=\langle \mathsf{If},\mathsf{Then}\rangle\in R} & \mathit{TriggeredWithPriorityBy}_r(\vec{x}) \\ & \wedge \textstyle\bigwedge_{j\in\{1,\ldots,n\}} \mathit{Matches}_{\mathbf{b_j}}^{\mathsf{Then}(\mathbf{b_j})}(y_j) \end{aligned} \right) \end{aligned}$$

4 Algorithms

We now introduce algorithms to handle the two main analysis tasks introduced in the previous section: detecting overlapping rules and (in)completeness. The proposed algorithms rely on a geometric interpretation of a DMN table. Every rule in a table is seen as an iso-oriented hyper-rectangle in an N-dimensional space (where N is a number of columns). Indeed, 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 column, 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 each value of the column's domain to a disjoint interval – 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 defines a hyper-rectangle, or potentially multiple hyper-rectangles in the case of a multi-valued categorical input entry (e.g., {"Refinancing", "Car leasing"}). The hyper-rectangles are iso-oriented because only constraints of the form "attribute operator literal" are allowed in S-FEEL and such constraints define iso-oriented lines or segments.

For example, the geometric interpretation of Table 1 is shown in Fig. 1. [5] The two dimensions, x and y, represent the two input columns (*Annual income* and *Loan size*) respectively. The table contains 4 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 1.

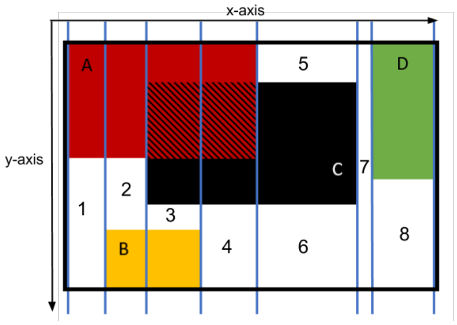


Fig. 1. Geometric representation of the DMN table shown in Table 1

The algorithms are presented for numeric columns. Minor adaptations (not discussed here) allow these algorithms to handle categorical columns as well.

4.1 Finding Overlapping Rules

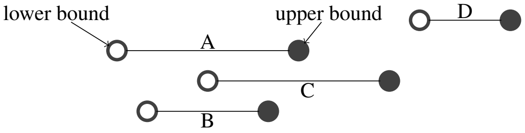
Algorithm 1 finds overlapping rules in a DMN table. This algorithm is an extension of line-sweep algorithm for two-dimensional spatial joins proposed in [1]. The idea of this latter algorithm is to pick one dimension (e.g., x-axis), project all hyper-rectangles into this dimension, and then sweep an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

1. ruleList, containing all rules of the input DMN table;

[5] For simplicity, the figure is purely schematic and does not preserve the scale along the axes.

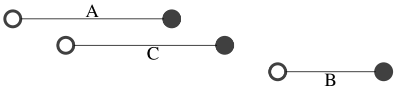
2. i, containing the index of the column under scrutiny;
3. N, representing the total number of columns;
4. OverlappingRuleList, storing the rules that overlap.

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^x_K and I^y_K respectively. All the intervals are represented in terms of upper and lower bounds. The bounds are sorted in ascending order (line 7). The algorithm iterates over the list of sorted bounds (line 8). In the case of Fig. 1, the rules projected over the x axis correspond are:



Considering the rules above, the algorithm first analyzes the lower bound of I^x_A. Therefore, I^x_A 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^x_B and I^x_B is added to $\mathcal{L}_x$. Then, the lower bound of I^x_C is processed and I^x_C is added to $\mathcal{L}_x$. Finally, the algorithm processes the upper bound of I^x_B. 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).

All the interval 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 as depicted below:



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^y_A. Therefore, I^y_A 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^y_C and adds I^y_C to $\mathcal{L}_y$. Then, the upper bound of I^y_C 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). If this is not the case, 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 14).

4.2 Finding Missing Rules

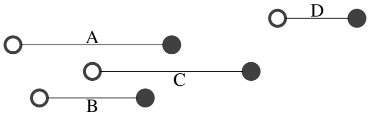
Algorithm 2 describes the procedure for finding missing rules, which is also based on the line-sweep principle. The algorithm takes as inputs 5 parameters:

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 return $overlappingRuleList$;
```

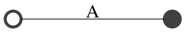
1. ruleList, containing all rules of the input DMN table;
2. missingIntervals, storing the current missing intervals;
3. i, containing the index of the column under scrutiny;
4. N, representing the total number of columns;
5. MissingRuleList, storing the missing rules.

The algorithm starts analyzing the first column of the table (axis x). Consider again the projection of the table in Fig. 1 on x:



Upper and lower bounds of each interval are sorted in ascending order (line 3). The algorithm iterates over the list of sorted bounds (line 5).

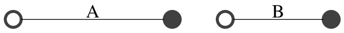
Considering the rules above, the algorithm first analyzes the lower bound of I^x_A. Therefore, I^x_A is added to an active list of intervals for the first column x, $\mathcal{L}_x$. An interval is added to the active list only if its lower bound is processed (line 16). If the upper bound of an interval is processed, the interval is removed from the list (line 18). Next, the algorithm processes the lower bound of I^x_B. Since $\mathcal{L}_x$ is not empty, I^x_B is not added to $\mathcal{L}_x$ yet (line 12). Starting from the interval $\mathcal{I}_{\mathcal{A},\mathcal{B}}$ (line 13) having the lower bound of I^x_A as lower bound and the lower bound of I^x_B as upper bound, the following column of the table is analyzed (in this case y) by invoking *findMissingRules* recursively (line 14). 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^y_A so that I^y_A 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 6). 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 6). 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 8). If this is the case the two areas are merged (line 9). If this is not the case, the area is added to a list of missing value areas (line 11). 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 exactly the same. In the example in Fig. 1, 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 previous processed bound is contiguous with the current one (line 6). 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 20).

5 Evaluation

We implemented the algorithms on top of dmn-js: an open-source rendering and editing toolkit for DMN tables.[6] In it current version, dmn-js does not support correctness verification. Our dmn-js extension with verification features can be found at

[6] https://github.com/bpmn-io/dmn-js

Algorithm 2: Procedure findMissingRules.

```
Input: $ruleList$; $missingIntervals$; $i$; $N$; $missingRuleList.$
if $i > N$ then
    define the current list of boundaries $\mathcal{L}_{x_i}$;
    $sortedListAllBoundaries$ = $ruleList$.sort($i$);
    $lastBound$ = 0;
    foreach $currentBound \in sortedListAllBoundaries$ do
        if !areContiguous($lastBound$, $currentBound$) then
            $missingIntervals[i]$ = constructInterval($lastBound$, $currentBound$);
            if $missingRuleList$.canBeMerged($missingIntervals$); then
                $missingRuleList$.merge($missingIntervals$);
            else
                $missingRuleList$.add($missingIntervals$);
        if !$\mathcal{L}_{x_i}$.isEmpty() ) then
            $missingIntervals$ [$i$] = constructInterval($lastBound$, $currentBound$);
            findMissingRules($\mathcal{L}_{x_i}$,$missingIntervals$,$i$ +1, $N$, $missingRuleList$); /* recursive
            invocation */
        if $currentBound$.isLower() then
            $\mathcal{L}_{x_i}$.put($currentBound$);
        else
            $\mathcal{L}_{x_i}$.delete($currentBound$);
        $lastBound$ = $currentBound$;
return $missingRuleList$;
```

`https://github.com/ulaurson/dmn-js` and a deployed version is available for testing at `http://dmn.cs.ut.ee`.

For the evaluation, we created decision tables from a loan dataset of LendingClub – a peer-to-peer lending marketplace.[7] 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 [5], we trained decision trees to classify the grade of each loan from a subset of the loan attributes. We then translated each trained decision tree into a DMN table by mapping each path from the root to a leaf of the tree into a rule. 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 has two categorical and five numerical input columns.

By construction, the generated tables do not contain overlapping or missing rules. To introduce missing 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

[7] `https://www.lendingclub.com/info/download-data.action`

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 2. 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 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 less rule overlaps, which in turn is due to the fact that the 5-columns tables have proportionally less categorical columns than the 3-columns ones.

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.

Table 3 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	297ms	6 475ms	24 530ms	200ms	1 621ms	5 374ms	5 715ms	6 793ms	30 736ms
missing time	160ms	611ms	1 672ms	163ms	820ms	1 942ms	2 173ms	7 029ms	18 263ms

Table 2: 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	131	447	812	110	225	378	139	227	371
	Signavio	1 226	10 920	23 115	679	3 692	8 921	23 175	22 002	62 217
#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 3: Number of reported errors of type “overlapping rules” & “missing rule”

6 Conclusion and Future Work

This paper presented a formal semantics of DMN decision tables, a notion of DMN table correctness, and algorithms that operationalize two core elements of this correctness notion: the detection of overlapping rules and of missing rules. The algorithms have been implemented atop the DMN toolkit dmn-js. An empirical evaluation on large decision tables has shown the potential for scalability of the proposed algorithms and their ability to generate non-redundant feedback that is more concise than the one generated by the Signavio DMN editor.

The proposed algorithms rely on a geometric interpretation of rules in decision tables, which we foresee could be used to tackle other analysis problems. In particular, we foresee that the problem of simplification of decision tables (rule merging) could be approached from a geometric standpoint. Indeed, if we see the rules as hyperrectangles, the problem of table simplification can be mapped to one of finding an optimal way of merging hyperrectangles with respect to some optimality notion. Another direction for future work is to extend the proposed formal semantics to encompass other aspects of the DMN standard, such as the concept of Decision Requirements Graphs (DRGs), which allow multiple decision tables to be linked in various ways.

Acknowledgement. This research was partly funded by an Institutional Grant of the Estonian Research Council.

References

1. Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Scalable sweeping-based spatial join. In *VLDB*, 1998.
2. Kimon Batoulis, Andreas Meyer, Ekaterina Bazhenova, Gero Decker, and Mathias Weske. Extracting decision logic from process models. In *Proc. of CAiSE*. Springer, 2015.
3. CODASYL Decision Table Task Group. *A Modern appraisal of decision tables : a CODASYL report*. ACM, 1982.
4. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. of TACAS*, pages 337–340. Springer, 2008.
5. 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.
6. Rattikorn Hewett and John Leuchner. Restructuring decision tables for elucidation of knowledge. *Data Knowl. Eng.*, 46(3):271–290, 2003.
7. D. N. Hoover and Zewei Chen. Tablewise, a decision table tool. In *Proc. of COMPASS*, pages 97–108, 1995.
8. Object Management Group. Decision Model and Notation (DMN) 1.0, 2015.
9. Zdzislaw Pawlak. Decision tables – a rough set approach. *Bulletin of the EATCS*, 33:85–95, 1987.
10. Udo W. Pooch. Translation of decision tables. *Comp. Surv.*, 6(2):125–151, 1974.
11. 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.
12. 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.
13. Abbas K. Zaidi and Alexander H. Levis. Validation and verification of decision making rules. *Automatica*, 33(2):155 – 169, 1997.