Semantics, Analysis and Simplification of DMN Decision Tables

Diego Calvanese[a], Marlon Dumas[b], Ülari Laurson[b], Fabrizio M. Maggi[b], Marco Montali[a], Irene Teinemaa[b]

[a]*Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy*
[b]*University of Tartu, J. Liivi 2, 50409 Tartu, Estonia*

Abstract

The Decision Model and Notation (DMN) is a standard notation to capture decision logic in business applications. 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 and refactoring tasks on these tables. This article puts forward a formal semantics for DMN decision tables and a formal definition of analysis tasks on such tables. The article then proposes a general approach to analyze and refactor decision tables based on a geometric interpretation thereof. This general approach is used to design efficient algorithms for two analysis tasks (detection of overlapping rules and of missing rules) and one refactoring task (simplification of tables via rule merging). The algorithms have been implemented in an open-source DMN editor and tested on large decision tables derived from a credit lending dataset.

Keywords: Decision Table, Decision Model, Notation, Sweep-Line

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

Email addresses: calvanese@inf.unibz.it (Diego Calvanese), marlon.dumas@ut.ee (Marlon Dumas), laurson17@gmail.com (Ülari Laurson), f.m.maggi@ut.ee (Fabrizio M. Maggi), montali@inf.unibz.it (Marco Montali), irene.teinemaa@ut.ee (Irene Teinemaa)

from the control-flow logic [1] and to capture it at a higher level of abstraction has motivated the emergence of the Decision Model and Notation (DMN) [16]. A central construct of DMN is that of a decision table, which stems from the concept of decision table proposed in the context of program decision logic specification in the 1960s [19]. 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 to capture complex and critical business decisions raises the need to support the analysis and refactoring of these tables. In this respect, two common correctness criteria imposed on decision tables are that their rules should be disjoint and complete, meaning that every possible input should match exactly one rule. A table violates these criteria if it has overlapping rules (multiple rules match a given input) or missing rules (no rule matches a given input). Meanwhile, a common refactoring task is to simplify a decision table by merging pairs of rules into a single more general rule.

Given the above setting, this article provides a foundation for analyzing and refactoring DMN decision tables. The contributions of the article are: (i) a formal semantics and a formalization of correctness criteria for DMN decision tables; and (ii) efficient algorithms for detecting overlapping rules and missing rules and for simplifying decision tables via rule merging. The latter algorithms are based on a novel geometric interpretation of DMN decision 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). Under this geometric interpretation, the problem of detecting overlapping rules is mapped to that of detecting maximal sets of overlapping hyper-rectangles; the problem of finding missing rules is mapped to that of determining if the union of the rules in the table (viewed as hyper-rectangles) covers the N-dimensional space identified by the cartesian product of the domains of the input columns of the table; and finally, the problem of simplifying a decision table is mapped to that of finding maximal sets of adjacent hyper-rectangles and merging them into a minimal set of hyper-rectangles covering the same space. Based on this geometric interpretation, the article presents scalable algorithms for the two analysis tasks and for the refactoring task.

The article is an extended and revised version of a previous conference article [5]. With respect to the conference version, the main additional contributions are: (i) the technique for decision table simplification and an empirical comparison against an approach to this problem proposed in the context of classical decision tables [18]; and (ii) an improved technique for detecting overlapping rules

that achieves lower execution times than the one proposed in the conference article. The techniques have been implemented atop the dmn-js editor and evaluated using decision tables of varying sizes derived from a credit lending dataset.

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

2. Background and Related Work

Below, we provide an overview of DMN decision tables and discuss previous work related to their analysis.

2.1. Overview of DMN Decision Tables

A DMN decision 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 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 (a.k.a. the *output entries*). For example, Table 1 shows a DMN decision table with two input columns, one output column and four rules.

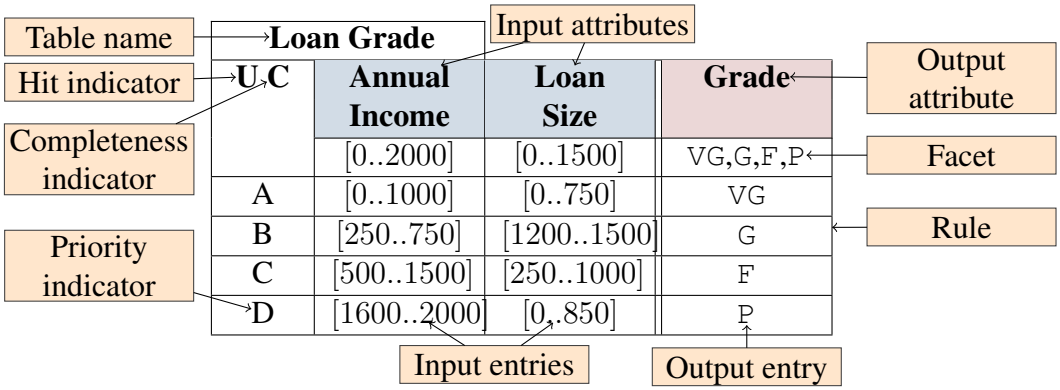


Table 1: Sample decision table with its constitutive elements

Given a vector of input 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 $\langle 500, 1400 \rangle$ matches rule B in Table 1, thus yielding G in the output column. To specify how output configurations are computed from input ones, a DMN decision table has a *hit indicator* and a *completeness indicator*.[1] 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 indicator. Similarly, if no rule matches an input configuration, this may contradict the completeness indicator. The former contradiction leads to *overlapping rules* while the latter leads to *missing rules*.

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 [6]. Several algorithms addressing these two analysis tasks have been proposed [17, 13, 24]. However, these algorithms require the domain of each input attribute to be either boolean or categorical. If an attribute has a numerical domain, meaning that the input entries of this attribute are intervals, then these intervals must be disjoint.

Concretely, the above approaches cannot handle situations where multiple overlapping intervals appear as input entries of the same attribute (e.g., attribute A has input entry [151..300] in one rule and [200..250] in another rule). If this happens, 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), [200..250) and [250..300]). In the worst case, this expansion increases the size of the table exponentially in the number of numerical attributes. Alternatively, instead of breaking down the entries under a numerical attribute into non-overlapping intervals, we can instead define one boolean variable for each interval. For example, if a numerical attribute A has input entries [151..300] in one rule and [200..250] in another rule, we can rewrite the table into an equivalent table with two boolean attributes, A1 and A2, where A1 is true iff the value of the original attribute A is in the interval [151..300], and A2 is true iff the value of the original

[1] In the new version of DMN released in 2016 (DMN 1.1), the notion of completeness indicator was eliminated. However, this is purely a standardization decision. The problem of identifying missing rules remains a relevant problem from a tooling perspective. In this article, we refer to the completeness and hit indicators of DMN 1.0.

attribute A is in the interval [200..250]. This rewriting approach does not increase the number of rules in the table, but instead it may significantly increase the number of columns. To the best of our knowledge, the approach for analyzing decision tables we propose is the first one that deals with numerical attributes without requiring the input entries of each numerical attribute to be previously broken down into disjoint intervals.

Similarly, several algorithms for simplifying decision tables via rule merging have been proposed [18, 20, 15, 22]. The Pollack's algorithm [18] selects two rules that have the same output and coincide on all inputs but one. Every time that such a pair of rules is identified, they are merged into a single rule by doing the union of the sets of values in the two cells where the difference occurs (all other cells remain the same in the merged rule). Shwayder [20] proposes an optimization of the Pollack's algorithm applicable in the case where certain rules do not depend on all attributes, but only on a subset of them. Maes [15] proposes further optimizations in cases where there are logical relations between the attributes (e.g., if two attributes are true, then a third attribute is also true). The latter approach specifically optimizes the order in which the attributes are scanned (to identify possible rules to be merged) so as to obtain a minimum set of rules after simplification. In this approach, only one pair of rules is merged at a time. Vanthienen et al. [22] extend the latter approach by considering situations where groups of more than two rules can be merged together into a smaller set of rules.

All these algorithms suffer from the same limitations mentioned above in the context of algorithms for finding missing and overlapping rules. In other words, these algorithms operate over attributes with boolean domains (and by extension they can be applied to categorical domains). If the table has numerical domains, the intervals under each attribute must be made disjoint as explained above.

From these observations, we can conclude that, while the verification and simplification of decision tables with discrete or discretized domains has received much attention, the case where columns have both discrete domains and numerical domains with arbitrary interval expressions has not been considered in the literature. In this article, we propose a geometric approach to diagnose and simplify decision tables that overcome the above limitations. Geometric approaches to analyze equivalence or overlap of expressions have been studied in the context of arithmetic expressions [11], but they have never been applied to decision tables before.

Another related body of research deals with using decision tables as output of classification algorithms (as an alternative to decision trees) [14]. Such techniques have been applied to extract DMN decision tables from business process event

logs [2]. This body of research however is not concerned with the analysis of decision tables, but rather with their discovery.

To conclude our analysis of the literature, we report that several tools are available for modeling executing, and analyzing classical decision tables. Prologa [21, 23] 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. However, the underlying techniques implemented in Prologa are designed for boolean and categorical attributes. When attributes are numerical, their input entries need to be decomposed as explained above.

Signavio's DMN editor[2] detects overlapping and missing rules without requiring the entries of numerical attributes to be broken down. 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[3] 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 [7]), 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.

3. Formal Semantics

In this section, we provide a formalization of DMN decision tables, defining their input/output semantics, and, at the same time, introducing several analysis tasks focused on correctness checking. We do not consider forms of aggregation for output values here, as they are orthogonal to correctness checking. Hence, we focus on decision tables returning the output of one rule only. These are called *single hit* tables. 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. This formalization is important per se, as it defines a clear, unambiguous semantics of

[2] `http://www.signavio.com`
[3] `http://openrules.com/`

decision tables, and also it represents 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 the conditions over such types expressed using the S-FEEL language. A data type $\mathcal{T}$ is a pair $\langle \Delta_{\mathcal{T}}, \Sigma_{\mathcal{T}} \rangle$, where $\Delta_{\mathcal{T}}$ is an *object domain*, and $\Sigma_{\mathcal{T}} = \Sigma^P_{\mathcal{T}} \uplus \Sigma^F_{\mathcal{T}}$ is a *signature*, consisting of 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. Similarly, 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 rigidly 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 with *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[4]:

- $\mathcal{T}_{\mathbb{S}} = \langle \mathbb{S}, \emptyset, \emptyset \rangle$ – strings.
- $\mathcal{T}_{\mathbb{B}} = \langle \{\mathsf{true}, \mathsf{false}\}, \emptyset, \emptyset \rangle$ – boolean attributes.
- $\mathcal{T}_{\mathbb{Z}} = \langle \mathbb{Z}, \{</2, >/2\}, \{+/2, -/2, \cdot/2, \div/2\} \rangle$ – integer numbers equipped with the usual comparison predicates and binary operations.
- $\mathcal{T}_{\mathbb{R}} = \langle \mathbb{R}, \{</2, >/2\}, \{+/2, -/2, \cdot/2, \div/2\} \rangle$ – real numbers equipped with the usual comparison predicates and binary operations.

The set of all such types is denoted by $\mathfrak{T}$.

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 $\mathcal{T}$ is:

$$
\begin{array}{rcl}
\mathcal{Q} & ::= & \text{“–”} \mid \mathit{Term} \mid \text{“}\texttt{not (}\text{”}\ \mathit{Term}\ \text{“}\texttt{)}\text{”} \mid \\
 & & \mathit{Comparison} \mid \mathit{Interval} \mid \mathcal{Q}_1\texttt{,}\ \mathcal{Q}_2 \\
\mathit{Comparison} & ::= & \mathit{COp}\ \mathit{Term} \\
\mathit{Interval} & ::= & (\text{“}\texttt{(}\text{”} \mid \text{“}\texttt{[}\text{”})\,\mathit{Term}_1\,\text{“..”}\,\mathit{Term}_2(\text{“}\texttt{)}\text{”} \mid \text{“}\texttt{]}\text{”}) \\
\mathit{Term} & ::= & v \mid f(\mathit{Term}_1, \ldots, \mathit{Term}_m)
\end{array}
$$

where *(i)* COp is a binary predicate symbol in $\Sigma_{\mathcal{T}}$, *(ii)* v is an object from $\Delta_{\mathcal{T}}$, and *(iii)* f is an m-ary function in $\Sigma_{\mathcal{T}}$.

[4]Date/time data types are also supported, but can be considered as simple numerical attributes.

Concretely, S-FEEL supports the following conditions on a given data type $\mathcal{T}$:

- "–" indicates *any value*, i.e., it holds for every object in $\Delta_{\mathcal{T}}$.
- "$=Term$" 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 "$=Term$".
- $Comparison$ is only applicable when $\mathcal{T}$ is a numerical data type, and indicates a *comparison condition*, which holds for all objects that are related via the comparison predicate to the object resulting from expression $Term$.
- $Interval$ is only applicable when $\mathcal{T}$ is numerical, and allows the modeler to capture membership conditions testing whether an input object belongs to the given interval.
- "$\mathcal{Q}_1$`,` $\mathcal{Q}_2$" indicates an *alternative condition*, which holds whenever one of the two conditions $\mathcal{Q}_1$ or $\mathcal{Q}_2$ 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 elder (i.e., older than 70 years) is captured by condition "`[0..18)` , $\geq$ `70`". ∎

3.2. DMN Decision Tables

We can now define 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*, respectively.[5]
- $\mathsf{Type} : I \uplus O \to \mathfrak{T}$ 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 also referred to as "optional lists of values".
- R is a finite set $\{r_1, \ldots, r_p\}$ of *rules*. Each rule r is a pair $\langle \mathsf{If}, \mathsf{Then}\rangle$, where If is an *input entry function* that associates each input attribute $\mathbf{a^{in}} \in I$ to a

[5] Attributes are called "expressions" in the DMN standard because the entries in the table are expressions over attributes.

condition over $\mathsf{Type}(\mathbf{a^{in}})$, and Then is an *output entry function* that associates each output attribute $\mathbf{a^{out}} \in O$ to an object in $\mathsf{Type}(\mathbf{a^{out}})$.

- $\mathsf{Priority} : R \to \{1, \ldots, |R|\}$ is a *priority function* injectively mapping rules in R to a corresponding rule number defining its priority.
- $C \in \{\mathtt{c}, \mathtt{i}\}$ is the *completeness indicator*, where $\mathtt{c}$ is the default value and stands for *complete* table, while $\mathtt{i}$ stands for *incomplete* table.
- $H \in \{\mathtt{u}, \mathtt{a}, \mathtt{p}, \mathtt{f}\}$ is the *(single) hit policy indicator* defining the policy for the rule application, where: *(i)* $\mathtt{u}$ 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*.

The notion of *priority* deserves a dedicated discussion. According to the DMN standard, two different notions of priority are respectively induced by a set of rules. The first notion of priority, which we call *rule priority*, is simply determined by the ordering of rules. The second notion of priority, which we call *output priority*, is induced by ordering the rules according to the lexicographic ordering of the output values (e.g., alphabetical order for strings or the usual total ordering for numbers). Since rule and output priorities never interact (either none or only one of them is actually used), we employ the abstract $\mathsf{Priority}$ function introduced before to accommodate both types of priority.

Next, we informally review the 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 attribute, then the rule *triggers*, and bounds the output attributes to the corresponding 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 $1\,230$, 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 deciding which output is selected when multiple rules trigger simultaneously.
- "Any hit" indicates that when multiple rules trigger, they must agree on the output objects, thus guaranteeing that the output is always unambiguously computed.
- "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 the highest output priority.
- "First hit" (or "Rule order") indicates that whenever multiple rules trigger, then the output is unambiguously computed by only considering the contribution of the triggered rule that has the highest rule priority.

3.3. Formalization of Rule Semantics and Analysis Tasks

We first define how conditions map to the 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 $\mathcal{T}$, the *condition formula for* $\mathcal{Q}$, written $\Phi_{\mathcal{Q}}$, is a formula that captures the semantics of $\mathcal{Q}$ by suitably using predicates/functions in $\Sigma_{\mathcal{T}}$ and objects from $\Delta_{\mathcal{T}}$, as well as (possibly) a single free variable. Specifically, $\Phi_{\mathcal{Q}} =$

$$\begin{cases} \mathit{true} & \text{if } \mathcal{Q} = \text{“–”} \\ \neg\Phi_{\mathit{Term}} & \text{if } \mathcal{Q} = \text{“}\texttt{not}(\mathit{Term})\text{”} \\ x = \mathit{Term} & \text{if } \mathcal{Q} = \mathit{Term} \\ x\ \mathit{COp}\ \mathit{Term} & \text{if } \mathcal{Q} = \text{“}\mathit{COp}\ \mathit{Term}\text{” and } \mathit{COp} \in \{<, >, \leq, \geq\} \\ x > \Phi_{\mathit{Term}_1} \wedge x < \Phi_{\mathit{Term}_2} & \text{if } \mathcal{Q} = \text{“(}\ \mathit{Term}_1..\mathit{Term}_2\ \text{)”} \\ x > \Phi_{\mathit{Term}_1} \wedge x \leq \Phi_{\mathit{Term}_2} & \text{if } \mathcal{Q} = \text{“(}\ \mathit{Term}_1..\mathit{Term}_2\ \text{]”} \\ x \geq \Phi_{\mathit{Term}_1} \wedge x < \Phi_{\mathit{Term}_2} & \text{if } \mathcal{Q} = \text{“[}\ \mathit{Term}_1..\mathit{Term}_2\ \text{)”} \\ x \geq \Phi_{\mathit{Term}_1} \wedge x \leq \Phi_{\mathit{Term}_2} & \text{if } \mathcal{Q} = \text{“[}\ \mathit{Term}_1..\mathit{Term}_2\ \text{]”} \\ \Phi_{\mathcal{Q}_1} \vee \Phi_{\mathcal{Q}_2} & \text{if } \mathcal{Q} = \text{“}\mathcal{Q}_1\texttt{,}\ \mathcal{Q}_2\text{”} \end{cases}$$

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

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

With condition formulae, we now formalize the key notions of: *(i)* correctness of rule specifications, *(ii)* semantics of rules, *(iii)* semantics of completeness, and *(iv)* semantics of hit indicators. These notions are, in turn, basic building blocks for a global 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 $R = \{r_1, \ldots, r_p\}$. Throughout the formalization, we use notation $\vec{x} = x_1, \ldots, x_m$ for object variables filling the m input attributes, and $\vec{y} = y_1, \ldots, y_n$ for object variables filling the n output attributes.

Facet correctness. We first consider the *facet correctness* of $\mathcal{D}$, which intuitively amounts to check whether all mentioned input conditions and output objects agree with the facets associated to their corresponding attributes. First, we say that object x is *legal* for attribute $\mathbf{a} \in I \cup O$, written $Legal_{\mathbf{a}}(x)$, if x belongs to the set of objects defined by the facet associated to $\mathbf{a}$:

$$Legal_{\mathbf{a}}(x) \triangleq \Phi_{\mathsf{Facet}(\mathbf{a})}(x)$$

Consider now a condition $\mathcal{Q}$ over $\mathbf{a}$. We say that object x *matches* $\mathcal{Q}$, written $Matches^{\mathcal{Q}}_{\mathbf{a}}(x)$, if x agrees with the facet associated to $\mathbf{a}$, and at the same time x satisfies $\mathcal{Q}$:

$$Matches^{\mathcal{Q}}_{\mathbf{a}}(x) \triangleq Legal_{\mathbf{a}}(x) \wedge \Phi_{\mathcal{Q}}(x)$$

Note that for an output attribute with associated object o, $\Phi_{\mathcal{Q}}(x)$ corresponds to an atomic, equality formula testing whether $x = \mathsf{o}$. Starting from this notion of matching, we formalize that condition $\mathcal{Q}$ is *compatible* with attribute $\mathbf{a}$, written $Compatible^{\mathcal{Q}}_{\mathbf{a}}$, if it is possible find at least one object that matches $\mathcal{Q}$ (i.e., it is not the case that the facet attached to $\mathbf{a}$ is completely disjoint to the set of objects satisfying $\mathcal{Q}$). Formally:

$$Compatible^{\mathcal{Q}}_{\mathbf{a}} \triangleq \exists x. Matches^{\mathcal{Q}}_{\mathbf{a}}(x)$$

Rule semantics. A rule $r = \langle \mathsf{If}, \mathsf{Then} \rangle \in R$ is *triggered by* a configuration $\vec{x}$ of input objects, written $TriggeredBy_r(\vec{x})$, whenever each such object matches the corresponding input condition:

$$TriggeredBy_r(\vec{x}) \triangleq \bigwedge_{i \in \{1,\ldots,m\}} Matches^{\mathsf{If}(\mathbf{a_i})}_{\mathbf{a_i}}(x_i)$$

Two configurations $\vec{x}$ and $\vec{y}$ of input and output objects respectively are *input-output related* by a rule $r = \langle \mathsf{If}, \mathsf{Then} \rangle \in R$, written $IORel_r(\vec{x}, \vec{y})$, if r is triggered

by the input configuration $\vec{x}$, and the output configuration $\vec{y}$ agrees with the output specified by r:

$$IORel_r(\vec{x},\vec{y}) \triangleq TriggeredBy_r(\vec{x}) \wedge \bigwedge_{j\in\{1,\ldots,n\}} Matches_{\mathbf{b_j}}^{\mathsf{Then}(\mathbf{b_j})}(y_j)$$

Completeness. When declaring that a table is (in)complete, there is no guarantee that the rules contained in the table indeed behave as declared. Checking whether the rules of the table imply completeness amounts to checking that the input conditions "cover" the domains of the attributes, as specified by their facets. Formally, we say that decision table $\mathcal{D}$ is *complete* if for every input configuration constituted by objects that agree, position-wise, with the facets of their corresponding attributes, there exists at least one rule in $\mathcal{D}$ that is triggered by that configuration:

$$Complete_{\mathcal{D}} \triangleq \forall\vec{x}.\Big(\bigwedge_{i\in\{1,\ldots,m\}} Legal_{\mathbf{a_i}}(x_i)\Big) \rightarrow \bigvee_{k\in\{1,\ldots,p\}} TriggeredBy_{r_k}(\vec{x})$$

Hit policies. Like for the completeness indicator, when declaring that a table works under a given (single) hit policy, there is no guarantee that the rules in the table actually behave in a way that is compatible with the policy itself. We then review each policy indicator, and introduce the corresponding formulae over the rules of the table so as to capture this notion of compatibility.

We start with the *unique hit* policy, which prescribes that each input configuration triggers at most one rule. To properly mirror such an indication, table $\mathcal{D}$ must be *unique*, written $Unique_{\mathcal{D}}$, i.e., $\mathcal{D}$ must guarantee that whenever a rule is triggered by a given input configuration, that input configuration does not trigger any other rule:

$$Unique_{\mathcal{D}} \triangleq \bigwedge_{i\in\{1,\ldots,p\}} \forall\vec{x}.\Big(TriggeredBy_{r_i}(\vec{x}) \rightarrow \bigwedge_{j\in\{1,\ldots,p\}\setminus\{i\}} \neg TriggeredBy_{r_j}(\vec{x})\Big)$$

We then continue with the *any hit* policy. To behave consistently with such a policy, $\mathcal{D}$ must be such that whenever multiple rules are triggered by the same input configuration, they must agree on the output. This is formalized as follows:

$$AgreesOnOutput_{\mathcal{D}} \triangleq \bigwedge_{i,j\in\{1,\ldots,p\},i\neq j} \forall\vec{x},\vec{y}.\left(\left(\begin{array}{l} TriggeredBy_{r_i}(\vec{x}) \\ \wedge\ TriggeredBy_{r_j}(\vec{x})\big) \\ \wedge\ IORel_{r_i}(\vec{x},\vec{y}) \end{array}\right) \rightarrow \big(IORel_{r_j}(\vec{x},\vec{y})\big)\right)$$

We now consider the case of priority and first hit policies. Both policies depend on (respectively, the output and rule) priority. Hence, in the presence of such hit policies, the rule semantics needs to be properly reformulated, by considering the Priority function. In particular, we say that rule $r \in R$ is *triggered with priority* by input configuration $\vec{x}$, written $TriggeredWithPriorityBy_r(\vec{x})$, if it is triggered by $\vec{x}$ in the normal sense, and in addition no rule of higher priority is triggered by the same input $\vec{x}$:

$$TriggeredWithPriorityBy_r(\vec{x}) \triangleq TriggeredBy_r(\vec{x}) \wedge \bigwedge_{r_h \in \{r' | r' \in R \text{ and } \mathsf{Priority}(r') > \mathsf{Priority}(r)\}} \neg TriggeredBy_{r_h}(\vec{x})$$

This new formula, in turn, can be used to define the input-output relation induced by the entire table $\mathcal{D}$ in the presence of priority. Specifically, we say that two configurations $\vec{x}$ and $\vec{y}$ of input and output objects are *input-output related* by $\mathcal{D}$ *in the presence of priority*, written $IORelP_{\mathcal{D}}(\vec{x}, \vec{y})$, if there exists a rule r in $\mathcal{D}$ such that r is the highest-priority rule triggered by $\vec{x}$, and $\vec{x}$ and $\vec{y}$ are input-output related by r (in the sense defined before):

$$IORelP_{\mathcal{D}}(\vec{x}, \vec{y}) \triangleq \bigvee_{r \in R} TriggeredWithPriorityBy_r(\vec{x}) \wedge IORel_r(\vec{x}, \vec{y})$$

In addition, we observe that the presence of priority may lead to the wrong situation where some rules are never triggered. This happens when other rules of higher priority have more general input conditions, thus subsuming the one having lower priority. We formalize this notion as follows: rule $r_1 \in R$ *is masked* by rule $r_2 \in R$, written $MaskedBy_{r_1}^{r_2}$, if for every input configuration, whenever r_1 is triggered by that input configuration, then this is also the case for r_2:

$$\begin{aligned} MaskedBy_{r_1}^{r_2} \triangleq\ & \mathsf{Priority}(r_2) > \mathsf{Priority}(r_1) \\ & \wedge \forall \vec{x}.\, TriggeredBy_{r_1}(\vec{x}) \rightarrow TriggeredBy_{r_2}(\vec{x}) \end{aligned}$$

Clearly, checking redundancies and subsumption between rules is of general interest, not just in the presence of priorities. This will be in fact extensively discussed in Section 4.

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*, written $Correct_{\mathcal{D}}$, 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 or f, all rules are "relevant", i.e., no rule is masked by a rule with higher priority.

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

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

Global input-output formula. We conclude our formalization by defining a single formula that captures the overall input-output relation induced by $\mathcal{D}$, considering its rules as well as its hit policy. Specifically, we say that an input configuration $\vec{x}$ and an output configuration $\vec{y}$ are *input-output related* by $\mathcal{D}$, written $\mathit{IORel}_{\mathcal{D}}(\vec{x}, \vec{y})$, if:

1. The hit policy is either u or a, and there exists a rule that relates $\vec{x}$ to $\vec{y}$.[6]
2. The hit policy is either p or f, there exists a rule r that relates $\vec{x}$ to $\vec{y}$, and there is no other rule with higher priority that is triggered by $\vec{x}$.

This is formalized as follows:

$$\begin{aligned} \mathit{IORel}_{\mathcal{D}}(\vec{x}, \vec{y}) \triangleq & \Big((H = \mathtt{u} \vee H = \mathtt{a}) \rightarrow \bigvee_{r \in R} \mathit{IORel}_r(\vec{x}, \vec{y}) \Big) \\ & \wedge \Big((H = \mathtt{p} \vee H = \mathtt{f}) \rightarrow \mathit{IORelP}_{\mathcal{D}}(\vec{x}, \vec{y}) \Big) \end{aligned}$$

[6] In the case of *any hit* policy, several matching rules may exist, but since they establish the same input-output relation, it is sufficient to pick one of them.

4. Analysis and Simplification Algorithms

In this section, we introduce a general approach to represent a DMN decision table in terms of geometric objects and we apply this approach to design algorithms for analyzing and refactoring DMN decision tables. First, we apply this general approach to design an algorithm for detecting overlapping rules in a DMN decision table. The detection of overlapping rules allows us to check if a given table fulfills a unique-hit policy. It also allows us to check if a table with an "any hit" policy is correct. Indeed a table with "any hit" should be such that any group of overlapping rules should have the same output. Finally, in the case of "priority hit" and "first hit" policies, this operation allows us to detect situations where a rule partially or totally masks a set of other rules, because it overlaps with these rules while having a higher priority or order. Next, we apply the general approach for representing DMN tables in order to design an algorithm for detecting missing rules. This operation allows us to verify the correctness of a table with a "Complete" indicator. Finally, we show how the approach can be used to simplify a decision table by merging multiple "adjacent" rules with the same output into a smaller set of rules.

4.1. General approach

The proposed algorithms rely on a geometric interpretation of a DMN decision table. Every rule in a table is seen as an iso-oriented hyper-rectangle in an N-dimensional space (where N is the number of columns). Indeed, an input entry in a rule can be seen as a constraint over one of the columns. 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 a 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.

The geometric interpretation of Table 1 is shown in Figure 1.[7] 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 $[500, 1000] \times [250, 750]$. The table also contains missing values. For example, vector $\langle 200, 1000 \rangle$ does not match any rule in Table 1.

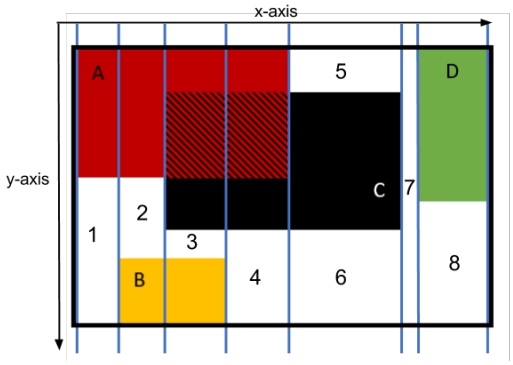


Figure 1: Geometric representation of the DMN decision table shown in Table 1. The x-axis is the annual income while the y-axis is the loan size. The colored rectangles A-D represent the four rules in the table. The white rectangles 1-8 represent a set of rectangles such that the union of rectangles A-D and 1-8 cover all possible combinations of annual income and loan amount.

4.2. Finding Overlapping Rules

Given the geometric interpretation of a decision table discussed above, the problem of detecting overlapping rules becomes that of finding intersections in a set of hyper-rectangles. A straightforward approach to this problem is to scan through all possible pairs of hyper-rectangles and to check each pair for intersection. This algorithm has a complexity of $O(d \cdot |R|^2)$, where $|R|$ is the number of rules in the table and d is the number of columns (which is also the number of dimensions in the hyper-space under consideration). A more sophisticated approach [9] combines a sweep-line algorithm with data structures for range queries in order to achieve a lower complexity: $O(|R| \cdot log^d |R|)$.

The above approaches produce as output the set of all pairs of overlapping rules in a table. This output is arguably not useful from an end-user perspective as this set of pairs can be too large for manual inspection. Also, if the goal is

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

ultimately to repair the table so as to obtain a non-overlapping one, it is not convenient to reason at the level of pairs of overlapping rules, but rather at the level of larger groups of rules that overlap with each other.

Inspired by this observation, we formulate the problem of finding overlapping rules in a DMN decision table as that of computing maximal subsets of rules such that every two rules in a subset overlap with each other. We approach this problem by first computing all intersecting pairs of hyper-rectangles in order to construct an *overlap graph*. In this graph, each vertex represents a rule (i.e., a hyper-rectangle). There is an edge between two vertices if their corresponding rules overlap. For example, consider the set of overlapping rectangles shown in the left-hand side of Figure 2, which includes the four rectangles in Figure 1 and two additional ones (E and F). The corresponding overlap graph and its maximal cliques are shown on the right-hand side of the figure. Note that rectangles B and D do not appear in the overlap graph since they do not overlap with any other rectangle.

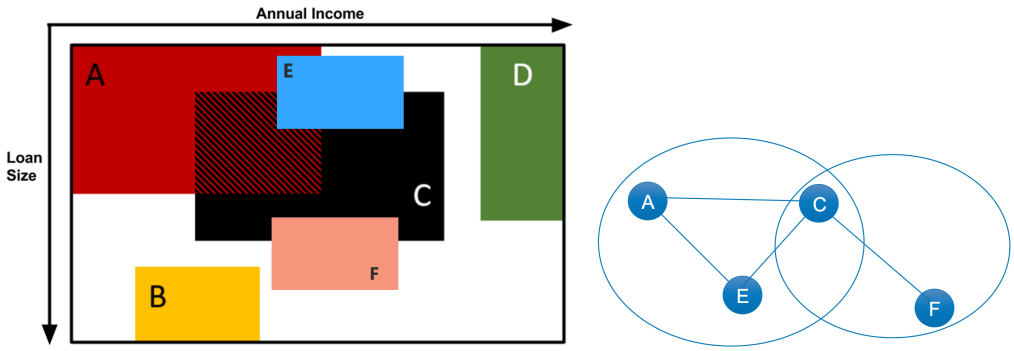


Figure 2: Extended version of the example in Figure 1 (left) and corresponding overlap graph with maximal cliques thereof (right)

The subsets of rules that we are looking for are cliques in the overlap graph (e.g., the two cliques shown in Figure 2). Since we aim at minimizing the number of such subsets, the problem is then that of listing the maximal cliques in the overlap graph. For example, the maximal cliques of the running example are delimited as ovals in Figure 2. Finding maximal cliques is an NP-hard problem [10]. However, existing algorithms such as the Bron-Kerbosch [4] algorithm perform well in practical scenarios, particularly when the input graph is sparse, which is likely to be the case in our setting, as we expect that rule overlaps will be an exception rather than a norm.

In summary, the proposed procedure to identify overlapping rules from a decision table is as follows:

1. Map the rules into hyper-rectangles in an N-dimensional space (where N is the number of columns in the table).

2. Build the overlap graph by computing all the pairs of intersecting hyper-rectangles (this can be done either by scanning through all possible pairs of hyper-rectangles and checking each pair for intersection or by using the algorithm in [9]) and eliminate rules that do not overlap with any other rule.

3. Generate the maximal cliques of the overlap graph using the Bron-Kerbosch algorithm.

4. Output each maximal clique as a set of overlapping rules.

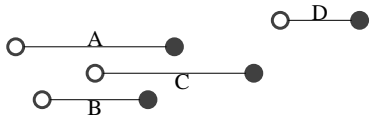
4.3. Finding Missing Rules

The proposed geometric interpretation of decision tables can also be applied to the problem of detecting missing rules. Below we outline an approach to this problem based on the sweep-line algorithmic paradigm [3]. The idea of sweep-line algorithms is to pick one dimension (e.g., x-axis), project all geometric objects (in this case hyper-rectangles) on 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 a hyper-rectangle starts or ends. In Figure 1, the stop points are depicted as vertical lines. When the line makes a "stop", we gather all (hyper-)rectangles that intersect the line. This set of hyper-rectangles is called the *active list* and they are such that they overlap along their x-axis projection. The idea of sweep-line is to analyze the hyper-rectangles in the active set, then move the line to the next position, and so on until the last hyper-rectangle has been processed along the x-axis.

The specific procedure we propose for missing rules detection is described in Algorithm 1. This algorithm takes as inputs five parameters:

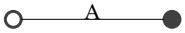
1. ruleList – the set of rules of the input DMN decision table;
2. missingIntervals – the current set of missing intervals;
3. i – the index of the dimension (column) that is being swept through;
4. N – the total number of columns;
5. MissingRuleList – the set of missing rules.

The algorithm starts by sweeping through the first column of the table (axis x). To illustrate this first pass, we consider the projection of the table in Figure 1 on the x axis:



The 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_A^x. Therefore, I_A^x is added to the active list of intervals for the first column x, $\mathcal{L}_x$. An interval is added to the active list 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_B^x. Since $\mathcal{L}_x$ is not empty, I_B^x 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_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 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_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 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 adjacent 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 the list of missing value areas. Note that the algorithm merges two areas of missing values only when the intervals corresponding to one column are adjacent and the ones corresponding to all the other columns are exactly the same. In the example in Figure 1, areas 4 and 6 are merged.

At this point, the recursion ends and the algorithm goes back to analyze the intervals in the projection along the x axis. The last bound processed was the

Algorithm 1: Procedure findMissingRules.

Input: $ruleList$; $missingIntervals$; i; N; $missingRuleList$.

```
1  if i < N then
2      L_{x_i} = []; // initializes the current list of bounds
3      sortedListAllBounds = ruleList.sort(i);
4      lastBound = 0;
5      foreach currentBound ∈ sortedListAllBounds do
6          if !areAdjacent(lastBound, currentBound) then
7              missingIntervals[i] = constructInterval(lastBound, currentBound);
8              if missingRuleList.canBeMerged(missingIntervals); then
9                  missingRuleList.merge(missingIntervals);
10             else
11                 missingRuleList.add(missingIntervals);
12         if !L_{x_i}.isEmpty() then
13             missingIntervals [i] = constructInterval(lastBound, currentBound);
14             findMissingRules(L_{x_i},missingIntervals,i +1, N, missingRuleList); /* recursive
                 invocation */
15         if currentBound.isLower() then
16             L_{x_i}.put(currentBound);
17         else
18             L_{x_i}.delete(currentBound);
19         lastBound = currentBound;
20 return missingRuleList;
```

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 adjacent to 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 adjacent to an area of missing values previously found and, if this is the case, the two areas are merged. If this is not the case, the area is added to the list of missing value areas (in our case area 2 is added to the list of missing value areas). The list of missing areas (stored in missingRuleList) is returned by the algorithm (line 20).

4.4. Decision Table Simplification

The proposed approach to table simplification proceeds in three phases as sketched in Figure 3. In the first phase, the set of rules are divided into groups on the basis of their output (i.e., all rules with the same output are put into one group). This grouping is needed, because two rules can be merged only if they have the same output. The second and third phases are applied to each group resulting from this first phase.

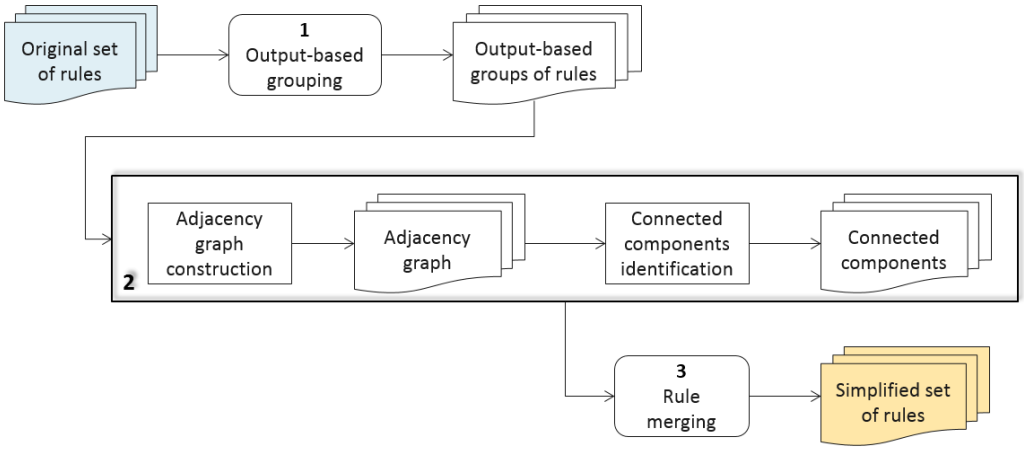


Figure 3: Decision table simplification phases

In the second phase, each rule is mapped to a hyper-rectangle as previously explained. Using this viewpoint, we construct an *adjacency graph*. In this graph, each rule in the table is represented as a vertex. Two vertices are connected by an edge if their corresponding hyper-rectangles are *adjacent*, meaning that they overlap across all dimensions but one, and they are contiguous along the dimension where they do not overlap (i.e., they share a common side). We observe that two adjacent hyper-rectangles can potentially be merged either into a single hyper-rectangle (if they fully share a side), or into multiple ones if they partially share a side. We also observe that, if two hyper-rectangles are in different connected

components of the adjacency graph, they cannot be merged into a single rule (neither fully nor partially). This observation allows us to apply a divide-and-conquer approach: instead of testing arbitrary pairs of rules to find potential rule merging opportunities, we consider one connected component of the adjacency graph at a time, and merge rules only within that connected component.

The computation of the adjacency graph can be done in a similar way as that of the overlap graph, either by testing every possible pair of hyper-rectangles to check if they have a common side – $O(d \cdot |R|^2)$ – or by using the algorithm in [9] – $O(|R| \cdot log^d |R|)$. The computation of the connected components is done using the Tarjan's algorithm, which has a linear-time complexity in the number of edges.

The output of the second phase is thus a set of connected components in the adjacency graph. The third phase will be applied for each connected component containing more than one rule. For example, Figure 4 shows a decision table (similar to the one in Table 1 but without overlaps) and its corresponding geometric interpretation. As depicted in the figure, the corresponding adjacency graph has three connected components called $G1$, $G2$, and $G3$. The latter two connected components contain only one rule each, so no simplification is possible. The rule merging will thus be applied to component $G1$ only.

Loan Grade			
U C	**Annual Income**	**Loan Size**	**Grade**
	≥ 0	≥ 0	VG,G,F,P
R1	[0..500)	[0..1000)	G
R2	[500..1000)	[0..250)	G
R3	[500..1000)	[250..1000)	G
R4	[1000..1500)	[250..1000)	G
R5	[1600..2000)	[0..850)	G
R6	[250..750)	[1200..1500)	G

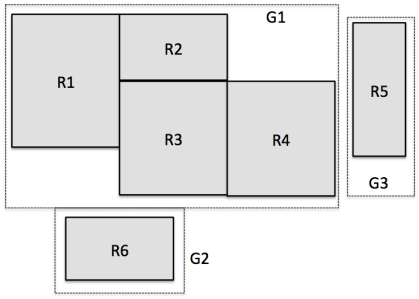


Figure 4: Decision table and its corresponding geometric interpretation

To this aim, we split the hyper-rectangles in the connected component in such a way that, if two hyper-rectangles are adjacent, they fully share one of their sides (as opposed to sharing only part of a side), and we then re-merge the resulting set of hyper-rectangles as much as possible. The first step in this third phase (splitting) is performed by running a sweep-line along each dimension in turn. Every time that the line crosses the projection of a hyper-rectangle over the axis along with the sweep is being performed, we break down the hyper-rectangle into

two parts (left-side and right-side) with respect to the current position of the line. As an example, if we apply this procedure to the rectangles shown in Figure 4 and assuming that all the rules have the same output (and thus can potentially be merged), we obtain the set of rectangles in Figure 5.

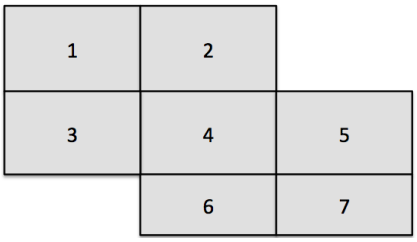


Figure 5: Decision table simplification example

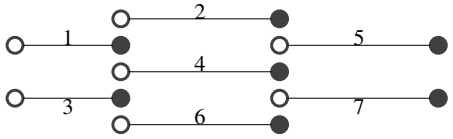
The procedure for re-merging the resulting set of rules is given in Algorithm 2. This procedure ($simplifyRules$) takes as inputs two parameters:

1. ruleList – the list of rules to be merged;
2. N – the total number of input columns in the table.
3. i – index of the column being analyzed. In the initial call to $simplifyRules$, this is 1.

Algorithm 2: Procedure simplifyRules.

```
   Input: ruleList; N; i
1  if i < N then
2      B = {};
3      F = {};
4      sortedListAllBounds = ruleList.sort(i);
5      foreach currentBound in sortedListAllBounds do
6          if currentBound.isIn(ruleList) then
7              if (currentBound.isUpper() && F.isEmpty()) then
8                  B.put(currentBound);
9              else if (currentBound.isLower() && !B.isEmpty() && (areAdjacent(lastBound,
                   currentBound) || (lastBound.value == currentBound.value)) then
10                 F.put(currentBound);
11             else if (!B.isEmpty() && !F.isEmpty()) then
12                 ruleList = mergeRules(ruleList, B, F, i);
13                 B = {};
14                 F = {};
15                 B.put(currentBound);
16             lastBound = currentBound;
17     ruleList = simplifyRules(ruleList, N, i+1);
18 return ruleList;
```

The algorithm starts analyzing the first column (axis x) of the considered connected component ($G1$). In the case of Figure 5, the rules of component $G1$ are projected on x as:



Upper and lower bounds of each interval are sorted in ascending order (line 4) and 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 and the lower bound of I_3^x. Then, the algorithm processes the upper bound of I_1^x. I_1^x is added to list $\mathcal{B}$ (line 8), because the bound processed is an upper bound and $\mathcal{F}$ is empty (line 7). Next, the algorithm processes the upper bound of I_3^x, which is also added to $\mathcal{B}$ (line 8). Then, the lower bound of I_2^x is processed and I_2^x is added to $\mathcal{F}$ (line 10), because the bound processed is a lower bound, it is adjacent to the last bound processed and $\mathcal{B}$ is not empty (line 9). Next, the algorithm processes the lower bound of I_4^x, which is added to $\mathcal{F}$ (line 10), because this is a lower bound with the same value as the previously processed bound and $\mathcal{B}$ is not empty (line 9). Similarly, I_6^x is also added to $\mathcal{F}$. Then, the algorithm processes the upper bound of I_2^x. At this point, the procedure $mergeRules$ (line 12) is invoked to merge rules corresponding to intervals in $\mathcal{B}$ and $\mathcal{F}$ (since all intervals in $\mathcal{B}$ and $\mathcal{F}$ are adjacent).

Procedure $mergeRules$ compares rules corresponding to intervals in $\mathcal{B}$ and rules corresponding to intervals in $\mathcal{F}$ in a pairwise manner. The comparison of two rules is needed to understand if they have the same inputs in all the input columns except for the current one (in which they are adjacent). If the inputs are the same, then the two rules are merged into one. In the example in Figure 5, since $\mathcal{B}$ contains $\{I_1^x, I_3^x\}$ and $\mathcal{F}$ contains $\{I_2^x, I_4^x, I_6^x\}$, rules corresponding to intervals I_1^x and I_2^x, and rules corresponding to I_3^x and I_4^x are merged into one rule (we call the new intervals I_2^x and I_4^x, respectively).

Procedure $mergeRules$ also tries to merge these resulting rules with each other. Two rules are merged if they have the same inputs in all the input columns except for the current one. In our example, rules corresponding to intervals I_2^x and I_4^x are merged (into I_2^x). $mergeRules$ returns $ruleList$, which contains I_2^x, I_6^x, I_5^x, and I_7^x (see Figure 6).

At this point, the algorithm makes $\mathcal{B}$ and $\mathcal{F}$ empty (lines 13, 14). The last processed bound was the upper bound of I_2^x, which is added to $\mathcal{B}$ (line 15). Next, the algorithm processes the upper bound of I_4^x. Since the rule corresponding to

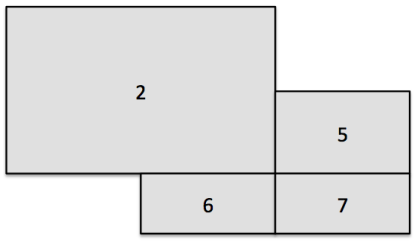


Figure 6: Rules after the first merge

this interval has already been merged with the rule corresponding to I_2^x (the rule is not in $ruleList$ anymore), the algorithm ignores it (line 6). Then, the upper bound of I_6^x is processed and is added to $\mathcal{B}$ (line 8). The algorithm processes the lower bound of I_5^x and adds it to $\mathcal{F}$ (line 10), because the bound processed is a lower bound and is adjacent to the last bound processed (line 9). Next, the lower bound of I_7^x is processed and this interval is added to $\mathcal{F}$ (line 10). At this point, the algorithm processes the upper bound of I_5^x and invokes $mergeRules$ again (line 12).

Again, procedure $mergeRules$ compares rules corresponding to intervals in $\mathcal{B}$ and $\mathcal{F}$ to check if there are rules that can be merged. In our example, $\mathcal{B}$ contains $\{I_2^x, I_6^x\}$ and $\mathcal{F}$ contains $\{I_5^x, I_7^x\}$. Then, rules corresponding to I_6^x and I_7^x are merged into one rule (I_7^x). Therefore, $ruleList$ contains $\{I_2^x, I_5^x, I_7^x\}$. In this case, it is not possible to further merge these rules with each other. Therefore, $mergeRules$ returns $ruleList$, which contains rules $\{I_2^x, I_5^x, I_7^x\}$. The resulting set of rules is shown in Figure 7.

At this point, the algorithm makes $\mathcal{B}$ and $\mathcal{F}$ empty (lines 13, 14). The last processed bound was the upper bound of I_5^x, which is added to $\mathcal{B}$ (line 15). Next, the algorithm processes the upper bound of I_7^x, which is added to $\mathcal{B}$ (line 8). At this stage, no rules can be merged along the current dimension. Since there are no other bounds to be processed along the current dimension, the algorithm starts sweeping the following dimension (line 17). The next dimension in our example is the y axis. All interval projections on y corresponding to rules in Figure 7 are shown in terms of upper and lower bounds below:

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) and processes the

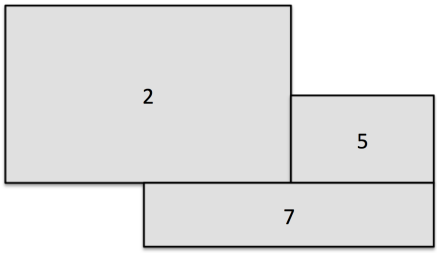


Figure 7: Rules after the second merge

bounds as for dimension x. In this case, there are no rules that can be merged. Therefore, the algorithm finishes and returns $ruleList$ (line 18), which in the running example consists of the rules depicted in Figure 7.

We observe that the end-to-end procedure for table simplification is dominated by the complexity of the third phase, i.e., of Algorithm 2, which has a complexity of $O(d \cdot |R|^2)$, where d is the number of dimensions and $|R|$ is the number of rules in the connected component (which in the worst case is equal to the number of rules in the entire table). This complexity comes from the fact that for each dimension, the algorithm computes the subsets of rules $\mathcal{B}$ and $\mathcal{F}$ and then compares the rules in these sets in a pairwise manner.

In computational geometry, the problem addressed by Algorithm 2 is known as the *optimal rectangulation of a hyper-polygon*, which means decomposing a hyper-polygon into a minimal set of hyper-rectangles. Indeed, if we merge all the hyper-rectangles in a connected component of the adjacency graph, we obtain an iso-oriented hyper-polygon (a.k.a. an orthogonal hyper-polygon). The goal of the simplification procedure is then to decompose this hyper-polygon into a minimal set of hyper-rectangles. As shown in [8], the problem of optimal rectangulation of an orthogonal hyper-polygon is, in general, NP-complete in the case of 3 or more dimensions. Algorithm 2 is hence a polynomial heuristic that attempts to find the best decomposition in a best-first approach based on one single "sweep" across each dimension.

In Algorithm 2, the dimensions are considered in one fixed order. We hereby call this approach *one-permutation* approach since it only considers one possible permutation of the set of dimensions. A more exhaustive approach is to apply Algorithm 2 once for each possible permutation of the dimensions (e.g., we first sweep along the x-axis followed by the y-axis, and then we repeat the procedure the other way around). Each permutation leads to a simplified set of rules. We can then pick the set with the smallest number of rules. In the running example, if we

apply Algorithm 2 starting from the y-axis, we would obtain three rectangles vertically aligned with each other. These rectangles are different from those shown in Figure 7, but the number of rectangles is still the same. In more complex examples however, the order in which dimensions are visited may affect the number of rectangles obtained after the simplification procedure.

The above *all-permutations* approach has a complexity of $O(d! \cdot |R|^2)$, meaning that it is combinatorial in the number of dimensions, but is still polynomial in the number of rules. Below, we empirically evaluate the performance of these one-permutation and all-permutation variants, in terms of their execution times and sizes of the simplified tables they produce.

5. Evaluation

We implemented the algorithms on top of dmn-js: an open-source rendering and editing toolkit for DMN decision tables.[8] 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 at `http://dmn.cs.ut.ee`.

Based on this implementation, we experimentally evaluated the scalability of the proposed algorithms and the conciseness of their output relative to existing baselines. We first discuss the evaluation of the algorithms for overlapping and missing rules detection, followed by that of the table simplification algorithm.

5.1. Evaluation of Overlapping and Missing Rules Detection

Dataset. For this evaluation, we created decision tables from a loan dataset of LendingClub – a peer-to-peer lending marketplace.[9] This dataset contains data about all loans issued in 2014 (235 629 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 [12], 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 decision 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 decision tables containing approx. 500, 1000

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

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

and 1500 rules and 3, 5, 7, 9, 11, 13, and 15 columns (21 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; the 7-dimensional tables have two categorical and five numerical input columns; the 9-dimensional tables have three categorical and six numerical input columns; the 11-dimensional tables have three categorical and eight numerical input columns; the 13-dimensional tables have four categorical and nine numerical input columns; and the 15-dimensional tables have five categorical and ten 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 randomly selected one column, and we injected noise into the input entry of 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.

Execution times. We ran the missing rule and the overlapping rule detection methods on each generated table and measured the 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 Tables 2 and 3. The execution times for overlapping rules are under 2 minutes, except for the 13-columns and 15-columns tables with $1\,500$ rules. Similarly, the execution times for missing rule detection are 1 minute and below, except for the 13-columns and 15-columns tables with $1\,500$ rules. These results suggest that the theoretical exponential complexity of the algorithms we employ does not prevent the analysis of tables with large numbers of rules, provided that the number of columns is relatively small (less than a dozen in the reported experiments). On the other hand, the algorithms do not scale well when confronted to tables with a over a dozen columns and a large number of rules.

Feedback conciseness. In addition to implementing our algorithms, we implemented the 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,

	3 COLUMNS			5 COLUMNS			7 COLUMNS			9 COLUMNS		
#rules	499	998	1 492	505	1 000	1 506	502	1 019	1 496	507	1 012	1 524
overlapping rules	117	503	1 263	160	600	1 370	1 374	2 069	13 405	1 100	4 935	30 554
missing rules	160	611	1 672	163	820	1 942	2 173	7 029	18 263	529	6 557	17 649

Table 2: Execution times (in milliseconds) for tables with 3, 5, 7, and 9 columns and noise 10%

	11 COLUMNS			13 COLUMNS			15 COLUMNS		
#rules	489	1 026	1 484	515	987	1 527	731	1 038	1 514
overlapping time	767	20 934	93 175	1 023	19 641	320 105	11 175	34 387	195 868
missing time	1 083	15 809	65 450	3 116	50 574	316 138	15 598	41 448	704 510

Table 3: Execution times (in milliseconds) for tables with 11, 13, and 15 columns and noise 10%

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 the case of missing rules, Signavio may report multiple missing rules separately even when these missing rules can be merged together. Our approach, instead, merges missing rules that are adjacent to each other into a smaller number of missing rules.

Tables 4–6 show the number of sets of overlapping rules and the number of missing rules identified by our approach vs. the baseline (i.e., the style of output implemented in Signavio). In all runs, both the number of overlapping and missing rules is drastically lower in our approach. Also, the results show a linear growth in the number of sets of overlapping and missing rules produced by our approach, compared to sharp jumps in the case of the baseline (e.g., from 1.2K for the 3×500 table to 10.9K for the 3×1000 one).

		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
	Baseline	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
	Baseline	668	2 655	5 386	563	2 022	4 832	5 201	18 076	43 552

Table 4: Number of reported errors of type "overlapping rules" & "missing rule" for tables with 3, 5, and 7 columns and noise 10%

We decided to inject noise in 10% of the rules in each table since our goal was to stress test the techniques by running them with very large numbers of rules and

		9 COLUMNS			11 COLUMNS		
#rules		507	1 012	1 524	489	1 026	1 484
#overlapping rule sets	Our approach	93	198	341	102	293	364
	Baseline	16 634	71 263	291 978	13 584	238 704	1 011 268
#missing rules	Our approach	126	195	176	106	211	374
	Baseline	3 359	36 398	101 905	5 667	150 256	361 861

Table 5: Number of reported errors of type "overlapping rules" & "missing rule" for tables with 9 and 11 columns and noise 10%

		13 COLUMNS			15 COLUMNS		
#rules		515	987	1 527	731	1 038	1 514
#overlapping rule sets	Our approach	95	212	589	129	211	371
	Baseline	14 683	252 083	1 652 964	121 813	361 389	1 152 632
#missing rules	Our approach	159	290	267	196	192	211
	Baseline	13 360	161 878	551 604	40 245	70 756	254 675

Table 6: Number of reported errors of type "overlapping rules" & "missing rule" for tables with 13 and 15 columns and noise 10%

high numbers of overlapping/missing rules. Indeed, as shown in Tables 4-6, the amount of overlapping/missing rules with this level of noise is already in the order of hundreds in the case of the table with 1500 rules and 15 columns (after merging the overlapping rules into maximal sets). At this level, the execution times reach several minutes. We repeated the same experiments by injecting noise in 15% of the rules in each table. As expected, the execution times were significantly higher (over an hour in the case of tables with 1500 rows and ≥ 11 columns) due to the large number of overlapping and missing rules.

5.2. Evaluation of Table Simplification

Dataset. To evaluate the table simplification approach, we constructed decision tables from the LendingClub dataset using a procedure similar to the one previously outlined. Specifically, we used Weka [12] to train decision trees to classify the grade of each loan from a subset of the loan attributes, and translated each decision tree into a DMN decision table. We tuned the pruning parameters to generate tables containing approximately 100 rules, with 3 input columns (one categorical and two numerical), 5 input columns (two categorical and three numerical), and 7 input columns (two categorical and five numerical).

Given the way decision trees are constructed, there are no rule merging opportunities in the resulting decision tables. Indeed, every time the decision tree learning algorithm splits an internal node into two leaves, it ensures that the class label of one leaf (which will become one rule) is different from the class label of the other leaf (which will become a rule with which it could potentially be

merged). To introduce rule merging opportunities, we inject noise as follows. We randomly select 10% of the rules. For each such rule r, we "enlarge" the range of a randomly selected column as described Section 5.1. As a result of this enlargement, rule r overlaps with other rules. Whenever the enlarged version of rule r overlaps with other rules, we break down these overlapping rules in such a way that the resulting rules do not overlap each other.

This procedure ensures that the rules in the resulting table do not overlap, while at the same time altering the geometric relations between the set of hyper-rectangles induced by the table in such a way that some pairs of hyper-rectangles with the same output are adjacent and hence can potentially be merged. In order to create a higher number of rule merging opportunities, we applied the same procedure as above, but instead of enlarging one column per selected rule, we enlarged 2 columns and 3 columns, respectively. As a result, we obtained nine tables: three with 3 columns, three with 5 columns, and three with 7 columns.

As a baseline, we took the decision table simplification approach proposed for classical decision tables by Pollack [18]. This approach simply selects two rules that have the same output and coincide in all inputs but one. When such a pair of rules is identified, they are merged into a single rule by doing the union of the sets of values in the two cells where the difference occurs (all other cells remain the same in the merged rule). The procedure is repeated until there is no pair of rules that can be merged. This approach was originally designed for tables over categorical domains. To make it applicable to tables with numerical domains, we further require that whenever two cells with numerical values are merged, they should have contiguous ranges. For example, if in one rule the range of the loan amount is [200..300), while in the other rule, the range of the loan amount is [500..1000), then these rules are not merged because the result would not be a single range. On the other hand, if these ranges are [200..300) and [300..1000), then the two rules are merged and the range of this column in the simplified decision table is [200..1000).

Execution times. We simplified each of the nine tables mentioned above and measured the execution times averaged over five runs on a 64-bit 2.6 GHz Intel Core i5-3230M processor with 4GB of RAM. The executions were timed-out after three hours for practical reasons. The results are shown in Table 7. The last three rows in the table show the execution times of: (i) our approach when performing the sweep-line following one single permutation of the set of columns; (ii) our approach when all possible permutations are explored; and (iii) the Pollack's approach. We observe that in all cases, the execution times of our approach (in both

variants) are considerably lower than those of the Pollack's approach. This speed-up can be attributed to the fact that we first compute the connected components in the graph of adjacent rules and then merge along each connected component separately, rather than globally as in the Pollack's approach. As expected, the execution times of the all-permutations variant of our approach are higher than those of the one-permutation variant. For the tables with 7 columns produced by enlarging two and three columns per row, the all-permutations variant reaches the three hours time-out. The Pollack's approach reaches the time-out for the table with 5 columns produced by enlarging three columns per row and for all the tables with 7 columns.

	3 COLUMNS			5 COLUMNS			7 COLUMNS		
#Columns enlarged:	1	2	3	1	2	3	1	2	3
Our approach (one-permutation)	0.52	1.1	5.7	3.5	5.5	30.1	5.6	61.5	774
Our approach (all-permutations)	1.2	2.1	10.3	8.5	13.5	660	6 480	-	-
Pollack	840	720	900	3 480	3 060	-	-	-	-

Table 7: Execution times (in seconds) for table simplification

The results suggest that the proposed rule simplification algorithm is applicable to tables with large number of rules, provided that the number of columns is small (less than 6). However, the algorithm does not scale up well for tables with larger number of columns. Further research is required to design algorithms that would address this latter limitation.

Size of simplified tables. Table 8 shows the number of rules obtained after simplification using both variants of our approach (one-permutation and all-permutations) and using the Pollack's approach [18]. The results show that both variants of our approach produce considerably fewer rules than the Pollack's approach. This is attributable to the fact that our approach tries to merge entire sets of adjacent rules, whereas the Pollack's approach performs local (pairwise) merging. When selecting a given pair of rules for merging, the Pollack's approach disables other possible rule merging opportunities that could lead to rules with larger ranges and hence to a lower overall number of rules. We also observe from the results that the all-permutations variant manages in all cases to reduce the number of rules with respect to the one-permutation, but the difference is relatively marginal. These results suggest that the one-permutation variant (which is more scalable) is sufficient for practical applications.

	3 COLUMNS			5 COLUMNS			7 COLUMNS		
#Columns modified	1	2	3	1	2	3	1	2	3
Our approach (one-permutation)	104	107	96	105	104	119	123	124	158
Our approach (all-permutations)	97	101	90	102	100	110	107	-	-
Pollack	178	178	158	165	163		-	-	-

Table 8: Number of rules after table simplification

6. Conclusion and Future Work

This article presented a formal semantics of DMN decision tables, a definition of decision table correctness based on this semantics, and a framework for analysis and refactoring of decision tables based on a geometric interpretation of the rules composing them. Specifically, the article showed how the rules of a decision table can be seen as iso-oriented hyper-rectangles in an N-dimensional space (where N is the number of input columns).

Given this geometric interpretation, the article outlined algorithms that operationalize two primitive operations for checking the correctness of decision tables, namely detection of overlapping rules and detection of missing rules. We also showed that this interpretation can be used to simplify decision tables by merging rules when they have the same output and the corresponding hyper-rectangles are adjacent (i.e., they share a common side).

The proposed algorithms have been implemented atop the dmn-js toolkit. An experimental evaluation on large decision tables has shown the potential for scalability of the proposed algorithms, their ability to generate concise feedback of overlapping and missing rules, and their ability to produce smaller tables relative to traditional approaches to decision table simplification [18].

The experimental evaluation also showed that the proposed algorithms scale well in practice to tables with a large number of rules. The overlapping and missing rule detection algorithms can handle tables with over a thousand rules and a large proportion of overlapping and missing rules. Similarly, the rule simplification algorithm can handle tables with hundreds of rules. However, the algorithms do not scale well as the number of attributes increases. This lack of scalability is particularly visible for the rule simplification algorithm, which does not scale well to tables with over half a dozen attributes and a few hundred rules. Further research is required to design algorithms that can handle this latter limitation.

The proposed geometric interpretation of DMN decision tables can be applied to design algorithms for other refactoring tasks besides those discussed in this article. For example, we foresee that the proposed geometric interpretation can be applied to transform decision tables with first-hit or priority-hit policies into

equivalent tables with unique-hit policy. Exploring further applications of the proposed general approach is a possible direction for future work.

Finally, the algorithms proposed in this article are limited to DMN decision tables where the expressions are written in S-FEEL (i.e., expressions of the form attribute-operator-literal). Hence, another direction for future work is to extend them in order to support more complex types of expressions supported in the more expressive FEEL version of the DMN standard.

Acknowledgement. This research was supported by the Estonian Research Council (grant IUT20-55) and by the Research Committee of the Free University of Bozen-Bolzano (project *Knowledge-driven ENterprise Distributed cOmputing* – KENDO).

References

[1] Kimon Batoulis, Andreas Meyer, Ekaterina Bazhenova, Gero Decker, and Mathias Weske. Extracting decision logic from process models. In *Proc. of CAiSE*. Springer, 2015.

[2] Ekaterina Bazhenova, Susanne Bülow, and Mathias Weske. Discovering decision models from event logs. In *Proc. of BIS*, volume 255 of *Lecture Notes in Business Information Processing*, pages 237–251. Springer, 2016.

[3] Jon Louis Bentley and Thomas Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Computers*, 28(9):643–647, 1979.

[4] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575?577, 1973.

[5] Diego Calvanese, Marlon Dumas, Ülari Laurson, Fabrizio Maria Maggi, Marco Montali, and Irene Teinemaa. Semantics and analysis of DMN decision tables. In *Proc. of BPM*, pages 217–233. Springer, 2016.

[6] CODASYL Decision Table Task Group. *A Modern appraisal of decision tables : a CODASYL report*. ACM, 1982.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. of TACAS*, pages 337–340. Springer, 2008.

[8] 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.

[9] Herbert Edelsbrunner. A new approach to rectangle intersections part II. *International Journal of Computer Mathematics*, 13(3-4):221–229, 1983.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Co, 1979.

[11] Mohammad Ali Ghodrat, Tony Givargis, and Alexandru Nicolau. Expression equivalence checking using interval analysis. *IEEE Trans. VLSI Syst.*, 14(8):830–842, 2006.

[12] 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.

[13] D. N. Hoover and Zewei Chen. Tablewise, a decision table tool. In *Proc. of COMPASS*, pages 97–108, 1995.

[14] Ron Kohavi and Dan Sommerfield. Targeting business users with decision table classifiers. In *Proc. of KDD*, pages 249–253, 1998.

[15] Rik Maes. An algorithmic approach to the conversion of decision grid charts into compressed decision tables. *Commun. ACM*, 23(5):286–293, 1980.

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

[17] Zdzislaw Pawlak. Decision tables – a rough set approach. *Bulletin of the EATCS*, 33:85–95, 1987.

[18] Solomon L. Pollack. Conversion of limited-entry decision tables to computer programs. *Commun. ACM*, 8(11):677–682, 1965.

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

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

[21] 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.

[22] Jan Vanthienen and Elke Dries. A branch and bound algorithm to optimize the representation of tabular decision processes. Technical Report 9602, Katholieke Universiteit Leuven, 1996.

[23] 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.

[24] Abbas K. Zaidi and Alexander H. Levis. Validation and verification of decision making rules. *Automatica*, 33(2):155 – 169, 1997.