



UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik



Prototyping a Verification Tool for Decision Model and Notation

Masterarbeit

Zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Wirtschaftsinformatik

vorgelegt von

Jonas Blatt

Erstgutachter: Prof. Dr. Patrick Delfmann
Institut für Wirtschafts- und Verwaltungsinformatik

Zweitgutachter: M.Sc. Carl Corea
Institut für Wirtschafts- und Verwaltungsinformatik

Koblenz, im April 2020

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

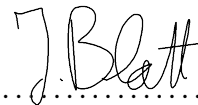
Ja Nein

Mit der Einstellung der Arbeit in die Bibliothek bin ich ein-
verstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich
zu.

Koblenz, 14. April 2020

.....
(Ort, Datum)



.....
(Jonas Blatt)

Abstract

The industry standard Decision Model and Notation (DMN) has enabled a new way for the formalization of business rules since 2015. Here, rules are modeled in so-called decision tables, which are defined by input columns and output columns. Furthermore, decisions are arranged in a graph-like structure (DRD level), which creates dependencies between them. With a given input, the decisions now can be requested by appropriate systems. Thereby, activated rules produce output for future use. However, modeling mistakes produces erroneous models, which can occur in the decision tables as well as at the DRD level. According to the Design Science Research Methodology, this thesis introduces an implementation of a verification prototype for the detection and resolution of these errors while the modeling phase. Therefore, presented basics provide the needed theoretical foundation for the development of the tool. This thesis further presents the architecture of the tool and the implemented verification capabilities. Finally, the created prototype is evaluated.

Zusammenfassung

Der Industriestandard Decision Model and Notation (DMN) ermöglicht seit 2015 eine neue Art der Formalisierung von Geschäftsregeln. Hier werden Regeln in sogenannten Entscheidungstabellen modelliert, die durch Eingabespalten und Ausgabespalten definiert sind. Zudem sind Entscheidungen in graphartigen Strukturen angeordnet (DRD Ebene), die Abhängigkeiten unter diesen erzeugen. Nun können, mit gegebenen Input, Entscheidungen von geeigneten Systemen angefragt werden. Aktivierte Regeln produzieren dabei einen Output für die zukünftige Verwendung. Jedoch erzeugen Fehler während der Modellierung fehlerhafte Modelle, die sowohl in den Entscheidungstabellen als auch auf der DRD Ebene auftreten können. Nach der Design Science Research Methodology fokussiert diese Arbeit eine Implementierung eines Verifikationsprototyps für die Erkennung und Lösung dieser Fehler während der Modellierungsphase. Die vorgestellten Grundlagen liefern die notwendigen theoretischen Grundlagen für die Entwicklung des Tools. Diese Arbeit stellt außerdem die Architektur des Werkzeugs und die implementierten Verifikationsfähigkeiten vor. Abschließend wird der erstellte Prototyp evaluiert.

Contents

1	Introduction	1
1.1	Motivation and Problem Statement	2
1.2	Research Aim	6
1.3	Research Approach	7
1.4	Structure of the Thesis	8
2	Basics	11
2.1	Decision Model and Notation (DMN)	12
2.2	Existing error type classifications	17
2.2.1	Verification and Validation by Vanthienen et al. (1998)	17
2.2.2	Verification Capabilities by Smit et al. (2019)	18
2.2.3	DMN Change Pattern by Hasić et al. (2020)	20
2.2.4	Terminology Mapping	21
2.3	Status Quo on DMN Verification	23
3	Implementation	25
3.1	DMN Verification API	26
3.1.1	Defining the Architecture	27
3.1.2	Overview of important Dependencies	29
3.1.3	Defining REST Endpoints	30
3.1.4	Defining the Verification Result Object	32
3.1.5	Defining the Abstract Verifier	34

3.2	Implementation of DMN Verifiers	36
3.2.1	Decision Requirements Diagram Level Verifiers	37
3.2.1.1	Lonely Data Input	37
3.2.1.2	Missing Input	38
3.2.1.3	Missing Input Column	40
3.2.1.4	Multiple Input Data	41
3.2.1.5	Wrong Data Type	43
3.2.2	Decision Logic Level Verifiers	44
3.2.2.1	Missing Input Value	44
3.2.2.2	Missing Output Value	46
3.2.2.3	Missing Predefined Value	47
3.2.2.4	Unused Predefined Value	48
3.2.2.5	Subsumption Rule	50
3.2.2.6	Identical Rule	54
3.2.2.7	Overlapping Rule	56
3.2.2.8	Partial Reduction	59
3.2.2.9	Missing Rule	61
3.2.2.10	Equivalent Strings	65
3.2.2.11	Empty Output	66
3.2.2.12	Missing Column	67
3.2.3	Syntax Level Verifiers	68
3.2.3.1	Date Format	68
3.2.3.2	Input Value Syntax	69
3.3	Front-end of the DMN Verification Tool	71
3.4	Demonstration	74
4	Evaluation	79
5	Conclusion	85

Bibliography		89
Appendices		93
A	Appendix: Project Information	93
B	Appendix: Implementation	94
B.I	Java Annotation REST Endpoint	94
B.II	JSON Result of Verification Types	95
B.III	JSON Result of Verification Configuration	96
B.IV	JSON Result of Action Types	97
B.V	JSON Result of Action Scopes	97
B.VI	JSON Result of Action Config	98
B.VII	JSON Result of Performance Metrics	99
B.VIII	JSON Result of Verification Request	100
B.IX	Java Example Verifier Implementation	103
C	Appendix: Evaluation	104
C.I	DMN Generator	104

List of Figures

1	Business Process Example	3
2	Business Process Example with Business Rule Task	3
3	Example Decision Table	4
4	Business Rule Management Lifecycle	4
5	Structure of the thesis	9
6	Example of a Decision Requirements Diagram (DRD)	14
7	Example of a decision table (Decision logic)	15
8	Verification and Validation classification by Vanthienen, Mues & Aerts (1998)	18
9	Verification Framework by Smit et al. (2019)	19
10	Terminology mapping of verification capabilities	22
11	Project Architecture	28
12	Result Object UML Class Diagram	33
13	Abstract Verifier UML Class Diagram	35
14	Minimal example – Lonely Data Input	37
15	Minimal example – Missing Input	39
16	Minimal example – Missing Input Column	40
17	Minimal example – Multiple Input Data	42
18	Minimal example – Wrong Data Type	43
19	Minimal example – Missing Input Value	45
20	Minimal example – Missing Output Value	46
21	Minimal example – Missing Predefined Value	48
22	Minimal example – Unused Predefined Value	49
23	Minimal example – Subsumption Rule	50
24	Minimal example – Identical Rule	54
25	Minimal example – Overlapping Rule	56

26	Minimal example – Partial Reduction	59
27	Minimal example – Missing Rule	61
28	Extended example – Missing Rule – Decision table	62
29	Extended example – Missing Rule – Geometric interpretation	62
30	Minimal example – Equivalent Strings	65
31	Minimal example – Empty Output	66
32	Minimal example – Missing Column	67
33	Minimal example – Date Format	69
34	Minimal example – Input Value Syntax	70
35	Front-end overview	71
36	Front-end – Connection to DMN repository	72
37	Demonstration - Initial DMN model	74
38	Demonstration - Overlapping rules	75
39	Demonstration - Missing Input	75
40	Demonstration - Missing Input Value (1/2)	76
41	Demonstration - Missing Input Value (2/2)	76
42	Demonstration - Missing Output	77
43	Demonstration - Missing Predefined Value	77
44	Performance test 1 – Run-time for the analysis of synthetic decision tables with up to 10 columns and 500 rows.	81
45	Performance test 2 – Run-time statistics for the analyzed synthetic decision models with up to 180 nodes on the DRD-level and 100 rules per table.	82

List of Tables

1	Overview of verification capabilities covered by existing approaches	23
2	External Project Dependencies	29

List of Algorithms

1	Lonely Data Input algorithm	38
2	Missing Input algorithm	39
3	Missing Input Column algorithm	41
4	Multiple Input Data algorithm	42
5	Wrong Data Type algorithm	43
6	Missing Input Value algorithm	45
7	Missing Output Value algorithm	47
8	Missing Predefined Value algorithm	48
9	Unused Predefined Value algorithm	49
10	Subsumption algorithm (1/5)	50
11	Subsumption algorithm (2/5)	51
12	Subsumption algorithm (a) (3/5)	52
13	Subsumption algorithm (b) (4/5)	53
14	Subsumption algorithm (c) (5/5)	53
15	Identical Rule algorithm	55
16	Overlapping algorithm (1/3)	57
17	Overlapping algorithm (2/3)	57
18	Overlapping algorithm (3/3)	58
19	Partial Reduction algorithm (1/2)	60
20	Partial Reduction algorithm (2/2)	60
21	Missing Rule algorithm (1/3)	63
22	Missing Rule algorithm (2/3)	63
23	Missing Rule algorithm (3/3)	64
24	Equivalent Strings algorithm	65
25	Empty Output algorithm	67
26	Missing Column algorithm	68

27	Date Format algorithm	69
28	Input Value Syntax algorithm	70

Chapter 1

Introduction

In the context of Business Process Management (BPM), business process models are used to prescribe how to perform company activities. Process models often contain complex decision-logic, governing the allowed execution of activities. Here, instead of including complicated gateways in the process model, decision logic is often abstracted from the process model representation to hide this complexity from the user, by means of linking process models to decision models. Then, during process execution, rule engines can compute how to proceed based on case-specific data and can route the process accordingly. The industry-standard Decision Model and Notation (DMN), which is defined by the Object Management Group (OMG), is a popular formalism to represent such decision models, and is the counterpart to the process modelling standard Business Process Model and Notation (BPMN). DMN decision models can be linked to the BPMN model via so-called business rule tasks, which can trigger the associated decision models, e.g. via DMN rule engines, to handle the company decision logic. However, as decision models are usually maintained by multiple modelers, such DMN models can contain various undiscovered errors, which impedes using the decision models for their intended purpose of governing process execution. Thus, these errors must be found and, if possible, resolved. This thesis therefore investigates a novel tool for the verification of DMN decision models. The following chapter motivates this topic and introduces the scope and structure of this thesis.

1.1 Motivation and Problem Statement

BPMN and DMN are popular standards to model process models, respectively decision models. Many vendors provide technologies, offer modelling- and repository tools. To name two, Signavio¹ or Camunda² supply software, which supports modeling and storing of process- and decision models (Both using the BPMN/DMN standard). Furthermore, Camunda provides a decision engine, which includes a repository for storing and executing these rules. Besides that decision engine, Camunda provides a workflow engine where Business Process Model and Notation (BPMN) models can be executed. As mentioned, BPMNs can con-

¹<https://www.signavio.com/>

²<https://camunda.com/>

tain special activities, named Business Rules Tasks, where the decision engine is used to execute connected DMNs. The decision engine receives the input from the workflow engine, executes the DMN, and returns the output of the executed DMN back to the process. Now the BPMN has processed a potentially complicated activity without using any complex gateway logic. This aspect is visually shown within the two BPMN models in Figure 1 and Figure 2.

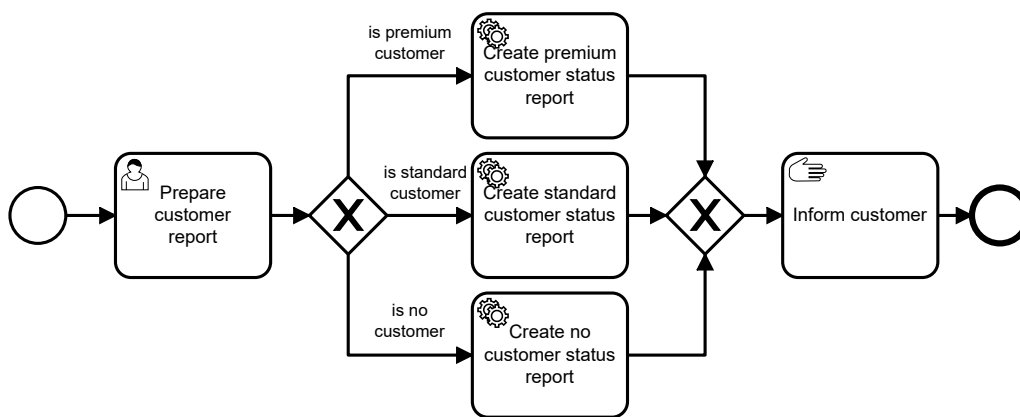


Figure 1 Business Process Example

Instead of explicitly modeling the gateways as in Figure 1, the three service tasks in the middle of the process can be replaced by a Business Rule Task in the process of Figure 2. Here, the (in this simple example) complexity of the gateway logic is resolved by the underlying DMN of the Business Rule Task.



Figure 2 Business Process Example with Business Rule Task

This task is associated with a decision model (or here, more specifically, a decision table), shown in Figure 3.

Decision 1		
Decision_1wpf0nw		
U	Input +	Output +
	customerStatus	report
	string	string
1	"premium"	"Dear special customer, ..."
2	"standard"	"Dear customer, ..."
3	"no customer"	"Dear, ..."
+	-	-

Figure 3 Example Decision Table

The individual table rows shown in Figure 3 can be read as “if (‘premium’), then (‘Dear special ...’)”, and allow to capture the company logic as so-called business rules. In short, a business rule has the form ‘*If A then X*’ and represents a policy or a procedure of an enterprise (Graham & Wiley 2006).

Decision models, respectively the contained business rules, are created and maintained within business rules management (BRM). BRM is a concept for an implemented system based on business rules (Graham & Wiley 2006). A Business Rule Management System (BRMS) stores business rules in repositories where the rule base is separated from data and the control logic (Graham & Wiley 2006). Following Schlosser et al. (2014), a typical BRM lifecycle usually contains the phases of elicitation, authoring, verification, deployment/implementation and monitoring. A possible lifecycle is inspired by these phases and shown in Figure 4.

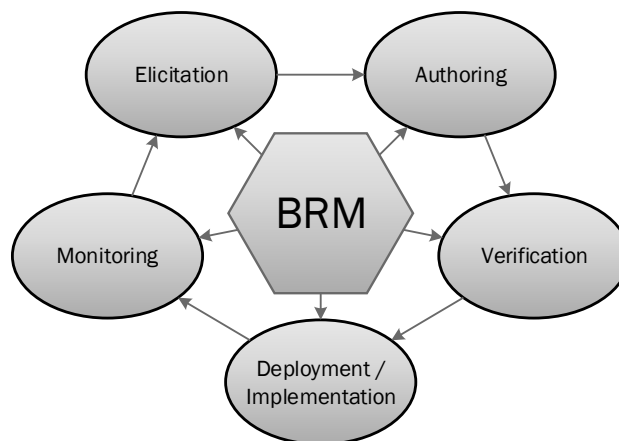


Figure 4 Business Rule Management Lifecycle inspired by Schlosser et al. (2014)

The phases are described by Schlosser et al. (2014) as BRM tasks and are here shortly summarized:

- *Elicitation*. The task in this phase is the requirements analysis for gathering all relevant information. Policies and related rules from various sources have to be identified.
- *Authoring*. The previously identified rules and policies have to be documented. This is done in a formalized way so that they are readable by business users and executable by special systems. The formal notation provides further the possibility for consistency and integrity checks.
- *Verification*. When policies have changed, rules usually must be changed, too. Because of the high risk of new arising inconsistencies, these rules have to be checked for errors. This phase deals with it.
- *Deployment / Implementation*. The rules need to be implemented in processes and deployed to a system that can execute them.
- *Monitoring*. Reporting relevant metrics is the task of this phase. While the execution of business rules, monitoring tools measure the performance and other KPIs for an efficient business behavior.

This thesis focuses on the verification phase within the BRM lifecycle. That is where the existing rules have to be analyzed. This analysis contains verification activities, where existing rules could be checked for inconsistencies or other anomalies. These verifications can also be done while modeling to support the modeler and to avoid mistakes within the rule base.

So, why is the verification of business rules an important issue? Resolving errors in rule bases while modeling before the rules are running in a production system prevents problems in processes of enterprises during run-time. For example, incomplete or inconsistent rules need to be detected and resolved by such a verification tool. The error types investigated in this work (cf. chapter 2 and 3) are not only theoretical, but they also exist in practice. For instance, Batoulis et al. (2017) analyzed 62 decision tables for automated refunding of invoices in a healthcare insurance company. The results of this study revealed that only 8% of the tables were complete, 27% of the rules were unreachable, and in 80% of

the tables exist overlapping rules. Furthermore, Smit et al. (2019) has conducted a survey in Dutch authorities. This confirmed the problems in practical environments, where multiple modelers produce erroneous decision models. Here, missing rules, overlapping rules, and unreachable rules were also found. Previously mentioned modeling tools and other tools support such verification capabilities only at a limited scope or provide no support at all. This aspect is further discussed in section 2.3.

Consequently, there is a need for a verification tool that assists a business rule modeler. This thesis should fill this gap and provide an implementation of such a verification tool. The following section introduces the research aim, which should lead through the thesis.

1.2 Research Aim

The research aim of this work is subdivided into four sub-questions, which scope the concrete chapters of this work and should lead through it. The main research aim **RA** is defined as:

RA:

Implementation of a verification tool for DMN models.

Thus, the goal is prototyping a verification tool for DMN models, which supports modelers in developing decision models in the DMN standard. Moreover, the theory of the prototype should be based on literature. Towards the main research aim, we raise five subsidiary research questions.

SQ1:

What are the major verification types?

SQ2:

How can verification errors be fixed by the verification tool?

SQ3:

How should an architecture for the verification tool look like?

SQ4:

How should the algorithms for the verifications look like?

SQ5:

How can a verification tool be evaluated?

The sub-questions **SQ1** and **SQ2** aim at the point of the theoretical foundation. They should lead the implementation of the prototype to a clear idea of which verification types are needed and what fixes should be provided. Furthermore, sub-question **SQ3** aims at the architecture of the tool, and **SQ4** aims at the used algorithms for the verification types. Finally, the last sub-question **SQ5** aims at the evaluation of the final prototype. Section 1.4 further introduces the structure of this work and arranges these questions into this structure. The following section introduces the research approach.

1.3 Research Approach

Following the presented research aim, this thesis has a design-oriented goal (Becker et al. 2003). Based on this design-oriented focus, the **Design Science Research Methodology** (DSRM), developed by Peffers et al. (2007), is adopted for this work. Fundamental elements of this methodology are earlier defined by Hevner et al. (2004). Furthermore, DSRM consists of six activities (A1 – A6), which are summarized by Hevner & Chatterjee (2010) and here shortly described:

A1 *Problem identification and motivation.*

The first activity includes the definition of the concrete problem and motivates the necessity of a solution, which should be realized by the development of an artifact.

A2 *Define the objectives for a solution.*

The objectives of a potential solution are derived by the problem identification and describe how the new artifact should solve the problem better than existing solutions or to not previous existing problems. This phase implicit defines the requirements, which are the basis for the artifact.

A3 *Design and development.*

The activity contains the description of the architecture, the implementation or creation of the specific artifact, which can either be a model, a method, or another construct, where the design contains the research contribution.

A4 *Demonstration.*

The artifact is used to solve the previously mentioned problems in experiments.

A5 *Evaluation.*

To measure how good the artifact solves the problems is the goal of this activity phase. Results from the demonstration activity can be used here as an input. The measurement is done with the help of metrics and analysis tools and can be quantitative performance measures.

A6 *Communication.*

The final activity in DSRM is to communicate the importance of the problem and the solution in form of the artifact to the publicity.

Consequently, this work also strives after the goal of developing an artifact within the DSRM. The artifact is a prototype of a DMN verification tool, which fulfills the objectives of the research aim. The DSRM activities are arranged in the structure of the thesis in the next section 1.4.

1.4 Structure of the Thesis

Figure 5 shows the coarse structure of this thesis, which is inspired by the previously mentioned research approach. Thus, the DSRM activities are assigned to the corresponding chapters. Also, all research questions are assigned to the corresponding chapters so that there is an overview, where the answers to these questions can be found. While the overall research aim **RA** refers to all chapters, the sub-questions refer to specific chapters. The next chapter 2 provides the theoretical foundation, based on literature and standard definitions and answers questions **SQ1** and **SQ2**. Besides the DMN standard, it further introduces two

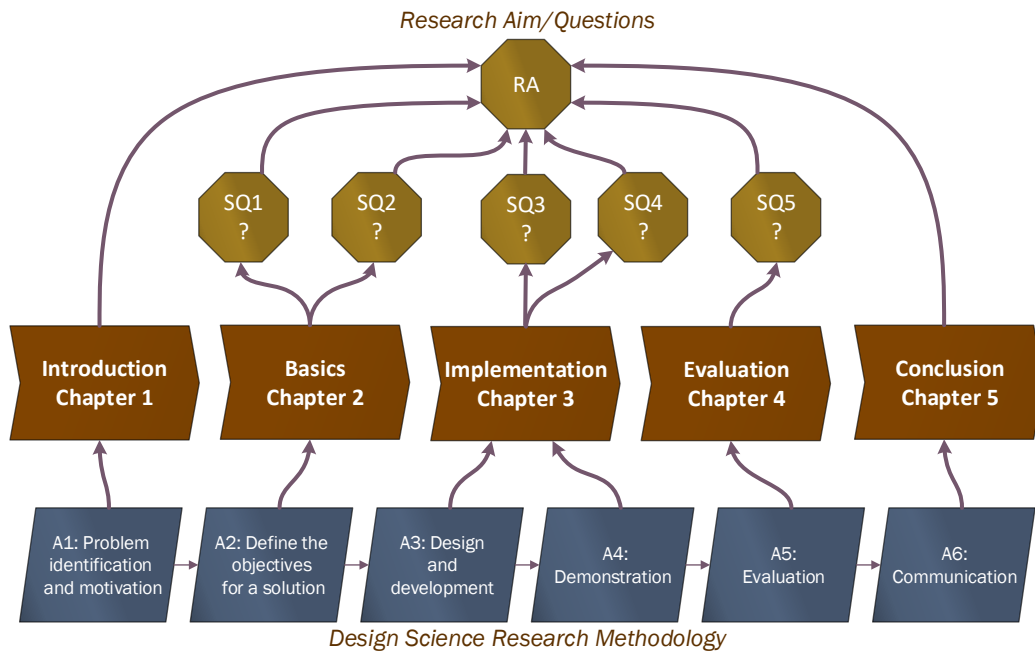


Figure 5 Structure of the thesis

frameworks, which contain theoretical ideas for finding and solving DMN modeling errors. These verification frameworks are partly used in chapter 3 to implement the DMN verification tool, which is the artifact of this work. That chapter firstly describes the general architecture of the prototype and then describe more precisely implementations of concrete realized verifiers. At the end of that chapter, the implementation of a front-end is presented and demonstrated. Therefore, the description of the architecture and the implementation of the verification tool is the answer to questions **SQ3** & **SQ4**. According to the research approach, the developed prototype is evaluated in chapter 4, which answers question **SQ5**. Finally, this thesis gives a conclusion of the whole work in chapter 5, including an outlook for future work.

Chapter 2

Basics

This chapter introduces general basics as relevant background information. After a definition about business rules, this chapter further presents the industry standard Decision Model and Notation (DMN) in section 2.1. The DMN standard is used within the prototype as the model language for business rules which makes it relevant for the implementation part in the next chapter 3. Section 2.2 introduces frameworks and other capabilities of relevant verification types which is the theoretical foundation of the prototype implementation. The last section 2.3 shows an overview about existing tools regarding verification of DMN.

Early definitions of business rules focus more in particular in rules for databases (Graham & Wiley 2006). Definition 1 is based on Morgans's (2002) definition, and Graham & Wiley (2006) extended it to address more than one business. Graham & Wiley (2006) define a business rule as follows:

Definition 1. A business rule is a compact, atomic, well-formed, declarative statement about an aspect of a business that can be expressed in terms that can be directly related to the business and its collaborators, using simple, unambiguous language that is accessible to all interested parties: business owners, business analysts, technical architects or customers. This simple language may include domain-specific jargon.

To use business rules in a business rule management system, they should be well-formed so that they can be executed (Graham & Wiley 2006). Furthermore, rules should be compact and atomic, which means that they "cannot be broken down without the loss of important information"(Graham & Wiley 2006, p.6). Concerning Graham & Wiley (2006) "Business rules are always interpreted against a defined domain ontology". All these aspects of the definition of business rules are realized in the Decision Model and Notation (DMN) standard, which is introduced in the next section.

2.1 Decision Model and Notation (DMN)

DMN is defined by the Object Management Group (OMG)¹ which is an international organisation for defining and developing enterprise standards concerning computer technologies in multiple industries (Object Management Group®

¹<https://omg.org>

2019a). The first version 1.0 of DMN was published in September 2015, version 1.1 in June 2016, and the current version 1.2 in January 2019. DMN provides a notation for defining business rules, where the language is able to express business decisions precisely (Object Management Group® 2019a). Furthermore, this language is understandable and easily readable by all business users (Object Management Group® 2019b). This includes stakeholders from technical developers who are responsible for implementations in information systems, to business people who define requirements and monitor these DMN models (Object Management Group® 2019a). DMNs are represented in XML, and the corresponding XSD supports interchangeability between organizations (Object Management Group® 2019b). The OMG also defines the business process model standard Business Process Management and Notation (BPMN), where DMN can work besides this standard with interfaces and mechanisms for an integrated system (Object Management Group® 2019a). In a BPMN model, a ‘Business Rule Task’ triggers the execution of a decision-making component. DMN provides such a component as a “bridge between business process models and decision logic models” (Object Management Group® 2019b, p.23). The purpose of the DMN is useful in many use cases. For instance, risk assessments could be evaluated with decision tables. Based on ages, salaries and other customer properties in bank institutions, credit limits or creditworthiness can easily be expressed by DMN (Rücker 2016). Other use cases are located at feasibility checks, approvals, validations, fraud detection or calculations.

Camunda published a public model API of the DMN specification 1.1. Furthermore, they provide in their integrated Business Process Management System APIs for accessing and executing DMN models. For that reason, DMN was chosen for representing business rules in the implementation of the prototype.

The relevant levels of DMN are the Decision Requirements Diagram (DRD) and the Decision logic level (DLL), which are precisely defined by Object Management Group® (2019b). They are summarized in the following paragraphs.

Decision Requirements Diagram (DRD). This level of abstraction shows how the elements are graphically arranged and how the relations between the components are structured. The DRD defines the dependencies of the elements. The main components (elements) in the DRD level are: Decision, Input Data,

Business Knowledge Model, Knowledge Source and Decision Service. However, for the scope of this thesis, only decisions and input data elements are relevant. A decision expresses an element that produces an output from a given input with a given decision logic. An Input data element corresponds to information that is used by one or more decisions. These two elements can be connected in a not circular graph with so-called Information Requirement arcs to express the dependencies. Figure 6 shows an example DRD where two Decisions are defined.

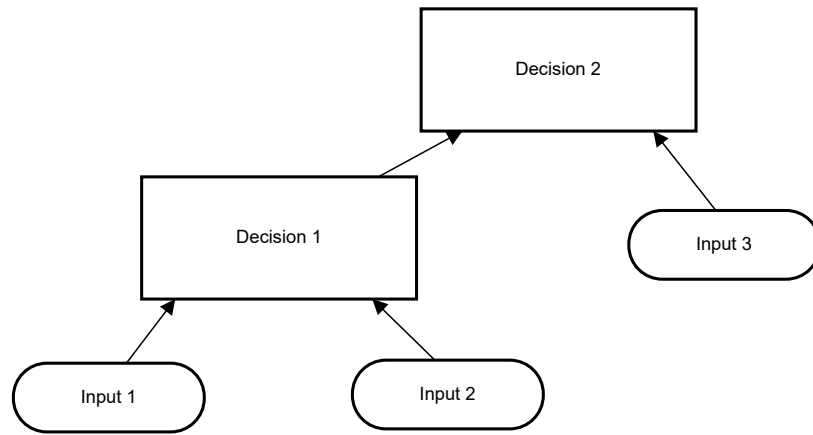


Figure 6 Example of a Decision Requirements Diagram (DRD)

While Decision 1 has two Input data nodes as input, Decision 2 has one Input data node and the output from Decision 1 as input. As seen in the figure, a decision could not only use Input data nodes as input but also outputs from other decisions.

A formal definition (without Business Knowledge Model, Knowledge Source and Decision Service) of a DRD is given as followed:

Definition 2. A Decision Requirement Diagram DRD is a tuple (N, IR, T, f) , where

- N is a set of nodes, where $N = D \cup I$. D is a set of decision nodes and I is a set of input data nodes.
- $IR \subseteq N \times N$ is a set of information requirements (directed arcs), which connects nodes to a non-circular graph:

- (I, D) : An information requirement between an input data node and a decision node
- (D, D) : An information requirement between two decision nodes
- T is a set of names
- f is a function $f : N \rightarrow T$, that maps the names to the nodes

Decision logic. This level describes a decision in a more granular level as a decision table. Decision tables define details about rules and their input and output columns. These rules define which specific input values apply to which output value. This is where the decision logic is defined. The various elements of a decision table is shown in Figure 7.

Decision 1 1			
Decision_0m4bk3a			
U 7	Input + 2		Output + 3
	Input 1	Input 2	Output z
	integer 6	string	string
1	<10	"a"	"z"
4 ²	[10..20] 5	"a"	"y"
3	[10..20]	"b"	"x"
4	>20	"a"	"f"
5	-	"c"	"g"
+	-	-	-

Figure 7 Example of a decision table (Decision logic)

A decision table has a name and a unique id 1 which is important for connecting the decision to a Business Rule Task in a BPMN. Input Columns 2 define the inputs of decision tables. They should correspond to the Input data nodes (or output columns of other decision tables) on the DRD level which are connected with the Information Requirement arc. Output Columns 3 define the outputs of the decision table. Both the input columns and the output columns also have a name. A single rule is defined in a row 4 where concrete values for the input and output columns are defined. The cells are called 'Input/Output entries'. These values are expressed in the so call Friendly Enough Expression Language (FEEL) 5. This expression language places for different data types 6 various

operations. For instance, comparisons or ranges for numerical data types are supported. Supported data types are: strings, integers, longs, doubles, booleans and dates. Columns with strings as data type can further define predefined values. These values are then suggested as input entries or output entries. Furthermore, a decision table has a Hit Policy (7), which further defines how the decision table is executed, or the output is computed. Some hit policies allow overlapping rules (e. g. ‘First’ or ‘Collect’); others do not allow overlapping rules and determine that the decision table is complete. In the scope of this thesis, the hit policy ‘Unique’ is most relevant. Unique means that at maximum, only one rule is allowed to be selected. For deeper insights about DMN, we refer the reader to the full definition of the standard in Object Management Group® (2019b).

Calvanese et al. (2018) provides a formal definition for a decision table. That definition is partly adapted as followed:

Definition 3. A decision table D is a tuple $(T, I, O, Type, R, Order, H)$, where

- T is the table name
- I and O are disjoint, finite sets of input and output attributes (represented as strings)
- $Type : I \uplus O \rightarrow \Gamma$ is a typing function that associates each input/output attribute to its corresponding data type (in Γ)
- R is a finite set of rules $\{r_1, \dots, r_n\}$. Each rule r is a pair $\langle If_i, Then_i \rangle$, where If_i is an input entry function that associates each input attribute $m \in I$ to a condition over $Type(m)$, and $Then_i$ is an output entry function that associates each output attribute $n \in O$ an object in $Type(n)$
- $Order: R \rightarrow \{1, \dots, |R|\}$ is a priority function injectively mapping rules in R to a corresponding rule number defining its priority (important for ‘first’ hit indicator)
- $H \in \{u, f, c\}$ is the hit indicator, where $u = unique$, $f = first$ and $c = collect$

2.2 Existing error type classifications

There are many pieces of research in literature presenting approaches and frameworks for classifying errors in business rules. Some depict more theoretical aspects, and others present concrete implementations of DMN verification tools. For the purpose of a theoretical foundation of the implementation in the next chapter, this section introduces the idea of an early approach by Vanthienen, Mues & Aerts (1998) in section 2.2.1, and two frameworks for business rule verification by Smit et al. (2019) (section 2.2.2) and Hasić et al. (2020b)² (section 2.2.3). At the end, section 2.2.4 lists an overview of major verification types from these frameworks and maps them to the verification types used in this thesis. These verification types should be implemented in the next chapter.

2.2.1 Verification and Validation by Vanthienen et al. (1998)

Vanthienen, Mues, Wets & Delaere (1998) investigated a tool which verifies and validates decision tables in PROLOGA (in the year 1998). For this, they define classifications of decision table anomalies, which are shortly presented in the following. First of all, they differ between ‘Intra-tabular anomalies’ and ‘Inter-tabular anomalies’. Intra-tabular anomalies exist between the components of one single table while Inter-tabular anomalies describe interactions between different tables (Vanthienen, Mues & Aerts 1998). This two levels can be adapted to the DRD level and the decision logic level of DMN. An overview of the anomalies inside these both categories is shown in Figure 8. For instance, in the Intra-tabular level they define redundancy anomalies of rules. The redundancy describe e.g. overlapping rules. Furthermore, they define ambivalence as anomalies, where the conclusion of rules differ even though the inputs are redundant. Moreover, they define on the Inter-tabular level unfirable columns, which means that a column of a decision is not used or not defined as an output column from previous tables. Other anomalies describe an unusable action row where given output values from previous tables do not cover all rows of the current decision table.

²co-author of this paper

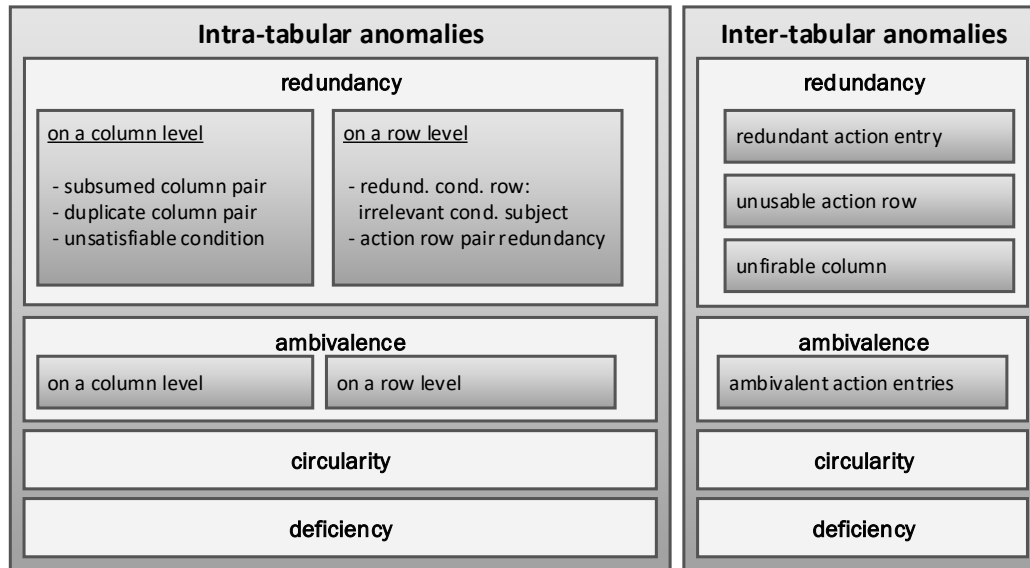


Figure 8 Verification and Validation classification by Vanthienen, Mues & Aerts (1998)

In section 2.2.4 these classifications are mapped to other classifications of the both next sections and to the verification types implemented in this thesis. These authors define more concrete verification types and provide ideas and actions to fix such verifications issues.

2.2.2 Verification Capabilities by Smit et al. (2019)

Smit et al. (2019) conducted a survey in Dutch public administration. Concretely, in that research, they addressed the DMN standard because just a little research was done for the verification steps there Smit et al. (2019). Furthermore, the adoption and use of DMN increases more and more (Smit et al. 2017). Five large Dutch government institutions take part in this round base survey. In focus groups, they discussed possible verification capabilities. The outcome was 28 verification capabilities of a verification framework, which is shown in Figure 9. They subdivide these 28 verification capabilities into four classifications. These categories are shortly described in the following. The Decision requirements level verification category addressed the highest level of abstraction and is related to the DRD level of DMN. For instance, the 'Input verification' capability checks the correctness of decisions inputs. The information requirements (connected input data or connected decisions) of a decision should be equal to the input columns

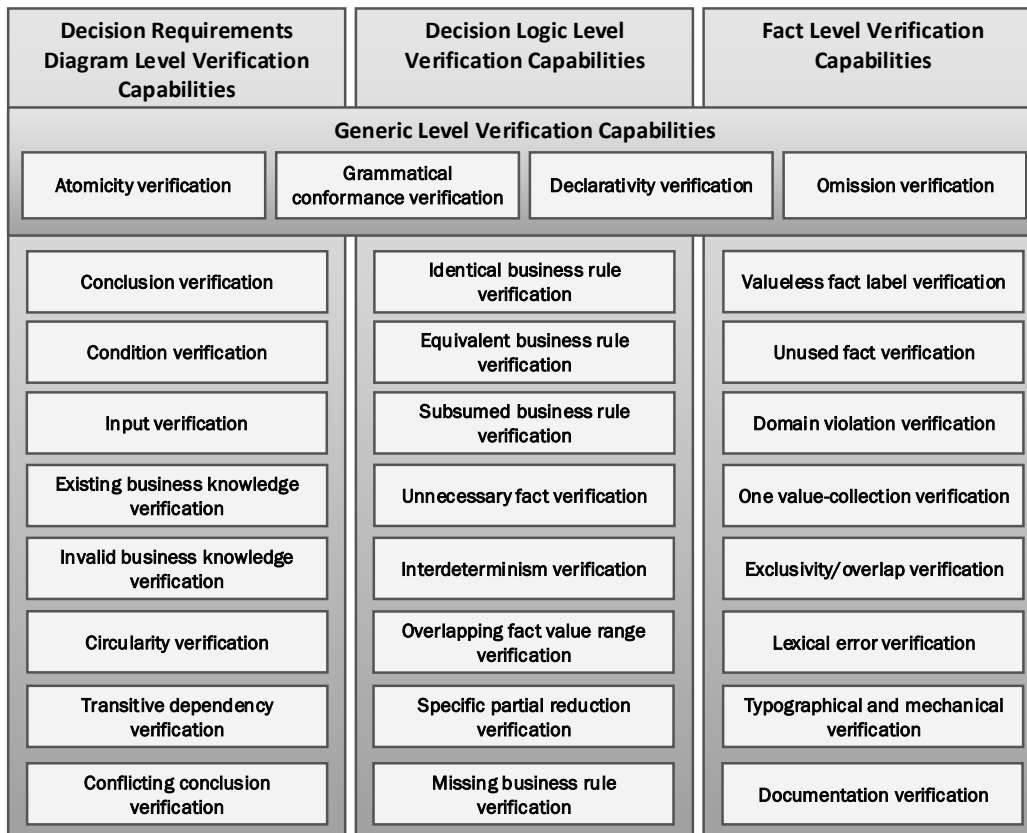


Figure 9 Verification Framework by Smit et al. (2019)

of the decision. Neither more nor less information requirements are allowed. The second category defines verifications on the decision logic level. For example, overlapping rules of a decision table are addressed in one capability. They distinguish between ‘overlapping’, ‘subsumption’, and ‘identical’ rules. While the first-mentioned should find rules where every input entry should be equal, the second one finds rules, where one or more rules completely subsume other rules. Furthermore, ‘Interdeterminism verification’ describes rules where the same input values lead to different conclusions. In the third category, they define verification capabilities on the fact level. The needed facts for these verifications types are present in a running instance. Documentation issues or lexical errors are also part of this category. The fourth classification aims general aspects. The generic level capabilities consider atomic design principle or general syntax checks.

Pre-work for this thesis is a verification tool, which cover the middle column of this framework (Corea et al. 2019). Beside the implementation of these veri-

fiers, this prototype is able to validate multiple tables verifications, and finds for instance overlapping rules, which are defined in different decision tables.

2.2.3 DMN Change Pattern by Hasić et al. (2020)

For the DMN standard, Hasić et al. (2020b) identify and analyse change patterns. Change patterns can occur in a DMN model and describe changes in both, the DRD level and DLL. The problem here is that every change could generate inconsistencies, which may cause problems. For instance, adding a new column in a decision table results in a missing input data node on the DRD level. Moreover, they define actions that should solve the inconsistency. These resolving recommendations are also changes that can trigger further change patterns, which leads to the next resolution actions. This cycle repeats until there are no more inconsistencies. One possible resolving solution for the missing input data from the previous example can easily be to add the missing input data node in the DRD. This knowledge about resolving inconsistencies is important for the subsidiary research questions **SQ2**. All concrete actions to resolve these inconsistencies are described in chapter 3. Based on the change pattern by Hasić et al. (2020b) the following categories of verification categories can be extracted:

- **Inconsistencies of input columns and corresponding information requirements on DRD level.** These information requirements can either be input data nodes or other decisions with corresponding output columns. For instance, an inconsistency exists if an input data node is connected to a decision wherein the decision table exists no matching input column. A resolution action is to delete the input data node or to add a new input column in the decision table.
- **Inconsistencies of output columns which are information requirements for following decisions.** If a decision A is connected to the following decision B and there is no matching input column in B for any output column in A then there exists an inconsistency. A possible solution here is to add a new matching input column in decision table B .
- **Inconsistencies in string-based columns between predefined values and value entries.** Predefined values define the possible values, which can be

used in the entries of the rules. Now, either the entries can use values that are not predefined values, or there can exist predefined values that are never used in the complete column. This leads to inconsistencies that can be resolved.

- **Inconsistencies between input entries of a decision table and output entries of a preceding decision table.** This verification type should check if values that are produced by output entries of a decision are covered in the following decision table. A possible resolution action for a uncovered output entry is to delete the corresponding rule.
- **Inconsistencies between output entries of a decision table and input entries of a succeeding decision table.** This verification type should check if values that are used by input entries of a decision are produced in the preceding decision table. A possible resolution action for a missing input entry is to add a new rule with this value entry.

Finally, these change patterns are great for the modeling phase because these changes occur every time the modeler perform a change action. For instance, if he adds a new rule, a column, a new input data node, or delete an output column. In the next subsection, all the verification types are also mapped to the other capabilities from the previous frameworks. Within the implementation part of the next chapter, these verification types are more precisely described and the implementations of the corresponding resolving recommendations are shown.

2.2.4 Terminology Mapping

The three previously presented verification frameworks differentiate in vocabulary so that a mapping between the relevant elements is needed. As a result, we define names for ‘verification capabilities’ and map the capabilities of the three frameworks to these verification capabilities. This mapping is visualized in Figure 10. At this stage of work, not all verification capabilities are covered. For instance, the DRD level does not allow circularises and Smit et al. (2019) define a verification type that looks at this aspect. Also Vanthienen, Mues & Aerts (1998) defines such a circularity anomaly on the Intra-tabular level. However, we identified the major verification capabilities as shown in Figure 10 which is the an-

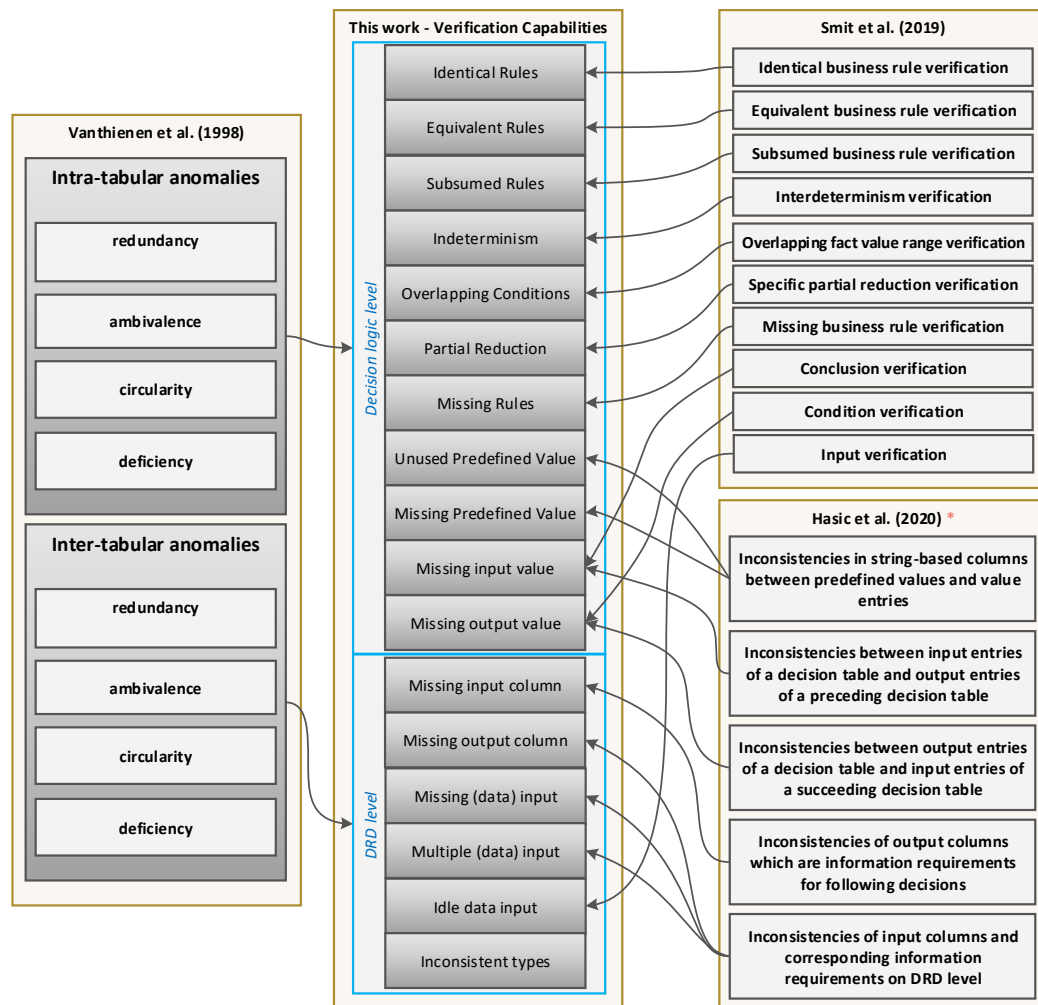


Figure 10 Terminology mapping of verification capabilities; * Co-author, providing a framework of resolving actions

answer to the subsidiary research questions **SQ1**. Future investigations and further research should take a closer look at the missing capabilities. This assumption results in a requirement for the prototype. The architecture of the prototype should be designed in a way that more verification types can be added easily. Such a design has the advantage that also other company- or branch-specific verification types can be implemented. As an example, the verification capability 'Inconsistent types' is not mentioned in any of these frameworks, but is an important aspect for a verification tool. This type should check if an output column has a same datatype to the connected input column of a succeeding decision table.

In this thesis, the term ‘verifier’ describes one implementation of an algorithm that fulfills an entirely or partly aspect of a verification capability. To avoid redundancy, in the next chapter 3, these verifiers are described precisely beside the implementation.

2.3 Status Quo on DMN Verification

Various researchers presented tools that solve errors in rule bases. Corea & Delfmann (2020) conducted a survey and analyzed existing tools solving such errors while design-time, during run-time, or after the execution of the rules. Table 1 shows an overview of some of these tools, where the used rule engine is DMN. As

<i>Literature</i>	Decision logic level										DRD level						
	Identical Rules	Equivalent Rules	Subsumed Rules	Indeterminism	Overlapping Conditions	Partial Reduction	Missing Rules	Unused Predefined Value	Missing Predefined Value	Missing input value	Missing output value	Missing input column	Missing output column	Idle data input	Missing (data) input	Multiple (data) input	Inconsistent types
Calvanese et al. (2016)	X	o	X		X		X										
Laurson & Maggi (2016)	X		X	o	X	o	X										
Batoulis & Weske (2017)	X		X		X		X										
Calvanese et al. (2017)	X	o	X	o	X		X										
Ochoa & González-Rojas (2017)									X	X							
Batoulis & Weske (2018b)	X		X		X		o										
Calvanese et al. (2018)	X	o	X	o	X	o	X										
Corea et al. (2019)*	X	X	X	X	X	X	X										
This work	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Table 1 Overview of verification capabilities covered by existing approaches (X = full support, o = partial support. * = co-authors of this work). Content based on Corea & Delfmann (2020) and Hasić et al. (2020a)

an initial step, this table was created by Corea & Delfmann (2020) and extended in Hasić et al. (2020a). These approaches aim at various modeling mistakes, which were previously presented in section 2.2.4 and defined as verification capabilities. For instance, Calvanese et al. (2016) present a tool, which finds overlapping rules and missing rules, based on a geometric algorithm on decision logic level. This

idea of a geometric interpretation of DMN tables is also used in some algorithms in this thesis. Batoulis & Weske (2018a) implemented a tool which algorithm converts a decision table with overlapping rules and the hit policy 'Rule Order' into a decision table without overlapping rules, with the hit policy 'Unique'. Furthermore, Corea & Delfmann (2018) developed a tool for searching for inconsistencies during process execution. All these tools uses Camunda as implementation provider for the DMN specification. Future approaches may contain more verification capabilities currently not listed in the table.

As can be seen, only a few capabilities are covered by previous approaches. Therefore, to address the missing capabilities, we present in the following a modeling tool, which covers all verification capabilities listed in the table.

Chapter 3

Implementation

As the central part of the thesis, this chapter introduces the implementation of the DMN verification tool, which refers to the sub-research-questions **SQ3** and **SQ4**. First, the architecture of the implementation is explained in section 3.1. Here, project-specific aspects are defined, and conceptual structures for the verification API are outlined. Because of the used programming paradigm ‘Representational State Transfer (REST)’, a web service has to be implemented. This paradigm is often used as a modern and suitable solution approach for distributed systems. Besides the components around the web service, the abstract definition of a verifier is defined (called ‘Abstract Verifier’). Also, the structure for the response, more precisely the output of the verifiers, is defined in this first sub-section. The response object defines what is wrong, which elements are concerned, and if possible, how can the error be fixed.

The second part of this chapter section 3.2 introduces the concrete implementations of the verification capabilities. The previously presented verification capabilities are implemented as so named verifiers. They use the abstract verifier as the basic structure. In addition to the description of the verifiers, the used algorithm is explained. Additionally, the respective resolving actions are presented, too.

Finally, section 3.3 describes a front-end solution for this DMN verification API shortly. This front-end displays a DMN and provides the functionality for editing the model. Furthermore, it uses the previously defined REST service for receiving the errors existing in the DMN. The proposed resolving actions for the model are implemented, too. Section 3.4 then demonstrates the usage of the front-end in a small scenario.

3.1 DMN Verification API

This section introduces the DMN verification API and is structured as followed. First, the general architecture of the whole project is described in section 3.1.1. This architecture gives an overview and depicts the responsibilities and the inner dependencies of the components. After that, the external dependencies required for the project are shortly described in section 3.1.2. As this API provides a web service, the REST endpoints for accessing the API are described in section 3.1.3. Furthermore, the result object for the verification endpoint of the webservice is

described in section 3.1.4. Finally, the abstract verifier, providing a skeleton for the concrete implementation of the verification capabilities, is explained in section 3.1.5.

3.1.1 Defining the Architecture

The foundation of the DMN verification API is the underlying architecture. Some of the verification algorithms may need processing power, so a server providing power as a central solution is suitable. As the API should further provide an interface for multiple clients, a scalable web service at this server is the component that provides the required architecture. Figure 11 shows the general architecture of the complete project. The components of the web service and the implementation of the verifiers is bundled in a Java Maven project, named 'Back-end'. This figure shows the 'Front-end' as a black box, which is later discussed in section 3.3.

Any front-end (e.g. the Camunda DMN Modeler or other DMN modeling tools) can send verification requests ① to the defined REST endpoints (cf. section 3.1.3). In the verification request, the complete DMN is contained as XML format. Other endpoints handle configuration settings or provide information about performance measurements. However, in this figure these aspects are faded out. The Quarkus project fulfills the requirements of a distributed system and provides a Java EE application server. This component has the needed functionalities already included. For instance, it handles HTTP-requests or provides mechanisms for defining the web service endpoints.

The central component of the API is the 'DMN Verification Service'. It is responsible for handling the organizational logic of the API, for instance, calling the requested verifiers. Hence, the endpoint method uses this service and forwards the request and the XML of the DMN model ②. Now, the DMN Verification Service uses the 'DMN Parser', which converts the XML into a Java object using the 'Camunda DMN model API' ③. The 'Verification DMN Model' (VDMN) is a component, which provides additional functionalities, which are later useful for the verifiers. The DMN Parser creates such an object ④, which is returned to the DMN Verification Service ⑤.

At this point, this central component calls the requested verifiers ⑥, which calculate the verification capabilities. These verifiers use the 'Abstract Verifier',

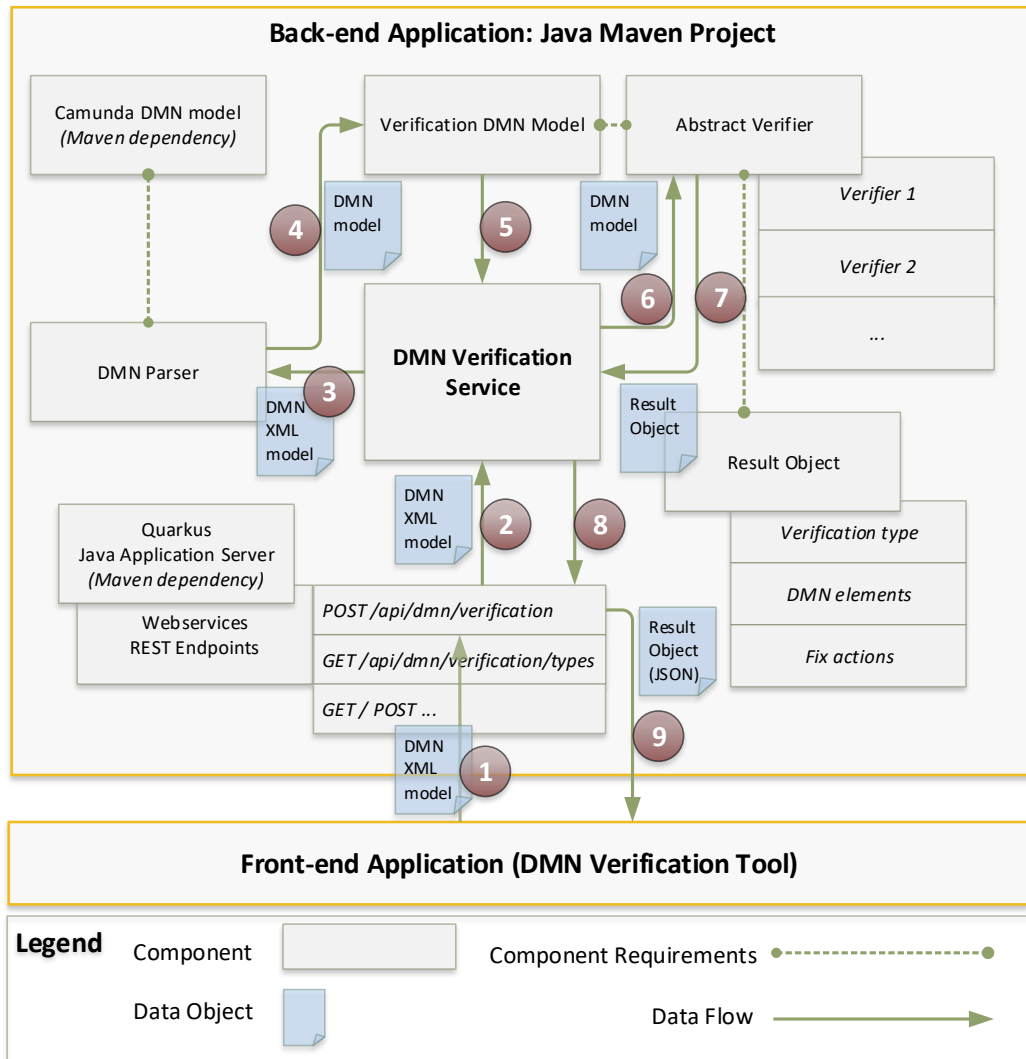


Figure 11 Project Architecture

which provides basic functionalities and interfaces for creating the ‘Result Object’ (cf. section 3.1.5). Moreover, all implemented verifiers are precisely described in section 3.2. Because of the independence of the verifiers, they can be executed in parallel threads to optimize the total execution time. Each verifier adds their verification output to the ‘Result Object’, which is returned to the DMN Verification Service (7). This object contains all the vital information about the identified errors (cf. section 3.1.4). After each verifier is executed, the final result object is now returned to the web service endpoint (8) and then returned to the front-end in text-based JSON format (9).

3.1.2 Overview of important Dependencies

Table 2 shows an overview about the required dependencies.

Table 2 External Project Dependencies

Dependency	Scope	Usage
Quarkus ¹	Back-end (Java / Maven)	This framework contains a Java EE server and has included many required extensions for developing Java REST web services (e.g. port listing, handling JSON, defining Rest Endpoints, handling security aspects, ...).
Camunda, DMN API ²	Back-end (Java / Maven)	This API provides a Java Object representation of the DMN standard. Furthermore, it provides functions to convert the XML representation into a Java Object.
Java Wordnet Interface ³	Back-end (Java / Maven)	This library provides functions for accessing the dictionary 'Wordnet'. It is needed for the verifier 'Equivalent Strings' to find synsets of given words.
JUnit 5 ⁴	Back-end (Java / Maven)	A Java test framework for unit testing.
dmn-js ⁵	Front-end (JavaScript)	Modeling environment framework for DMN models including DRD level and table level.
jQuery ⁶	Front-end (JavaScript)	JavaScript framework for accessing and manipulating HTML documents.

¹<https://quarkus.io/>

²<https://github.com/camunda/camunda-bpm-platform/tree/master/model-api/dmn-model>

³<https://projects.csail.mit.edu/jwi/> - (Finlayson 2014)

⁴<https://junit.org/junit5/>

⁵<https://github.com/bpmn-io/dmn-js>

⁶<https://jquery.com/>

For the back-end a Java Maven⁷ project, is set up. Maven provide a functionality to include external dependencies by just require them inside the Maven 'pom.xml'. These external resources, in the form of frameworks, provide needed functionalities. There is no need for redeveloping those functions because many developers thoroughly test these frameworks. Some of them are mentioned in the previous section. However, for an overview, they are here further described with their scope and their role in this project. Moreover, their usage is shortly described so that they can be classified.

Besides the dependencies for the back-end, the two essential front-end dependencies are listed, too. These two JavaScript frameworks are important for section 3.3.

3.1.3 Defining REST Endpoints

As the web service provides interfaces for the verification of DMN models, the corresponding REST endpoints have to be defined. Quarkus offer a simple annotation construct to define such endpoints which is shown in appendix B.I. We divide the definition of the endpoints into four categories, which are described as followed.

Verifier Information.

First of all, the front-end has to know which verifier types can be requested. For that, a single endpoint is defined, which provides a list of all enabled verifiers. This list contains all verifiers that can be requested by any front-end to calculate the verification errors of DMN models.

```
GET
/api/dmn/verification/types
Produces: application/json
```

The HTTP-GET request does not contain any further parameters and produces a JSON list with the verification types. Beside the name of this verifier, a formatted name, a description and the classification (e.g. DRD or Decision logig) of the verifier is given. An example of the result list is partly given in appendix B.II.

⁷<https://maven.apache.org/>

Verification Request.

The DMN verification can be requested at two different endpoints. First, if the front-end likes to request all verification types, this endpoint is used.

```
POST
/api/dmn/verification?[token={token}]
Consumes: text/xml
Produces: application/json
```

Second, if the front-end just need some verification capabilities, a preselected set of verifiers can be submitted by parameters. Here, just the name of the verifier is sufficient after the 'typeName' parameter name.

```
POST
/api/dmn/verification/types?typeName={a}&typeName={b}.. [&token={token}]
Consumes: text/xml
Produces: application/json
```

Both endpoints require the DMN in XML format as body parameter and produce a JSON result which is described in section 3.1.4. Furthermore, they can receive the optional 'token' parameter which can be used by clients to measure the execution time performance.

Configuration.

A few configuration endpoints provide the possibility to make a few adjustments to the web service. The first endpoint provide a list of all verifiers, that are registered in the current environment. This list contains a string - boolean pair and defines, if a verifier is enabled or not. An example output is listed in appendix B.III.

```
GET
/api/dmn/verification/config
Produces: application/json
```

To enable or disable one verifier, the following endpoint can be called, where 'verifier' is the name and 'enabled' the boolean value.

```
POST
/api/dmn/verification/config/{verifier}/{enabled}
```

Furthermore, verifiers may create fix actions. Fix actions provide a solution for solving the inconsistency, which was found by the verifier. These fix actions are defined with an action type (show / create / update / delete) and an action scope (decision, input data, output column, ..) For instance, an action can suggest cre-

ating a new rule. To get a list of action types this endpoint can be requested (cf. appendix B.IV):

```
GET
/api/dmn/verification/actions/actionTypes
Produces: application/json
```

To get a list of action scopes this endpoint can be requested (cf. appendix B.V):

```
GET
/api/dmn/verification/actions/actionScopes
Produces: application/json
```

To get the configuration for allowed actions scopes and action types this endpoint can be called. An example output is listed in appendix B.VI.

```
GET
/api/dmn/verification/actions/allowedActions
Produces: application/json
```

To set the boolean value to enable or disable the allowed action type / action scope, this endpoint can be called:

```
POST
/api/dmn/verification/actions/allowedActions/scope/type/value
Produces: text/plain
```

Performance Metrics.

If the verification request is called with a token parameter, this endpoint provides a small performance overview of the executions. Besides the total and average execution time, the total and the average number of detected errors are provided.

```
GET
/api/dmn/verification/metrics?token=token
Produces: application/json
```

A part of the JSON result is listed in appendix B.VII.

3.1.4 Defining the Verification Result Object

The verification API has to define a suitable data structure of the verifiers' output. In this context, the output determines the definition of the found errors and possible fix actions of these errors. Simultaneous, the data structure is used as the

output format of the defined verification web service endpoints. Therefore, the result object has to be defined. A class-based structure defines this result object, which is shown in a UML class diagram in Figure 12.

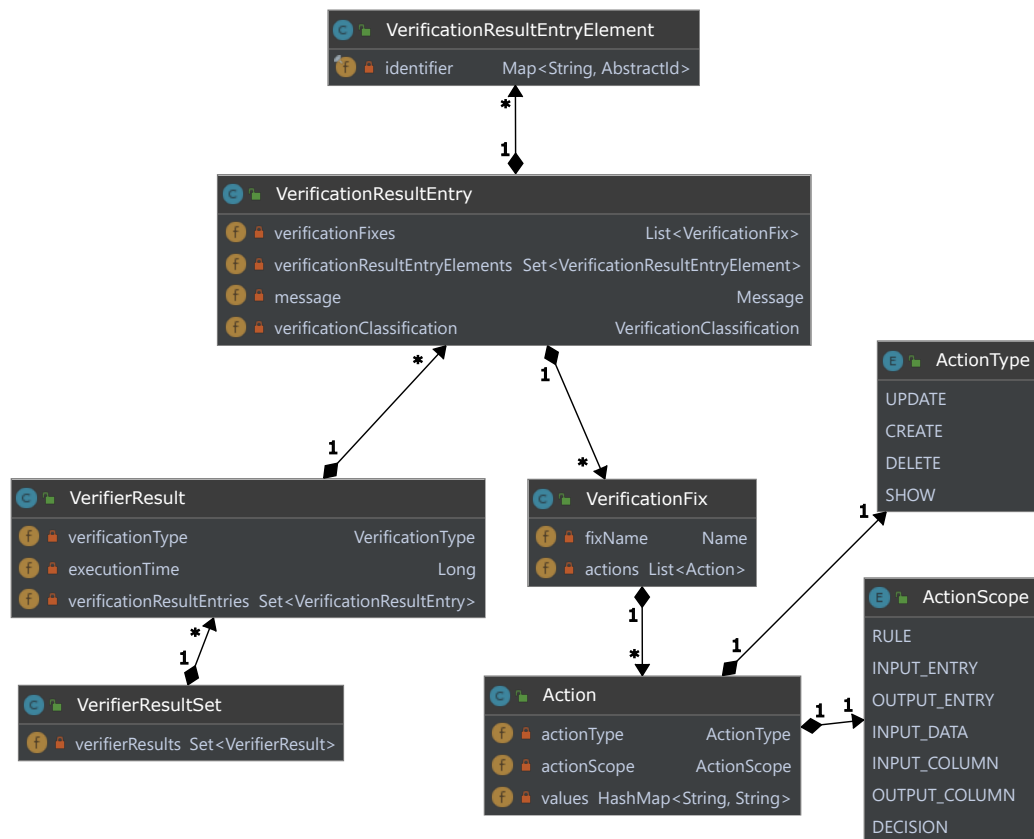


Figure 12 Result Object UML Class Diagram

For the web service endpoint, Quarkus offer the possibility to convert this Java Object automatically to the JSON format, so that there is nothing more to do. This automatic conversation is made by an annotation, which is shown in appendix B.I (Line 8).

In a nutshell, the central architecture component 'DMN Verification Service' creates the root component of the result object, and each activated verifier adds its output to the structure. The root element is the 'VerifierResultSet', which contains only a set of 'VerifierResults'. One 'VerifierResult' represents a container for the output of one verifier. Beside the verification type (name, description, ..), this element contains also the execution time (determines how long the veri-

fier calculated the output) and the set of 'VerificationResultEntries'. A 'VerificationResultEntry' describes one single error found by one verifier. This element contains a human-readable message "There is an error in..." and a set of entry elements. The 'VerificationResultEntryElement' contains a map of strings and Ids. The string value points to a DMN element (e.g. Input data), and the value of the unique Id identifies the specific element. Moreover, the 'VerificationResultEntryElement' contains a 'VerificationClassification' which declare a result entry as 'WARNING' or 'ERROR'. Furthermore, the 'VerificationResultEntryElement' has a list of 'VerificationFixes' included. This list represents all fixes, which can be activated by a user to resolve the error (or display the error in the DMN model). One fix contains a list of one or more actions, which describes what the front-end has to do (enum ActionType) and which element is involved (enum ActionScope). Furthermore, it contains a map of string pairs for further information. Each element has further a unique Id and additionally the number of containing elements (Not shown in this diagram). An example of a JSON formatted representation of this data structure is shown in appendix B.VIII.

3.1.5 Defining the Abstract Verifier

As the verification capabilities are each independent and as they require different algorithms to compute the errors, an appropriate solution is necessary. Furthermore, an easy way to give the verifier a name and a description is needed. For this, the template method pattern in combination with a Java annotation is adapted. This construct is illustrated as an UML class diagram in Figure 13, where besides the 'AbstractVerifier' and the annotation 'DMNVerifier', three concrete verifiers are displayed. The 'AbstractVerifier' class defines fundamental methods and attributes, which are important for the central component 'DMN Verification Services' and the implementation of concrete verifiers. The verification service class call the 'verify' method to generate a new 'VerifierResult' object (cf. section 3.1.4). Moreover, the abstract verifier calculate the execution time of the concrete verifiers by measuring the time for the execution of the 'doVerification()' method. This method is the template method, and the point where the implementation of the algorithm takes part. Other attributes and methods are helpers for the verifier implementation (e.g. access to the DMN model, or methods for creating results). The annotation 'DMNVerifier' defines the name, the descrip-

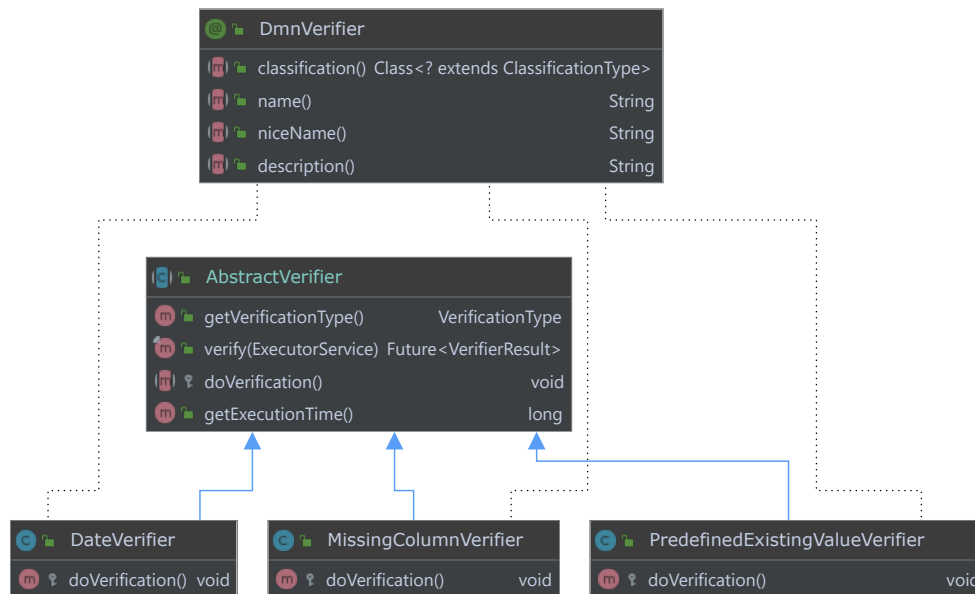


Figure 13 Abstract Verifier UML Class Diagram

tion, and the corresponding classification (DRD level, Decision Logic Level, ..) of a verifier. The attributes of the annotation are then automatically converted into a 'VerificationType' object, which is then handed over to the result object of the verifier. Furthermore, this annotation is used by the DMN verification service class for scanning for verifiers. If a class has this annotation, it is automatically added to the pool of verifiers.

As a result, defining a new verifier requires three actions:

1. Creating a new Class, which extends AbstractVerifier.
2. Adding the annotation 'DmnVerifier' and defining the *name*, the *classification*, the *description*, and a *niceName* for the verifier.
3. Implementing the 'doVerification()' method. There, the found errors are added to the result object.

An example verifier class is shown in appendix B.IX. The implementations of the 'doVerification()' method of all implemented verifiers are discussed in the following section.

3.2 Implementation of DMN Verifiers

Now, as the abstract verifier is defined as an underlying construct, this section describes all implemented verifiers. Each subsection considers a single verifier and is structured as followed. A short description introduces the verifier, which contains the classification of the related verification capability and the designation of the involved DMN elements. Furthermore, each description includes a small minimal example, so that the intention of the verifier is clear. It follows a presentation of a formal algorithm and the corresponding description of the algorithm. Furthermore, feasible verification fixes are introduced. The algorithm represents the 'doVerification()' method as a pseudocode. All defined variables within the algorithms are written lowercase. For an overall consistent naming of the DMN elements, the following shortcuts are defined:

- M* The complete DMN model
- N* One (abstract) Node of a DMN model (decisions & input data)
m.n is a list of *N*, where *m* is a *M*
- D* One Decision of a DMN model
m.d is a list of *D*, where *m* is a *M*
- ID* One Input Data of a DMN model
m.id is a list of *ID*, where *m* is a *M*
- IC* One Input Column of a Decision
d.ic is a list of *IC*, where *d* is a *D*
- OC* One Output Column of a Decision
d.oc is a list of *OC*, where *d* is a *D*
- R* One Rule of a Decision
d.r is a list of *R*, where *d* is a *D*
- IE* One Input Entry of a Rule / Input Column
r.ie is a list of *IE*, where *r* is a *R*
ic.ie is a list of *IE*, where *ic* is a *IC*
- OE* One Output Entry of a Rule / Output Column
r.or is a list of *OE*, where *r* is a *R*
oc.or is a list of *OE*, where *oc* is a *OC*
- PV* One Predefined Value of a string-based column
ic.pv is a list of *PV*, where *ic* is a *IC*
oc.pv is a list of *PV*, where *oc* is a *OC*

DT The Data Type of a column

ic.dt is the data type *DT*, where *ic* is a *IC*

oc.dt is the data type *DT*, where *oc* is a *OC*

IR one Information Requirement between an Input Data or Decision and an other Decision

This section is subdivided by the DMN levels. First, section 3.2.1 defines verifiers for the Decision Requirements Diagrams level, then section 3.2.2 verifiers for the Decision Logic Level, and at the end two further syntax checking verifiers in section 3.2.3.

3.2.1 Decision Requirements Diagram Level Verifiers

This section introduces five verifiers, which are related to the Decision Requirements Diagrams Level. These verifier are: 'Lonely Data Input' (section 3.2.1.1), 'Missing Input Data' (section 3.2.1.2), 'Missing Input Column' (section 3.2.1.3), 'Multiple Input Data' (section 3.2.1.4), and 'Wrong Data Type' (section 3.2.1.5).

3.2.1.1 Lonely Data Input

This verifier detects input data nodes, which has no connection to at least one decision table. On the one hand, this modeling error is trivial. On the other hand, this verification capability offers fixes, which can help the modeler while modeling which is indeed essential. The related verification capability of this verifier is the **Idle data input** capability. For the error detection, only the input data nodes and the information requirements are relevant. However, for the determination of fixes, decision tables and their input columns are relevant, too.

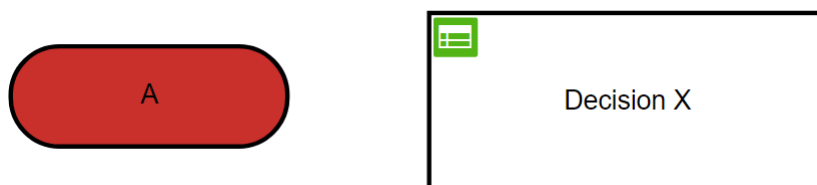


Figure 14 Minimal example – Lonely Data Input

Algorithm. The algorithm is defined as followed:

Algorithm 1 Lonely Data Input algorithm

```

1: function DOVERIFICATION( $M m$ )
2:   for all  $ID id : m.id$  do
3:     if  $id.ir.isEmpty()$  then                                ▷ No information requirement is connected to the input data
4:        $addToResult(id)$                                        ▷ Add input data to result
5:        $createFixes(id)$                                        ▷ Add feasible fixes

```

This verifier iterates over all input data nodes (line 2) and checks for each of these nodes if there exists no information requirement which is connected to the input data node (line 3). If this condition is met, the input data node can be added to the result object (line 4) and the feasible fixes can be created (line 5).

Feasible Fixes. These fixes are created by the presented verifier:

- Show the 'lonely' input data node.
- Delete the 'lonely' input data node.
- Create a new information requirement to a decision, which has a equal naming input column .
- Create a new decision (with a corresponding input column) and a new information requirement between the input data node and the new decision node.

3.2.1.2 Missing Input

This verifier is responsible for detecting input columns of decisions, which has no reference to any input data node. Therefore, this verifier is related to the **Missing (data) input** verification capability. However, in this implementation, this verifier also checks if the input column has simultaneously no connection to any output column of a connected decision, which is related to the **Missing output column** verification capability. This has the reason that input columns, which have an information requirement to an input data node, do not need to have a connected output column and the outer way around. As a result, the relevant elements are the input data nodes, and the (incoming) information requirements with their input data, or decisions as a source.

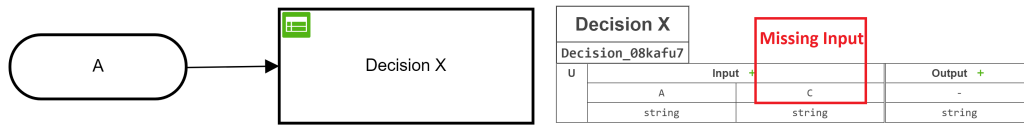


Figure 15 Minimal example – Missing Input

Algorithm. The algorithm is defined as followed:

Algorithm 2 Missing Input algorithm

```

1: function DOVERIFICATION( $M m$ )
2:   for all  $D d : m.d$  do ▷ For each decision
3:      $inNodes \leftarrow d.ir.sourceNodes$  ▷ All incoming nodes (input data & decisions)
4:     for all  $IC ic : d.ic$  do ▷ For each input column
5:        $checkExistingInput(ic, inNodes)$ 
6: function CHECKEXISTINGINPUT( $IC ic, N inNodes$ )
7:   if  $ic.hasName()$  then
8:      $columnName \leftarrow ic.name$ 
9:     for all  $N node : inNodes$  do ▷ Check for each node
10:      if  $node.isInputData() \& node.name = columnName$  then ▷ Names are equal
11:        return ▷ Exit function
12:      if  $node.isDecision()$  then
13:        for all  $OC oc : node.oc$  do ▷ Check for each output column
14:          if  $oc.name = columnName$  then ▷ Names are equal
15:            return ▷ Exit function
16:    $addToResult(ic)$  ▷ Add input column to result
17:    $createFixes(ic)$  ▷ Add feasible fixes

```

First, this verifier iterates over all input columns of all decisions (lines 2 & 5), and then checks for each source nodes of the information requirements, if there exists a related element (lines 3 & 5). In this check, these source nodes are iterated and checked (lines 9 – 15). If a node is an input data node, and the name of this node is equal to the name of the current input column, then there is no error for this input column present (lines 10 & 11). Also, if the node is a decision and has an output column with an equal name with the current input column, then there is no error for this input column present, too (lines 12 – 15). Only if the algorithm passed this check, the input column is added to the result object and the fixes are calculated.

Feasible Fixes. These fixes are created by the presented verifier:

- Show the input column.
- Name an output column of a connected decision with the name of the input column. This output column needs to have no name and the same data type of the input column.

- Rename the input column with the name of a connected input data node or an output column of connected decisions.
- Delete the input column.
- Create a new input data node with the name of the input column and an information requirement for the current decision.
- Create a new decision with a equal named output column and an information requirement for the current decision.
- Create an information requirement from an equal named existing input data node.
- Create an information requirement from an existing decision with an equal named output column.
- Create a new equal named output column in an connected decision.

3.2.1.3 Missing Input Column

The aim of this verifier is finding missing input columns. Hence, **Missing input column** is the related verification capability. The verifier detects input data nodes and decisions, which have an information requirement to decisions with no matching input columns. Therefore, this verifier has two steps. The first step is considering input data nodes and their connected decisions. If the connected decisions have no equal named input column, then there exists an error in the DRD. The second step is considering decisions (A) and their connected decisions (B) ($A \rightarrow B$). If a decision B has no input column, which has an equal name of any output column of a decision A , then there exists an error in the DRD, too.

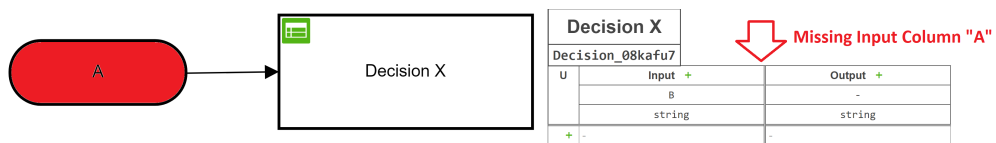


Figure 16 Minimal example – Missing Input Column

Algorithm. The algorithm is defined as followed:

As mentioned, this verifier checks first all input data nodes (lines 2 – 4) and then all decisions (lines 5–7). While the check of the input data requires only the name of the node (line 3), the check for each decision requires all names of the

Algorithm 3 Missing Input Column algorithm

```

1: function DOVERIFICATION( $M m$ )
2:   for all  $ID id : m.id$  do                                     ▷ Check for each input data node
3:      $outNames = singletonList(id.name)$                        ▷ Name of the input data (List of Strings)
4:      $checkInput(id, outNames)$                                ▷ Check the input data node
5:   for all  $D d : m.d$  do                                       ▷ Check for each decision
6:      $outNames = d.oc.names$                                    ▷ Names of all output columns (List of Strings)
7:      $checkInput(d, outNames)$                                ▷ Check the decision node
8: function CHECKINPUT( $N node, List<String> outNames$ )
9:   for all  $D d : node.ir.d$  do                                   ▷ Check for each information providing decision
10:     $inNames = d.ic.names$                                      ▷ Names of all input columns (List of Strings)
11:    if  $outNames.doesNotContainAnyOf(inNames)$  then
12:       $addToResult(node, d)$                                    ▷ Add node and decision to result
13:       $createFixes(node, d)$                                    ▷ Add feasible fixes

```

output columns (line 6). The check itself iterates over all connected (outgoing information requirements) decisions (line 9) with its input column names (line 10), and tests if there exists an equal string pair in both lists (line 11). If this is not the case, the current node (input data or decision) and the connected decision is added to a new result object.

Feasible Fixes. These fixes are created by the presented verifier:

- Show the input data / decision with its connected decision, where the input column is missing.
- Delete the input data node.
- Create a new equal named input column in the connected decision.
- Delete the output column of the source decision.
- Delete the source decision.

3.2.1.4 Multiple Input Data

This verifier is related to the **Multiple (data) input** verification capability and detects input columns that have more than one input data (or output columns from decisions) as information requirement. Hence, the relevant elements are the input columns with the information requirements of the decisions.

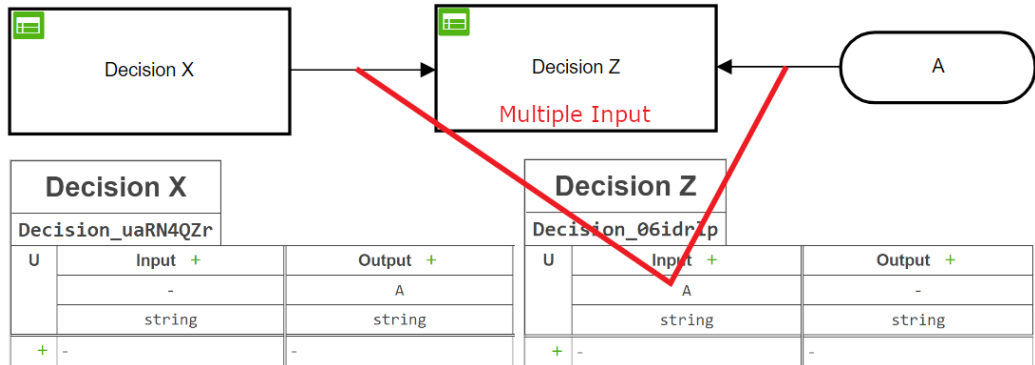


Figure 17 Minimal example – Multiple Input Data

Algorithm. The algorithm is defined as followed:

Algorithm 4 Multiple Input Data algorithm

```

1: function DOVERIFICATION( $M\ m$ )
2:   for all  $D\ d : m.d$  do                                     ▷ For each decision
3:      $inNodes \leftarrow d.ir.sourceNodes$                        ▷ All incoming nodes (input data & decisions)
4:     for all  $IC\ ic : d.ic$  do                                 ▷ For each input column
5:        $checkMultipleInputData(ic, inNodes)$ 
6: function CHECKMULTIPLEINPUTDATA( $IC\ ic, List<N>\ inNodes$ )
7:    $mn \leftarrow \emptyset$                                        ▷ New empty list for matching nodes
8:   for all  $N\ node : inNodes$  do                               ▷ For each node (of the information requirement)
9:     if  $node.isInputData() \& ic.name = node.name$  then
10:       $mn.add(node)$ 
11:     if  $node.isDecision()$  then
12:       for all  $OC\ oc : node.oc$  do                             ▷ For each output column
13:         if  $ic.name = oc.name$  then
14:            $mn.add(node)$ 
15:   if  $mn.size() > 1$  then                                       ▷ Amount of found nodes greater than one
16:      $addToResult(ic, mn)$                                        ▷ Add input column and related elements to result object
17:      $createFixes(ic, mn)$                                        ▷ Add feasible fixes

```

In the first statements, the algorithm iterates over all decisions (line 2) and over all output columns (line 4). The verifier creates a new empty list for caching nodes for later (line 7). Then, the check function iterates over the nodes of incoming information requirements of the current decision (line 8). If the node is an input data, and the name of the input data is equal to the input column name, then the input data node is added to the list (lines 9,10). If the node is a decision, then all output columns are checked for an equal name. Also, here, if the test passed, this decision node is added to the list. Finally, if the size of the list is larger than one, the input column and all detected nodes are added to a new result object.

Feasible Fixes. The only fix that is created by the ‘Multiple Input Data’ verifier is the ‘Show’ action. Here, all elements that are relevant for the verification error need to be highlighted. These elements are the input column and all detected related nodes (input data nodes and decision nodes).

3.2.1.5 Wrong Data Type

The task of this verifier is the detection of output columns and their connected input columns (of connected decisions) where the data types differentiate (**Inconsistent types** verification capability). If an output column produces a string value and a connected input column requires integer values, then it implies a significant modeling error. Relevant elements are all decisions with their information requirements and their input/output columns.

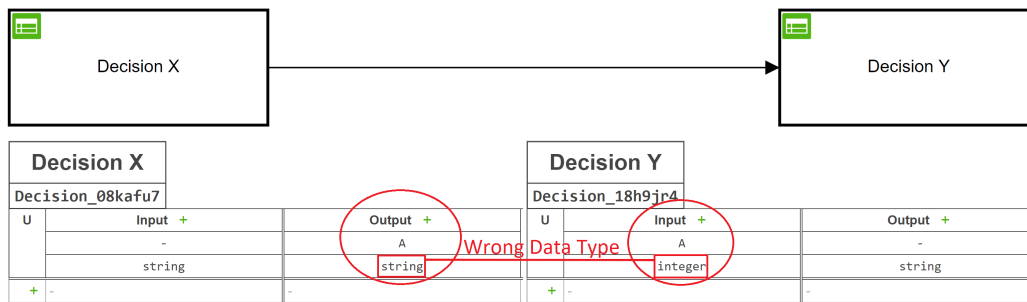


Figure 18 Minimal example – Wrong Data Type

Algorithm. The algorithm is defined as followed:

Algorithm 5 Wrong Data Type algorithm

```

1: function DOVERIFICATION( $M$   $m$ )
2:   for all  $D$   $d$  :  $m.d$  do ▷ For each decision
3:     for all  $D$   $fd$  :  $d.ir.followingDecisions$  do ▷ For each following decision
4:       checkWrongDataTypes( $d$ ,  $fd$ )
5: function CHECKWRONGDATATYPES( $D$   $d$ ,  $D$   $fd$ )
6:   for all  $OC$   $oc$  :  $d.oc$  do ▷ For each output column
7:     for all  $IC$   $ic$  :  $fd.ic$  do ▷ For each input column
8:       if  $oc.name = ic.name$  &  $oc.dt \neq ic.dt$  then ▷ Names are equal and data types are not equal
9:         addToResult( $ic$ ,  $oc$ ) ▷ Add input column and output column to result object
10:        createFixes( $ic$ ,  $oc$ ) ▷ Add feasible fixes

```

First, this verifier iterates over all decisions d (line2) and in an inner loop over the following corresponding decisions fd of the information requirements

(line 3). The check function iterates over all output columns of decision d (line 6) and in an inner loop over all input columns of decision fd (line 7). Then, if the names of both columns are equal, and simultaneously the data types are not equal, then these two columns are added to a new result object of this verifier.

Feasible Fixes. These fixes are created by the presented verifier:

- Show the erroneous output and input column.
- Update one of the columns with a new data type (from the respective other column data type).

3.2.2 Decision Logic Level Verifiers

This subsection presents the verification capabilities of the Decision Logic Level of DMN models. Looking at a single decision table or looking at rules and entries of single tables is the focus of this abstraction level. Therefore, the following verifiers are introduced: Missing Input Value (section 3.2.2.1), Missing Output Value (section 3.2.2.2), Missing Predefined Value (section 3.2.2.3), Unused Predefined Value (section 3.2.2.4), Subsumption Rule (section 3.2.2.5), Identical Rule (section 3.2.2.6), Overlapping Rule (section 3.2.2.7), Partial Reduction (section 3.2.2.8), Missing Rule (section 3.2.2.9), Equivalent Strings (section 3.2.2.10), Empty Output (section 3.2.2.11), Missing Column (section 3.2.2.12).

3.2.2.1 Missing Input Value

The Missing Input Value verifier detects output values of output entries in decision tables which are never used in the connected decision table in input entries. This has especially interests for string columns for ensuring, that all provides values are also used. **Missing input value** is the equal named related verification capability. Relevant elements are the input and output entries and the information requirements to find the connections between the decisions.

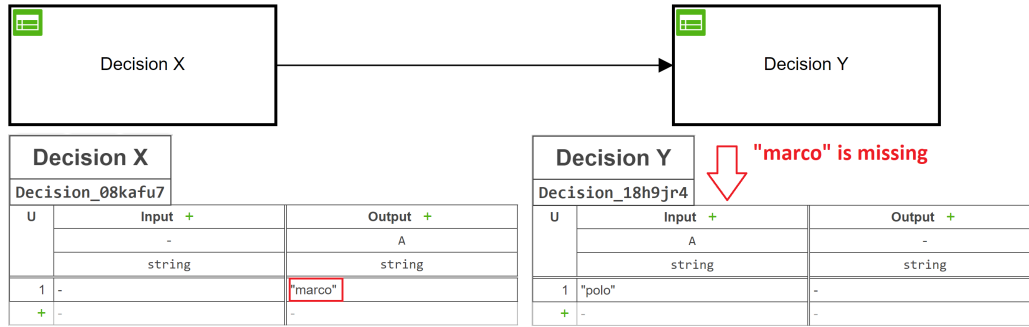


Figure 19 Minimal example – Missing Input Value

Algorithm. The algorithm is defined as followed:

Algorithm 6 Missing Input Value algorithm

```

1: function DOVERIFICATION( $M m$ )
2:   for all  $D d : m.d$  do ▷ For each decision
3:     for all  $D fd : d.ir.followingDecisions$  do ▷ For each following decision
4:       checkMissingInputValues( $d, fd$ )
5:   function CHECKMISSINGINPUTVALUES( $D d, D fd$ )
6:     for all  $OC oc : d.oc$  do ▷ For each output column
7:       for all  $IC ic : fd.ic$  do ▷ For each input column
8:         if  $oc.name = ic.name \& oc.dt = ic.dt$  then ▷ Names are equal and data types are equal
9:           checkColumns( $oc, ic$ )
10:    function CHECKCOLUMNS( $OC oc, IC ic$ ) ▷ Output Column & Input Column
11:      outerLoop:
12:        for all  $OE oe : oc.oe$  do ▷ For each output entry
13:          for all  $OE ie : ic.ie$  do ▷ For each input entry
14:            if  $ie.isInContactWith(oe)$  then ▷ Check if  $ie$  and  $oe$  are in contact
15:              continue outerLoop ▷ Next output entry
16:            addToResult( $ic, oe$ ) ▷ Add input column and output entry to result object
17:            createFixes( $ic, oe$ ) ▷ Add feasible fixes

```

First, this verifier iterates over all decisions d (line 2) and in an inner loop over the following corresponding decisions fd of the information requirements (line 3). Then, the called function iterates over the output columns of the first-mentioned decision (line 6), and in an inner loop over the input columns of the second-mentioned decision (line 7). If a column mapping was found (line 8), all output entries and input entries of both columns are iterated (lines 12, & 13). Here, if the output entire is used in any input entry (line 13 & 14), then there is no existing problem for this output entry, and the next output entry is looked up (line 15). If all input entries are iterated, this output entry is added to a new result object (as missing input entry), and the fixes are created.

Feasible Fixes. These fixes are created by the presented verifier:

- Show the output entry and the input column, where the output entry is not covered.
- Delete the rule, that contains the output entry.
- Add a new rule in the following decision that covers the output entry.

3.2.2.2 Missing Output Value

This verifier provides the opposite of the previously presented verifier. Here, the aim is the detection of input entries in input columns that are not defined in the decision of an information requirement as output value. **Missing output value** is the equal named related verification capability. Relevant elements are the input and output entries and the information requirements to find the connections between the decisions.

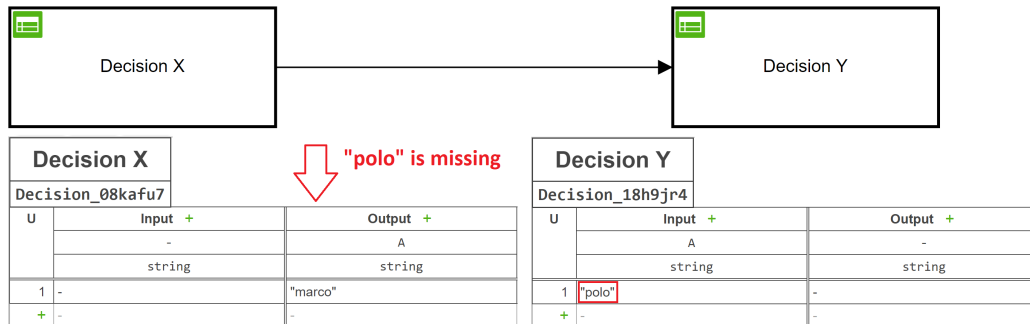


Figure 20 Minimal example – Missing Output Value

Algorithm. The algorithm is defined as followed:

The idea is the same as the previous presented 'Missing input value' in section 3.2.2.1. First, this verifier iterates over all decisions d (line 2) and in an inner loop over the following corresponding decisions fd of the information requirements (line 3). Then, the called function iterates over the input columns of the second-mentioned decision (line 6), and in an inner loop over the output columns of the first-mentioned decision (line 7). If a column mapping was found (line 8), all output entries and input entries of both columns are iterated (lines 12,& 13). However, the loops inside the 'checkColumns' function are reversed, so that the focus is on the missing out values. Here, if the input entire is used in any output

Algorithm 7 Missing Output Value algorithm

```

1: function DOVERIFICATION( $M m$ )
2:   for all  $D d : m.d$  do                                     ▷ For each decision
3:     for all  $D fd : d.ir.followingDecisions$  do             ▷ For each following decision
4:       checkMissingOutputValues( $d, fd$ )
5: function CHECKMISSINGOUTPUTVALUES( $D d, D fd$ )
6:   for all  $IC ic : fd.ic$  do                                 ▷ For each input column
7:     for all  $OC oc : d.oc$  do                               ▷ For each output column
8:       if  $oc.name = ic.name \& oc.dt = ic.dt$  then        ▷ Names are equal and data types are equal
9:         checkColumns( $oc, ic$ )
10: function CHECKCOLUMNS( $OC oc, IC ic$ )                    ▷ Output Column & Input Column
11:   outerLoop:
12:     for all  $OE ie : ic.ie$  do                             ▷ For each input entry
13:       for all  $OE oe : oc.oe$  do                           ▷ For each output entry
14:         if  $oe.isInContactWith(ie)$  then                 ▷ Check if  $oe$  and  $ie$  are in contact
15:           continue outerLoop                             ▷ Next input entry
16:         addToResult( $oc, ie$ )                             ▷ Add output column and input entry to result object
17:         createFixes( $oc, ie$ )                             ▷ Add feasible fixes

```

entry (line 13& 14), then there is no existing problem for this input entry, and the next input entry is looked up (line 15). If all output entries are iterated, this input entry is added to a new result object (as missing output entry), and the fixes are created.

Feasible Fixes. These fixes are created by the presented verifier:

- Show the input entry and the output column, where the input entry is not covered.
- Delete the rule, that contains the input entry.
- Add a new rule in the decision (connected with the information requirement) that covers the input entry.

3.2.2.3 Missing Predefined Value

String columns in DMN models provide the functionality of predefined values. After defining string values in the column head, these values then can be used in the entries of the column. However, not all defined values may be used in the entries. The task of the verifier is the detection of string values that are not defined in the list of predefined values of the column. The related verification capability is the **Missing Predefined Value** capability, and the related elements are the string-based input and output columns with their entries.

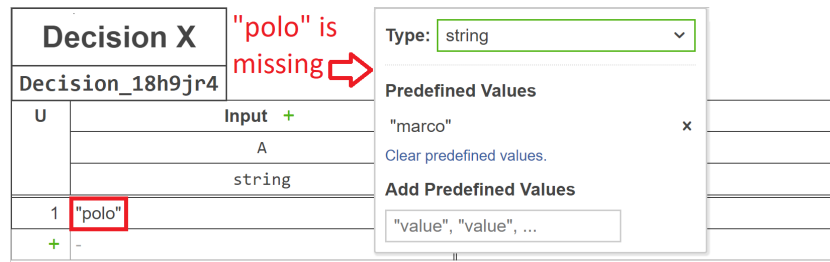


Figure 21 Minimal example – Missing Predefined Value

Algorithm. The algorithm is defined as followed:

Algorithm 8 Missing Predefined Value algorithm

```

1: function DOVERIFICATION( $M$   $m$ )
2:    $stringColumns \leftarrow m.getAllStringColumns()$            ▷ Get all string columns existing in the model
3:   for all  $Column$   $sc$ :  $stringColumns$  do                   ▷ For each (string) column
4:      $predVal \leftarrow sc.getPredefinedValues()$            ▷ Get the predefined values of the column
5:     for all  $Entry$   $entry$ :  $sc.entries$  do                 ▷ For each entry in the string column
6:       if  $!predVal.contains(entry.values)$  then           ▷ Check if the string values are not predefined
7:          $addToResult(entry)$                                ▷ Add the entry to the result object
8:          $createFixes(entry)$                                ▷ Add feasible fixes

```

The verifier requires a list of all string-based columns existing in the complete model. Hence, the abstract function '*getAllStringColumns()*' provides this list (line 2). Then, this list is iterated (line 3), and the list of predefined values of the current column is requested (line 4). In an inner loop, all entries are iterated. If any value of an entry does not exist in the list of predefined values (line 6), the entry is added to the result list.

Feasible Fixes. These fixes are created by the presented verifier:

- Show the entry, where the string value is missing in the predefined values.
- Add the missing string to the predefined values.
- Delete the rule, which contains the missing string value.

3.2.2.4 Unused Predefined Value

The counterpart of the previous presented verifier is the Unused Predefined Value verifier with the equal named **Unused Predefined Value** verification capability. This time, the verifier detects predefined string values, which itself are never used

in entries at the column. Here, the related elements are the string-based input and output columns with their entries, too.

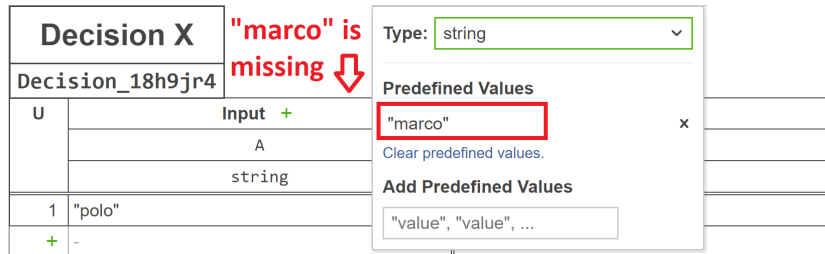


Figure 22 Minimal example – Unused Predefined Value

Algorithm. The algorithm is defined as followed:

Algorithm 9 Unused Predefined Value algorithm

```

1: function DOVERIFICATION( $M$   $m$ )
2:    $stringColumns \leftarrow m.getAllStringColumns()$            ▷ Get all string columns existing in the model
3:   for all  $Column$   $sc : stringColumns$  do                   ▷ For each (string) column
4:      $predVal \leftarrow sc.getPredefinedValues()$            ▷ Get the predefined values of the column
5:     for all  $Entry$   $entry : sc.entries$  do                 ▷ For each entry in the string column
6:       if  $predVal.contains(entry.values)$  then           ▷ Check if the string values is predefined
7:          $predVal.remove(entry.values)$                    ▷ Remove the value from the list of predefined values
8:     for all  $String$   $val : predVal$  do                     ▷ For each remaining predefined value
9:        $addToResult(val)$                                    ▷ Add the string value result object
10:     $createFixes(val)$                                      ▷ Add feasible fixes

```

Like the previous verifier, this verifier requests a list of all string-based columns by the abstract function '*getAllStringColumns()*' (line 2). Then, all string-based columns are iterated (line 3), and the list of predefined values is saved into a new list (line 4). After that, all entries of the column are iterated (line 5). If the values of the entry exist in the list of the predefined values (line 6), they are removed from that list (line 7). Finally, all remaining strings in the list are unused and added to the result object (line 8).

Feasible Fixes. These fixes are created by the presented verifier:

- Show the column with the unused predefined string value.
- Remove the unused value from the list of predefined values.
- Create a new rule, where the unused predefined string value is used.

3.2.2.5 Subsumption Rule

The term ‘subsumption’ indicates the inclusion of values. Accordingly, ‘subsumption rule’ expresses the detection of individual rules which are subsumed by other rules. For instance, rules containing wildcards often render more specific rules unnecessary due to subsumption. Subsumed rules often mean that they are not necessary, and if they have equal output, they can be deleted. As the name of this verifier is saying, it is related to the **Subsumed Rules** verification capability. Moreover, if the output of a subsumed rule differentiates to the subsuming rule, a **Indeterminism** verification capability is detected. The relevant elements are rules of decisions and their input entries. The output entries are relevant for the second mentioned verification capability.

Decision X		
Decision_iVQwkCLc		
U	Input +	Output +
	in	out
	integer	string
1	[0..10]	"a"
2	[2..8]	"b"
+	-	-

Rule 1 subsumes rule 2

Figure 23 Minimal example – Subsumption Rule

Algorithm. The algorithm is defined as followed:

Algorithm 10 Subsumption algorithm (1/5)

```

1: function DOVERIFICATION( $M m$ )
2:   for all  $D d : m.d$  do ▷ For each decision
3:     checkSubsumptions( $d.ic, 0, d.r, false, \emptyset$ ) ▷ → algorithm 11

```

The idea of this algorithm is adapted from Calvanese et al. (2016) and uses an extended version of a line-sweep algorithm for two-dimensional spatial joins which is introduced by Arge et al. (1998). However, this modified algorithm only finds subsumptions and no overlapping or identical rules. In general, each decision table is checked for subsuming rules (algorithm 10 lines 2–3).

Algorithm 11 shows the main construct of the verifier and outlines the complete function ‘checkSubsumptions()’. This algorithm is more complicated than the previous presented. As a result, all parts of the function ‘checkSubsumptions()’ are discussed in their own parts (a → Algorithm 12, b → Algorithm 13 &

Algorithm 11 Subsumption algorithm (2/5)

```

1: function CHECKSUBSUMPTIONS(List<IC> ic,                                ▷ Input columns
                               int i,                                  ▷ Index for current input column
                               List<R> cr,                          ▷ List of current rules
                               boolean hasS,                       ▷ Boolean flag for found subsumption
                               List<R> rootSR)                       ▷ Current subsuming root rules
2:   if i = ic.size() then                                          ▷ All columns processed?
3:     if hasS & cr.size() > rootSR.size() then                    ▷ Is a subsumption present?
4:       dc ← differentConclusions(cr)                               ▷ Check if different output
5:       addToResult(cr, dc)                                       ▷ Add the subsuming rules to result object
6:       createFixes(cr, dc)                                       ▷ Add feasible fixes
7:     else
8:       List sb ← getColumnEntriesByFilteredRules(ic[i], cr)      ▷ List of current entries
9:       List areS ← ∅                                               ▷ List for indication of subsumption
10:      List subC ← ∅                                               ▷ 2-dimensional list for subsumption clusters
11:      List subV ← ∅                                               ▷ 2-dimensional list for subsumption values
12:      if rootSR.isEmpty() then                                     ▷ If subsuming rules are existing
13:        → a → Algorithm 12
14:      else                                                         ▷ No subsumption was found yet
15:        → b → Algorithm 13
16:      removeDuplicates(subC)                                       ▷ Remove duplicate clusters
17:      → c → Algorithm 14

```

$c \rightarrow$ Algorithm 14). This recursive function has five parameters and is called for each decision individually. The first parameter ic contains a list of all input columns of the current decision and is never changed in the function. The index parameter i defines the current index of the column and is responsible for leaving the recursive call stack. The third parameter cr contains a list of rules, which are currently activated. Initially, this list contains all the rules of the decision. Later, this list contains only rules, which are detected as a set of subsuming rules. The boolean parameter $hasS$ indicates, if the current recursive call contains already a subsumption. Finally, the parameter $rootSR$ contains a list of rules, which are the root rules, that subsumes all other founded rules in cr . This list is initially empty.

Inside this function, the first check (line 2) tests the end of the recursion. If i is higher than the size of the list of columns, the next check (line 3) tests the existence of a subsumption. If this is the case, the trivial function ‘differentConclusion()’ (line 4) checks all outputs of the found rules, and these rules are added to the result (line 5, 6).

If the end of the recursion is not reached, four lists are created, which are used in the following parts of the algorithm. First, sb is a list, which contains all column entries of the current column ($ic[i]$) and the current selected rules (cr). Then, $areS$ is a list of boolean values, which indicates if the current index of the following lists is a subsumption or not. Furthermore, $subC$ and $subV$ are 2-dimensional lists,

which contain clusters of rules, where subsumptions were found. Where $subC$ contains all involved rules, $subV$ only contains the subsuming rules. List $subC$ may contain rules, and $subV$ does not contain any rule. Then only identical rules were found yet, which means that there can still exist subsumptions when other columns are considered. Both lists are initially empty. The next part depends on the size of the list $rootSR$. If this list is empty, the next statements are described in algorithm 12, else in algorithm 13. Finally, the recursive call is described in algorithm 14.

Algorithm 12 Subsumption algorithm (a) (3/5)

```

1: for all  $IE\ sbe1 : sb$  do                                ▷ For each current entries
2:    $List\ c \leftarrow \emptyset$                                ▷ Subsumption cluster
3:    $foundSub \leftarrow false$ 
4:   for all  $IE\ sbe2 : sb$  do                                ▷ For each current entries
5:      $fs \leftarrow sbe1.subsumes(sbe2)$                     ▷ Check if  $sbe1$  subsumes  $sbe2$ 
6:      $foundSub = foundSub \vee fs$ 
7:     if  $fs$  then
8:        $c.add(sbe)$ 
9:    $List\ rsl \leftarrow \emptyset$                                ▷ List for subsumption values
10:  for all  $IE\ sbe2 : sb$  do                                ▷ For each current entries
11:    if  $sbe1.isIdentical(sbe2)$  then
12:       $rsl.add(sbe2)$ 
13:      if  $foundSub$  then
14:         $c.add(sbe2)$ 
15:    if  $c.size() > 1$  then                                   ▷ Add cluster to list of clusters
16:       $subC.add(c)$ 
17:       $areS.add(foundSub)$ 
18:      if  $foundSub$  then
19:         $subV.add(rsl)$ 
20:      else
21:         $subV.add(\emptyset)$ 

```

Algorithm 12 is executed if no subsumption was found yet. The whole part iterates over all input entries $sbe1$ of the current rules in sb (line 1). An empty list c for a cluster of subsuming rules is created (line 2), and a boolean flag $foundSub$ is set to false (line 3). Then, in an inner loop over sb (line 4), each input entry $sbe2$ is checked to be a subsumption of $sbe1$ (line 5). If this is the case, $sbe2$ is added to the current cluster (line 8), and the flag of founded subsumption is set to true (line 6). An empty list rsl for the root subsumption rules is created (line 9). Then in a second inner loop, all input entries are iterated again (line 10). Now, if the entries $sbe1$ and $sbe2$ are identical (line 11), the entry $sbe2$ is added the list rsl (line 12). Furthermore, if previously a subsumption was found, the identical entry is added to the current cluster c (line 14). The last step checks if the size of the cluster has more than one element (line 15). If this is the case, the created

cluster c is added to the 2-dimensional list $subC$ (line 16). Furthermore, the flag of the found subsumption is saved for the next recursion step, and if a subsumption was found, the root elements of the found subsumption are saved, too (line 19). If no subsumption was found (only identical rules), an empty list is added (line 21).

Algorithm 13 Subsumption algorithm (b) (4/5)

```

1: for all  $R r : rootSR$  do                                ▷ For each subsumption rule
2:    $entry \leftarrow r.get(i)$                                ▷ Current entry
3:    $List\ rsl \leftarrow \emptyset$                             ▷ List for subsumption values
4:    $List\ c \leftarrow \emptyset$                              ▷ Subsumption cluster
5:    $foundSub \leftarrow false$ 
6:   for all  $IE\ sbe : sb$  do                               ▷ For each current entries
7:      $fs \leftarrow entry.subsumes(sbe)$                    ▷ Check if  $r$  subsumes  $sbe$ 
8:      $foundSub = foundSub \vee fs$ 
9:     if  $fs$  then
10:       $c.add(sbe)$ 
11:     else if  $entry.isIdentical(sbe)$  then
12:       $c.add(sbe)$ 
13:     if  $rootSR.contains(sbe)$  then
14:       $rsl.add(sbe)$ 
15:   if  $c.size() > 1$  then                                ▷ Add cluster to list of clusters
16:      $subC.add(c)$ 
17:      $areS.add(foundSub)$ 
18:      $subV.add(rsl)$ 

```

Algorithm 13 is executed if a previous recursion step found a subsumption. This time, the whole part iterates over the list $rootSR$ (line 1). This list contains rules, which subsumes all rules in the list sr . The variable $entry$ is the input entry of the current rule of $rootSR$ and current column ic (line 2). Like the previous part, two empty lists rsl and c , and a boolean flag $foundSub$, are created. Now, in an inner loop, all entries of sb are iterated (line 6) and checked for a subsumption with $entry$ (line 7). If this is the case, the current entry of sb is added to the cluster (line 10). Furthermore, if an identical rules is detected (line 11), it is also added to the cluster. If the rule is already detected as root rule, it is added to the rsl list (line 14). Like before, if a subsumption is detected, the lists $subC$, $subV$ and $areS$ are filled with the detected rules.

Algorithm 14 Subsumption algorithm (c) (5/5)

```

1: for  $int\ x = 0; x < subC.size(); x++$  do                ▷ For each found cluster
2:    $checkSubsumptions(ic,$                                 ▷ Next column
      $i + 1,$                                               ▷ Next column index
      $subC[x].getAllRules(),$                                ▷ Subsuming rules of the current cluster
      $areS[x] \vee hasS,$                                     ▷ Flag if a subsumption was found
      $subV[x].getAllRules()$                                ▷ Root subsumption rules of current cluster
   )

```

The last part of the verifier is shown in algorithm 14. Here all clusters are iterated (line 1) and for each cluster the function *checkSubsumptions()* is called recursively. The input columns are passed through as they are. However, the index for the current column is incremented. The parameter of the clusters of the current iteration step (the lists *subC* and *subV*) are passed, too. Furthermore, the boolean value *areS*, or is passed *hasS*. However, if the parameter *hasS* was true before, it still remains true.

Feasible Fixes. These fixes are created by the presented verifier:

- Show the rules, which are detected as subsuming rules. The message text of a result entry should contain a description of which rules subsumes other rules. Furthermore, if the output entries are different, an **Indeterminism** verification capability is detected and has to be displayed.
- Delete all subsumed rules, if the output entries of all detected rules are the same.

3.2.2.6 Identical Rule

This verifier is related to the **Identical Rules** verification capability and searches rules, that are identical in their inputs. Meaning, that all detected rules are triggered for each possible input variation at the same time. Detecting rules which have an identical input is essential for detecting redundancies. If the output of the found rules is identical, too, rules can be deleted. If not, there exists an inconsistency so that a **Indeterminism** verification capability is found. The relevant elements are all input and output entries of all rules of all decisions.

Decision X			
Decision_iVQwkCLc			
U		input +	Output +
		in	out
		integer	string
1	=10		"a"
2	=10		"b"
+	-		-

Rules 1 and 2 are identical

Figure 24 Minimal example – Identical Rule

Algorithm. The algorithm is defined as followed:

Algorithm 15 Identical Rule algorithm

```

1: function DOVERIFICATION( $M m$ )
2:   for all  $D d : m.d$  do                                     ▷ For each decision
3:     checkForIdenticalRules( $d.ic, 0, d.r$ )
4:   function CHECKFORIDENTICALRULES( $List<IC> ic,$                 ▷ Input columns
                                      $int i,$                     ▷ Index for current input column
                                      $List<R> cr$ )                ▷ List of rules

5:   if  $i = ic.size()$  then                                     ▷ All columns processed?
6:     addToResult( $cr$ )                                       ▷ Add the rules to result object
7:     createFixes( $cr$ )                                       ▷ Add feasible fixes
8:   else
9:      $List curInVals \leftarrow \emptyset$                        ▷ List for identical rules
10:     $List sortInVals \leftarrow getColumnEntriesByFilteredRules(ic[i], cr)$  ▷ Get all current input entries
11:     $sortInVals.sort()$                                        ▷ Sort list by values of entries
12:     $lastVal \leftarrow null$ 
13:    for all  $IE cv : sortInVals$  do                             ▷ For each sorted input entry  $cv$ 
14:      if  $lastVal \neq null$  then
15:        if  $cv.values.equals(lastVal)$  then                   ▷ Check for equal input entries
16:           $curInVals.add(lastVal)$ 
17:        else
18:          if  $curInVals.size() > 1$  then
19:             $curInVals.add(lastVal)$                            ▷ Add last entry
20:            checkForIdenticalRules( $ic, i + 1, curInVals.rules$ ) ▷ Recursive call
21:             $curInVals \leftarrow \emptyset$ 
22:             $lastVal \leftarrow cv$                              ▷ Set  $lastValue$  to current value  $cv$ 
23:          if  $curInVals.size() > 1$  then
24:             $curInVals.add(lastVal)$                            ▷ Add last entry
25:            checkForIdenticalRules( $ic, i + 1, curInVals.rules$ ) ▷ Recursive call

```

Like the subsumption algorithm, the idea of this verifier is adapted from Calvanese et al. (2016). The verifier checks identical rules for each decision (line 2). The recursive check function is called with three parameters: The input columns ic of the decisions, the index i of the current column, and the list of rules cr . This list initially contains all the rules of the decisions. In further recursion steps, only candidates for identical rules. The first step is the check of the recursion ending (line 5). The algorithm tests if the last column was processed in the previous iteration. If the check is true, the rules are added to the result object, and fixes are created. If the check is false, a new empty list $curInVals$ is created, which is a temporary list. The list $sortInVals$ contains all entries of the current rules and the current column (line 9). This list is sorted by the values of the entries (line 10) so that all identical entries are one after the other. Furthermore, the variable $lastVal$ is initially null and represents the previous entry of the next loop. This loop iterates over the sorted list (line 13) of entries. The first entry is ignored (line 14) but set as $lastVal$ in line 22. Then the current entry is compared with the last entry,

and if they are identical, the *lastVal* is added to the *curInVals* list (line 16). If the entries are not equal and *curInVals* has more than one element (line 18), the function is called recursively with the new identical rules of *curInVals* and the incremented column index. Furthermore, the list *curInVals* is cleared (line 21), because the entries are not identical. After the loop, if *curInVals* contains more than one element, the function is called recursive again, like inside the loop.

Feasible Fixes. These fixes are created by the presented verifier:

- Show the identical rules in their decisions.
- Delete all but one identical rules.

3.2.2.7 Overlapping Rule

The detection of overlapping rules is an essential aspect of finding inconsistencies in decision tables. The term ‘overlapping’ in this context means that at least two rule conditions are at least at one point in contact. This type of error may occur quickly while modeling. If one rule has the condition ‘ ≤ 10 ’ and the modeler creates a second rule with the condition ‘ ≥ 10 ’, then the values 10 is the overlapping part of these two rules. Also, if a decision is complex and has many input columns, this type of error can arise fast. Thereby, it is important that no subsuming or identical rules are found because other verifiers detect them. As a result, this verifier does not show identical rules or subsumptions. **Overlapping Conditions** is the related verification capability. Furthermore, if the overlapping rules produce different outputs, a **Indeterminism** verification capability is found.

Decision X			
Decision_ivQwkCLc			
U	Input +	Output +	
	in	out	
	integer	string	
1	< 10	"a"	
2	> 5	"b"	
+	-	-	

Rule 1 and rule 2 are overlapping

Figure 25 Minimal example – Overlapping Rule

Algorithm. The algorithm is defined as followed:

Algorithm 16 Overlapping algorithm (1/3)

```

1: function DOVERIFICATION( $M\ m$ )
2:   for all  $D\ d : m.d$  do                                     ▷ For each decision
3:     checkOverlapping( $d.ic, 0, d.r, false, \emptyset$ )           ▷ → algorithm 17

```

Like the subsumption algorithm and the identical rule algorithm, the idea of this verifier is adapted from Calvanese et al. (2016). However, this algorithm only finds real overlaps. Meaning, that subsuming or identical rules are deliberately not detected by this verifier. First, the overlaps are checked for each decision in algorithm 16.

Algorithm 17 Overlapping algorithm (2/3)

```

1: function CHECKOVERLAPPING( $List<IC>\ ic,$                                ▷ Input columns
                                $int\ i,$                                ▷ Index for current input column
                                $List<R>\ cr,$                          ▷ List of current rules
                                $boolean\ hasO,$                      ▷ Boolean flag for found overlapping rules
                                $List<R>\ sR$ )                       ▷ Current subsuming rules
2:   if  $i = ic.size()$  then                                       ▷ All columns processed?
3:     if  $hasO$  then                                               ▷ Is a overlap present?
4:        $dc \leftarrow differentConclusions(cr)$                    ▷ Check if different output
5:       addToResult( $cr, dc$ )                                     ▷ Add the overlapping rules to result object
6:       createFixes( $cr, dc$ )                                     ▷ Add feasible fixes
7:     else
8:       → Algorithm 18

```

The parameters of the function ‘checkOverlapping’ in algorithm 17 are similar to the subsumption algorithm. The first parameter ic are all input columns of the current decisions, while i is the index for this column. Furthermore, cr contains all rules, that are marked to be overlapping. The flag $hasO$ marks the current status of the existence of an overlap. Because subsuming rules can also exist, the list sR contains these rules.

The first step of this function is the test of the recursion ending. So, if the index i reaches the size of the column list (line 2), then no more columns can be processed. If this is the case and overlap exists in the list of rules cr (line 3), then these rules are added to the result object. Moreover, if these rules have different conclusions (line 4) an **Indeterminism** verification capability is detected.

The second part of this function is shown in algorithm 18. An empty list $curOvVals$ is created to cache the overlapping rules for the next recursive calls. Next, the input entries $sortInVals$ of the current rules and current input column

Algorithm 18 Overlapping algorithm (3/3)

1: <i>List</i> <i>curOvVals</i> $\leftarrow \emptyset$	▷ List for identical rules
2: <i>List</i> <i>sortInVals</i> \leftarrow <i>getColumnEntriesByFilteredRules</i> (<i>ic</i> [<i>i</i>], <i>cr</i>)	▷ Get all current input entries
3: <i>sortInVals.sort</i> ()	▷ Sort list by values of entries
4: <i>z</i> $\leftarrow 0$	
5: <i>foP</i> \leftarrow <i>false</i>	▷ Flag for ‘found overlap’ for prev. iteration
6: for all <i>IE</i> <i>ie</i> : <i>sortInVals</i> do	▷ For each input entry
7: <i>z</i> \leftarrow <i>z</i> + 1	
8: <i>fo</i> \leftarrow <i>hasO</i>	▷ Flag for ‘found overlap’
9: <i>List</i> <i>curOvValsCp</i> \leftarrow <i>curOvVals</i>	▷ Copy of the list
10: <i>curOvVals.add</i> (<i>ie</i>)	
11: <i>fnic</i> \leftarrow <i>false</i>	▷ Flag for ‘found not in contact’
12: for <i>int</i> <i>x</i> = 0; <i>x</i> < <i>curOvVals.size</i> (); <i>x</i> = <i>x</i> + 1 do	
13: if <i>ie.isNotInContact</i> (<i>curOvVals</i> [<i>x</i>]) then	▷ Are both entries not in contact?
14: <i>curOvVals.remove</i> (<i>x</i>)	▷ Remove the entry at <i>x</i>
15: <i>fnic</i> \leftarrow <i>true</i>	
16: else	
17: <i>fo</i> \leftarrow <i>fo</i> <i>ie.isOverlapping</i> (<i>curOvVals</i> [<i>x</i>])	▷ Is overlapping?
18: if <i>fnic</i> & <i>curOvValsCp.size</i> () > 1 then	
19: <i>List</i> <i>nSR</i> \leftarrow <i>searchSubsumingElements</i> (<i>curOvValsCp</i>)	
20: <i>checkOverlapping</i> (<i>ic</i> , <i>i</i> + 1, <i>curOvValsCp.rules</i> ,	▷ Recursive call
<i>hasO</i> <i>fo</i>	▷ Existing overlap? Or..
<i>noDuplicateRules</i> (<i>nSR</i> , <i>sR</i>),	▷ .. subsuming rules changed
<i>nSR</i>)	▷ New subsuming rules
21: if <i>z</i> = <i>sortInVals.size</i> () & <i>curOvVals.size</i> () > 1 then	
22: <i>List</i> <i>nSR</i> \leftarrow <i>searchSubsumingElements</i> (<i>curOvVals</i>)	
23: <i>checkOverlapping</i> (<i>ic</i> , <i>i</i> + 1, <i>curOvVals.rules</i> ,	▷ Recursive call
<i>hasO</i> <i>fo</i>	▷ Existing overlap? Or..
<i>noDuplicateRules</i> (<i>nSR</i> , <i>sR</i>),	▷ .. subsuming rules changed
<i>nSR</i>)	▷ New subsuming rules
24: <i>foP</i> \leftarrow <i>foP</i> <i>fo</i>	

are requested (line 2). These entries are sorted by their values (line 3). The variable *z* (line 4) is a counter variable and incremented each iteration (line 7). The variable *foP*, which caches the status of a found overlap in a previous iteration, is initially set to false (line 5). The loop (line 6) iterates over all sorted entries in *sortInVals*. The variable *fo* indicates a found overlap in this iteration step and is set to the value of *hasO* (line 8). A list *curOvValsCp* is created as a copy of the list *curOvVals* (line 9), and then the current input entry is added to the list *curOvVals* (line 10). The variable *fnic* is a flag for indicating that the current entry has no contact with any element of the newly created list *curOvValsCp*. This aspect is checked in the inner loop (line 12). Here all entries of *curOvVals* are iterated and checked for ‘no contact’ (line 13). If this is the case, the compared value is removed from *curOvVals* and *fnic* is set to true (line 15). If this is not the case, the flag *fo* is set to true if an overlap to the current entry is present (line 17).

After the loop, it is checked whether the function should call itself. This is the case when *fnic* is true, and if the list *curOvValsCp* contains more than one ele-

ment (line 18). Also, when the last entry was processed, and the list *curOvVals* contains more than one element (line 21), the function calls itself. However, a new list *nSR*, which contains all subsuming entries of the current entries, is created before. This list is essential for finding only overlapping rules, which are no subsumptions. The *hasO* variable is true for the next step if it was previously true, an overlap was found, or the subsuming rules differentiate to previous subsumption entries. Now, the next column of the decision table is processed, and the algorithm starts in algorithm 17 again.

Feasible Fixes. The only reasonable fix for this verifier is the ‘Show’ fix. It enables a lookup and fast finding of the overlapping rules in the DMN model for the user. To combine overlapping (if possible), identical, or subsuming rules, the following verifier Partial Reduction provides the fixes to do that.

3.2.2.8 Partial Reduction

This verifier provides a simplification fix by searching rules, that can be combined. It checks whether ranges can be combined to simplify decision tables. Subsuming, overlapping or identical rules, presented before, may be candidates to be part of this verifier. An essential aspect of this verifier is that the output of possible rule combinations has to be identical. For instance, if a rule specifies an integer entry to be lower than 100 (< 100) as input, and another rule with the same output specifies the input entry to be between 100 and 200 ($[100..200]$), then these two rules can be combined as a new rule with an input entry < 200 . The relevant elements are all input and output entries of all rules of all decisions. **Partial Reduction** is the related verification capability.

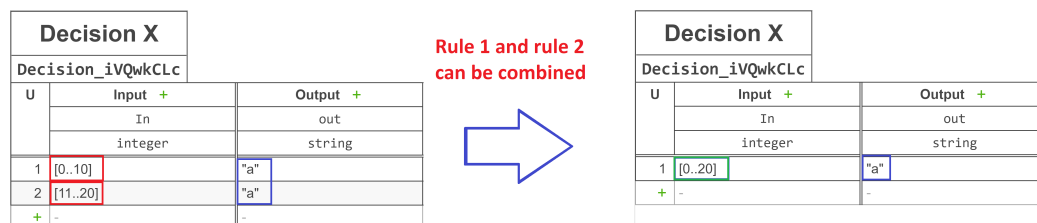


Figure 26 Minimal example – Partial Reduction

Algorithm. The algorithm is defined as followed:

Algorithm 19 Partial Reduction algorithm (1/2)

```

1: function DOVERIFICATION( $M\ m$ )
2:   for all  $D\ d : m.d$  do                                     ▷ For each decision
3:      $List\ ior = getIdentialOutputRulePairs(d)$              ▷ Nested list of rule pairs with identical outputs
4:     for all  $List<R>\ rules : ior$  do
5:        $findPartialReduction(d.ic, 0, rules, false, null)$    ▷ → algorithm 20

```

The first part of the verifier is described in algorithm 19. This verifier checks for each decision in the model if there exist rules that can be combined. Therefore, the first step is to iterate over these decisions (line 2) and create a nested list of rule pairs. The function *getIdentialOutputRulePairs()* generates such a list, where rule pairs are received that have identical output. For each of these rule pairs the function *findPartialReduction()* is called, which is described in algorithm 20.

Algorithm 20 Partial Reduction algorithm (2/2)

```

1: function FINDPARTIALREDUCTION( $List<IC>\ ic,$                                      ▷ Input columns
                                 $int\ i,$                                        ▷ Index for current input column
                                 $List<R>\ cr,$                                        ▷ List of rules
                                 $boolean\ hasC,$                                        ▷ Boolean flag for found combination
                                 $R\ subR$ )                                           ▷ Current subsuming root rules
2:   if  $i = ic.size()$  then
3:      $addToResult(cr)$                                        ▷ All columns processed?
4:      $createFixes(cr)$                                        ▷ Add the rules to result object
5:   else
6:      $List\ entries \leftarrow getColumnEntriesByFilteredRules(in[i], cr)$            ▷ Get entries of both rules
7:      $found \leftarrow false$ 
8:      $comb \leftarrow true$ 
9:     if  $!hasC \mid subR = entries[0]$  then
10:      if  $entries[0].subsumes(entries[1])$  then                                       ▷ Check for subsumption entries
11:        if  $subR = null \mid subR = entries[0]$  then
12:           $subR \leftarrow entries[0]$ 
13:           $found \leftarrow true$ 
14:      else if  $!hasC \mid subR = entries[1]$  then
15:        if  $!hasC \ \&\ entries[1].subsumes(entries[0])$  then                                       ▷ Check for subsumption entries
16:          if  $subR = null \mid subR = entries[1]$  then
17:             $subR \leftarrow entries[1]$ 
18:             $found \leftarrow true$ 
19:      else if  $entries[1].isIdentical(entries[0])$  then                                       ▷ Check for identical entries
20:         $comb \leftarrow false$ 
21:         $found \leftarrow true$ 
22:      else if  $entries[0].combinationPossible(entries[1])$  then                                       ▷ Check for combination of entries
23:         $found \leftarrow true$ 
24:      if  $found$  then
25:         $findPartialReduction(ic, i + 1, rules, comb, subR)$ 

```

This recursive function has five parameters: *ic* all columns of the current decision, *i* the index for the current column, *cr* the rule pair with the identical output, *hasC* a boolean flag for the indication of a found combination, and *subR* the rule

for the indication of a subsumption. First, the exit condition for the recursion is testes. If all columns are processed, and the index is equal to the size of the column array *ic* (line 2), both rules in *cr* can be combined and are added to the result object. If the end of the recursion is not reached, several tests are processed to evaluate the possible existence of a combination of the two rules. First, both rules are checked to be a subsumption (lines 10 & 15). However, this can only be done if there was no combination previously found, or the current rule is the rule, which was a subsumption before (lines 9 & 14). The next checks test if the entries are identical (line 19) or the entries can be combined (line 22). If one of these tests are passed, the flag *found* is set to true. At this point, if *found* is true (line 24), the function calls itself recursively (line 25) with the incremented index *i* for the column *ic*.

Feasible Fixes. These fixes are created by the presented verifier:

- Show both rules, that can be combined.
- Delete both rules, that can be combined, and add a new rule, which covers both rules.

3.2.2.9 Missing Rule

The detection of missing business rules is essential for error-free decision tables. So the verifier detects whether rules are missing for expected inputs. **Missing Rules** is the equal named verification capability. The crucial elements for this algorithm are the input entries of all decision tables. For instance, in Figure 27 exists no output for an input value of 100. As a result, the missing rule has the input entry ≥ 10 .

Decision X		Rule with " ≥ 10 " is missing	
Decision_iVQwkCLc			
U	Input +	Output +	
	in	out	
	integer	string	
1	<10	"a"	
+	-	-	

Figure 27 Minimal example – Missing Rule

In a more complex example, shown in Figure 28, the determination of the missing rules is more complicated. Hence, a geometric interpretation helps to find these missing rule areas.

Decision 1		
Decision_1k8zkli		
U	Input +	
	Value1	Value2
	double	double
1	[0..3]	[1..4]
2	>6	[2..5]
+	-	-

Figure 28 Extended example – Missing Rule – Decision table

Figure 29 shows this geometric interpretation. The green areas on the left side show the defined rules, and the red area covers the missing rules. The problem now is to define these areas such that no overlapping rules are created. The right side of the figure shows with the colored areas one possible solution for defining the missing rules. The following algorithm detects exactly these missing rules.

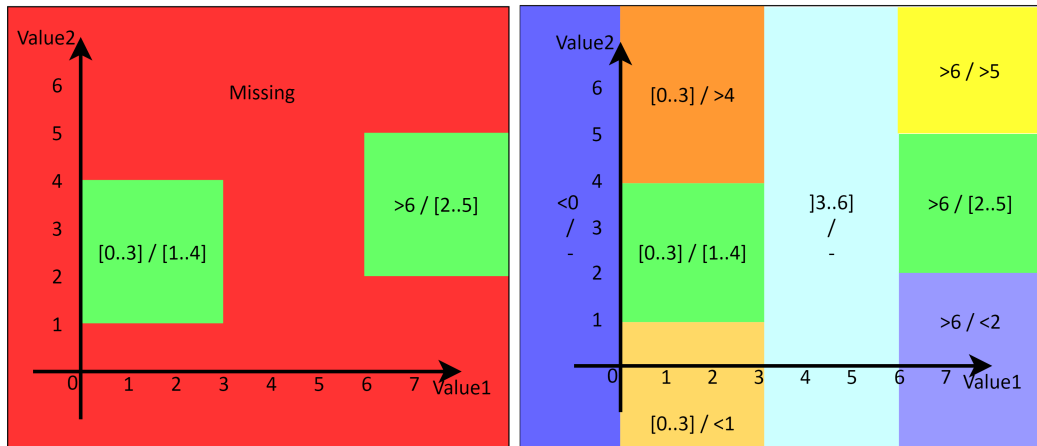


Figure 29 Extended example – Missing Rule – Geometric interpretation

Algorithm. The algorithm is defined as followed:

Algorithm 21 Missing Rule algorithm (1/3)

```

1: function DOVERIFICATION( $M\ m$ )
2:   for all  $D\ d : m.d$  do                                     ▷ For each decision
3:      $missingRules(d)$                                        ▷ → algorithm 22

```

Algorithm 21 shows that the missing rule algorithm is executed for each decision of the DMN model.

Algorithm 22 Missing Rule algorithm (2/3)

```

1: function MISSINGRULES( $D\ d$ )
2:    $List\ lmr \leftarrow \emptyset$                                      ▷ List of missing rules
3:    $lmr.add(createNewDefaultRule())$                              ▷ Rule with only wildcards
4:   for all  $R\ r1 : d.r$  do
5:      $List\ lnmr \leftarrow getInContactRules(lmr, r1)$            ▷ Get all 'in contact' rules
6:     for all  $R\ r2 : lnmr$  do
7:        $lmr.remove(r2)$ 
8:       for all  $IC\ cic : d.ic$  do
9:          $R\ nr \leftarrow createNewRule(LO, cic, d.ic, r1, r2)$    ▷ → algorithm 23
10:         $lmr.addIfNotNull(nr)$ 
11:         $R\ nr \leftarrow createNewRule(HI, cic, d.ic, r1, r2)$    ▷ → algorithm 23
12:         $lmr.addIfNotNull(nr)$ 
13:    $merge(lmr)$ 
14:    $addToResult(lmr)$                                            ▷ Add the rules to result object
15:    $createFixes(lmr)$                                            ▷ Add feasible fixes

```

The first step of the algorithm 22 is the creation of an empty list lmr (line 2), which contains later the missing rules. Then, a first rule, which only contains 'wildcards' as entries, is added to this newly created list (line 3). A wildcard entry means that all possible input values can trigger this entry as applicable. The next step is an iteration over all loops existing in the decision (line 4), where $r1$ represents the rule of the current iteration step. Now, in line 5, the list $lnmr$ is created from all current missing rules (lmr), which are in contact with the current rule $r1$ (cf. subsumption, overlap, or identical algorithm). This created list is iterated in an inner loop (line 6), where $r2$ represents the current rule of this list. Rule $r2$ is removed from the list lmr because the next steps create new, more specific missing rules (line 7). The next inner loop is an iteration over all input columns of the current decision (line 8). Now, the algorithm tries to create new rules and add these new rules to the list lmr (lines 9–12). This creation is done for each current input column twice: One time for the part that is lower (LO) than the current entry and the second time for the part that is higher (HI) than the current entry.

This rule generation is explained below in algorithm 23. At the end of this algorithm, when all rules of the decision are processed, the created rules are merged *lmr* because of subsuming, or overlapping rules (line 13). Now, all missing rules *lmr* are added to the result object, and the fixes are created.

Algorithm 23 Missing Rule algorithm (3/3)

```

1: function CREATENEWRULE(ENUM state, IC cic, List<IC> ic, R r1, R r2)
2:   R nr ← null
3:   for all IC ic2 : ic do
4:     if ic2.index < cic.index then
5:       nr.ie ← r1.ie
6:     else if ic2.index > cic.index then
7:       nr.ie ← r2.ie
8:     else if ic2.index == cic.index then
9:       if state = LO then
10:        nr.ie ← (r1.lowerBound, r2.lowerBound.rev())
11:       else if state = HI then
12:        nr.ie ← (r2.upperBound.rev(), r1.upperBound)
13:   return nr | null on Exception

```

The function ‘createNewRule’ in algorithm 23 describes the creation of a new rule for the missing rule verifier. Here, the variable *nr* describes a new rule, which is initially *null* (line 2). The function iterates over all columns (iteration value *ic2*) of the decision (line 3) and sets the respective input entry for the new rule. The values of the input entries depend on the parameter *cic*, which is the input column from the inner iteration in algorithm 22. If the column index of *ic2* is lower than the column index of *cic*, than the input entry of *r1* is used as the new input entry (lines 4 & 5). If the column index of *ic2* is higher than the column index of *cic*, than the input entry of *r2* is used as the new input entry (lines 6 & 7). However, if the index is equal, two opportunities are possible (line 8). If the *state* value is *LO*, than the new input entry is created with the lower bounds of the rule *r1* and the reversed lower bound of rule *r2* (line 10). If the *state* value is *HI* than the new input entry is created with the reversed upper bound of the rule *r2* and the upper bound of rule *r1* (line 12).

For instance, an entry of *r1* has the range [0..20] and an entry of *r2* has the range [5..15]. Then, the new entry for *LO* equals [0..5[and the new entry for *UP* equals [15..20].

If one of these actions is not possible, the complete function returns *null*, and no rule is created (line 13).

Feasible Fixes. These fixes are created by the presented verifier:

- Show the decision, which has the missing rule. The message text of an result entry should contain a description of the values of the missing rule.
- Add the detected missing rule in the decision.

3.2.2.10 Equivalent Strings

This verifier detects string-based entries which are not identical, but still semantically equivalent. Here, the verifier checks if there exists a pair of entries in the same column which use synonyms as inputs and are therefore equivalent, based on synonym relations via Wordnet. It may accrue, that rules, which have apparent equal, are not detected as equal by humans. For instance, the word 'invoice' is used in one rule, and in the other rule, the word 'bill'. These words are synonyms and may have the same intention. However, they are not detected as identical by the rule engine. **Equivalent Rules** is the related verification capability, and all string based columns with their entries are the relevant elements.

Decision X		
Decision_ivQwkCLc		
U	Input +	Output +
	in	out
	string	string
1	"bill"	"a"
2	"invoice"	"a"
+	-	-

"bill" and "invoice" may be equivalent

Figure 30 Minimal example – Equivalent Strings

Algorithm. The algorithm is defined as followed:

Algorithm 24 Equivalent Strings algorithm

```

1: function DOVERIFICATION( $M$   $m$ )
2:    $stringColumns \leftarrow m.getAllStringColumns()$            ▷ Get all string columns existing in the model
3:   for all  $Column$   $sc$ :  $stringColumns$  do                       ▷ For each (string) column
4:      $checkForSynonyms(sc)$ 
5:   function CHECKFORSYNONYMS( $Column$   $sc$ )
6:     for int  $i = 0; i < sc.entries.size(); i = i + 1$  do
7:       for int  $u = i + 1; u < sc.entries.size(); u = u + 1$  do
8:         if  $Dictionary.areSynonyms(sc.entries[i], sc.entries[u])$  then           ▷ Check for synonyms
9:            $addToResult(sc.entries[i], sc.entries[u])$            ▷ Add both strings to result object
10:           $createFixes(sc.entries[i], sc.entries[u])$            ▷ Add feasible fixes

```

This verifier checks straight forward for each string-based column (line 3) if input entries contain pairs of synonyms. Therefore, each entry is compared directly with each input entire in two nested loops (lines 6 & 7). So, if any values of the compared entries are synonyms (line 8), they are added to the result object. The *Dictionary.areSynonyms()* function is provided by a service, which has access to the Wordnet⁸ dictionary, and provides the functionality to check if two words are synonyms or not. This function further reduces the words to the base form so that more matching cases are covered. Future versions may consider predefined values or include all columns at once to guarantee a model-wide consistency.

Feasible Fixes. These fixes are created by the presented verifier:

- Show and highlight the entries, where the synonyms occur.
- Rename one of both values inside the entries (not implemented yet).

3.2.2.11 Empty Output

This verifier detects rules with empty output values. An empty output value can create inconsistencies because unknown outputs for the following decisions or processes can result in unexpected issues. However, this verifier is not related to a defined verification capability, because it is more a recommendation for the modeler. Related elements are the output columns with their output entries.

Decision X		Rule 2 has an empty output	
Decision_iVQwkCLc			
U	Input +	Output +	
	in	out	
	integer	string	
1	>10	"a"	
2	<=10	-	
+	-	-	

Figure 31 Minimal example – Empty Output

Algorithm. The algorithm is defined as followed:

The strait forward iteration over all decisions (line 2), over their output column (line 3), and finally over the output entries (line 4), is the basic structure of this

⁸<https://projects.csail.mit.edu/jwi/>

Algorithm 25 Empty Output algorithm

```

1: function DOVERIFICATION( $M m$ )
2:   for all  $D d : m.d$  do                                     ▷ For each decision
3:     for all  $OC oc : d.oc$  do                               ▷ For each output column
4:       for all  $OE oe : oc.oe$  do                           ▷ For each output entry
5:         if  $oe.value.isEmpty()$  then                       ▷ Check if the output entry contains no value
6:            $addToResult(oe)$                                 ▷ Add the output entry to the result object
7:            $createFixes(oe)$                                 ▷ Add feasible fixes

```

verifier. The check tests if the output entry is empty, meaning that the triggered rule does not provide a value.

Feasible Fixes. These fixes are created by the presented verifier:

- Show the detected output entry.
- Delete the rule, which contains the empty output entry.

3.2.2.12 Missing Column

As presented verifiers provide fixes, which deletes columns in decisions, these decisions may have no input column or output column remaining. As a consequence, the aim of this verifier is the detection of decision which has no input columns or no output columns. This verifier is not related to a specific verification capability. However, it is necessary for previously executed fixes of other verifiers (Deletion of columns). The relevant elements are the decisions with their columns.

Decision X		
Decision_iVQwkCLc		
U	Input	Output +
	+	out
		string
+	-	-

No input column

Figure 32 Minimal example – Missing Column

Algorithm. The algorithm is defined as followed:

First, the algorithm iterates over all decisions (line 2) and secondly checks if the current decision has no input column or no output column (line 3). Both tests lead to adding the decision to the result object.

Algorithm 26 Missing Column algorithm

```

1: function DOVERIFICATION(M m)
2:   for all D d : m.d do                                     ▷ For each decision
3:     if d.ic.size() = 0 | d.oc.size() = 0 then             ▷ Amount of input or output columns is zero
4:       addToResult(d)                                       ▷ Add decision to result object
5:       createFixes(d)                                         ▷ Add feasible fixes

```

Feasible Fixes. These fixes are created by the presented verifier:

- Show the decision, where no input column or output column is present.
- Delete the detected decision.

3.2.3 Syntax Level Verifiers

This subsection introduces verifiers for syntactical checks of a DMN model. The current version of the `dmn.js` modeler⁹ only provides limited syntactical checks, which are defined by the XSD definition¹⁰ of DMN. For instance, it is not possible to create an information requirement arc between two input data nodes. However, concrete syntactical checks of entries are not part of this definition. This thesis only describes two syntactical aspects, which are defined and implemented in this section. Section 3.2.3.1 defines a date format check and section 3.2.3.2 describes a verifier for a general syntax validation of input values. Further aspects can be easily added with the definition of new verifiers.

3.2.3.1 Date Format

This verifier performs an example syntactical verification by checking the specific formal syntax of date-based entries. It checks the correctness of the date format for all date entries in columns, which require dates as values. Each date value need to be in the format “data and time(yyyy-MM-ddTHH:mm:ss)”. Besides this syntactical correctness, it checks the veracity of the concrete date. For instance, modeling an unknown date like “data and time(2020-02-30T00:00:00)” makes no sense. This date does not exist and is as a result of this recognition not valid.

⁹<https://bpmn.io/toolkit/dmn-js/>

¹⁰<https://www.omg.org/spec/DMN/20180521/DMN12.xsd>

Decision X		Date 2020-02-30 does not exist	
Decision_iVQwkCLc			
U	Input +	Output +	
	DateCol	out	
	date	string	
1	date and time("2020-02-30T00:00:00")	"a"	
+	-	-	

Figure 33 Minimal example – Date Format

Algorithm. The algorithm is defined as followed:

Algorithm 27 Date Format algorithm

```

1: function DOVERIFICATION( $M$   $m$ )
2:    $dateColumns \leftarrow m.getAllDateColumns()$            ▷ Get all date columns existing in the model
3:   for all  $Column$   $dc$ :  $dateColumns$  do                 ▷ For each (date) column
4:     for all  $Entry$   $entry$ :  $dc.entries$  do             ▷ For each entry in the date column
5:       if !  $checkCorrectDate(entry.values)$  then       ▷ Check if the date values are correct
6:          $addToResult(entry)$                            ▷ Add the entry to the result object
7:          $createFixes(entry)$                              ▷ Add feasible fixes

```

The verifier iterates straight forward all the date-based columns (lines 2 & 3). Then it checks for each entry (line 4) if the date values are correct (line 5). First, the date is checked for syntactical correctness with a regular expression. Second, the program tries to parse the date value to a Java date object. If this is not possible, the semantic validation for the date value failed. Non-existing dates (e. g. 2020/02/30) are automatically detected by catching a parsing exception. If a date value is syntactical or semantical incorrect, the entry is added to the result object.

Feasible Fixes. The only reasonable fix for this verifier is the ‘Show’ fix. It enables a lookup and fast finding of the error in the DMN model for the user.

3.2.3.2 Input Value Syntax

This verifier checks the syntactical correctness of all the input entries. The Camunda Modeler does not directly detect syntactical errors in the model. However, the model produces errors when the engine executes the DMN. At this point, it is too late to maintain the DMN model, because it is already deployed to the BRM system. The modeler has to redeploy the model. Therefore, it is more significant to detect these errors directly while modeling.

Decision X		"56p" is not an integer value	
Decision_ivQwkCLc			
U	Input +	Output +	
	IntCol	out	
	integer	string	
1	= 56p	"a"	
+	-	-	

Figure 34 Minimal example – Input Value Syntax

Algorithm. The algorithm is defined as followed:

Algorithm 28 Input Value Syntax algorithm

```

1: function DOVERIFICATION( $M\ m$ )
2:   for all  $D\ d : m.d$  do ▷ For each decision
3:     for all  $R\ r : d.r$  do ▷ For each rule
4:       for all  $IE\ ie : r.ie$  do ▷ For each input entry
5:         if  $ie.isValid()$  then ▷ Check for syntactical correctness
6:           addToResult( $ie$ ) ▷ Add input entry to result

```

This algorithm checks straight forward for each existing input entry the syntactical correctness (lines 2–4). The important part is the *isValid()* function (line 5), which depends on the data type of the current column. For instance, strings has to start and end with double quotes. Other syntactical specifications are defined by the FEEL standard¹¹ and here black boxed as *isValid()* function. However, this validation test ignores the date values, because they are checked by the previous presented verifier.

Feasible Fixes. The only fix that is created by the ‘Input Value Syntax’ verifier is the ‘Show’ action. Here, all elements that have syntactical errors need to be highlighted so that users can easily find them and correct these errors manually.

¹¹<https://docs.camunda.org/manual/7.4/reference/dmn11/feel/language-elements/>

3.3 Front-end of the DMN Verification Tool

The previous sections introduced the back-end system as well as the algorithms of the implemented verifiers. This back-end works as a standalone system, which provides a web service for the detection of errors in DMN models. Though this is entirely sufficient for the automatic detection of errors, humans are not able to interact directly with this web service. Therefore, third-party systems, which have to interact with DMN models, are now able to request these verification capabilities and can handle them. Users of these systems are then able to recognize and improve the faulty models. Hence, a first modeling tool using this web service is introduced in the following.

This web-based prototype is a modeling tool for DMN models, which considers the verification aspect. An overview of this tool is shown in Figure 35. It is divided into three components, which are described separately.

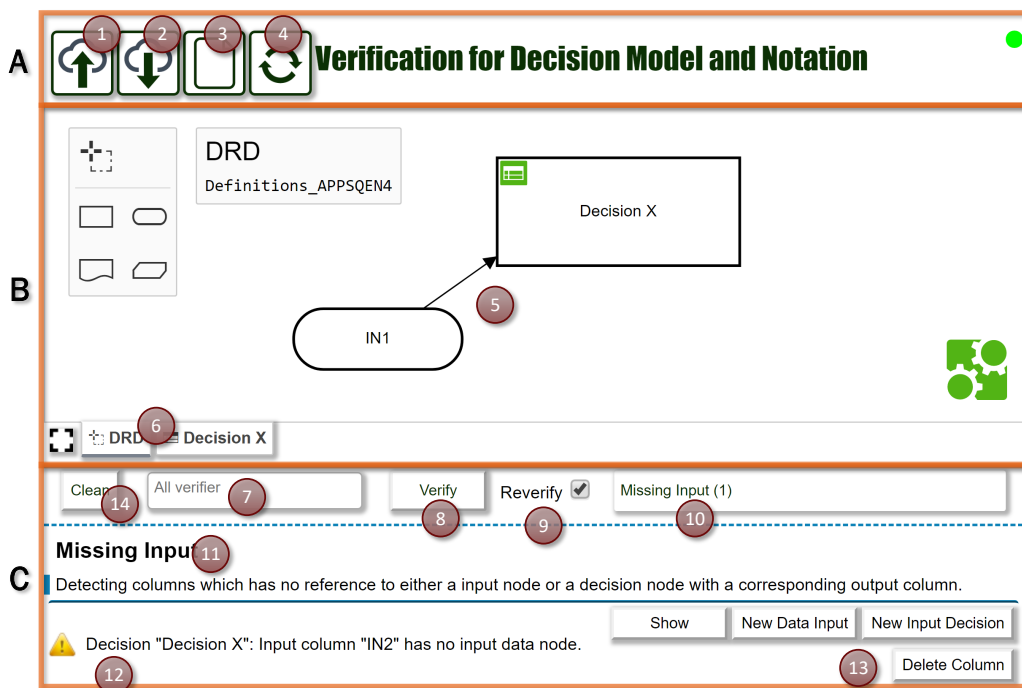


Figure 35 Front-end overview

Component A. This upper part of the tool provides some general functions. It contains besides the headline four buttons for loading or for saving the current

DMN model. Button ① offers an upload function so that the user can upload a DMN model from his personal computer. The counterpart provides the button ②, which starts the download of the current model. Furthermore, button ③ creates a new empty DMN model. This model only contains one empty decision table. Finally, button ④ provides some further functions to connect the prototype with a DMN repository. These functionalities are shown in Figure 36.

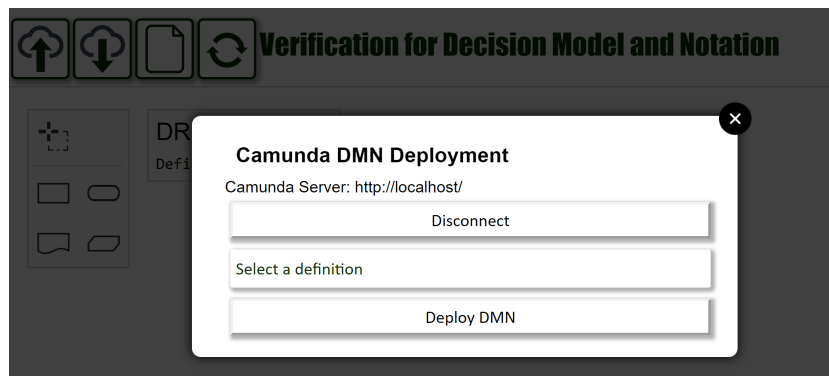


Figure 36 Front-end – Connection to DMN repository

In this tool, the Camunda process engine, which also includes a DMN engine, can be requested to load existing models. Furthermore, the ‘Deploy DMN’ button redeploys the changed model to the DMN engine. The redeployment increments the version of the model inside the DMN engine, where this model is directly executable.

Component B. The center part considers the modeling aspect of the prototype. Here, the DMN modeling library *dmn-js*¹² is used to display the model ⑤. It provides all needed functions for editing the Decision Requirement Diagram (DRD) (shown in Figure 35), and all needed functions for editing the decision tables (e.g. adding or deleting of columns / rules). Furthermore, tabs at the bottom of the modeling area ⑥ helps to navigate between the DRD and the decision models.

Component C. The part below the modeling section is all about the verification aspect. Several functions enable access to the verification service and provide

¹²<https://github.com/bpmn-io/dmn-js>

the functionality for the calculated fix actions. First of all, a multi-select combo box ⑦ is filled with all verifiers, which are provided by the back-end, and allows the user to preselect the needed verifiers. Now, the user can press the 'Verify' button ⑧ manually. The front-end sends the complete DMN model (as XML) to the back-end, which does the verification magic (cf. section 3.1), and receive the verification result as JSON. Furthermore, the checkbox 'Reverify' ⑨ enables an automatic verification mechanism, so that after each change on the model, the verification results are refreshed. The combo box ⑩ provides the functionality of selecting a single verifier so that the verification results are listed in the next lines. Alternatively, all results can be displayed. Each verification result starts with a name and a short description of the verifier ⑪, which is also provided by the web service. The list below now contains all verification results, which were found by the verifier ⑫. Furthermore, the fix actions are rendered as buttons ⑬. If the user clicks the button, the front-end executes the intentioned behavior of the button and shows the relevant elements in the model or does other actions (e.g. deletion of a column). If the 'Show' button was clicked, the relevant elements in the model are highlighted. The 'Clear' action ⑭ removes these highlights again. Some of the actions are demonstrated in the following section.

3.4 Demonstration

The following scenario is introduced to illustrate the usage of the modeling tool in combination with the verification aspect. An insurance company calculates the insurance rate with the help of a DMN model. This model is shown in Figure 37.

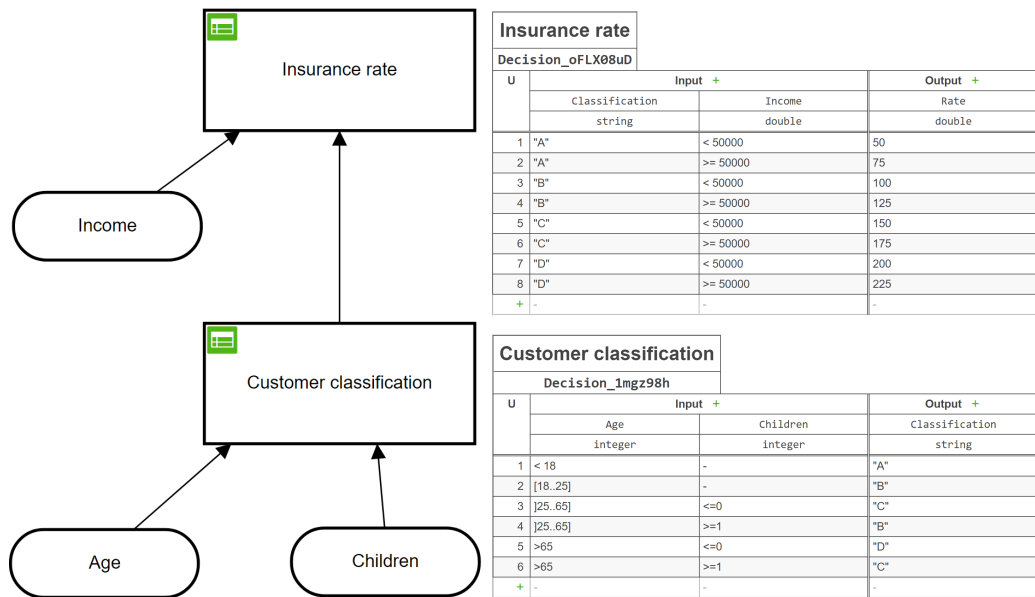


Figure 37 Demonstration - Initial DMN model

Several inputs are used to calculate this rate. First, the 'Customer classification' decision abstracts the age of the customer and the number of children into a classification. The 'Insurance rate' decision uses this classification as input. Furthermore, it uses the income of the customer as the second input to calculate the rate.

Now, a new requirement has to be implemented by an employee. The boolean value of the criminal past (yes or no) should affect the calculation. If a customer has a criminal past, the classification should be the value 'E', which is currently not present in the model. However, the employee adds the new input column 'CriminalPast' to the decision 'Customer classification' and adds the new rule, which fulfills the requirement. However, the verifiers have just found several errors. One verifier detects that the new added rule is overlapping with other rules, so the employee has to change the other rules, too. This overlap with rule

six is shown in Figure 38. The employee can now take a close look and can correct the error.

Verification for Decision Model and Notation

U	Input +			Output +	Annotation
	Age	Children	CriminalPast	Classification	
	integer	integer	boolean	string	
1	< 18	-	-	"A"	-
2	[18..25]	-	-	"B"	-
3]25..65]	<=0	-	"C"	-
4]25..65]	>=1	-	"B"	-
5	>65	<=0	-	"D"	-
6	>65	>=1	-	"C"	-
7	-	-	true	"E"	-
+	-	-	-	-	-

DRD: Insurance rate, Customer classification

Clean All verifier Verify Reverify Overlapping Rules (4)

Overlapping Rules
 Detecting whether there are any overlaps in rule conditions. This verifier does not show identical, overlaps or subsumptions. This verifier only works for unique hit policies.

Decision "Customer classification": The rules 6, 7 are overlapping. Show

Figure 38 Demonstration - Overlapping rules

The 'Missing Input' verifier detects another error. Although the employee added the new column (cf. Figure 38), he forgets to add the Input data node in the DRD level. Figure 39 shows the execution of the fix action 'New Data Input', which creates a new Input Data node in the DRD.

Missing Input
 Detecting columns which has no reference to either a input node or a decision node with a corresponding output column.

Decision "Customer classification": Input column "CriminalPast" has no input data node. Show New Data Input New Input Decision Delete Column

Missing Input
 Detecting columns which has no reference to either a input node or a decision node with a corresponding output column.

DRD: Insurance rate, Customer classification

Clean All verifier Verify Reverify Missing Input (0)

Figure 39 Demonstration - Missing Input

Furthermore, Figure 40 shows the finding of the Missing Input Values verifier. The classification value 'E' is never used in the following 'Insurance rate' decision table (cf. Figure 37), which may cause problems.

Verification for Decision Model and Notation

	Age	Children	CriminalPast	Classification	Annotation
	integer	integer	boolean	string	
1	< 18	-	false	"A"	-
2	[18..25]	-	false	"B"	-
3]25..65]	<=0	false	"C"	-
4]25..65]	>=1	false	"B"	-
5	>65	<=0	false	"D"	-
6	>65	>=1	false	"C"	-
7	-	-	true	"E"	-
8	-	-	-	-	-
9	-	-	-	-	-

DRD Insurance rate Customer classification

Clean All verifier Verify Reverify Missing Input Value (1)

Missing Input Value

Detecting output values of output columns in decision tables which are not used in the connected decision table as input values.

Decision "Insurance rate": Input column "Classification" never handles the value "E".

Show Delete rule in decision "Customer classification"

Add rule in decision "Insurance rate" with input value "E"

Figure 40 Demonstration - Missing Input Value (1/2)

Therefore, the employee can click the green highlighted button to create a new rule, which contains that missing value. This newly created rule is shown in Figure 41.

Verification for Decision Model and Notation

	string	double	double	Annotation
1	"A"	< 50000	50	-
2	"A"	>= 50000	75	-
3	"B"	< 50000	100	-
4	"B"	>= 50000	125	-
5	"C"	< 50000	150	-
6	"C"	>= 50000	175	-
7	"D"	< 50000	200	-
8	"D"	>= 50000	225	-
9	"E"	-	-	-

DRD Insurance rate Customer classification

Clean All verifier Verify Reverify Missing Input Value (0)

Missing Input Value

Detecting output values of output columns in decision tables which are not used in the connected decision table as input values.

Figure 41 Demonstration - Missing Input Value (2/2)

However, the verifier does not know which output value (the rate) is valid for this rule because this is knowledge, which should be provided by the employee. Another verifier detects this empty output value and creates a verification result

for this type of error (cf. Figure 42). Now, the employee can add this value manually.

Verification for Decision Model and Notation

	string	double	double	Annotation
1	"A"	< 50000	50	-
2	"A"	>= 50000	75	-
3	"B"	< 50000	100	-
4	"B"	>= 50000	125	-
5	"C"	< 50000	150	-
6	"C"	>= 50000	175	-
7	"D"	< 50000	200	-
8	"D"	>= 50000	225	-
9	"E"	-	-	-
+	-	-	-	-

DRD Insurance rate Customer classification

Clean All verifier Verify Reverify Empty Output (1)

Empty Output
 Detection rules with empty output values.
 Decision "Insurance rate", Rule "9": Output value is empty. Show Delete rule

Figure 42 Demonstration - Missing Output

Finally, the verifier 'Missing Predefined Value' indicates the value 'E' as missing predefined value in the corresponding column definition (cf. Figure 43). A click on the button 'Add pred. value "E"' fixes this issue.

Verification for Decision Model and Notation

	Input	Output	Annotation
	Classification	Rate	
1	"A"	50	-
2	"A"	75	-
3	"B"	100	-
4	"B"	125	-
5	"C"	150	-
6	"C"	175	-
7	"D"	200	-
8	"D"	225	-
9	"E"	-	-
+	-	-	-

DRD Insurance rate Customer classification

Clean All verifier Verify Reverify Missing Predefined Value (2)

Missing Predefined Value
 Detecting string values which are not defined in the list of predefined values of the column.
 Decision "Insurance rate", Column "Classification": String value "E" was not found in the list of predefined values. Show Add pred. value "E" Delete rule 9

Figure 43 Demonstration - Missing Predefined Value

As seen in this short demonstration, a simple change in a DMN model can result in further required changes. This prototype can support the employee in his modeling work by highlighting modeling errors or suggesting various fixes.

Chapter 4

Evaluation

An essential task in the Design Science Research Methodology (DSRM) is the evaluation. Testing the created artifact is a crucial component of the research process (Hevner et al. 2004). This chapter aims at the research question **SQ5**. Hence, the verification prototype, as the artifact of this thesis, has to be evaluated, too. "IT artifacts can be evaluated in terms of functionality, completeness, consistency, accuracy, performance, reliability, usability, fit with the organization, and other relevant quality attributes" (Hevner et al. 2004). Moreover, Corea & Delfmann (2020) summarised a guideline for the evaluation in Business Process Compliance. First, a demonstration should show the approach in action conducting an example. The demonstration of the DMN verification tool was presented in the previous section 3.4. Second, the approach should be applied in case-studies, where the tool can be used in real-world scenarios. Comparisons with other approaches, a complexity analysis, or experiments with qualitative feedback are further techniques to evaluate the tool (Corea & Delfmann 2020). However, in this thesis, run-time experiments are used as a first evaluation step, which is presented in the following. Other evaluation techniques (e.g. Case-Studies or Experiments) are planned for future work with this tool.

Setting up the evaluation. The general idea for this evaluation is to measure the execution time of the implemented verifiers with different input parameters. Then, the measured run-times of the different inputs can be plotted to illustrate the performance of the verification tool. The inputs of these performance tests are synthetic DMN models. Parameters should define the DMN model and determine the number of decisions and input data nodes, the number of columns, and the number of rules. As a result, randomly generated DMN models are needed. Therefore, a small Java library for the generation of random DMN models was created. The generator object in appendix C.I demonstrates a small example for the generation of random DMN models. Different parameters determine the number of decisions, the possible data types, the number of rules, and further settings for the randomization. Then, the generator generates a DMN model as XML String or as DMN Model object with the given parameters. The entries are filled with randomly ranges for number-based columns or random words for string-based columns. Input data nodes and decisions are connected with information requirements randomly, too. In this way, erroneous DMN models are created automatically.

For the experiments, the verification tool is requested with different input parameters multiple times. Two parameters determine the dimensions of the plotted graph, and the third dimension determines the execution time. To suppress random irregularities, each input parameter combination is executed multiple times, from which the average execution time is calculated.

Results. The evaluation is divided into two parts. The first experiment was executed for the previous version of the prototype, which was used as the demo tool in Corea et al. (2019). This prototype mainly focuses on the decision logic level and the verification capabilities by Smit et al. (2017). As a result, the synthetic DMN models always contain one decision, and the running parameters for the generation are the number of rules (50, 100, ..., 500) and the number of input columns (1, 2, ..., 10). The algorithms used in that tool are the same as the algorithms used in this thesis. The following decision level verifiers are part of this evaluation: Subsumption Rule, Identical Rule, Overlapping Rule, Identical Rule, Partial Reduction, Missing Rule, and Equivalent Strings. This experiment was run on a Windows 10 PC with an i7 processor and 16GB DDR4 RAM. As

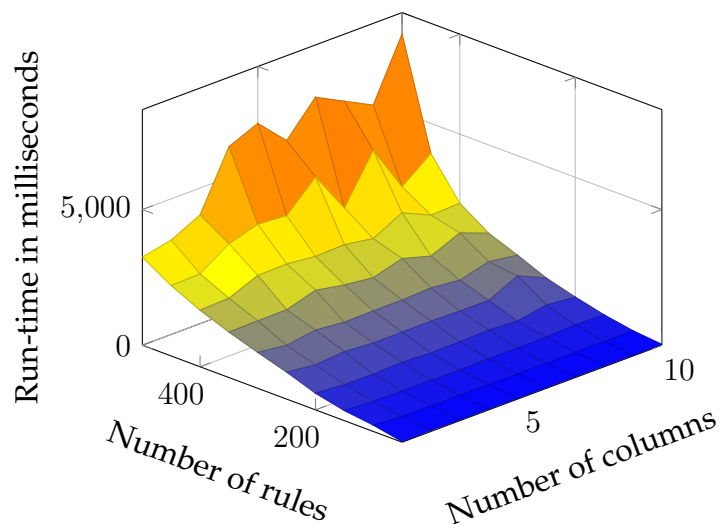


Figure 44 Performance test 1 – Run-time for the analysis of synthetic decision tables with up to 10 columns and 500 rows.

seen in Figure 44, the run-time mainly depends on the number of rules, where the number of columns is mostly not relevant. The longest run-time with 500

rules is around five seconds, which is acceptable for synthetic decisions. An evaluation with real data-sets from industry sectors should perform better because they should have much less noise in their models, which is less stressful for the verifiers. However, this is a task for future work.

The second experiment was executed with the verification tool, which was created for this thesis and presented in the previous chapter 3. This prototype was created as a proof of concept in Hasić et al. (2020b) and evaluated in Hasić et al. (2020a). For this evaluation, the number of nodes (decisions and input data nodes) is most relevant. Therefore, one dimension displays the number of nodes (0, 20, ..., 180; ratio 1:4 of input data : decisions), and the second dimension displays the number of rules per decision (0, 10, ..., 100). The evaluated verifiers for this part are: Lonely Data Input, Missing Input, Missing Input Column, Multiple Input Data, Wrong Data Type, Missing Input Value, Missing Output Value, Missing Column, Date Format Input Value Syntax, Missing Predefined Value, Unused Predefined Value, and Empty Output. The experiment was run on Ubuntu Xenial with E312 processor and 16GB RAM. As seen in Figure 45, the execution time

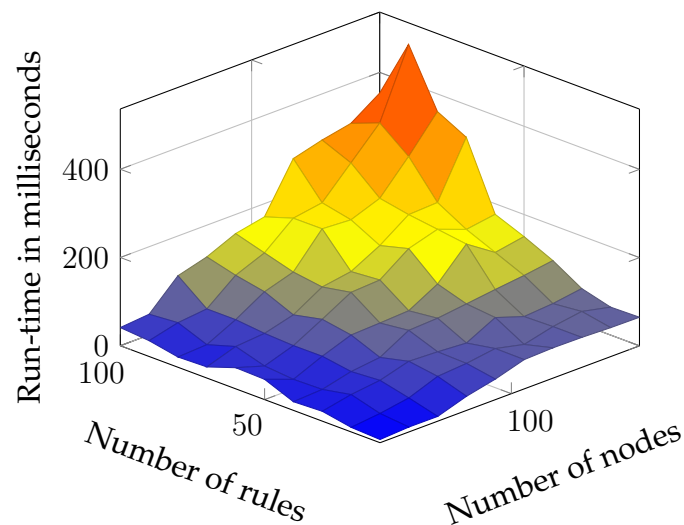


Figure 45 Performance test 2 – Run-time statistics for the analyzed synthetic decision models with up to 180 nodes on the DRD-level and 100 rules per table.

depends on both: the number of nodes and the number of rules. With 200 nodes and 100 rules per decision, the run-time is still under 500 milliseconds, which is feasible for production tasks.

Future tests may include more differentiation of input parameters. For instance, changing the number of output columns, differentiate between data types, or changing the probabilities of connected nodes.

Chapter 5

Conclusion

This thesis introduced the implementation of a verification tool for DMN models. This aim was subdivided into five subsidiary research questions. These questions were answered with the content of the related chapters. First, fundamental basics and verification frameworks were presented in chapter 2. Besides these basics, major verification types have been identified, which was the answer to the research question **SQ1**. Moreover, approaches for fixes were mainly applied from Hasić et al. (2020b) (related to research question **SQ2**). The next chapter 3 presented the architecture of the tool (answer to research question **SQ3**), explained the algorithms of the implemented verifiers (answer to research question **SQ4**), and described the front-end. Finally, chapter 4 answers research question **SQ5** with the evaluation of the created tool. Furthermore, the created prototype was demonstrated in section 3.4. With this tool, errors in rule bases can be resolved while modeling before the rules are running in a production system. So it prevents problems in processes of enterprises during run-time.

Further developments. The prototype has several points for improvements and adjustments that can be future development extensions. First, the back-end was dynamically implemented, so that new verifiers are addable without much effort. The abstract verifier provides a skeleton for further verifier, which enables new verification checks. On the one hand, missing verification capabilities can be implemented as new verifiers. For instance, the Decision Requirement Diagram (DRD) prohibits a cyclic arrangement of the decisions, and a new verifier can detect these circles. An error message created by the new verifier can help an inexperienced modeler to avoid these types of errors. On the other hand, potential company requirements to rule bases can also be implemented in the form of a new verifier that checks the individual constraints. For instance, a company has the requirement that a decision should only have at maximum 50 rules, and the new verifier produces a warning for those oversized decisions. More than these amount of rules indicates that the decision can be simplified by implementing new sub-decisions or by remodeling the decision with a new hit policy.

Furthermore, the Camunda Modeler, which is a modeling environment for BPMN or DMN models, provides extensions points for plugins for further functionalities. As the current prototype is designed in two layers (back-end & front-end), the front-end can easily be replaced. With a new plugin as the new front-

end, the verification API (back-end) can be requested to show modeling errors directly in the Camunda Modeler.

Finally, the used DMN version in this prototype is 1.2, which is the current supported version of the Camunda DMN model API. However, the new DMN version 1.3, which was introduced in early 2019, should be supported in a future version.

Further evaluation. The first steps of an evaluation of the presented artifact were presented in chapter 4. Here, performance tests with synthetic DMN models were conducted. Future tests should also use real models from industry to test the performance of the algorithms. However, as mentioned by Corea & Delfmann (2020), further techniques can be applied for the evaluation of such artifacts. These techniques can further test and improve the prototype. For instance, in addition to the performance experiments, complexity analysis of the algorithms helps to evaluate the verifiers.

Further ways for the evaluation are usability experiments, which can indicate if the verification tool creates additional benefits for modelers. Compared to a non-verification-detection supported DMN modeling tool, a faster finding of errors in DMN models can be the result of this evaluation. Furthermore, case-studies with a survey of employees, which use this tool in production systems, can create more insights about usability and relevance in practice.

Miscellaneous aspects. As the prototype can detect and resolve several errors in DMN models, it is an added value for modelers. However, some of the verifiers are very straight forward and does not require a lot of calculation power. Hence, verifications like the 'Lonely Data Input' (cf. section 3.2.1.1) can be implemented directly inside the modeling tool and not by the external DMN verification API. Here, the manufacturers of such software are in demand.

Furthermore, adjustments in the DMN standard can prevent modelers from modeling some of these errors from the beginning. For instance, an information requirement currently only contains information about the source and sink of the edge. As a suggestion, the information requirement has to be extended with more information about the source and sink by adding the name of the related column of the connected decisions. Now, the modeler can create a more precise model,

where the ‘Missing Input Column’ verification (cf. section 3.2.1.3) can never occur. This aspect may be included in future versions of the DMN specification.

To conclude, a well-working prototype was developed during this thesis. Several verification aspects were covered by the tool, and easy mechanisms for resolving these errors are implemented, too.

Bibliography

- Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T. & Vitter, J. S. (1998), 'Scalable Sweeping-Based Spatial Join', *VLDB* (November 1998).
- Batoulis, K., Nesterenko, A., Repitsch, G. & Weske, M. (2017), 'Decision management in the insurance industry: Standards and tools', *CEUR Workshop Proceedings* **1985**, 52–63.
- Batoulis, K. & Weske, M. (2017), A Tool for Checking Soundness of Decision-Aware Business Processes., in 'BPM (Demos)', pp. 1–5.
- Batoulis, K. & Weske, M. (2018a), 'A tool for the uniqueification of DMN decision tables', *CEUR Workshop Proceedings* **2196**, 116–119.
- Batoulis, K. & Weske, M. (2018b), Disambiguation of {DMN} decision tables, in 'International Conference on Business Information Systems', Springer, pp. 236–249.
- Becker, J., Holten, R., Knackstedt, R. & Niehaves, B. (2003), Forschungsmethodische Positionierung in der Wirtschaftsinformatik: epistemologische, ontologische und linguistische Leitfragen, Technical report, Arbeitsberichte des Instituts für Wirtschaftsinformatik, Westfälische~. . . .
- Calvanese, D., Dumas, M., Laurson, Ü., Maggi, F. M., Montali, M. & Teinmaa, I. (2016), 'Semantics and analysis of DMN decision tables', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **9850 LNCS(i)**, 217–233.
- Calvanese, D., Dumas, M., Laurson, Ü., Maggi, F. M., Montali, M. & Teinmaa, I. (2018), 'Semantics, analysis and simplification of DMN decision tables', *Information Systems* **78**, 112–125.

- Calvanese, D., Dumas, M., Maggi, F. M. & Montali, M. (2017), Semantic {DMN}: Formalizing Decision Models with Domain Knowledge, *in* 'International Joint Conference on Rules and Reasoning', Springer, pp. 70–86.
- Corea, C., Blatt, J. & Delfmann, P. (2019), A Tool for Decision Logic Verification in DMN Decision Tables, *in* 'Proceedings of the Dissertation Award, Doctoral Consortium, and Demonstration Track at BPM 2019 co-located with 17th International Conference on Business Process Management, BPM 2019, Vienna, Austria, September 1-6, 2019.', pp. 169–173.
URL: <http://ceur-ws.org/Vol-2420/papeDT11.pdf>
- Corea, C. & Delfmann, P. (2018), A Tool to Monitor Consistent Decision-Making in Business Process Execution, *in* 'Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018 co-located with 16th International Conference on Business Process Management (BPM 2018), Sydney, Australia, September 9-14, 2018.', pp. 76–80.
URL: http://ceur-ws.org/Vol-2196/BPM_2018_paper_16.pdf
- Corea, C. & Delfmann, P. (2020), 'Using Business Rule Organizing Approaches for Business Process Compliance', *Submitted to: Enterprise Modelling and Information Systems Architectures* .
- Finlayson, M. A. (2014), 'Java libraries for accessing the Princeton Wordnet: Comparison and evaluation', *GWC 2014: Proceedings of the 7th Global Wordnet Conference* pp. 78–85.
- Graham, I. & Wiley, J. (2006), *Business rules management and service oriented architecture: a pattern language*, John wiley & sons.
- Hasić, F., Corea, C., Blatt, J., Delfmann, P. & Serral, E. (2020a), 'A Tool for the Verification of Decision Modeland Notation (DMN) Models', *Research Challenges in Information Science* .
- Hasić, F., Corea, C., Blatt, J., Delfmann, P. & Serral, E. (2020b), 'Decision Model Change Patterns for Dynamic System Evolution', *Knowledge and Information Systems* .

- Hevner, A. R. & Chatterjee, S. (2010), *Design Research in Information Systems: Theory and Practice*, Vol. 2.
URL: <http://link.springer.com/10.1007/978-1-4419-6108-2>
- Hevner, A. R., March, S. T., Park, J., Ram, S. & Ram, S. (2004), 'Design Science in Information Systems Research', *MIS Quarterly* **28**(1), 75–105.
URL: <https://www.jstor.org/stable/25148625>
- Laurson, Ü. & Maggi, F. M. (2016), A Tool for the Analysis of {DMN} Decision Tables., in 'BPM (Demo Track)', pp. 56–60.
- Morgan, T. (2002), *Business rules and information systems: aligning IT with business goals*, Addison-Wesley Professional.
- Object Management Group® (2019a), 'Datasheet OMG Standards for Decision Model and Notation OMG Standard for Decision Model and Notation'.
URL: <https://www.omg.org/intro/DMN.pdf>
- Object Management Group® (2019b), 'Decision Model and Notation Version 1.2', p. 208.
URL: <http://www.omg.org/spec/DMN/>
- Ochoa, L. & González-Rojas, O. (2017), Analysis and Re-configuration of Decision Logic in Adaptive and Data-Intensive Processes (Short Paper), in 'OTM Confederated International Conferences', Springer, pp. 306–313.
- Peffer, K., Tuunanen, T., Rothenberger, M. A. & Chatterjee, S. (2007), 'A Design Science Research Methodology for Information Systems Research', *Journal of Management Information Systems* **24**(3), 45–77.
- Rücker, B. (2016), 'Decision Model and Notation : Digitalisierung von Entscheidungen mit DMN', *OBJEKTSpektrum* pp. 40–45.
- Schlosser, S., Baghi, E., Otto, B. & Oesterle, H. (2014), 'Toward a functional reference model for business rules management', *Proceedings of the Annual Hawaii International Conference on System Sciences* pp. 3837–3846.
- Smit, K., Zoet, M. & Berkhout, M. (2017), 'Verification capabilities for business rules management in the Dutch governmental context', *International Conference on Research and Innovation in Information Systems, ICRIIS* pp. 1–6.

- Smit, K., Zoet, M. & Berkhout, M. (2019), 'A Verification Framework for Business Rules Management in the Dutch Government Context', *International Journal on Advances in Systems and Measurements* **12**(1), 101–112.
- Vanthienen, J., Mues, C. & Aerts, A. (1998), 'An illustration of verification and validation in the modelling phase of KBS development', *Data and Knowledge Engineering* **27**(3), 337–352.
- Vanthienen, J., Mues, C., Wets, G. & Delaere, K. (1998), 'A tool-supported approach to inter-tabular verification', *Expert systems with applications* **15**(3-4), 277–285.

A Appendix: Project Information

Bookmarks. The following lists relevant links of the DMN verification tool.

- GitLab project
`https://gitlab.uni-koblenz.de/jonasblatt/ma-jonasblatt-dmn-verifier`
- Running back-end
`http://dmn.fg-bks.uni-koblenz.de:8080`
- Running front-end
`http://dmn.fg-bks.uni-koblenz.de`
- Configuration for front-end
`http://dmn.fg-bks.uni-koblenz.de/config.html`
- Metrics of verifiers
`http://dmn.fg-bks.uni-koblenz.de/metrics.html`

Packages. The following lists major packages of the DMN verification project.

The base package is `de.unikoblenz.fgbks.*`.

- DMN Rest API
`*.api`
- DMN Verification Model
`*.core.dmn.domain.vdmn.*`
- DMN Verification Service
`*.core.dmn.verificaiton.*`
- DMN Verifier
`*.core.dmn.verificaiton.verifier.*`
- DMN Verification Result structure
`*.core.dmn.verificaiton.result.*`
- DMN Generator
`*.dmn.generator`
- Performance test
`*.dmn.performancetest`

B Appendix: Implementation

B.I Java Annotation REST Endpoint

```
1 @Path("/api/dmn/verification")
2 public class VerificationApi {
3
4     @Inject
5     protected DmnVerificationService dmnVerificationService;
6
7     @POST
8     @Produces(MediaType.APPLICATION_JSON)
9     @Consumes(MediaType.TEXT_XML)
10    public Response verifyAll(String payload) {
11        return dmnVerificationService.generate(payload);
12    }
13    ..
14 }
```

B.II JSON Result of Verification Types

```
1 [
2   {
3     "description": "Detecting .. values.",
4     "name": "MissingInputValueVerification",
5     "niceName": "Missing Input Value",
6     "classification": {
7       "description": "..",
8       "name": "LogicModelingLevel",
9       "niceName": "Logic Modeling Level"
10    }
11  },
12  {
13    "description": "Detecting rules which ..",
14    "name": "EquivalentStringVerification",
15    "niceName": "Equivalent Strings",
16    "classification": {
17      "description": "..",
18      "name": "LogicModelingLevel",
19      "niceName": "Logic Modeling Level"
20    }
21  },
22  ..
23 ]
```

B.III JSON Result of Verification Configuration

```
1 {
2   "PredefinedExistingValueVerification": true,
3   "EquivalentStringVerification": true,
4   "OverlappingRulesVerification": true,
5   "MultipleInputDataVerification": true,
6   "SubsumptionRulesVerification": true,
7   "EmptyOutputVerification": true,
8   "MissingInputDataVerification": true,
9   "InputValueSyntaxVerification": true,
10  "PredefinedMissingValueVerification": true,
11  "DateFormatVerification": false,
12  "IdenticalRuleVerification": false,
13  "MissingInputValueVerification": true,
14  "LonelyDataInputVerification": true,
15  "WrongDataTypeVerification": true,
16  "MissingOutputValueVerification": true,
17  "MissingColumnVerification": false,
18  "MissingInputColumnVerification": true,
19  "PartialReductionVerification": false,
20  "MissingRulesVerification": false
21 }
```

B.IV JSON Result of Action Types

```
1 [
2   "UPDATE" ,
3   "CREATE" ,
4   "DELETE" ,
5   "SHOW"
6 ]
```

B.V JSON Result of Action Scopes

```
1 [
2   "RULE" ,
3   "INPUT_ENTRY" ,
4   "OUTPUT_ENTRY" ,
5   "INPUT_DATA" ,
6   "INPUT_COLUMN" ,
7   "OUTPUT_COLUMN" ,
8   "DECISION"
9 ]
```

B.VI JSON Result of Action Config

```
1 {
2   "RULE": {
3     "UPDATE": true,
4     "DELETE": false,
5     "CREATE": false,
6     "SHOW": false
7   },
8   "INPUT_COLUMN": {
9     "UPDATE": true,
10    "DELETE": true,
11    "CREATE": true,
12    "SHOW": true
13  },
14  "OUTPUT_COLUMN": {
15    "UPDATE": true,
16    "DELETE": true,
17    "CREATE": true,
18    "SHOW": true
19  },
20  "DECISION": {
21    "UPDATE": true,
22    "DELETE": true,
23    "CREATE": true,
24    "SHOW": true
25  },
26  "INPUT_DATA": {
27    "UPDATE": true,
28    "DELETE": true,
29    "CREATE": true,
30    "SHOW": true
31  }
32  ..,
33 }
```


B.VII JSON Result of Performance Metrics

```
1 {
2   "verificationMetrics": [
3     {
4       "averageExecutionTime": 638700.0,
5       "averageExecutionTimeInMs": 0.6387,
6       "averageNumberOfElements": 1.4444444444444444,
7       "numberOfExecutions": 18,
8       "totalExecutionTime": 11496600,
9       "totalExecutionTimeInMs": 11,
10      "totalNumberOfElements": 26,
11      "type": {
12        "description": "Detecting columns ..",
13        "name": "MissingInputDataVerification",
14        ..
15      }
16    },
17    {
18      "averageExecutionTime": 160728.57142857142,
19      "averageExecutionTimeInMs": 0.16072857142857142,
20      "averageNumberOfElements": 0.0,
21      "numberOfExecutions": 14,
22      "totalExecutionTime": 2250200,
23      "totalExecutionTimeInMs": 2,
24      "totalNumberOfElements": 0,
25      "type": {
26        "description": "Detecting rules ..",
27        "name": "IdenticalRuleVerification",
28        ..
29      }
30    },
31    ..
32  ]
33 }
```

B.VIII JSON Result of Verification Request

```

1 {
2   "id":988,
3   "size":2,
4   "verifier":[
5     {
6       "id":998,
7       "entries":[
8         {
9           "id":1043,
10          "elements":[
11            {
12              "id":1041,
13              "identifier":{
14                "inputId":"input_1",
15                "decisionId":"Decision_d5wskPnC",
16                "ruleId":"DecisionRule_1goyka6",
17                "decisionTableId":"decisionTable_1",
18                "inputEntryId":"UnaryTests_01mpi41",
19                "definitionId":"Definitions_kj2sMCzC"
20              }
21            }
22          ],
23          "message":"Decision \"Decision 1\", Column \"x\": .. was not found.....",
24          "size":1,
25          "verificationClassification":"WARNING",
26          "verificationFixes":[
27            {
28              "actions":[
29                {
30                  "id":92,
31                  "actionScope":"INPUT_ENTRY",
32                  "actionType":"SHOW",
33                  "actionValues":{ }
34                }
35              ],
36              "fixName":"Show"
37            },
38            {
39              "actions":[
40                {
41                  "id":1045,
42                  "actionScope":"INPUT_COLUMN",
43                  "actionType":"UPDATE",
44                  "actionValues":{
45                    "inputId":"input_1",
46                    "addPredVal":"x",
47                    "decisionId":"Decision_d5wskPnC"
48                  }
49                }
50              ],
51              "fixName":"Add pred. value \"x\""
52            },
53            {
54              "actions":[
55                {
56                  "id":1048,
57                  "actionScope":"RULE",
58                  "actionType":"DELETE",
59                  "actionValues":{
60                    "decisionId":"Decision_d5wskPnC",
61                    "ruleId":"DecisionRule_1goyka6"
62                  }
63                }
64              ],

```

```

65         "fixName":"Delete rule 1"
66     }
67 ]
68 }
69 ],
70 "executionTime":285901,
71 "size":1,
72 "type":{
73     "description":"Detecting string ...",
74     "name":"PredefinedExistingValueVerification",
75     "niceName":"Predefined Existing Value",
76     "classification":{
77         ...
78     }
79 }
80 ],
81 {
82     "id":990,
83     "entries":[
84         {
85             "id":1013,
86             "elements":[
87                 {
88                     "id":1009,
89                     "identifier":{
90                         "inputId":"input_1",
91                         "decisionId":"Decision_d5wskPnC",
92                         "decisionTableId":"decisionTable_1",
93                         "definitionId":"Definitions_kj2sMCzC"
94                     }
95                 }
96             ],
97             "message":"Decision 1: Input column has no input data node.",
98             "size":1,
99             "verificationClassification":"WARNING",
100             "verificationFixes":[
101                 {
102                     "actions":[
103                         {
104                             "id":98,
105                             "actionScope":"INPUT_COLUMN",
106                             "actionType":"SHOW",
107                             "actionValues":{ }
108                         }
109                     ],
110                     "fixName":"Show"
111                 },
112                 {
113                     "actions":[
114                         {
115                             "id":1019,
116                             "actionScope":"INPUT_DATA",
117                             "actionType":"CREATE",
118                             "actionValues":{
119                                 "name":"x",
120                                 "decisionId":"Decision_d5wskPnC"
121                             }
122                         }
123                     ],
124                     "fixName":"New Data Input"
125                 }
126             ],
127             "actions":[
128                 {
129                     "id":1026,
130                     "actionScope":"DECISION",
131                     "actionType":"CREATE",

```

```
132         "actionValues":{
133             "name":"Decision x",
134             "outputColumnName":"x",
135             "outputColumnTypeRef":"string",
136             "toDecisionId":"Decision_d5wskPnC"
137         }
138     },
139 ],
140     "fixName":"New Input Decision"
141 },
142 {
143     "actions":[
144         {
145             "id":1037,
146             "actionScope":"DECISION",
147             "actionType":"DELETE",
148             "actionValues":{
149                 "decisionId":"Decision_d5wskPnC"
150             }
151         }
152     ],
153     "fixName":"Delete decision"
154 }
155 ]
156 }
157 ],
158 "executionTime":526299,
159 "size":1,
160 "type":{
161     "description":"...",
162     "name":"MissingInputDataVerification",
163     "niceName":"Missing Input",
164     "classification":{
165         ...
166     }
167 }
168 ]
169 ]
170 }
```

B.IX Java Example Verifier Implementation

```
1 @DmnVerifier(  
2     classification = DrdModelingLevelVerification.class,  
3     name = "ExampleVerification",  
4     niceName = "Example DRD verification",  
5     description = "Detection of awesome input data nodes.")  
6 public class ExampleVerifier extends AbstractVerifier {  
7  
8     @Override  
9     protected void doVerification() {  
10        // iterate over all InputData nodes  
11        for (VDmnInputData i :  
12            dmnObjectContainer.getVDmnDefinition()  
13                .getVDmnInputData()) {  
14            // add element to result  
15            if (awesomeCheckFunction(i)) {  
16                // Add the id of the input data to the result object  
17                vreFactory  
18                    .addElement(VerificationResultEntryElement.create(i))  
19                    // add show fix  
20                    .addVerificationFix(SHOW_INPUT_DATA)  
21                    // add to result object with message  
22                    .addToEntry(INFO, "Input data is awesome!.",  
23                        i.getName());  
24            }  
25        }  
26  
27        private boolean awesomeCheckFunction(VDmnInputData i) {  
28            // awesome check  
29            return i.getName().get().getValue().contains("awesome");  
30        }  
31    }
```

C Appendix: Evaluation

C.I DMN Generator

```
1 DmnGenerator dmnGenerator = new DmnGenerator();
2 List<String> dmns = dmnGenerator.generateModelsAsString(
3     GeneratorConfiguration.getBuilder()
4         .withNumberOfDefinitions(8)
5         .withNumberOfDecisions(4)
6         .withNumberOfInputData(1)
7         .withProbabilityOfConnectionDecision(0.2d)
8         .withProbabilityOfConnectionInputData(0.5d)
9         .withNumberOfUniqueNames(4)
10        .withNumberOfUniqueStringValue(10)
11        .withMinNumberOfInputColumns(3)
12        .withMaxNumberOfInputColumns(3)
13        .withMinNumberOfOutputColumns(2)
14        .withMaxNumberOfOutputColumns(2)
15        .withMinNumberOfRules(10)
16        .withMaxNumberOfRules(100)
17        .addHitPolicy(HitPolicy.UNIQUE)
18        .addPossibleTypeRef("string")
19        .addPossibleTypeRef("integer")
20        .addPossibleTypeRef("boolean")
21        .build());
```