

An OMG® Decision Model and Notation™ Publication



Decision Model and Notation

Version 1.7 Beta 1

OMG Document Number: [d1c/2026-06-02](#)

Release Date: June 2026

Standard Document URL: <https://www.omg.org/spec/DMN>

Copyright © 2019-2021, 88solutions
Copyright © 2019-20264, BOC Products & Services AG
Copyright © 2021-2024, BPM Advantage Consulting
Copyright © 2015-20264, Camunda Services GmbH
Copyright © 2013-20264, Blue Polaris Inc, formerly Decision Management Solutions
Copyright © 2019-2021, Department of Veterans Affairs
Copyright © 2013-2019, Escape Velocity LLC
Copyright © 2013-20264, Fair Isaac Corporation
Copyright © 2019-2024, GfSE e.V.
Copyright © 2013-20264, International Business Machines Corporation
Copyright © 2013-20264, KU Leuven
Copyright © 2013-2019, Model Systems Limited
Copyright © 2015-2019, Oracle Incorporated
Copyright © 2019-2024, PNA Group
Copyright © 2020-20264, processCentric GmbH
Copyright © 2013-2023, Red Hat Inc
Copyright © 2013-20264, Sapiens Decision NA
Copyright © 2019-2021, Signavio GmbH
Copyright © 2022-20264, Softeam
Copyright © 2019-20264, Sparx Systems Pty Ltd
Copyright © 2019-20264, Thematrix Partners LLC
Copyright © 2014-2019, TIBCO Software Inc.
Copyright © 2015-20264, Trisotech
Copyright © 2015-2025, Object Management Group, Inc.

Commented [AF1]: Editorial

USE OF SPECIFICATION – TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 9C Medway Rd, PMB 274, Milford, MA 01757, U.S.A.

TRADEMARKS

CORBA®, CORBA logos®, FIBO®, Financial Industry Business Ontology®, FINANCIAL INSTRUMENT GLOBAL IDENTIFIER®, IIOP®, IMM®, Model Driven Architecture®, MDA®, Object Management Group®, OMG®, OMG Logo®, SoaML®, SOAML®, SysML®, UAF®, Unified Modeling Language®, UML®, UML Cube Logo®, VSIPL®, and XMI® are registered trademarks of the Object Management Group, Inc.

For a complete list of trademarks, see: https://www.omg.org/legal/tm_list.htm. All other products or company names mentioned are used for identification purposes only and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process, we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page <https://www.omg.org>, under Specifications, Report a Bug/Issue.

Table of Contents

Preface	x
1 Scope	1
2 Conformance	1
2.1 Conformance levels.....	1
2.2 General conformance requirement	2
2.2.1 Visual appearance.....	2
2.2.2 Decision semantics	2
2.2.3 Attributes and model associations	2
3 References	3
3.1 Normative	3
3.2 Non-normative.....	4
4 Additional information	5
4.1 Acknowledgements	5
4.2 IPR and Patents	7
4.3 Guide to the Specification	7
5 Introduction to DMN	10
5.1 Context.....	10
5.2 Scope and uses of DMN	12
5.2.1 Modeling human decision-making	12
5.2.2 Modeling requirements for automated decision-making	13
5.2.3 Implementing automated decision-making	14
5.2.4 Combining applications of modelling.....	14
5.3 Basic concepts.....	15
5.3.1 Decision requirements level	15
5.3.2 Decision logic level	16
5.3.3 Decision services.....	18
6 Requirements (DRG and DRD)	22
6.1 Introduction.....	22
6.2 Notation	22
6.2.1 DRD Elements	24
6.2.2 DRD Requirements.....	25
6.2.3 Connection rules	27
6.2.4 Partial views and hidden information.....	28
6.2.5 Decision service.....	29
6.2.6 Identifying Collections.....	31
6.3 Metamodel.....	32
6.3.1 DMN Element metamodel.....	32
6.3.2 Definitions metamodel.....	34
6.3.3 Import metamodel	36
6.3.4 Element Collection metamodel	37
6.3.5 DRG Element metamodel	37
6.3.6 Artifact metamodel.....	38
6.3.7 Decision metamodel.....	39
6.3.8 Business Context Element metamodel	41
6.3.9 Business Knowledge Model metamodel	43
6.3.10 Decision service metamodel	44
6.3.11 Input Data metamodel	46
6.3.12 Knowledge Source metamodel.....	47
6.3.13 Information Requirement metamodel	47
6.3.14 Knowledge Requirement metamodel.....	48
6.3.15 Authority Requirement metamodel	49
6.3.16 Extensibility	50
6.4 Examples.....	51

7	Relating Decision Logic to Decision Requirements.....	53
7.1	Introduction.....	53
7.2	Notation	55
7.2.1	Expressions.....	55
7.2.2	Boxed literal expression.....	55
7.2.3	Boxed invocation.....	56
7.3	Metamodel.....	57
7.3.1	Expression metamodel.....	58
7.3.2	UnaryTests Metamodel	59
7.3.3	ItemDefinition metamodel.....	59
7.3.4	InformationItem metamodel.....	62
7.3.5	Literal expression metamodel.....	63
7.3.6	Invocation metamodel	64
7.3.7	Binding metamodel.....	65
7.3.8	Error Handling	65
8	Decision Table	68
8.1	Introduction.....	68
8.2	Notation	71
8.2.1	Line style and color.....	71
8.2.2	Table orientation.....	71
8.2.3	Input expressions	73
8.2.4	Input values	73
8.2.5	Information Item names, output labels, and output component names.....	74
8.2.6	Output values	74
8.2.7	Multiple outputs	74
8.2.8	Input entries	75
8.2.9	Merged input entry cells.....	75
8.2.10	Output entry	76
8.2.11	Hit policy	76
8.2.12	Default output values	78
8.3	Metamodel.....	79
8.3.1	Decision Table metamodel	79
8.3.2	Decision Table Input and Output metamodel	80
8.3.3	Decision Rule metamodel.....	82
8.4	Examples.....	83
9	Simple Expression Language (S-FEEL)	86
9.1	Introduction.....	86
9.2	S-FEEL syntax	86
9.3	S-FEEL data types.....	88
9.4	S-FEEL semantics.....	88
9.5	Use of S-FEEL expressions	89
9.5.1	Item definitions	89
9.5.2	Invocations.....	89
9.5.3	Decision tables	89
10	Expression Language (FEEL).....	92
10.1	Introduction.....	92
10.2	Notation	92
10.2.1	Boxed Expressions	92
10.2.2	FEEL	103
10.3	Full FEEL Syntax and Semantics.....	104
10.3.1	Syntax	105
10.3.2	Semantics.....	111
10.3.3	XML Data	141
10.3.4	Built-in functions	144
10.3.5	FEEL Libraries.....	163
10.4	Execution Semantics of Decision Services	164
10.5	Metamodel.....	165
10.5.1	Context metamodel.....	166

10.5.2	ContextEntry metamodel.....	166
10.5.3	FunctionDefinition metamodel.....	166
10.5.4	List metamodel.....	167
10.5.5	Relation metamodel.....	167
10.5.6	Conditional metamodel.....	168
10.5.7	ChildExpression metamodel.....	168
10.5.8	Filter metamodel.....	168
10.5.9	Iterator metamodel.....	169
10.5.10	For metamodel.....	169
10.5.11	Quantified metamodel.....	169
10.5.12	Every metamodel.....	169
10.5.13	Some metamodel.....	170
10.6	Examples.....	170
10.6.1	Context.....	170
10.6.2	Calculation.....	171
10.6.3	If, In.....	171
10.6.4	Sum entries of a list.....	171
10.6.5	Invocation of user-defined PMT function.....	171
10.6.6	Sum weights of a recent credit history.....	171
10.6.7	Determine if credit history contain a bankruptcy event.....	172
11	B-FEEL.....	174
11.1	Introduction.....	174
11.2	Operator and built-in functions returning a Boolean.....	174
11.3	Built-in functions returning a number.....	175
11.4	Built-in functions returning a string.....	176
11.5	Built-in functions returning a date and time, date and time.....	176
11.6	Built-in functions returning a duration.....	176
11.7	Built-in functions returning a collection and expressions using a collection.....	177
11.8	Built-in functions returning a range.....	177
11.9	Semantics of addition and subtraction.....	177
11.10	Semantics of multiplication and division.....	178
11.11	Semantics of exponentiation.....	179
12	DMN Examples.....	181
12.1	Example 1: Originations.....	181
12.1.1	Introduction.....	181
12.1.2	The business process model.....	181
12.1.3	The decision requirements level.....	182
12.1.4	The decision logic level.....	194
12.1.5	Executing the Decision Model.....	206
12.2	Example 2: Ranked Loan Products.....	207
13	Exchange Formats.....	226
13.1	Interchanging Incomplete Models.....	226
13.2	Machine Readable Files.....	226
13.3	XSD.....	226
13.3.1	Document Structure.....	226
13.3.2	References within the DMN XSD.....	227
14	DMN Diagram Interchange (DMN DI).....	230
14.1	Scope.....	230
14.2	Diagram Definition and Interchange.....	230
14.3	How to read this chapter.....	230
14.4	DMN Diagram Interchange Meta-Model.....	230
14.4.1	Overview.....	230
14.4.2	DMNDI [Class].....	231
14.4.3	DMNDiagram [Class].....	232
14.4.4	DMNDiagramElement [Class].....	233
14.4.5	DMNShape [Class].....	234
14.4.6	DMNEdge [Class].....	235

14.4.7 DMNLabel [Class]	236
14.4.8 DMNStyle [Class]	237
14.5 Notation Depiction Library and Abstract Element Resolutions	238
14.5.1 Labels.....	239
14.5.2 DMNShape Resolution	239
14.5.3 DMNEdge Resolution.....	242
ANNEXES	244
Annex A Relation to BPMN (informative).....	247
A.1 Goals of BPMN and DMN	247
A.2 BPMN Tasks and DMN Decisions	247
A.3 Types of BPMN Tasks relevant to DMN	248
A.4 Process gateways and Decisions.....	249
A.5 Linking BPMN and DMN Models	249
a) Associating Decisions with Tasks and Processes	250
b) Decision Services	250
Annex B Glossary	252

Preface

About the Object Management Group

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Meta-model); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at <https://www.omg.org/>.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Formal Specifications are available from this URL: <https://www.omg.org/spec>

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
9C Medway Road, PMB 274
Milford, MA 01757
USA

Tel: +1-781-444-0404
Fax: +1-781-444-0320

Email: pubs@omg.org

Certain OMG specifications are also available as ISO/IEC standards. Please consult: <http://www.iso.org>

Issues

The reader is encouraged to report and technical or editing issues/problems with this specification to:
https://www.omg.org/report_issue.htm

1 Scope

The primary goal of **DMN** is to provide a common notation that is readily understandable by all business users, from the business analysts needing to create initial decision requirements and then more detailed decision models, to the technical developers responsible for automating the decisions in processes, and finally, to the businesspeople who will manage and monitor those decisions. **DMN** creates a standardized bridge for the gap between the business decision design and decision implementation. **DMN** notation is designed to be usable alongside the standard **BPMN** business process notation.

Another goal is to ensure that decision models are interchangeable across organizations via an XML representation.

The authors have brought forth expertise and experience from the existing decision modeling community and have sought to consolidate the common ideas from these divergent notations into a single standard notation.

2 Conformance

2.1 Conformance levels

Software may claim compliance or conformance with **DMN** if and only if the software fully matches the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim that the software was based on this specification but may not claim compliance or conformance with this specification.

The specification defines three levels of conformance, namely **Conformance Level 1**, **Conformance Level 2**, and **Conformance Level 3**.

An implementation claiming conformance to Conformance Level 1 is not required to support Conformance Level 2 or Conformance Level 3. An implementation claiming conformance to Conformance Level 2 is not required to support Conformance Level 3.

An implementation claiming conformance to **Conformance Level 1** SHALL comply with all of the specifications set forth in clauses 6 (Decision Requirements), 7 (Decision Logic) and 8 (Decision Table) of this document. An implementation claiming conformance to Conformance Level 1 is never required to interpret expressions (modeled as an Expression elements) in decision models. However, to the extent that an implementation claiming conformance to Conformance Level 1 provides an interpretation to an expression, that interpretation SHALL be consistent with the semantics of expressions as specified in clause 7.

An implementation claiming conformance to **Conformance Level 2** SHALL comply with all of the specifications set forth in clauses 6 (Decision Requirements), 7 (Decision Logic) and 8 (Decision Table) of this document. In addition, it is required to interpret expressions in the simple expression language (S-FEEL) specified in clause 9.

An implementation claiming conformance to **Conformance Level 3** SHALL comply with all of the specifications set forth in clauses 6 (Decision Requirements), 7 (Decision Logic), 8 (Decision Table) and 10 (Expression language) of this document. An implementation does NOT need to support any Function Kind other than FEEL to claim conformance to Level 3, i.e. support for Java, PMML, and ONNX is optional. Notice that the simple expression language that is specified in clause 9 is a subset of FEEL, and that, therefore, an implementation claiming conformance to Conformance Level 3 can also claim conformance to Conformance Level 2 (and to Conformance Level 1).

In addition, an implementation claiming conformance to any of the three **DMN** conformance levels SHALL comply with all of the requirements set forth in Clause 2.2.

2.2 General conformance requirement

2.2.1 Visual appearance

A key element of **DMN** is the choice of shapes and icons used for the graphical elements identified in this specification. The intent is to create a standard visual language that all decision modelers will recognize and understand. An implementation that creates and displays decision model diagrams SHALL use the graphical elements, shapes, and markers illustrated in this specification.

There is flexibility in the size, color, line style, and text positions of the defined graphical elements, except where otherwise specified.

The following extensions to a **DMN** Diagram are permitted:

- New markers or indicators MAY be added to the specified graphical elements. These markers or indicators could be used to highlight a specific attribute of a DMN element or to represent a new subtype of the corresponding concept.
- A new shape representing a new kind of artifact MAY be added to a Diagram, but the new shape SHALL NOT conflict with the shape specified for any other DMN element or marker.
- Graphical elements MAY be colored, and the coloring may have specified semantics that extend the information conveyed by the element as specified in this standard.
- The line style of a graphical element MAY be changed, but that change SHALL NOT conflict with any other line style required by this specification.

2.2.2 Decision semantics

This specification defines many semantic concepts used in defining decisions and associates them with graphical elements, markers, and connections.

To the extent that an implementation provides an interpretation of some **DMN** diagram element as a semantic specification of the associated concept, the interpretation SHALL be consistent with the semantic interpretation herein specified.

2.2.3 Attributes and model associations

This specification defines a number of attributes and properties of the semantic elements represented by the graphical elements, markers, and connections. Some attributes are specified as mandatory but have no representation or only optional representation. And some attributes are specified as optional.

For every attribute or property that is specified as mandatory, a conforming implementation SHALL provide some mechanism by which values of that attribute or property can be created and displayed. This mechanism SHALL permit the user to create or view these values for each **DMN** element specified to have that attribute or property.

Where a graphical representation for that attribute or property is specified as required, that graphical representation SHALL be used. Where a graphical representation for that attribute or property is specified as optional, the implementation MAY use either a graphical representation or some other mechanism.

If a graphical representation is used, it SHALL be the representation specified. Where no graphical representation for that attribute or property is specified, the implementation MAY use either a graphical representation or some other mechanism. If a graphical representation is used, it SHALL NOT conflict with the specified graphical representation of any other **DMN** element.

3 References

3.1 Normative

BMM

- *Business Motivation Model (BMM), Version 1.2*, OMG Document number: formal/2014-05-01, May 2014 <https://www.omg.org/spec/BMM/1.2>

BPMN 2.0

- *Business Process Model and Notation, version 2.0*, OMG Document Number: formal/2011-01-03, January 2011 <https://www.omg.org/spec/BPMN/2.0>

CQL

- *Clinical Quality Language*, V1.4, HL7 <https://cql.hl7.org/09-b-cqlreference.html#interval-operators-3>

IEEE 754

- *IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic*, International Electrical and Electronics Engineering Society, December, 2008 <https://www.techstreet.com/ieee/searches/5835853>

ISO 8601

- *ISO 8601:2004, Data elements and interchange formats -- Information interchange -- Representation of dates and times*, International Organization for Standardization, 2004 https://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=40874

ISO EBNF

- *ISO/IEC 14977:1996, Information technology -- Syntactic metalanguage -- Extended BNF*, International Organization for Standardization, 1996 [https://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996\(E\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip)

Java

- *The Java Language Specification, Java SE 7 Edition*, Oracle Corporation, February 2013 <https://docs.oracle.com/javase/specs/jls/se7/jls7.pdf>

ONNX

- <https://onnx.ai/>

PMML

- *Predictive Model Markup Language (PMML)*, Data Mining Group, May, 2014 <https://www.dmg.org/v4-2-1/GeneralStructure.html>

RFC 3986

- *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*. Berners-Lee, T., Fielding, R., and Masinter, L, editors. Internet Engineering Task Force, 2005. <https://www.ietf.org/rfc/rfc3986.txt>

SCE

- *Specification Common Elements (SCE), version 1.0* <https://www.omg.org/spec/SCE/1.0>

UML

- *Unified Modeling Language (UML), v2.4.1*, OMG Document Number formal/2011-08-05, August 2011 <https://www.omg.org/spec/UML/2.4.1>

XBASE

- *XML Base (Second Edition)*. Jonathan Marsh and Richard Tobin, editors. World Wide Web Consortium, 2009. <https://www.w3.org/TR/xmlbase/>

XML

- *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, W3C Recommendation 26 November 2008 <https://www.w3.org/TR/xml/>

XML Schema

- *XML Schema Part 2: Datatypes Second Edition*, W3C Recommendation 28 October 2004 <https://www.w3.org/TR/xmlschema-2/>

XPath Data Model

- *XQuery 3.1+0 and XPath 3.1+0 Data Model (XDM) (Second Edition)*, W3C Recommendation 14 December 2010 <https://www.w3.org/TR/xpath-datamodel/>

Commented [AF3]: DMN17-73

XQuery and XPath Functions and Operators

- *XQuery 3.1+0 and XPath 3.1+0 Functions and Operators (Second Edition)*, W3C Recommendation 14 December 2010 <https://www.w3.org/TR/xpath-functions/XQuery>

Commented [AF4]: DMN17-73

3.2 Non-normative

JSON

- *ECMA-404 The JSON Data Interchange Standard*, European Computer Manufacturers Association, October, 2013 <https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>

PRR

- *Production Rule Representation (PRR), Version 1.0*, December 2009, OMG document number formal/2009-12-01 <https://www.omg.org/spec/PRR/1.0/>

RIF

- *RIF production rule dialect*, Ch. de Sainte Marie et al. (Eds.), W3C Recommendation, 22 June 2010. <https://www.w3.org/TR/rif-prd/>

SBVR

- *Semantics of Business Vocabulary and Business Rules (SBVR)*, V1.2, OMG document number formal/2013-11-04, November 2013 <https://www.omg.org/spec/SBVR/1.2/>

SQL

- ISO/IEC 9075-11:2011, Information technology -- Database languages -- SQL -- Part 11: Information and Definition Schemas (SQL/Schemata), International Organization for Standardization, 2011 https://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=5368

XPath

- *XML Path Language (XPath) Version 1.0*, W3C Recommendation 16 November 1999 <https://www.w3.org/TR/xpath>

4 Additional information

4.1 Acknowledgements

The following companies submitted version 1.0 of this specification:

- Decision Management Solutions
- Escape Velocity
- FICO
- International Business Machines
- Oracle

The following companies supported this specification:

- KU Leuven
- Knowledge Partners International
- Model Systems
- TIBCO

The following persons were members of the core team that contributed to the content specification: Martin Chapman, Bob Daniel, Alan Fish, Larry Goldberg, John Hall, Barbara von Halle, Gary Hallmark, Dave Ings, Christian de Sainte Marie, James Taylor, Jan Vanthienen, Paul Vincent. In addition, the following persons contributed valuable ideas and feedback that improved the content and the quality of this specification: Bas Janssen, Robert Lario, Pete Rivett.

Version 1.1 was developed by the following persons and companies: Elie Abi-Lahoud, University College Cork; Justin Brunt, TIBCO; Alan Fish, FICO; John Hall, Rule ML Initiative; Denis Gagne, Trisotech; Gary Hallmark, Oracle; Elisa Kendall, Thematrix Partners LLC; Manfred Koethe, 88solutions; Falko Menge, Camunda Services GmbH; Zbigniew Misiak, BOC Information Technologies Consulting; Sijr Nijssen, PNA Group; Mihail Popov, MITRE; Pete Rivett, Adaptive; Bruce Silver, Bruce Silver Associates; Bastian Steinert, Signavio GmbH; Tim Stephenson, Omny Link; James Taylor, Decision Management Solutions; Jan Vanthienen, K.U. Leuven; Paul Vincent, Knowledge Partners, Inc.

Version 1.2 was developed by the following persons and companies: Alan Fish, FICO; Denis Gagne, Trisotech; Gary Hallmark, Oracle; Elisa Kendall, Thematrix Partners LLC; Manfred Koethe, 88solutions; Falko Menge, Camunda Services GmbH; Zbigniew Misiak, BOC Products & Services AG; Sijr Nijssen, PNA Group; Octavian Patrascoiu, Goldman Sachs; Bruce Silver, Bruce Silver Associates; Gil Ronen, Sapiens DECISION; Caroline Scharf, Tom Sawyer Software; Bastian Steinert, Signavio GmbH; James Taylor, Decision Management Solutions; Edson Tirelli, Red Hat; Jan Vanthienen, K.U. Leuven; Stephen White, Department of Veterans Affairs.

Version 1.3 was developed by the following persons and companies: Alan Fish, FICO; Denis Gagne, Trisotech; Gary Hallmark, Oracle; Uwe Kaufmann, GfSE e.V.; Elisa Kendall, Thematrix Partners LLC; Manfred Koethe, 88solutions; Robert Lario, Department of Veterans Affairs; Falko Menge, Camunda Services GmbH; Zbigniew Misiak, BOC Products & Services AG; Matteo Mortari, Red Hat; Sijr Nijssen, PNA Group; Octavian Patrascoiu, Goldman Sachs; Bruce Silver, Bruce Silver Associates; Gil Segal, Sapiens Decision NA; Bastian Steinert, Signavio GmbH; James Taylor, Decision Management Solutions; Edson Tirelli, Red Hat; Jan Vanthienen, K.U. Leuven; Stephen White, Department of Veterans Affairs.

Version 1.4 was developed from December 2019 to December 2021 by the following persons and companies:

- (chair) Falko Menge, Camunda Services GmbH
- (chair) Alan Fish, FICO
- Bastian Steinert, Signavio GmbH
- Denis Gagne, Trisotech
- Edson Tirelli, Red Hat
- Elisa Kendall, Thematrix Partners LLC
- Gil Segal, Sapiens Decision NA
- J.D. Baker, Sparx Systems Pty Ltd
- James Taylor, Decision Management Solutions

- Jan Vanthienen, K.U. Leuven
- Manfred Koethe, 88solutions
- Matteo Mortari, Red Hat
- Pete Rivett, agnos.ai UK Ltd
- Serge Schiltz, processCentric GmbH
- Sijr Nijssen, PNA Group
- Stephen White, Department of Veterans Affairs
- Uwe Kaufmann, GfSE e.V.
- Zbigniew Misiak, BOC Products & Services AG

In addition, the following persons contributed valuable ideas and feedback that improved the content and the quality of version 1.4 of this specification:

- Daniel Tanner, ACTICO GmbH
- Greg McCreath, Montera
- Keith Swenson, Fujitsu
- Philipp Ossler, Camunda Services GmbH
- Simon Ringuette, Trisotech

Version 1.5 was developed from December 2021 to March 2023 by the following persons and companies:

- (chair) Falko Menge, Camunda Services GmbH
- (chair) Alan Fish, FICO
- Alessandra Bagnato, Softeam
- Denis Gagne, Trisotech
- Elisa Kendall, Thematix Partners LLC
- Gil Segal, Sapiens Decision NA
- J.D. Baker, Sparx Systems Pty Ltd
- James Taylor, Decision Management Solutions
- Jan Vanthienen, K.U. Leuven
- Matteo Mortari, Red Hat
- Octavian Patrascoiu, Goldman Sachs
- Pete Rivett, agnos.ai UK Ltd
- Serge Schiltz, processCentric GmbH
- Sijr Nijssen, PNA Group
- Stephen White, BPM Advantage Consulting
- Tibor Zimanyi, International Business Machines
- Uwe Kaufmann, GfSE e.V.
- Zbigniew Misiak, BOC Products & Services AG

In addition, the following persons contributed valuable ideas and feedback that improved the content and the quality of version 1.5 of this specification:

- Maciej Barelkowski, Camunda Services GmbH
- Philipp Ossler, Camunda Services GmbH
- Simon Ringuette, Trisotech

Version 1.6 was developed from March 2023 to May 2024 by the following persons and companies:

- (chair) Falko Menge, Camunda Services GmbH
- (chair) Alan Fish, FICO
- Alessandra Bagnato, Softeam
- Denis Gagne, Trisotech
- Elisa Kendall, Thematix Partners LLC
- Gil Segal, Sapiens Decision NA
- J.D. Baker, Sparx Systems Pty Ltd
- James Taylor, Decision Management Solutions
- Jan Vanthienen, K.U. Leuven
- Octavian Patrascoiu, Goldman Sachs
- Serge Schiltz, processCentric GmbH
- Sijr Nijssen, PNA Group
- Stephen White, BPM Advantage Consulting
- Tibor Zimanyi, International Business Machines
- Uwe Kaufmann, GfSE e.V.

- Zbigniew Misiak, BOC Products & Services AG

In addition, the following persons contributed valuable ideas and feedback that improved the content and the quality of version 1.6 of this specification:

- Maciej Barelkowski, Camunda Services GmbH
- Philipp Ossler, Camunda Services GmbH
- Simon Ringuette, Trisotech

Version 1.7 was developed from February 2025 to June 2026 by the following persons and companies:

- (chair) Falko Menge, Camunda Services GmbH
- (chair) Alan Fish, FICO
- Alessandra Bagnato, Softeam
- Denis Gagne, Trisotech
- Elisa Kendall, Thematrix Partners LLC
- Gil Segal, Sapiens Decision NA
- J.D. Baker, Sparx Systems Pty Ltd
- James Taylor, Blue Polaris Inc
- Jan Vanthienen, K.U. Leuven
- Octavian Patrascioiu, Goldman Sachs
- Serge Schiltz, processCentric GmbH
- Tibor Zimanyi, International Business Machines
- Zbigniew Misiak, BOC Products & Services AG

In addition, the following persons contributed valuable ideas and feedback that improved the content and the quality of version 1.7 of this specification:

- Maciej Barelkowski, Camunda Services GmbH
- Philipp Ossler, Camunda Services GmbH
- Simon Ringuette, Trisotech

Commented [AF5]: Editorial

4.2 IPR and Patents

The submitters contributed this work to OMG on a RF on RAND basis.

4.3 Guide to the Specification

Clause 1 summarizes the goals of the specification.

Clause 2 defines three levels of conformance with the specification: Conformance Level 1, Conformance Level 2, and Conformance Level 3.

Clause 3 lists normative references.

Clause 4 provides additional information useful in understanding the background to and structure of the specification.

Clause 5 discusses the scope and uses of **DMN** and introduces the principal concepts, including the two levels of **DMN**: the decision requirements level and the decision logic level.

Clause 6 defines the decision requirements level of **DMN**: the Decision Requirements Graph (DRG) and its notation as a Decision Requirements Diagram (DRD).

Clause 7 introduces the principles by which decision logic may be associated with elements in a DRG: i.e., how the decision requirements level and decision logic level are related to each other. Clauses 8, 9 and 10 then define the decision logic level of **DMN**:

- Clause 8 defines the notation and syntax of Decision Tables in **DMN**.

- Clause 9 defines S-FEEL: a subset of FEEL to support decision tables.
- Clause 10 defines the full syntax and semantics of FEEL: the default expression language used for the Decision Logic level of **DMN**.

Clause 11 provides examples of **DMN** used to model human and automated decision-making.

Clause 12 addresses exchange formats and provides references to machine-readable files (XSD and XMI). The Annexes provide non-normative background information:

- Annex A. discusses the relationship between **DMN** and **BPMN**.
- Annex B. provides a glossary of terms.

This page intentionally left blank.

5 Introduction to DMN

5.1 Context

The purpose of **DMN** is to provide the constructs that are needed to model decisions, so that organizational decision-making can be readily depicted in diagrams, accurately defined by business analysts, and (optionally) automated.

Decision-making is addressed from two different perspectives by existing modeling standards:

- Business process models (e.g., **BPMN**) can describe the coordination of decision-making within business processes by defining specific tasks or activities within which the decision-making takes place.
- Decision logic (e.g., PRR, PMML) can define the specific logic used to make individual decisions, for example as business rules, decision tables, or executable analytic models.

However, a number of authors (including members of the submission team) have observed that decision-making has an internal structure which is not conveniently captured in either of these modeling perspectives. Our intention is that **DMN** will provide a third perspective – the Decision Requirements Diagram – forming a bridge between business process models and decision logic models:

- Business process models will define tasks within business processes where decision-making is required to occur.
- Decision Requirements Diagrams will define the decisions to be made in those tasks, their interrelationships, and their requirements for decision logic.
- Decision logic will define the required decisions in sufficient detail to allow validation and/or automation.

Taken together, Decision Requirements Diagrams and decision logic can provide a complete decision model which complements a business process model by specifying in detail the decision-making carried out in process tasks. The relationships between these three aspects of modeling are shown in Figure 5-1.

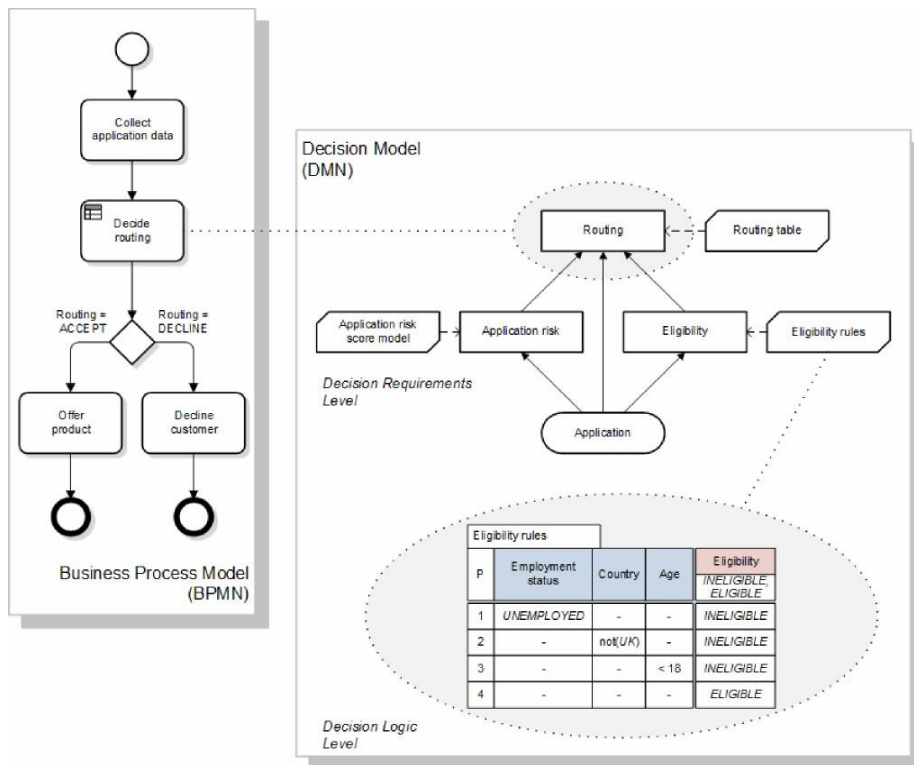


Figure 5-1: Aspects of modeling

The resulting connected set of models will allow detailed modeling of the role of business rules and analytic models in business processes, cross-validation of models, top-down process design and automation, and automatic execution of decision-making (e.g., by a business process management system calling a decision service deployed from a business rules management system).

Although Figure 5-1 shows a linkage between a business process model and a decision model for the purposes of explaining the relationship between **DMN** and other standards, it must be stressed that **DMN** is not dependent on **BPMN**, and its two levels – decision requirements and decision logic – may be used independently or in conjunction to model a domain of decision-making without any reference to business processes (see Figure 5-2).

DMN will provide constructs spanning both decision requirements and decision logic modeling. For decision requirements modeling, it defines the concept of a Decision Requirements Graph (DRG) comprising a set of elements and their connection rules, and a corresponding notation: The Decision Requirements Diagram (DRD). For decision logic modeling it provides a language called FEEL for defining and assembling decision tables, calculations, if/then/else logic, simple data structures, and externally defined logic from Java, ONNX and PMML into executable expressions with formally defined semantics. It also provides a notation for decision logic (“boxed expressions”) allowing components of the decision logic level to be drawn graphically and associated with elements of a Decision Requirements Diagram. The relationship between these constructs is shown in Figure 5-2.

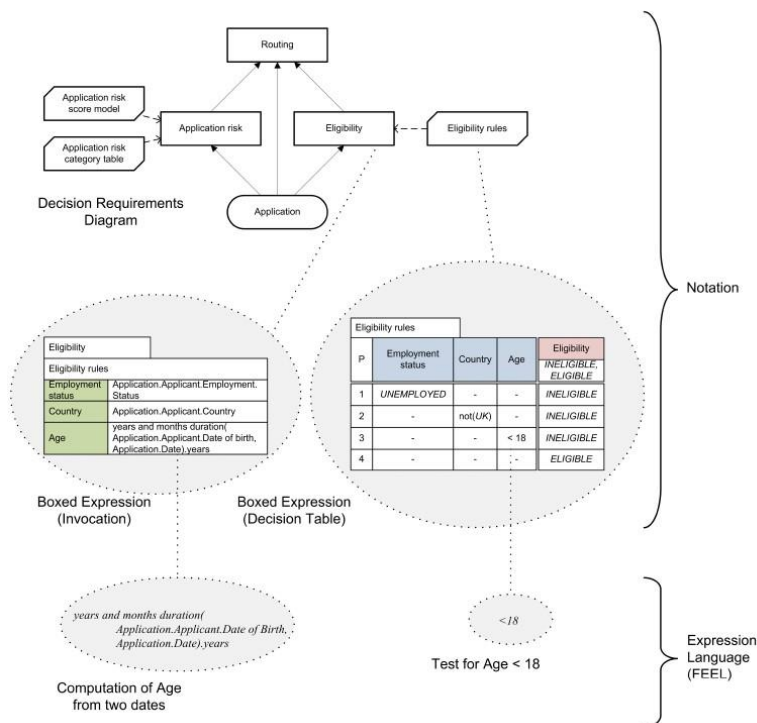


Figure 5-2: DMN Constructs

5.2 Scope and uses of DMN

Decision modeling is carried out by business analysts in order to understand and define the decisions used in a business or organization. Such decisions are typically operational decisions made in day-to-day business processes, rather than the strategic decision-making for which fewer rules and representations exist.

Three uses of **DMN** can be discerned in this context:

1. For modeling human decision-making.
2. For modeling the requirements for automated decision-making.
3. For implementing automated decision-making.

5.2.1 Modeling human decision-making

DMN may be used to model the decisions made by personnel within an organization. Human decision-making can be broken down into a network of interdependent constituent decisions and modeled using a DRD. The decisions in the DRD would probably be described at quite a high level, using natural language rather than decision logic.

Knowledge sources may be defined to model governance of decision-making by people (e.g., a manager), regulatory bodies (e.g., an ombudsman), documents (e.g., a policy booklet) or bodies of legislation (e.g., a government statute). These knowledge sources may be linked together, for example to show that a decision is governed (a) by a set of regulations defined by a regulatory body, and (b) by a company policy document maintained by a manager.

Business knowledge models may be used to represent specific areas of business knowledge drawn upon when making decisions. This will allow **DMN** to be used as a tool for formal definition of requirements for knowledge management. Business knowledge models may be linked together to show the interdependencies between areas of knowledge (in a manner similar to that used in the existing technique of Knowledge Structure Mapping). Knowledge sources may be linked to the business knowledge models to indicate how the business knowledge is governed or maintained, for example to show that a set of business policies (the business knowledge model) is defined in a company policy document (the knowledge source).

In some cases, it may be possible to define specific rules or algorithms for the decision-making. These may be modeled using decision logic (e.g., business rules or decision tables) to specify business knowledge models in the DRD, either descriptively (to record how decisions are currently made, or how they were made during a particular period of observation) or prescriptively (to define how decisions should be made or will be made in the future).

Decision-making modeled in **DMN** may be mapped to tasks or activities within a business process modeled using **BPMN**. At a high level, a collaborative decision-making task may be mapped to a subset of decisions in a DRD representing the overall decision-making behavior of a group or department. At a more detailed level, it is possible to model the interdependencies between decisions made by a number of individuals or groups using **BPMN** collaborations: each participant in the decision-making is represented by a separate pool in the collaboration and a separate DRD in the decision model. Decisions in those DRDs are then mapped to tasks in the pools, and input data in the DRDs are mapped to the content of messages passing between the pools.

The combined use of **BPMN** and **DMN** thus provides a graphical language for describing multiple levels of human decision-making within an organization, from activities in business processes down to a detailed definition of decision logic. Within this context **DMN** models will describe collaborative organizational decisions, their governance, and the business knowledge required for them.

5.2.2 Modeling requirements for automated decision-making

The use of **DMN** for modeling the requirements for automated decision-making is similar to its use in modeling human decision-making, except that it is entirely prescriptive, rather than descriptive, and there is more emphasis on the detailed decision logic.

For full automation of decisions, the decision logic must be complete, i.e., capable of providing a decision result for any possible set of values of the input data.

However, partial automation is more common, where some decision-making remains the preserve of personnel. Interactions between human and automated decision-making may be modeled using collaborations as above, with separate pools for human and automated decision-makers, or more simply by allocating the decision-making to separate tasks in the business process model, with user tasks for human decision-making and business rule tasks for automated decision-making. So, for example, an automated business rules task might decide to refer some cases to a human reviewer; the decision logic for the automated task needs to be specified in full but the reviewer's decision-making could be left unspecified.

Once decisions in a DRD are mapped to tasks in a **BPMN** business process flow, it is possible to validate across the two levels of models. For example, it is possible to verify that all input data in the DRDs are provided by previous tasks in the business process, and that the business process uses the results of decisions only in subsequent tasks or gateways. **DMN** models the relationships between Decisions and Business Processes so that the Decisions that must be made for a Business Process to complete can be identified and so that the specific decision-making tasks that perform or execute a Decision can be specified. No formal mapping of **DMN** `ItemDefinition` or **DMN** `InputData` to **BPMN** `DataObject` is proposed but an implementation could include such a check in a situation where such a mapping could be determined.

Together, **BPMN** and **DMN** therefore allow specification of the requirements for automated decision-making and its interaction with human decision making within business processes. These requirements may be specified at any level of detail, or at all levels. The three-tier mapping between business process models, DRDs and decision logic will allow the definition of these requirements to be supported by model-based computer-aided design tools.

5.2.3 Implementing automated decision-making

If all decisions and business knowledge models are fully specified using decision logic, it becomes possible to execute decision models.

One possible scenario is the use of “decision services” deployed from a Business Rules Management System (BRMS) and called by a Business Process Management System (BPMS). A decision service encapsulates the decision logic supporting a DRD, providing interfaces that correspond to subsets of input data and decisions within the DRD. When called with a set of input data, the decision service will evaluate the specified decisions and return their results. The constraint in **DMN** that all decision logic is free of side-effects means that decision services will comply with SOA principles, simplifying system design. Note that decision services may also be invoked internal to the decision model, a trait that they share with business knowledge models.

The structure of a decision model, as visualized in the DRD, may be used as a basis for planning an implementation project. Specific project tasks may be included to cover the definition of decision logic (e.g., rule discovery using human experts, or creation of analytic models), and the implementation of components of the decision model.

Some decision logic representing the business knowledge encapsulated in decision services needs to be maintained over time by personnel responsible for the decisions, using special “knowledge maintenance interfaces”. **DMN** supports the effective design and implementation of knowledge maintenance interfaces: any business knowledge requiring maintenance should be modeled as business knowledge models in the DRD, and the responsible personnel as knowledge sources. DRDs then provide a specification of the required knowledge maintenance interfaces and their users, and the decision logic specifies the initial configuration of the business knowledge to be maintained.

Other decision logic needs to be refreshed by regular analytic modeling. The representation of business knowledge models as functions in **DMN** makes the use of analytic models in decision services very simple: any analytic model capable of representation as a function may be directly called by or imported into a decision service.

5.2.4 Combining applications of modelling

The three contexts described above are not mutually exclusive alternatives; a large process automation project might use **DMN** in all three ways.

First, the decision-making within the existing process might be modeled, to identify the full extent of current decision making and the areas of business knowledge involved. This “as-is” analysis provides the baseline for process improvement.

Next, the process might be redesigned to make the most effective use of both automated and human decision-making, often using collaboration between the two (e.g., using automated referrals to human decision-makers, or decision support systems which advise or constrain the user). Such a redesign involves modeling the requirements for the decision making to occur in each process task and the roles and responsibilities of individuals or groups in the organization. This model provides a “to-be” specification of the required process and the decision-making it coordinates.

Comparison of the “as-is” and “to-be” models will indicate requirements not just for automation technology, but for change management: changes in the roles and responsibilities of personnel, and training to support new or modified business knowledge.

Finally, the “to-be” model will be implemented as executable system software. Provided the decision logic is fully specified in FEEL and/or other external logic (e.g., externally defined Java methods or PMML models), components of the decision model may be implemented directly as software components.

DMN does not prescribe any particular methodology for carrying out the above activities; it only supports the models used for them.

5.3 Basic concepts

5.3.1 Decision requirements level

The word “decision” has two definitions in common use: it may denote the act of choosing among multiple possible options; or it may denote the option that is chosen. In this specification, we adopt the former usage: a **decision** is the act of determining an **output** value (the chosen option), from a number of **input** values, using logic defining how the output is determined from the inputs. This **decision logic** may include one or more **business knowledge models** which encapsulate business know-how in the form of business rules, analytic models, or other formalisms. This basic structure, from which all decision models are built, is shown in Figure 5-3.

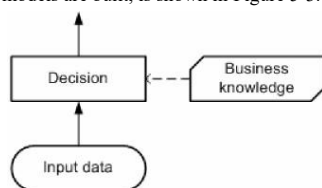


Figure 5-3: Basic elements of a decision model

For simplicity and generality, many of the figures in this specification show each decision as having a single associated business knowledge model, but it should be noted that **DMN** does not require this to be the case. The use of business knowledge models to encapsulate decision logic is a matter of style and methodology, and decisions may be modeled with no associated business knowledge models, or with several. Similar to business knowledge models, decision services may also be used to encapsulate decision logic for reuse inside the decision model, but for simplicity such examples will be presented starting in the section describing decision services.

Authorities may be defined for decisions or business knowledge models, which might be (for example) domain experts responsible for defining or maintaining them, or source documents from which business knowledge models are derived or sets of test cases with which the decisions must be consistent. These are called **knowledge sources** (see Figure 5-4).

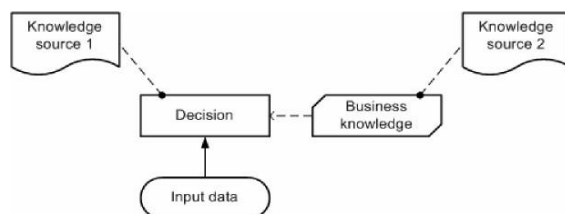


Figure 5-4: Knowledge sources

A decision is said to “require” its inputs in order to determine its output. The inputs may be **input data**, or the outputs of other decisions. (In either case they may be data structures, rather than just simple data items.) If the inputs of a decision Decision1 include the output of another decision Decision2, Decision1 “requires” Decision2. Decisions may therefore be connected in a network called a **Decision Requirements Graph (DRG)**, which may be drawn as a **Decision Requirements Diagram (DRD)**. A DRD shows how a set of decisions depend on each other, on input data, and on business knowledge models. A simple example of a DRD with only two decisions is shown in Figure 5-5.

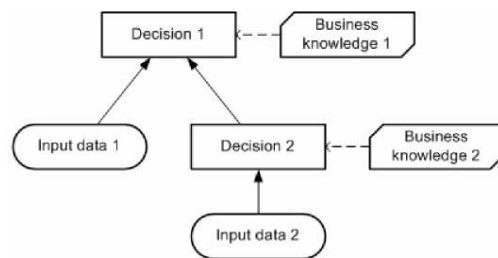


Figure 5-5: A simple Decision Requirements Diagram (DRD)

A decision may require multiple business knowledge models, and a business knowledge model may require multiple other business knowledge models, as shown in Figure 5-6.

This will allow (for example) the modeling of complex decision logic by combining diverse areas of business knowledge, and the provision of alternative versions of decision logic for use in different situations.

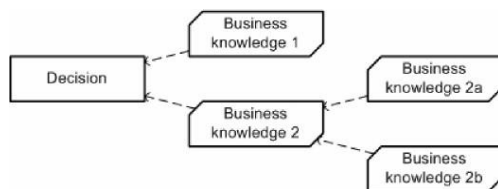


Figure 5-6: Combining business knowledge models

DRGs and their notation as DRDs are specified in detail in clause 6.

5.3.2 Decision logic level

The components of the decision requirements level of a decision model may be described, as they are above, using only business concepts. This level of description is often sufficient for business analysis of a domain of decision-making, to identify the business decisions involved, their interrelationships, the areas of business knowledge and data required by them, and the sources of the business knowledge. Using decision logic, the same components may be specified in greater detail, to capture a complete set of business rules and calculations, and (if desired) to allow the decision making to be fully automated.

Decision logic may also provide additional information about how to display elements in the decision model. For example, the decision logic element for a decision table may specify whether to show the rules as rows or as columns. The decision logic element for a calculation may specify whether to line up terms vertically or horizontally.

The correspondence between concepts at the decision requirements level and the decision logic level is described below. Please note that in the figures below, as in Figure 5-1 and Figure 5-2, the grey ellipses and dotted lines are drawn only to indicate correspondences between concepts in different levels for the purposes of this introduction. They do *not* form part of the notation of **DMN**, which is formally defined in clauses 6.2, 8.2, and 10.2. It is envisaged that implementations will provide facilities for moving between levels of modeling, such as “opening”, “drilling down” or “zooming in”, but **DMN** does not specify how this should be done.

At the decision logic level, every decision in a DRG is defined using a **value expression** which specifies how the decision’s output is determined from its inputs. At that level, the decision is considered to *be* the evaluation of the expression. The value expression may be notated using a **boxed expression**, as shown in Figure 5-7.

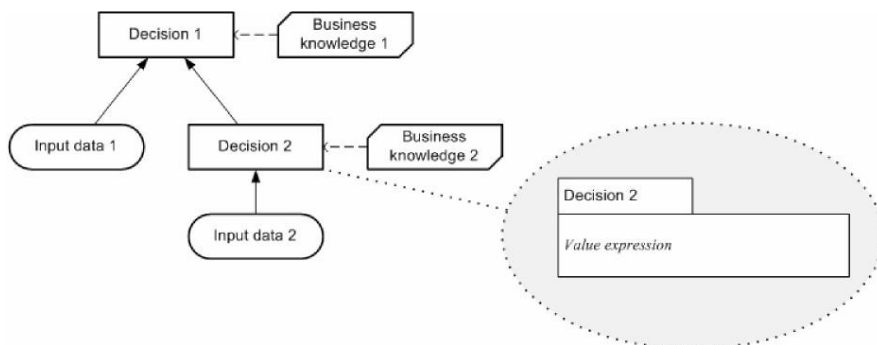


Figure 5-7: Decision and corresponding value expression

In the same way, at the decision logic level, a business knowledge model is defined using a value expression that specifies how an output is determined from a set of inputs. In a business knowledge model, the value expression is encapsulated as a function definition, which may be invoked from a decision's value expression.

The interpretation of business knowledge models as functions in **DMN** means that the combination of business knowledge models as in Figure 5-6 has the clear semantics of functional composition. The value expression of a business knowledge model may be notated using a **boxed function** definition, as shown in Figure 5-8. Similar to a business knowledge model, the decision service element can also be invoked from a decision's value expression (see clause 5.3.3).

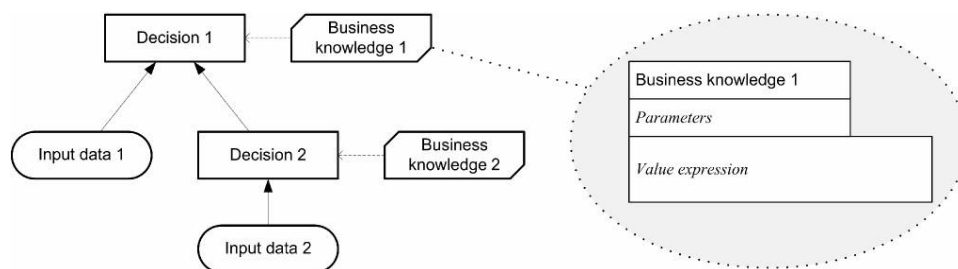


Figure 5-8: Business knowledge model and corresponding value expression

A business knowledge model may contain any decision logic which is capable of being represented as a function. This will allow the import of many existing decision logic modeling standards (e.g., for business rules and analytic models) into **DMN**. An important format of business knowledge, specifically supported in **DMN**, is the Decision Table. Such a business knowledge model may be notated using a **Decision Table**, as shown in Figure 5-9.

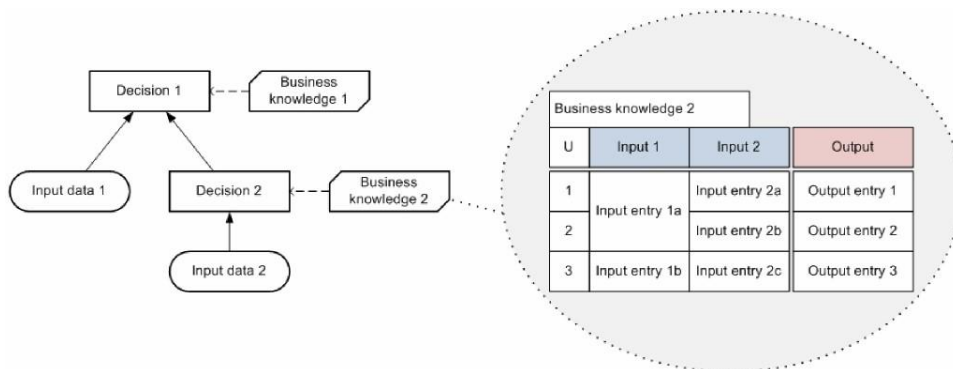


Figure 5-9: Business knowledge model and corresponding decision table

In most cases, the logic of a decision is encapsulated into business knowledge models, and the value expression associated with the decision specifies how the business knowledge models are invoked, and how the results of their invocations are combined to compute the output of the decision. The decision's value expression may also specify how the output is determined from its input entirely within itself, without invoking a business knowledge model: in that case, no business knowledge model is associated with the decision (neither at the decision requirements level nor at the decision logic level).

An expression language for defining decision logic in **DMN**, covering all the above concepts, is specified fully in clause 10. This is **FEEL**: The Friendly Enough Expression Language. The notation for Decision Tables is specified in detail in clause 8.

5.3.3 Decision services

A decision service defines reusable logic within the decision model. A decision service exposes one or more decisions from a decision model as a reusable element, a service, which might be consumed (for example) internally by another decision in the decision model, or externally by a task in a **BPMN** process model. When the service is called with the necessary input data and decision results, it returns the outputs of the exposed decisions. Any decision service encapsulating a **DMN** decision model will be stateless and have no side effects.

One important use of **DMN** will be to define decision-making logic to be automated using decision services. When the decision service is invoked externally, it might be implemented, for example, as a web service. **DMN** does not specify how such services should be implemented, but it allows the functionality of a service to be defined against a decision model. The decision service therefore must be defined in a DRD. When invoked internally from a decision the decision service is invoked, similar to a BKM, by binding expressions in the logic of the calling decision to parameters in the invoked decision service.

It is assumed that the client requires a certain set of decisions to be made, and that the service is created to meet that requirement. The sole function of the decision service is to return the results of evaluating that set of decisions (the "output decisions"). The service may be provided with the results of decisions evaluated externally to the service (the "input decisions"). The service must encapsulate not just the output decisions but also any decisions in the DRG directly or indirectly required by the output decisions which are not provided in the input decisions (the "encapsulated decisions").

The interface to the decision service will consist of:

- Input data: instances of all the input data required by the encapsulated decisions.
- Input decisions: instances of the results of all the input decisions.
- Output decisions: the results of evaluating (at least) all the output decisions, using the provided input decisions and input data.

When the service is called, providing the input data and input decisions, it returns the output decisions.

Note that to define a decision service it is only necessary to specify the output decisions and either the input decisions or the encapsulated decisions. The remaining attributes (the required input data, and whichever of the

encapsulated or input decisions was not specified) may then be inferred from the decision model against which the service is defined. Alternatively, if more attributes are defined than are strictly necessary, they may be validated against the decision model.

Figure 5-10 shows a decision service defined against a decision model that includes three decisions. The output decisions for this service are {Decision 1}, and the input decisions are {}, that is, the service returns the result of Decision 1 and is not provided with the results of any external decisions. Since Decision 1 requires Decision 2, which is not provided to the service as input, the service must also encapsulate Decision 2. Decision 3 is not required to be encapsulated. The encapsulated decisions are therefore {Decision 1, Decision 2}. The service requires Input data 1 and Input data 2 but not Input data 3.

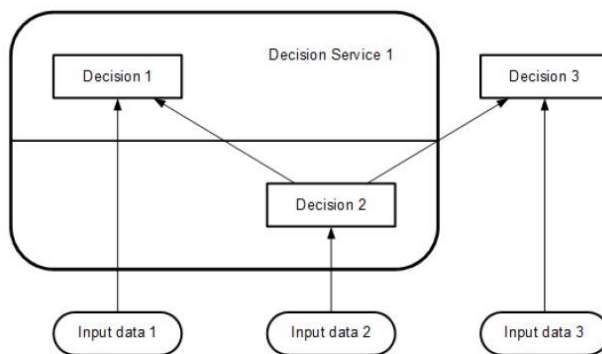


Figure 5-10: A decision service

Multiple decision services may be defined against the same decision model. Figure 5-11 shows a decision service defined against the same decision model, whose output decisions are {Decision 1} and whose input decisions are {Decision 2}. The encapsulated decisions for this service are {Decision 1}. The service requires Input data 1 but not Input data 2 or Input data 3.

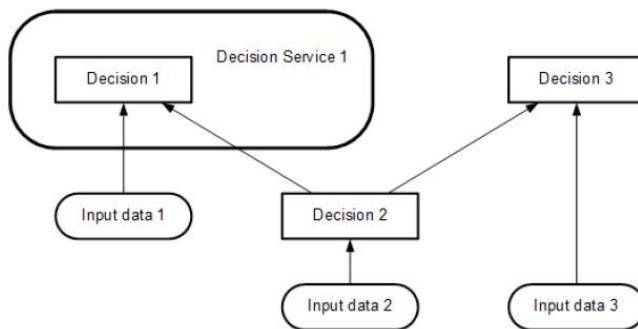


Figure 5-11: A decision service taking a decision as input

In its simplest form a decision service would always evaluate all the decisions in the output set, set and return all their results.

For computational efficiency various improvements to this basic interpretation can be imagined, for example:

- An optional input parameter specifying a list of “requested decisions” (a subset of the minimal output set). Only the results of the requested decisions would be returned in the output context.

- An optional input parameter specifying a list of “known decisions” (a subset of the encapsulation set), with their results. The decision service would not evaluate these decisions but would use the provided input values directly.

All such implementation details are left to the software provider.

A decision service is “complete” if it contains decision logic for evaluating all the encapsulated decisions on all possible input data values. A request to the service is “valid” if instances are provided for all the input decisions and input data required by those decisions which need to be evaluated, i.e., (in the simple case) all the encapsulated decisions, or (assuming the optional parameters above) any requested decisions and any encapsulated decisions required by them which are not already known.

This page intentionally left blank.

6 Requirements (DRG and DRD)

6.1 Introduction

The decision requirements level of a decision model in **DMN** consists of a Decision Requirements Graph (DRG) depicted in one or more Decision Requirements Diagrams (DRDs).

A DRG models a domain of decision-making, showing the most important elements involved in it and the dependencies between them. The elements modeled are decisions, areas of business knowledge, sources of business knowledge, input data and decision services:

- A **Decision** element denotes the act of determining an output from a number of inputs, using decision logic which may reference one or more Business Knowledge Models.
- A **Business Knowledge Model** element denotes a function encapsulating business knowledge, e.g., as business rules, a decision table, or an analytic model.
- An **Input Data** element denotes information used as an input by one or more Decisions.
- A **Knowledge Source** element denotes an authority for a Business Knowledge Model or Decision.
- A **Decision Service** element denotes a set of reusable decisions that can be invoked internally or externally.

The dependencies between these elements express three kinds of requirements: information, knowledge, and authority:

- An **Information Requirement** denotes Input Data or Decision output being used as input to a Decision.
- A **Knowledge Requirement** denotes the invocation of a Business Knowledge Model or Decision Service by the decision logic of a Decision.
- An **Authority Requirement** denotes the dependence of a DRG element on another DRG element that acts as a source of guidance or knowledge.

DRDs may also contain any number of artifacts representing annotations of the diagram:

- A Text Annotation is modeler-entered text used for comment or explanation.
- An Association is a dotted connector used to link a Text Annotation to a DRG Element
- A Group is a visual mechanism to group elements of a diagram informally.

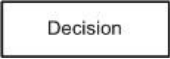
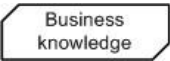
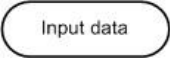
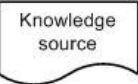
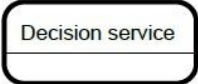
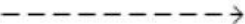
These components are summarized in Table 1 and described in more detail in clause 6.2.

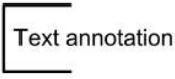
A DRG is a graph composed of elements connected by requirements and is self-contained in the sense that all the modeled requirements for any Decision in the DRG (its immediate sources of information, knowledge, and authority) are present in the same DRG. It is important to distinguish this complete definition of the DRG from a DRD presenting any particular view of it, which may be a partial or filtered display: see clause 6.2.4.

6.2 Notation

The notation for all components of a DRD is summarized in Table 1 and described in more detail below.

Table 1: DRD components

Component		Description	Notation
Elements	Decision	A decision denotes the act of determining an output from a number of inputs, using decision logic which may reference one or more business knowledge models.	
	Business Knowledge Model	A business knowledge model denotes a function encapsulating business knowledge, e.g., as business rules, a decision table, or an analytic model.	
	Input Data	An input data element denotes information used as an input by one or more decisions. When enclosed within a knowledge model, it denotes the parameters to the knowledge model. The default representation of the Input Data is an oval symbol. For visual coherence with BPMN and CMMN, the representation as a paper symbol with folded corner is possible. This specification uses the default representation in all examples.	 or alternatively  Input Data
	Knowledge Source	A knowledge source denotes an authority for a business knowledge model or decision.	
	Decision Service (expanded)	A decision service may enclose a set of reusable decisions (not shown in the element to the right) that can be invoked internally by another decision or externally, e.g., by a BPMN process.	
	Decision Service (collapsed)	A decision service denotes a set of reusable decisions (that may be hidden using the element to the right).	
Requirements	Information Requirement	An information requirement denotes input data, or a decision output being used as one of the inputs of a decision.	
	Knowledge Requirement	A knowledge requirement denotes the invocation of a business knowledge model.	
	Authority Requirement	An authority requirement denotes the dependence of a DRD element on another DRD element that acts as a source of guidance or knowledge.	

Artifacts	Text Annotation	A Text Annotation consists of a square bracket followed by modeler-entered explanatory text or comment.	
	Association	An Association connector links a Text Annotation to the DRG Element it explains or comments on.	
	Group	A Group consists of a rounded corner rectangle drawn with a solid dashed line that groups element together informally.	

6.2.1 DRD Elements

6.2.1.1 Decision notation

A Decision is represented in a DRD as a rectangle, normally drawn with solid lines, as shown in Table 1. The Name of the Decision MUST be displayed inside the shape unless it is overridden by the text attribute of the associated DMNDI:DMNLabel element, which MUST be displayed instead.

If the Listed Input Data option is exercised (see 6.2.1.3), all the Decision's requirements for Input Data SHALL be listed beneath the Decision's Name and separated from it by a horizontal line, as shown in Figure 6-1. The listed Input Data names SHALL be clearly inside the shape of the DRD element.

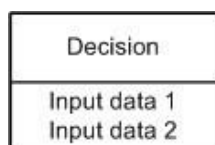


Figure 6-1: Decision with Listed Input Data option

The properties of a Decision are listed and described in 6.3.6.

6.2.1.2 Business Knowledge Model notation

A Business Knowledge Model is represented in a DRD as a rectangle with two clipped corners, normally drawn with solid lines, as shown in Table 1. The Name of the Business Knowledge Model MUST be displayed inside the shape unless it is overridden by the text attribute of the associated DMNDI:DMNLabel element, which MUST be displayed instead.

The properties of a Business Knowledge Model are listed and described in 6.3.8.

6.2.1.3 Input Data notation

An Input Data element is represented in a DRD as a shape with two parallel straight sides and two semi-circular ends, normally drawn with solid lines, as shown in Table 1. The Name of the Input Data element MUST be displayed inside the shape unless it is overridden by the text attribute of the associated DMNDI:DMNLabel element, which MUST be displayed instead.

An alternative compliant way to display requirements for Input Data, especially useful when DRDs are large or complex, is that Input Data are not drawn as separate notational elements in the DRD but are instead listed on those Decision elements which require them. For convenience in this specification this is called the "Listed Input Data" option. Implementations MAY offer this option. Figure 6-2 shows two equivalent DRDs, one drawing Input Data

elements, the other exercising the Listed Input Data option. Note that if an Input Data element is not displayed it SHALL be listed on all Decisions which require it (unless it is deliberately hidden as discussed in 6.2.4).

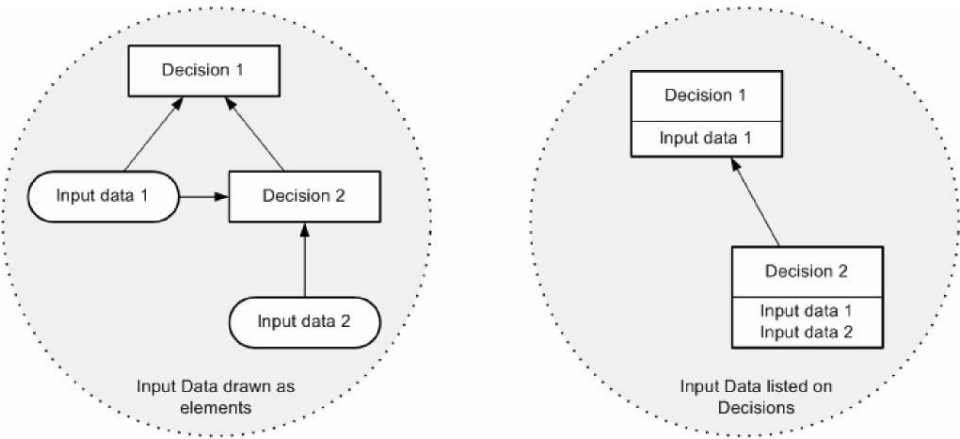


Figure 6-2: The Listed Input Data option

The properties of an Input Data element are listed and described in 6.3.11.

6.2.1.4 Knowledge Source notation

A Knowledge Source is represented in a DRD as a shape with three straight sides and one wavy one, normally drawn with solid lines, as shown in Table 1. The Name of the Knowledge Source MUST be displayed inside the shape unless it is overridden by the text attribute of the associated DMNDI:DMNLabel element, which MUST be displayed instead.

The properties of a Knowledge Source element are listed and described in 6.3.12.

6.2.2 DRD Requirements

6.2.2.1 Information Requirement notation

Information Requirements may be drawn from Input Data elements to Decisions, and from Decisions to other Decisions. They represent the dependency of a Decision on information from input data or the results of other Decisions. They may also be interpreted as data flow: a DRD displaying only Decisions, Input Data and Information Requirements is equivalent to a dataflow diagram showing the communication of information between those elements at evaluation time. The Information Requirements of a valid DRG form a directed acyclic graph.

An Information Requirement is represented in a DRD as an arrow drawn with a solid line and a solid arrowhead, as shown in Table 1. The arrow is drawn in the direction of information flow, i.e., towards the Decision that requires the information.

6.2.2.2 Knowledge Requirement notation

Knowledge Requirements may be drawn from invocable elements (Business Knowledge Models or Decision Services) to Decisions and from invocable elements to Business Knowledge Models. They represent the invocation of an invocable element when making a decision. If *e* is a decision or a BKM in some DRD, and *e* contains a knowledge requirement on some invocable element *b*, then the logic of *e* must contain an invocation expression of *b*, including expressions for each of *b*'s parameters.

A Knowledge Requirement is represented in a DRD as an arrow drawn with a dashed line and an open arrowhead, as shown in Table 1. The arrows are drawn in the direction of the information flow of the result of evaluating the function, i.e., toward the element that requires the business knowledge.

6.2.2.3 Authority Requirement notation

Authority Requirements may be used in two ways:

- They may be drawn from Knowledge Sources to Decisions, Business Knowledge Models, and other Knowledge Sources, where they represent the dependence of the DRD element on the knowledge source. This might be used to record the fact that a set of business rules must be consistent with a published document (e.g., a piece of legislation or a statement of business policy), or that a specific person or organizational group is responsible for defining some decision logic, or that a decision is managed by a person or group. An example of this use of Knowledge Sources is shown in Figure 6-3: in this case the Business Knowledge Model requires two sources of authority – a policy document and legislation – and the policy document requires the authority of a policy group.

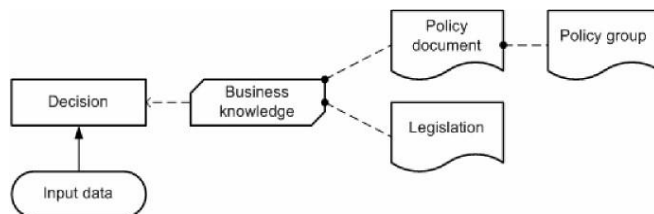


Figure 6-3: Knowledge Sources representing authorities

- They may be drawn from Input Data and Decisions to Knowledge Sources, where, in conjunction with use (a), they represent the derivation of Business Knowledge Models from instances of Input Data and Decision results, using analytics. The Knowledge Source typically represents the analytic model (or modeling process); the Business Knowledge Model represents the executable logic generated from or dependent on the model. An example of this use of a Knowledge Source is shown in Figure 6-4: in this case a business knowledge model is based on an analytic model which is derived from input data and the results of a dependent decision.

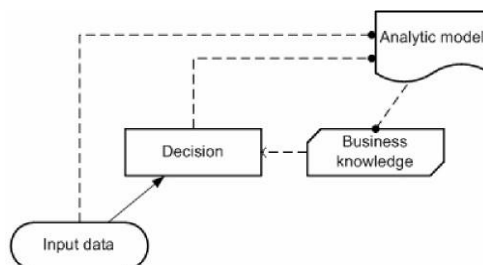


Figure 6-4: Knowledge source representing predictive analytics

However, the figures above are only examples. There are many other possible use cases for Authority Requirements (and since Knowledge Sources and Authority Requirements have no execution semantics their interpretation is necessarily vague), so this specification leaves the details of their application to the implementer.

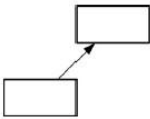
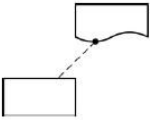
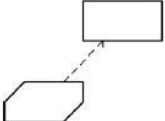
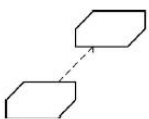
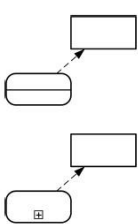
An Authority Requirement is represented in a DRD as an arrow drawn with a dashed line and a filled circular head, as shown in Table 1. The arrows are drawn from the source of authority to the element governed by it.

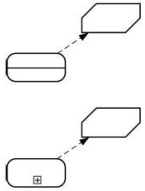
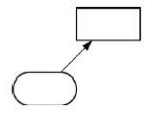
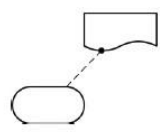
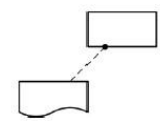
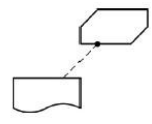
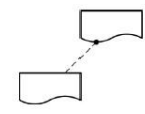
6.2.3 Connection rules

The rules governing the permissible ways of connecting elements with requirements in a DRD are described in Clause 6.2.2 above and summarized in Table 2. For clarity, a simple DRD is shown for each permissible connection. In each of these diagrams, the upper (“to”) element requires the lower (“from”) element.

Note that no requirements may be drawn terminating in Input Data, that is, input data may have no requirements. Note also that the type of the requirement is uniquely determined by the types of the two elements connected.

Table 2: Requirements connection rules

From	To (Required by)	Requirement	Diagram
Decision	Decision	Information	
Decision	Knowledge Source	Authority	
Business Knowledge Model	Decision	Knowledge	
Business Knowledge Model	Business Knowledge Model	Knowledge	
Decision Service	Decision	Knowledge	

Decision Service	Business Knowledge Model	Knowledge	
Input data	Decision	Information	
Input data	Knowledge Source	Authority	
Knowledge Source	Decision	Authority	
Knowledge Source	Business Knowledge Model	Authority	
Knowledge Source	Knowledge Source	Authority	

6.2.4 Partial views and hidden information

The metamodel (see 6.3) provides properties for each of the DRG elements which would not normally be displayed on the DRD but provide additional information about their nature or function.

For example, for a Decision these include properties specifying which **BPMN** processes and tasks make use of the Decision. Implementations SHALL provide facilities for specifying and displaying such properties.

For any significant domain of decision-making a DRD representing the complete DRG may be a large and complex diagram. Implementations MAY provide facilities for displaying DRDs which are partial or filtered views of the DRG, e.g., by hiding categories of elements, or hiding or collapsing areas of the network.

DRG Elements with requirements not displayed on the current DRD SHOULD be notated with an ellipsis (...) to show that this is the case. For example, see Figure 12-5.

Two examples of DRDs providing partial views of a DRG are shown in Figure 6-5: DRD 1 shows only the immediate requirements of a single decision; DRD 2 shows only Information Requirements and the elements they connect.

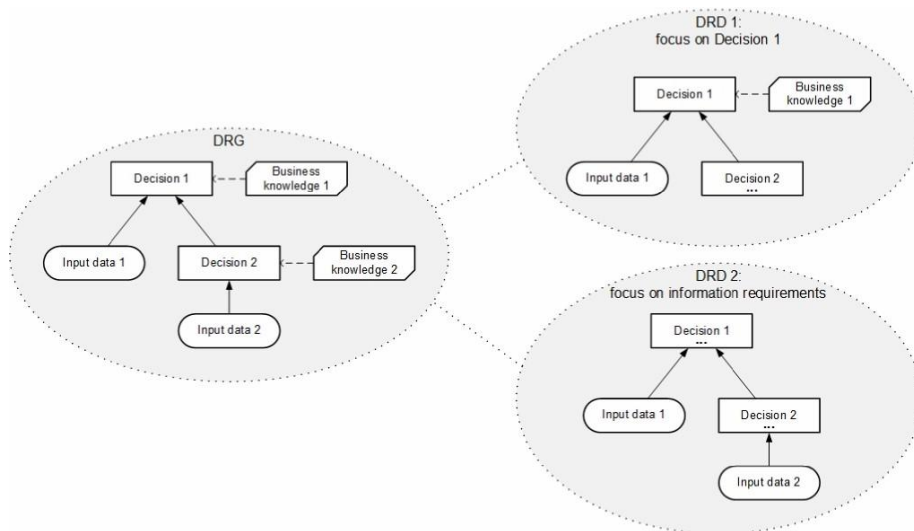


Figure 6-5: DRDs as partial views of a DRG

DRDs can be interchanged using the Diagram Interchange mechanism defined in section 14.

6.2.5 Decision service

A Decision Service is represented in a DRD as rectangle with rounded corners, drawn with a heavy solid border. The Name of the Decision Service MUST be displayed inside the shape unless it is overridden by the text attribute of the associated DMNDI:DMNLabel element, which MUST be displayed instead. The border SHALL enclose all the encapsulated decisions, and no other decisions or input data. The border MAY enclose other DRG elements, but these will not form part of the definition of the Decision Service.

If the set of output decisions is smaller than the set of encapsulated decisions, the Decision Service SHALL be divided into two parts with a straight solid line. One part SHALL enclose only the output decisions and the Decision Service's Name; the other part SHALL enclose all the encapsulated decisions which are not in the set of output decisions. Either part MAY enclose other DRG elements, but these will not form part of the definition of the Decision Service.

Figure 6-6 shows a Decision Service with two output decisions; other examples (with a single output decision) are shown in Figure 5-10 and Figure 5-11.

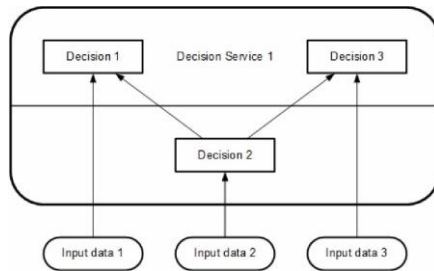
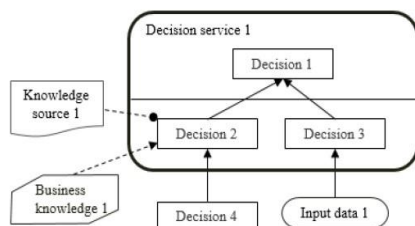


Figure 6-6: Decision Service notation

A decision service may be defined in one DRD and then shown in a different DRD when invoked internally within the decision model by another decision. In the case of a decision service invocation internal to the decision model, a decision service may also be shown without the details of its definition, as in a “collapsed state”. Figure 6-7 consists of two separate diagrams: DRD 1 shows the definition of Decision service 1. In DRD 2, the same Decision service 1 is shown as invoked by Decision 5. In DRD 2, Decision service 1 is shown in a collapsed form.

DRD 1



DRD 2

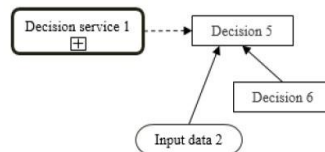


Figure 6-7: A decision service in expanded and collapsed form

DRD 1 in Figure 6-7 shows that Decision service 1 has 2 inputs: Decision 4 and Input data 1. It is therefore inferred that Decision Service 1 has 2 input parameters with matching characteristics to Decision 4 and Input data 1. DRD 2 in Figure 6-7 shows that Decision 5 has 2 dependencies but whether these are mapped as parameters for the invocation of Decision Service 1 cannot be determined from the diagram.

The information and authority requirements defined on Decision 2 in DRD 1 are not depicted in the collapsed form of Decision Service 1 shown in DRD 2.

DRD 3

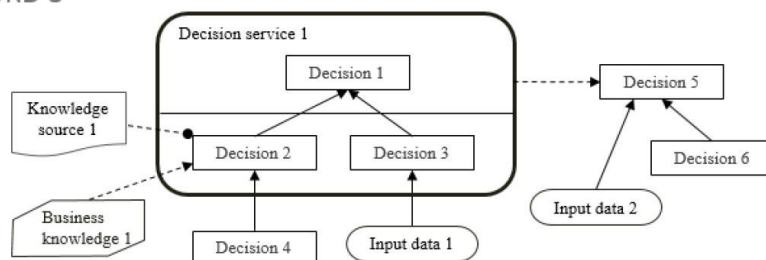


Figure 6-8: A decision service invoked in an expanded form

DRDs 1 and 2 in Figure 6-7 and DRD 3 in Figure 6-8 are all congruent within the same DRG. They all show different aspects of Decision Service 1. DRD 3 shows an expanded form Decision service 1 being invoked by Decision 5.

The constraint imposed on the rendering of decision services within a DRD is that the same decision service **MUST NOT** be rendered both expanded and collapsed within the same DRD. This stems from the general restriction disallowing the same DMN Element to be present twice in the same diagram.

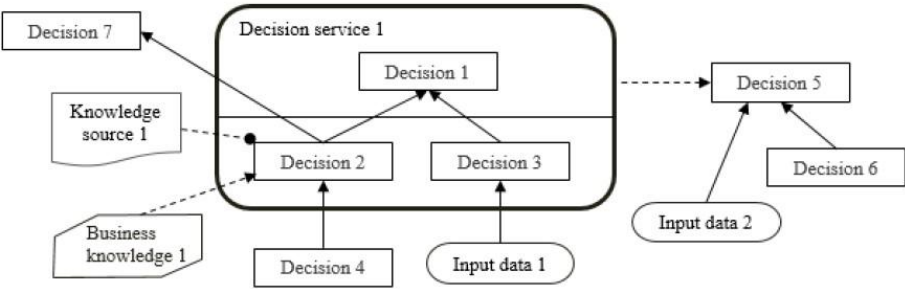


Figure 6-9: A decision service defined as an overlay

Decision services are defined as overlays and therefore do not encapsulate the decisions within them. Therefore, the richness of connections depicted in Figure 6-9 is allowed. In this DRD, Decision 7 is dependent on Decision 2.

6.2.6 Identifying Collections

Decisions and Input Data elements on a DRD can represent collections of elements. Implementations **MAY** show this with the addition of ||| in the shape. Implementations **SHALL** show this on all such DRD elements within a DRG **OR** on no DRD elements.

A Decision is considered to represent a collection if the Decision's decisionOutput InformationItem references an ItemDefinition with isCollection = TRUE.

An InputData is considered to represent a collection if the InputData's variable InformationItem references an ItemDefinition with isCollection = TRUE.

Two examples, a Decision and an Input Data, are shown in Figure 6-10.



Figure 6-10: Decision and Input Data showing collection marker

6.3 Metamodel

6.3.1 DMN Element metamodel

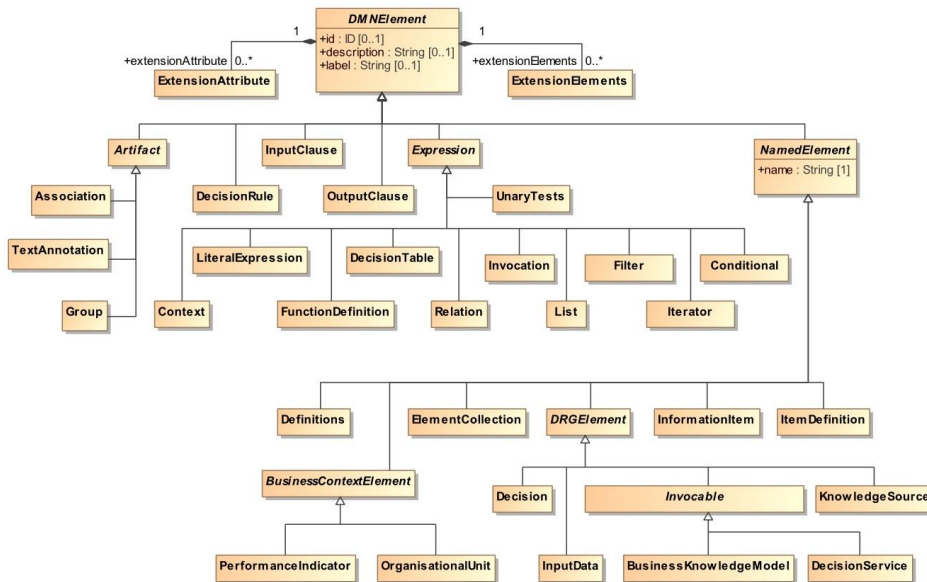


Figure 6-11: DMNElement Class Diagram

DMNElement is the abstract superclass for the decision model elements. It provides the optional attributes `id`, `description` and `label`, which are Strings which other elements will inherit. The `id` of a DMNElement is further restricted to the syntax of an XML ID (<https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#ID>), and SHALL be unique within the decision model.

DMNElement has abstract specializations `NamedElement` and `Expression`. `NamedElement` adds the required attribute `name`, and includes the abstract specializations `BusinessContextElement` and `DRGElement`, as well as concrete specializations `Definitions`, `ItemDefinition`, `InformationItem`, `ElementCollection` and `DecisionService`.

Table 3 presents the attributes and model associations of the DMNElement element.

Table 3: DMNElement attributes and model associations

Attribute	Description
id: ID [0..1]	Optional identifier for this element. SHALL be unique within its containing <code>Definitions</code> element.
description: String [0.. 1]	A description of this element.

label: String [0.. 1]	An alternative short description of this element. It should primarily be used on elements that do not have a name attribute, e.g., an Input Expression. Similar to the description attribute, it has no notation defined and is neither related to the DMNLabel element that is used in Diagram Interchange nor to the outputLabel attribute of a Decision Table.
extensionElements: ExtensionElement [0..1]	This attribute is used as a container to attach additional elements to any DMN Element. See 6.3.16 for additional information on extensibility.
extensionAttributes: ExtensionAttribute [0..*]	This attribute is used to attach named extended attributes and model associations. This association is not applicable when the XML schema interchange is used, since the XSD mechanism for supporting "anyAttribute" from other namespaces already satisfies this requirement. See 6.3.16 for additional information on extensibility.

Table 4: NamedElement attributes and model associations

Attribute	Description
Name: string	The name of this element

6.3.2 Definitions metamodel

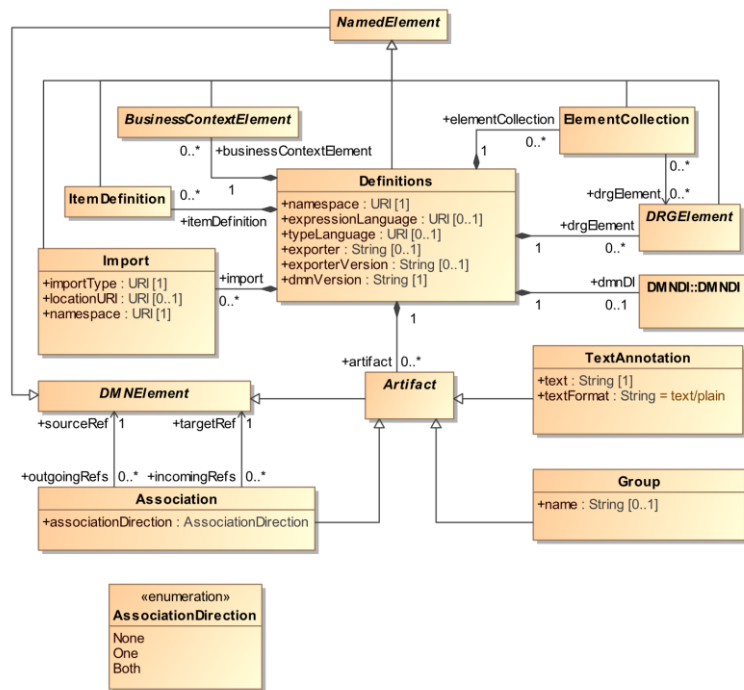


Figure 6-12: Definitions Class Diagram

The Definitions class is the outermost containing object for all elements of a **DMN** decision model. It defines the scope of visibility and the namespace for all contained elements. Elements that are contained in an instance of Definitions have their own defined life-cycle and are not deleted with the deletion of other elements. The interchange of **DMN** files will always be through one or more Definitions.

Definitions is a kind of NamedElement, from which an instance of Definitions inherits the name and optional id, description, and label attributes, which are Strings.

An instance of Definitions has a namespace, which is a String. The namespace identifies the default target namespace for the elements in the Definitions and follows the convention established by XML Schema.

An instance of Definitions may specify an expressionLanguage, which is a URI that identifies the default expression language used in elements within the scope of this Definitions. This value may be overridden on each individual LiteralExpression. The language SHALL be specified in a URI format. The default expression language is FEEL (clause 10), indicated by the URI: ["https://www.omg.org/spec/DMN/FEEL/"](https://www.omg.org/spec/DMN/FEEL/). If FEEL is used in other languages, e.g. as an expression language in a BPMN process model, a version number MUST be added to the URL, i.e. ["https://www.omg.org/spec/DMN/FEEL/1.7/"](https://www.omg.org/spec/DMN/FEEL/1.7/). ["https://www.omg.org/spec/DMN/20240513/FEEL/"](https://www.omg.org/spec/DMN/20240513/FEEL/). The simple expression language S-FEEL (clause 9), being a subset of FEEL, is indicated by the same URI. **DMN** provides a URI for expression languages that are not meant to be interpreted automatically (e.g., pseudo-code that may resemble FEEL but is not): ["http://www.omg.org/spec/DMN/uninterpreted/20140801/"](http://www.omg.org/spec/DMN/uninterpreted/20140801/).

An instance of Definitions may specify a typeLanguage, which is a URI that identifies the default type language used in elements within the scope of this Definitions. For example, a typeLanguage value of

Commented [AF6]: DMN17-65

Commented [AF7]: DMN17-65

"http://www.w3.org/2001/XMLSchema" indicates that the data structures defined within that Definitions are, by default, in the form of XML Schema types. If unspecified, the default typeLanguage is FEEL. This value may be overridden on each individual ItemDefinition.

The typeLanguage SHALL be specified in a URI format (the URI for FEEL is "https://www.omg.org/spec/DMN/20230324/FEEL/"; the URI "http://www.omg.org/spec/DMN/uninterpreted/20140801" can be used to indicate that a type definition is not meant to be interpreted)).

An instance of Definitions may specify an exporter and exporterVersion, which are Strings naming the tool and version used to create the XML serialization. In standards such as **BPMN**, this has been found to aid in model interchange between tools.

An instance of Definitions is composed of zero or more drgElements, which are instances of DRGElement, zero or more elementCollections, which are instances of ElementCollection, zero or more itemDefinitions, which are instances of ItemDefinition and of zero or more businessContextElements, which are instances of BusinessContextElement.

It may contain any number of associated import, which are instances of Import. Imports are used to import elements defined outside of this Definitions, e.g., in other Definitions elements, and to make them available for use by elements in this Definitions.

Definitions inherits all the attributes and model associations from NamedElement. Table 5 presents the additional attributes and model associations of the Definitions element.

Table 5: Definitions attributes and model associations

Attribute	Description
namespace: anyURI [1]	This attribute identifies the namespace associated with this Definitions and follows the convention established by XML Schema.
dmnVersion: string [1]	This attribute identifies the concrete DMN version used in this Definitions (see clause 13.3.1 for its use in the XML serialization).
expressionLanguage: anyURI [0.. 1]	This attribute identifies the expression language used in LiteralExpressions within the scope of this Definitions. The Default is FEEL (clause 10). This value MAY be overridden on each individual LiteralExpression. The language SHALL be specified in a URI format.
typeLanguage: anyURI [0.. 1]	This attribute identifies the type language used in LiteralExpressions within the scope of this Definitions. The Default is FEEL (clause 10). This value MAY be overridden on each individual ItemDefinition. The language SHALL be specified in a URI format.
exporter: string [0..1]	This attribute names the tool used to export the XML serialization.
exporterVersion: string [0.. 1]	This attribute names the version of the tool used to export the XML serialization.
itemDefinition: ItemDefinition [*]	This attribute lists the instances of ItemDefinition that are contained in this Definitions.

Commented [AF8]: DMN17-65

drgElement: DRGElement [*]	This attribute lists the instances of DRGElement that are contained in this Definitions.
businessContextElement: BusinessContextElement [*]	This attribute lists the instances of BusinessContextElement that are contained in this Definitions.
elementCollection: ElementCollection [*]	This attribute lists the instances of ElementCollection that are contained in this Definitions.
import: Import [*]	This attribute is used to import externally defined elements and make them available for use by elements in this Definitions.
artifact: Artifact [0..*]	Artifacts include text annotations, groups, and associations among DMN elements.
dmnDI: DMNDI [0..1]	This attribute contains the Diagram Interchange information contained within this Definitions (see Clause 13 for more information on the DMN Diagram Interchange).

6.3.3 Import metamodel

The `Import` class is used when referencing external elements, either **DMN** `DRGElement` or `ItemDefinition` instances contained in other `Definitions` elements, or non-**DMN** elements, such as an XML Schema or a PMML file. Imports SHALL be explicitly defined.

An instance of `Import` has an `importType`, which is a `String` that specifies the type of import associated with the element. For example, a value of “`http://www.w3.org/2001/XMLSchema`” indicates that the imported element is an XML schema. The **DMN** namespace indicates that the imported element is a **DMN** `Definitions` element.

The location of the imported element may be specified by associating an optional `locationURI` with an instance of `Import`. The `locationURI` is a `URI`.

An instance of `Import` has a `namespace`, which is a `URI` that identifies the namespace of the imported element, and also a `name`, inherited from `NamedElement`, which is a string that serves as a prefix in namespace-qualified names, such as `typeRefs` specifying imported `ItemDefinitions` and expressions referencing imported `InformationItems`. The `namespace` value should be globally unique, but the import name, which is typically a short business-friendly name, must be distinct from the names of other imports, decisions, input data, business knowledge models, decision services, and item definitions within the importing model only. Multiple imports with empty import names are allowed in the default namespace and their precedence is resolved according to their definition order.

When the import name attribute is an empty string, the elements are imported in the default namespace of the model. When a name collision occurs between an element in the default namespace and an imported element, the imported element does not replace the one already in the default namespace while the elements without name collision are imported.

Table 6 presents the attributes and model associations of the `Import` element.

Table 6: Import attributes and model associations

Attribute	Description
importType: anyURI	Specifies the style of import associated with this <code>Import</code> .

locationURI: anyURI [0.. 1]	Identifies the location of the imported element.
namespace: anyURI	Identifies the namespace of the imported element.

6.3.4 Element Collection metamodel

The `ElementCollection` class is used to define named groups of `DRGElement` instances. `ElementCollections` may be used for any purpose relevant to an implementation, for example:

- To identify the requirements subgraph of a set one or more decisions (i.e., all the elements in the closure of the requirements of the set).
- To identify the elements to be depicted on a DRD.

`ElementCollection` is a kind of `NamedElement`, from which an instance of `ElementCollection` inherits the name and optional `id`, `description`, and `label` attributes, which are `Strings`. The `id` of an `ElementCollection` element SHALL be unique within the containing instance of `Definitions`.

An `ElementCollection` element has any number of associated `drgElements`, which are the instances of `DRGElement` that this `ElementCollection` defines together as a group. Notice that an `ElementCollection` element must reference the instances of `DRGElement` that it collects, not contain them: instances of `DRGElement` can only be contained in `Definitions` elements.

`ElementCollection` inherits all the attributes and model associations from `NamedElement`. Table 7 presents the additional attributes and model associations of the `ElementCollection` element.

Table 7: `ElementCollection` attributes and model associations

Attribute	Description
drgElement: <code>DRGElement</code> [*]	This attribute lists the instances of <code>DRGElement</code> that this <code>ElementCollection</code> groups.

6.3.5 DRG Element metamodel

`DRGElement` is the abstract superclass for all **DMN** elements that are contained within `Definitions` and that have a graphical representation in a DRD. All the elements of a **DMN** decision model that are not contained directly in a `Definitions` element (specifically: all three kinds of requirement, bindings, clause and decision rules, import, and objective) SHALL be contained in an instance of `DRGElement`, or in a model element that is contained in an instance of `DRGElement`, recursively.

The specializations of `DRGElement` are `Decision`, `InputData`, `Invocable`, and `KnowledgeSource`. `Invocable` is further specialized into `BusinessKnowledgeModel` and `DecisionService`.

`DRGElement` is a specialization of `NamedElement`, from which it inherits the name and optional `id`, `description`, and `label` attributes. The `id` of a `DRGElement` element SHALL be unique within the containing instance of `Definitions`.

A **Decision Requirements Diagram (DRD)** is the diagrammatic representation of one or more instances of `DRGElement` and their information, knowledge, and authority requirement relations. The instances of `DRGElement` are represented as the vertices in the diagram; the edges represent instances of `InformationRequirement`, `KnowledgeRequirement` or `AuthorityRequirement` (see clauses 6.3.13, 6.3.14, and 6.3.15). The connection rules are specified in 6.2.3).

`DRGElement` inherits all the attributes and model associations of `NamedElement`. It does not define additional attributes and model associations of the `DRGElement` element.

6.3.6 Artifact metamodel

Artifacts are used to provide additional information about a Decision Model. DMN provides two standard Artifacts: Association and Text Annotation. Associations can be used to link Artifacts to any DMNElement.

6.3.6.1 Association

An Association is used to link information and Artifacts with DMN graphical elements. Text Annotations and other Artifacts can be associated with the graphical elements. An arrowhead on the Association indicates a direction of flow (e.g., data), when appropriate.

The Association element inherits the attributes and model associations of DMNElement (see Table 3). Table 8 presents the additional attributes and model associations for an Association.

Table 8: Association attributes and model associations

Attribute	Description
associationDirection: AssociationDirection = None {None One Both}	associationDirection is an attribute that defines whether or not the Association shows any directionality with an arrowhead. The default is None (no arrowhead). A value of One means that the arrowhead SHALL be at the Target Object. A value of Both means that there SHALL be an arrowhead at both ends of the Association line.
sourceRef: DMNElement [1]	The DMNElement that the Association is connecting from.
targetRef: DMNElement [1]	The DMNElement that the Association is connecting to.

6.3.6.2 Group

The Group object is an Artifact that provides a visual mechanism to group elements of a diagram informally. Groups are often used to highlight certain sections of a Diagram without adding additional constraints for performance. The highlighted (grouped) section of the Diagram can be separated for reporting and analysis purposes. Groups do not affect the execution of the Decisions.

As an Artifact, a Group is not a DRGElement, and, therefore, cannot be connected to/from an Information Requirement, Knowledge Requirement, or Authority Requirement. It can only be connected to/from an Association.

The Group element inherits the attributes and model associations of Artifact. Table 9 presents the additional attributes and model associations for a Group.

Table 9: Group model associations

Attribute	Description
Name: String[0.. 1]	The descriptive name of the element.

6.3.6.3 Text Annotation

Text Annotations are a mechanism for a modeler to provide additional text information for the reader of a DMN Diagram.

The `TextAnnotation` element inherits the attributes and model associations of `DMNElement` (see Table 3). Table 10 presents the additional attributes for a `TextAnnotation`.

Table 10: `TextAnnotation` attributes

Attribute	Description
text: string	Text is an attribute that is text that the modeler wishes to communicate to the reader of the Diagram.
textFormat: string = "text/plain"	This attribute identifies the format of the text. It SHALL follow the mime-type format. The default is "text/plain."

6.3.7 Decision metamodel

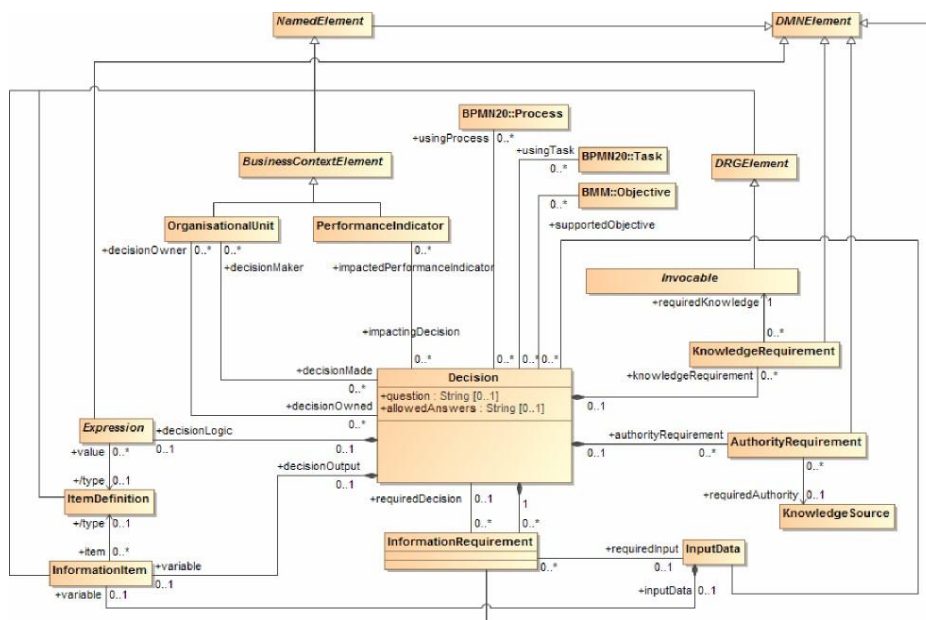


Figure 6-13: Decision Class Diagram

The class `Decision` is used to model a decision.

`Decision` is a concrete specialization of `DRGEElement` and it inherits the name and optional id, description and label attributes from `NamedElement`. The name of a `Decision` must be different from the name of any other invocable, input data, decision, or import in the decision model.

In addition, it may have a `question` and `allowedAnswers`, which are all strings. The optional `description` attribute is meant to contain a brief description of the decision-making embodied in the `Decision`. The optional `question` attribute is meant to contain a natural language question that characterizes the `Decision` such that the output of the `Decision` is an answer to the question. The optional `allowedAnswers` attribute is meant to

contain a natural language description of the answers allowed for the question such as Yes/No, a list of allowed values, a range of numeric values etc.

In a DRD, an instance of *Decision* is represented by a **decision** diagram element.

A *Decision* element is composed of an optional *decisionLogic*, which is an instance of *Expression*, and of zero or more *informationRequirement*, *knowledgeRequirement* and *authorityRequirement* elements, which are instances of *InformationRequirement*, *KnowledgeRequirement* and *AuthorityRequirement*, respectively.

In addition, a *Decision* defines an *InformationItem* representing its output. This *InformationItem* may include an optional *typeRef*, which references an *ItemDefinition* or other type definition specifying the datatype of the possible outcomes of the *Decision*.

The **requirement subgraph** of a *Decision* element is the directed graph composed of the *Decision* element itself, its *informationRequirements*, its *knowledgeRequirements*, and the union of the requirement subgraphs of each *requiredDecision* or *requiredKnowledge* element: that is, the requirement subgraph of a *Decision* element is the closure of the *informationRequirement*, *requiredInput*, *requiredDecision*, *knowledgeRequirement* and *requiredKnowledge* associations starting from that *Decision* element.

An instance of *Decision* – that is, the model of a decision – is said to be **well-formed** if and only if all of its *informationRequirement* and *knowledgeRequirement* elements are well-formed. That condition entails, in particular, that the requirement subgraph of a *Decision* element SHALL be acyclic, that is, that a *Decision* element SHALL not require itself, directly or indirectly.

Besides its logical components, information requirements, decision logic etc, the model of a decision may also document a business context for the decision (see clause 6.3.8 and Figure 6-14).

The business context for an instance of *Decision* is defined by its association with any number of *supportedObjectives*, which are instances of *Objective* as defined in OMG BMM, any number of *impactedPerformance Indicators*, which are instances of *Performance Indicator*, any number of *decisionMaker* and any number of *decisionOwner*, which are instances of *OrganisationalUnit*.

In addition, an instance of *Decision* may reference any number of *usingProcess*, which are instances of *Process* as defined in OMG BPMN 2.0, and any number of *usingTask*, which are instances of *Task* as defined in OMG BPMN 2.0, and which are the *Processes* and *Tasks* that use the *Decision* element.

Decision inherits all the attributes and model associations from *DRGElement*. Table 11 presents the additional attributes and model associations of the *Decision* class.

Table 11: Decision attributes and model associations

Attribute	Description
question: string [0..1]	A natural language question that characterizes the <i>Decision</i> such that the output of the <i>Decision</i> is an answer to the question.
allowedAnswers: string [0..1]	A natural language description of the answers allowed for the question such as Yes/No, a list of allowed values, a range of numeric values etc.
variable: <i>InformationItem</i>	The instance of <i>InformationItem</i> that stores the result of this <i>Decision</i> .
decisionLogic: <i>Expression</i> [0..1]	The instance of <i>Expression</i> that represents the decision logic for this <i>Decision</i> .

informationRequirement: InformationRequirement [*]	This attribute lists the instances of InformationRequirement that compose this Decision.
knowledgeRequirement: KnowledgeRequirement [*]	This attribute lists the instances of KnowledgeRequirement that compose this Decision.
authorityRequirement: AuthorityRequirement [*]	This attribute lists the instances of AuthorityRequirement that compose this Decision.
supportedObjective: BMM::Objective [*]	This attribute lists the instances of BMM::Objective that are supported by this Decision.
impactedPerformanceIndicator: PerformanceIndicator [*]	This attribute lists the instances of PerformanceIndicator that are impacted by this Decision.
decisionMaker: OrganisationalUnit [*]	The instances of OrganisationalUnit that make this Decision.
decisionOwner: OrganisationalUnit [*]	The instances of OrganisationalUnit that own this Decision.
usingProcesses: BPMN::process [*]	This attribute lists the instances of BPMN::process that require this Decision to be made.
usingTasks: BPMN::task [*]	This attribute lists the instances of BPMN::task that make this Decision.

6.3.8 Business Context Element metamodel

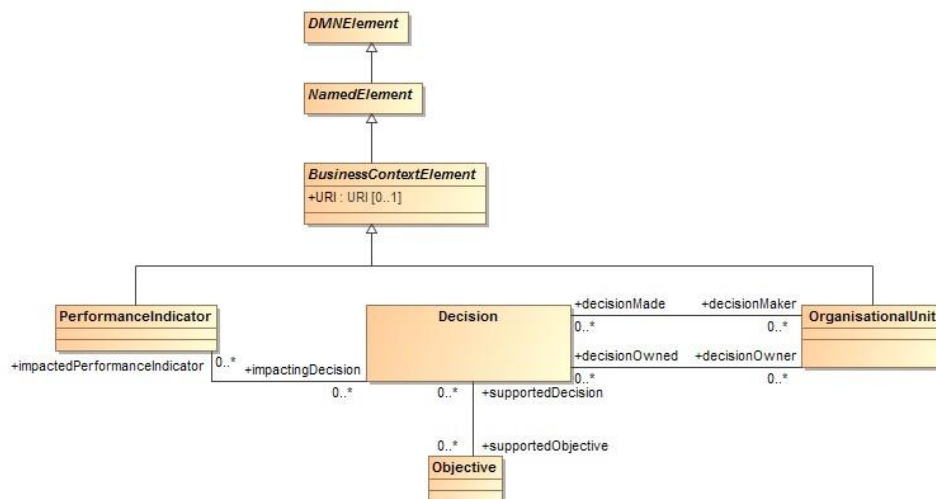


Figure 6-14: BusinessContextElement class diagram

The abstract class `BusinessContextElement`, and its concrete specializations `PerformanceIndicator` and `OrganizationUnit` are placeholders, anticipating a definition to be adopted from other OMG meta-models, such as OMG OSM when it is further developed.

`BusinessContextElement` is a specialization of `NamedElement`, from which it inherits the name and optional `id`, `description`, and `label` attributes.

In addition, instances of `BusinessContextElements` may have a `URI`, which is a `URI`, and

- an instance of `PerformanceIndicator` references any number of `impactingDecision`, which are the `Decision` elements that impact it;
- an instance of `OrganisationalUnit` references any number of `decisionMade` and of `decisionOwned`, which are the `Decision` elements that model the decisions that the organization unit makes or owns.

`BusinessContextElement` inherits all the attributes and model associations from `NamedElement`. Table 12 presents the additional attributes and model associations of the `BusinessContextElement` class.

Table 12: `BusinessContextElement` attributes and model associations

Attribute	Description
URI: anyURI [0..1]	The URI of this <code>BusinessContextElement</code> .

`PerformanceIndicator` inherits all the attributes and model associations from `BusinessContextElement`. Table 13 presents the additional attributes and model associations of the `PerformanceIndicator` class.

Table 13: `PerformanceIndicator` attributes and model associations

Attribute	Description
impactingDecision: <code>Decision</code> [*]	This attribute lists the instances of <code>Decision</code> that impact this <code>PerformanceIndicator</code> .

`OrganisationalUnit` inherits all the attributes and model associations from `BusinessContextElement`. Table 14 presents the additional attributes and model associations of the `OrganisationalUnit` class.

Table 14: `OrganisationalUnit` attributes and model associations

Attribute	Description
decisionMade: <code>Decision</code> [*]	This attribute lists the instances of <code>Decision</code> that are made by this <code>OrganisationalUnit</code> .
decisionOwned: <code>Decision</code> [*]	This attribute lists the instances of <code>Decision</code> that are owned by this <code>OrganisationalUnit</code> .

6.3.9 Business Knowledge Model metamodel

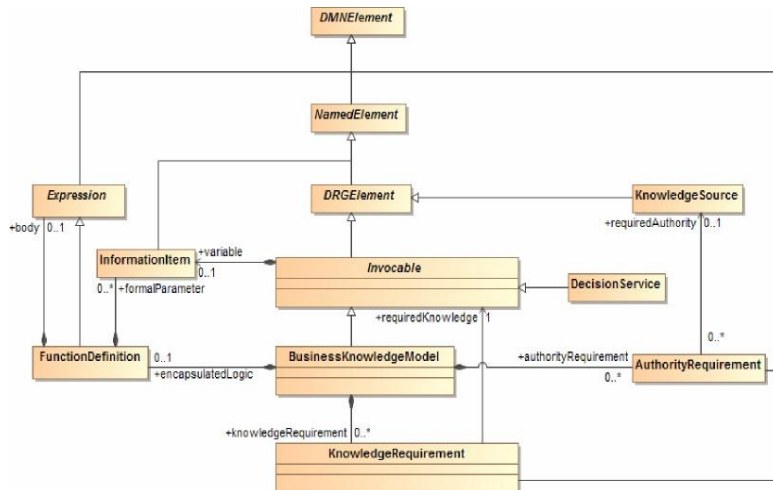


Figure 6-15: BusinessKnowledgeModel class diagram

A business knowledge model has an abstract part, representing reusable, invocable decision logic, and a concrete part, which mandates that the decision logic must be a single FEEL boxed function definition. A decision service is also an invocable element and thus can be invoked as required knowledge from other decisions and business knowledge models.

The class `Invocable` is used to model an invocable element and the class `BusinessKnowledgeModel` is used to model a business knowledge model.

`Invocable` is a specialization of `DRGEElement` and it inherits the name and optional id, description, and label attributes from `NamedElement`. The name of an `Invocable` must be different from the name of any other invocable, input data, decision, or import in the decision model. `BusinessKnowledgeModel` is a specialization of `Invocable` from which it additionally inherits the variable attribute.

A `BusinessKnowledgeModel` element may have zero or more `knowledgeRequirement`, which are instance of `KnowledgeRequirement`, and zero or more `authorityRequirement`, which are instances of `AuthorityRequirement`. These model elements are described below.

The **requirement subgraph** of a `BusinessKnowledgeModel` element is the directed graph composed of the `BusinessKnowledgeModel` element itself, its `knowledgeRequirement` elements, and the union of the requirement subgraphs of all the `requiredKnowledge` elements that are referenced by its `knowledgeRequirements`.

An instance of `BusinessKnowledgeModel` is said to be **well-formed** if and only if, either it does not have any `knowledgeRequirement`, or all of its `knowledgeRequirement` elements are well-formed. That condition entails, in particular, that the requirement subgraph of a `BusinessKnowledgeModel` element SHALL be acyclic, that is, that a `BusinessKnowledgeModel` element SHALL not require itself, directly or indirectly.

At the decision logic level, a `BusinessKnowledgeModel` element contains a `FunctionDefinition`, which is an instance of `Expression` containing zero or more parameters, which are instances of `InformationItem`. The `FunctionDefinition` that is contained in a `BusinessKnowledgeModel` element is the reusable module of decision logic that is represented by this `BusinessKnowledgeModel` element. An `Invocable` element contains an `InformationItem` that holds an invocable reference to the abstract business knowledge, which allows a `Decision` to invoke it by name. The name of that `InformationItem` SHALL be the same as the name of the `Invocable` element. `Invocable` inherits all the attributes and model associations from `DRGEElement`.

Table 15 presents the additional attributes and model associations of the `Invocable` class. Table 16 presents the additional attributes and model associations of the `BusinessKnowledgeModel` class.

Table 15: `Invocable` attributes and model associations

Attribute	Description
variable: <code>InformationItem</code>	This attribute defines a variable that is bound to the function defined by the <code>FunctionDefinition</code> , allowing decision logic to invoke the function by name.

Table 16: `BusinessKnowledgeModel` attributes and model associations

Attribute	Description
encapsulatedLogic: <code>FunctionDefinition</code> [0..1]	The function that encapsulates the logic encapsulated by this <code>BusinessKnowledgeModel</code> .
knowledgeRequirement: <code>KnowledgeRequirement</code> [*]	This attribute lists the instances of <code>KnowledgeRequirement</code> that compose this <code>BusinessKnowledgeModel</code> .
authorityRequirement: <code>AuthorityRequirement</code> [*]	This attribute lists the instances of <code>AuthorityRequirement</code> that compose this <code>BusinessKnowledgeModel</code> .

6.3.10 Decision service metamodel

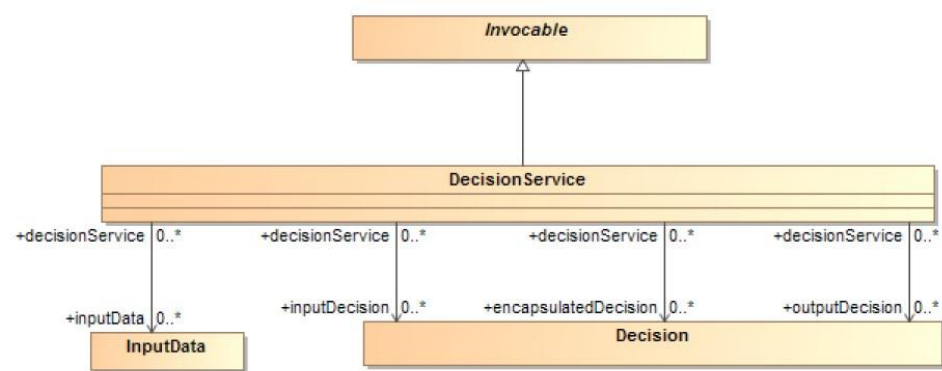


Figure 6-16: `DecisionService` class diagram

The `DecisionService` class is used to define named decision services against the decision model contained in an instance of `Definitions`.

`DecisionService` is a kind of `Invocable` element, from which an instance of `DecisionService` inherits the name and optional id, description, and label attributes, which are Strings, and a variable, which is an `InformationItem`. The id of a `DecisionService` element SHALL be unique within the containing instance of `Definitions`.

The name of the variable and the name of the `DecisionService` SHALL be the same. This name may be used to invoke a `DecisionService` from the decision logic of another decision or business knowledge model.

A `DecisionService` element has one or more associated `outputDecisions`, which are the instances of `Decision` required to be output by this `DecisionService`, i.e., the `Decisions` whose results the `Decision Service` must return when called.

A `DecisionService` element has zero or more `encapsulatedDecisions`, which are the instances of `Decision` required to be encapsulated by this `DecisionService`, i.e., the `Decisions` to be evaluated by the `Decision Service` when it is called.

A `DecisionService` element has zero or more `inputDecisions`, which are the instances of `Decision` required as input by this `DecisionService`, i.e., the `Decisions` whose results will be provided to the `Decision Service` when it is called.

A `DecisionService` element has zero or more `inputData`, which are the instances of `InputData` required as input by this `DecisionService`, i.e., the `Input Data` which will be provided to the `Decision Service` when it is called.

The `encapsulatedDecisions`, `inputDecisions` and `inputData` attributes are optional. At least one of the `encapsulatedDecisions` and `inputDecisions` attributes SHALL be specified.

The **requirement subgraph** of a `DecisionService` element is the directed graph composed of the `DecisionService` element itself and the union of the requirement subgraphs of all the `Decision` elements that are referenced by its `encapsulatedDecisions` and `outputDecisions`.

An instance of `DecisionService` is said to be **well-formed** if and only if its requirement subgraph is acyclic, that is, that a `DecisionService` element SHALL not require itself, directly or indirectly.

`DecisionService` inherits all the attributes and model associations from `Invocable`. Table 17 presents the additional attributes and model associations of the `DecisionService` element.

Table 17: `DecisionService` attributes and model associations

Attribute	Description
outputDecisions: <code>Decision</code> [1..*]	This attribute lists the instances of <code>Decision</code> required to be output by this <code>DecisionService</code> .
encapsulatedDecisions: <code>Decision</code> [0..*]	If present, this attribute lists the instances of <code>Decision</code> to be encapsulated in this <code>DecisionService</code>
inputDecisions: <code>Decision</code> [0..*]	If present, this attribute lists the instances of <code>Decision</code> required as input by this <code>DecisionService</code> .
inputData: <code>InputData</code> [0..*]	If present, this attribute lists the instances of <code>InputData</code> required as input by this <code>DecisionService</code>

6.3.11 Input Data metamodel

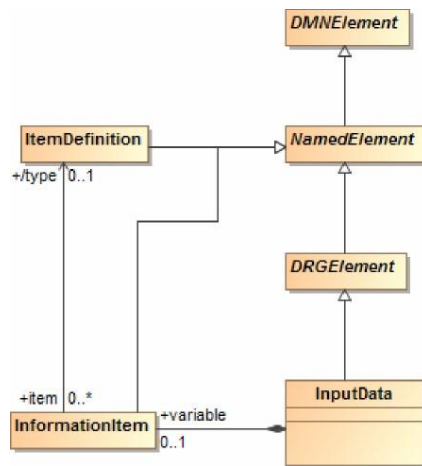


Figure 6-17: InputData class diagram

DMN uses the class `InputData` to model the inputs of a decision whose values are defined outside of the decision model.

`InputData` is a concrete specialization of `DRGElement` and it inherits the name and optional id, description and label attributes from `NamedElement`. The name of an `InputData` must be different from the name of any other decision, input data, business knowledge model, decision service, or import in the decision model.

An instance of `InputData` defines an `InformationItem` that stores its value. This `InformationItem` may include a `typeRef` that specifies the type of data that is this `InputData` represents, either an `ItemDefinition`, base type in the specified `expressionLanguage`, or imported type.

In a DRD, an instance of `InputData` is represented by an **input data** diagram element. An `InputData` element does not have a **requirement subgraph**, and it is always **well-formed**.

`InputData` inherits all the attributes and model associations from `DRGElement`. Table 18 presents the additional attributes and model associations of the `InputData` class.

Table 18: InputData attributes and model associations

Attribute	Description
variable: <code>InformationItem</code>	The instance of <code>InformationItem</code> that stores the result of this <code>InputData</code> .

6.3.12 Knowledge Source metamodel

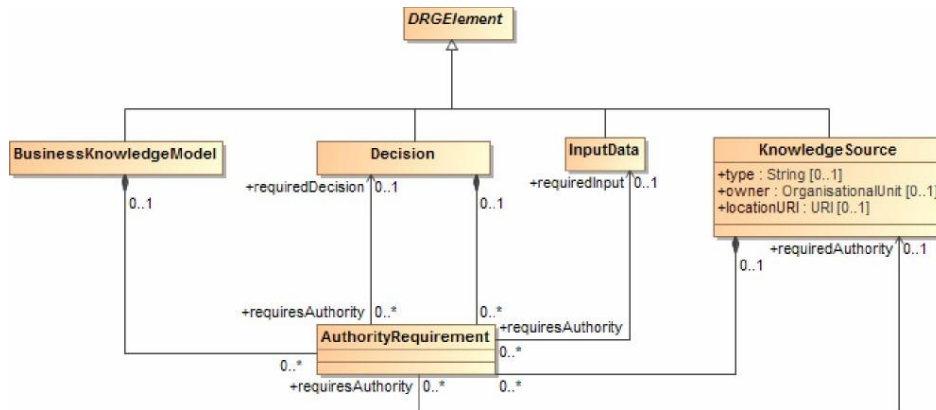


Figure 6-18: KnowledgeSource class diagram

The class KnowledgeSource is used to model authoritative knowledge sources in a decision model. In a DRD, an instance of KnowledgeSource is represented by a **knowledge source** diagram element.

KnowledgeSource is a concrete specialization of DRGElement, and thus of NamedElement, from which it inherits the name and optional id, description, and label attributes. In addition, a KnowledgeSource has a locationURI, which is a URI. It has a type, which is a string, and an owner, which is an instance of OrganisationalUnit. The type is intended to identify the kind of the authoritative source, e.g., Policy Document, Regulation, Analytic Insight.

A KnowledgeSource element is also composed of zero or more authorityRequirement elements, which are instances of AuthorityRequirement.

KnowledgeSource inherits all the attributes and model associations from DRGElement. Table 19 presents the attributes and model associations of the KnowledgeSource class.

Table 19: KnowledgeSource attributes and model associations

Attribute	Description
locationURI: anyURI [0.. 1]	The URI where this KnowledgeSource is located. The locationURI SHALL be specified in a URI format.
type: string [0..1]	The type of this Knowledge Source.
owner: OrganisationalUnit [0..1]	The owner of this KnowledgeSource.
authorityRequirement: AuthorityRequirement [*]	This attribute lists the instances of AuthorityRequirement that contribute to this KnowledgeSource.

6.3.13 Information Requirement metamodel

The class InformationRequirement is used to model an **information requirement**, as represented by a plain arrow in a DRD. InformationRequirement is a specialization of DMNElement, from which it inherits the optional id, description, and label attributes.

An `InformationRequirement` element is a component of a `Decision` element, and it associates that requiring `Decision` element with a `requiredDecision` element, which is an instance of `Decision`, or a `requiredInput` element, which is an instance of `InputData`.

An `InformationRequirement` element references an instance of either a `Decision` or `InputData`, which defines a variable. That variable, which is an instance of `InformationItem`, represents the `InformationRequirement` element at the decision logic level.

Notice that an `InformationRequirement` element must reference the instance of `Decision` or `InputData` that it associates with the requiring `Decision` element, not contain it: instances of `Decision` or `InputData` can only be contained in `Definitions` elements.

An instance of `InformationRequirement` is said to be **well-formed** if and only if all of the following are true:

- It references a `requiredDecision` or a `requiredInput` element, but not both.
- The referenced `requiredDecision` or `requiredInput` element is well-formed.
- The `Decision` element that contains the instance of `InformationRequirement` is not in the requirement subgraph of the referenced `requiredknowledge` element, if this `InformationRequirement` element references one.
- The referenced `requiredDecision` or `requiredInput` element is defined in the same decision model or in an imported decision model.

Table 20 presents the attributes and model associations of the `InformationRequirement` element.

Table 20: `InformationRequirement` attributes and model associations

Attribute	Description
requiredDecision: <code>Decision</code> [0..1]	The instance of <code>Decision</code> that this <code>InformationRequirement</code> associates with its containing <code>Decision</code> element.
requiredInput: <code>InputData</code> [0..1]	The instance of <code>InputData</code> that this <code>InformationRequirement</code> associates with its containing <code>Decision</code> element.

6.3.14 Knowledge Requirement metamodel

The class `KnowledgeRequirement` is used to model a **knowledge requirement**, as represented by a dashed arrow in a DRD. `KnowledgeRequirement` is a specialization of `DMNElement`, from which it inherits the optional `id`, `description`, and `label` attributes.

A `KnowledgeRequirement` element is a component of a `Decision` element or of a `BusinessKnowledgeModel` element, and it associates that requiring `Decision` or `BusinessKnowledgeModel` element with a `requiredKnowledge` element, which is an instance of `Invocable`.

Notice that a `KnowledgeRequirement` element must reference the instance of `Invocable` that it associates with the requiring `Decision` or `BusinessKnowledgeModel` element, not contain it: instances of `BusinessKnowledgeModel` can only be contained in `Definitions` elements.

An instance of `KnowledgeRequirement` is said to be **well-formed** if and only if all of the following are true:

- It references a `requiredKnowledge` element.
- The referenced `requiredKnowledge` element is well-formed.
- If the `KnowledgeRequirement` element is contained in an instance of `BusinessKnowledgeModel`, that `BusinessKnowledgeModel` element is not in the requirement subgraph of the referenced `requiredKnowledge` element.

- The referenced `requiredKnowledge` element is defined in the same decision model or in an imported decision model.

Table 21 presents the attributes and model associations of the `KnowledgeRequirement` element.

Table 21: KnowledgeRequirement attributes and model associations

Attribute	Description
requiredKnowledge: <code>Invocable</code>	The instance of <code>Invocable</code> that this <code>KnowledgeRequirement</code> associates with its containing <code>Decision</code> or <code>BusinessKnowledgeModel</code> element.

6.3.15 Authority Requirement metamodel

The class `AuthorityRequirement` is used to model an **authority requirement**, as represented by an arrow drawn with a dashed line and a filled circular head in a DRD. `AuthorityRequirement` is a specialization of `DMNElement`, from which it inherits the optional `id`, `description`, and `label` attributes.

An `AuthorityRequirement` element is a component of a `Decision`, `BusinessKnowledgeModel` or `KnowledgeSource` element, and it associates that requiring `Decision`, `BusinessKnowledgeModel` or `KnowledgeSource` element with a `requiredAuthority` element, which is an instance of `KnowledgeSource`, a `requiredDecision` element, which is an instance of `Decision`, or a `requiredInput` element, which is an instance of `InputData`.

Notice that an `AuthorityRequirement` element must reference the instance of `KnowledgeSource`, `Decision` or `InputData` that it associates with the requiring element, not contain it: instances of `KnowledgeSource`, `Decision` or `InputData` can only be contained in `Definitions` elements.

Table 22 presents the attributes and model associations of the `AuthorityRequirement` element.

Table 22: AuthorityRequirement attributes and model associations

Attribute	Description
requiredAuthority: <code>KnowledgeSource</code> [0.. 1]	The instance of <code>KnowledgeSource</code> that this <code>AuthorityRequirement</code> associates with its containing <code>KnowledgeSource</code> , <code>Decision</code> or <code>BusinessKnowledgeModel</code> element.
requiredDecision: <code>Decision</code> [0..1]	The instance of <code>Decision</code> that this <code>AuthorityRequirement</code> associates with its containing <code>KnowledgeSource</code> element.
requiredInput: <code>InputData</code> [0.. 1]	The instance of <code>InputData</code> that this <code>AuthorityRequirement</code> associates with its containing <code>KnowledgeSource</code> element.

6.3.16 Extensibility

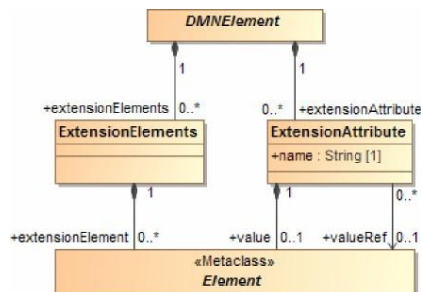


Figure 6-19: Extensibility class diagram

The **DMN** metamodel is aimed to be extensible. This allows **DMN** adopters to extend the specified metamodel in a way that allows them to be still **DMN**-compliant. It provides a set of extension elements, which allows **DMN** adopters to attach additional attributes and elements to standard and existing **DMN** elements. This approach results in more interchangeable models because the standard elements are still intact and can still be understood by other **DMN** adopters. It's only the additional attributes and elements that MAY be lost during interchange.

A **DMN** extension can be done using two different elements:

1. ExtensionElements
2. ExtensionAttribute

ExtensionElements is a container for attaching arbitrary elements from other metamodels to any **DMN** element. **ExtensionAttribute** allows these attachments to also have name. This allows **DMN** adopters to integrate any metamodel into the **DMN** metamodel and reuse already existing model elements.

6.3.16.1 ExtensionElements

The **ExtensionElements** element is a container to aggregate elements from other metamodels inside any **DMNElement**. Table 23 presents the attributes and model associations for the **ExtensionElements** element.

Table 23: **ExtensionElements** attributes and model associations

Attribute	Description
extensionElement: Element [0..*]	The contained Element. This association is not applicable when the XML schema interchange is used, since the XSD mechanism for supporting "any" elements from other namespaces already satisfies this requirement.

The **ExtensionAttribute** element contains an **Element** or a reference to an **Element** from another metamodel. An **ExtensionAttribute** also has a name to define the role or purpose of the associated element. This type is not applicable when the XML schema interchange is used, since the XSD mechanism for supporting "anyAttribute" from other namespaces already satisfies this requirement. Table 24 presents the model associations for the **ExtensionAttribute** element.

Table 24: `ExtensionAttribute` attributes and model associations

Attribute	Description
name: string	The name of the extension attribute.
value: <code>Element</code> [0..1]	The contained <code>Element</code> . This attribute SHALL NOT be used together with <code>valueRef</code> .
valueRef: <code>Element</code> [0..1]	A reference to the associated <code>Element</code> . This attribute SHALL NOT be used together with <code>value</code> .

6.4 Examples

Examples of DRDs are provided in clause 12.1.3.

The page intentionally left blank

7 Relating Decision Logic to Decision Requirements

7.1 Introduction

Clause 6 described how the decision requirements level of a decision model – a DRG represented in one or more DRDs – may be used to model the structure of an area of decision making. However, the details of how each decision's outcome is derived from its inputs must be modeled at the decision logic level. This section introduces the principles by which decision logic may be associated with elements in the DRG. Specific representations of decision logic (decision tables and FEEL expressions) are then defined in clauses 8, 9 and 10.

The decision logic level of a decision model in **DMN** consists of one or more value expressions. The elements of decision logic modeled as value expressions include tabular expressions such as decision tables and invocations, and literal (text) expressions such as *age > 30*.

- A **literal expression** represents decision logic as text that describes how an output value is derived from its input values. The expression language may, but need not, be formal or executable: examples of literal expressions include a plain English description of the logic of a decision, a first order logic proposition, a Java computer program and a PMML document or ONNX file. Clause 10 specifies an executable expression language called **FEEL**. Clause 9 specifies a subset of FEEL (S-FEEL) that is the default language for literal expressions in **DMN** decision tables (clause 8).
- A **decision table** is a tabular representation of decision logic, based on a discretization of the possible values of the inputs of a decision, and organized into rules that map discretized input values onto discrete output values (see clause 8).
- An **invocation** is a tabular representation of how decision logic that is represented by a business knowledge model or a decision service is invoked by a decision, or by another business knowledge model. An invocation may also be represented as a literal expression, but usually the tabular representation will be more understandable.

Tabular representations of decision logic are called *boxed expressions* in the remainder of this specification.

All three **DMN** conformance levels include all the above expressions. At **DMN** Conformance Level 1, literal expressions are not interpreted and, therefore, free. At **DMN** Conformance Level 2, literal expressions are restricted to S-FEEL. Clause 10 specifies additional boxed expressions available at **DMN** Conformance Level 3.

Decision logic is added to a decision model by including a value expression component in some of the decision model elements in the DRG:

- From a decision logic viewpoint, a decision is a piece of logic that defines how a given question is answered, based on the input data. As a consequence, each **decision** element in a decision model may include a value expression that describes how a decision outcome is derived from its required input, possibly invoking a business knowledge model;
- From a decision logic viewpoint, a business knowledge model is a piece of decision logic that is defined as a function allowing it to be re-used in multiple decisions. As a consequence, each **business knowledge model** element may include a value expression, which is the body of that function.

Another key component of the decision logic level is the **variable**: Variables are used to store values of Decisions and InputData for use in value expressions. InformationRequirements specify variables in scope via reference to those Decisions and InputData, so that value expressions may reference these variables. Variables link information requirements in the DRG to the value expressions at the decision logic level:

- From a decision logic viewpoint, an information requirement is a requirement for an externally provided value to be assigned to a free variable in the decision logic, so that a decision can be evaluated. As a consequence, each **information requirement** in a decision model points to a Decision or InputData, which in turn defines a variable that represents the associated data input in the decision's expression.

- The variables that are used in the body of the function defined by a business knowledge model element in the DRG must be bound to the information sources in each of the requiring decisions. As a consequence, each **business knowledge model** includes zero or more variables that are the parameters of the function.

The third key element of the decision logic level are the **item definitions** that describe the types and structures of data items in a decision model: **input data** elements in the DRG, and **variables** and **value expressions** at the decision logic level, may reference an associated item definition that describes the type and structure of the data expected as input, assigned to the variable or resulting from the evaluation of the expression.

Notice that **knowledge sources** are not represented at the decision logic level: knowledge sources are part of the documentation of the decision logic, not of the decision logic itself.

The dependencies between decisions, required information sources and business knowledge models, as represented by the information and knowledge requirements in a DRG, constrain how the value expressions associated with these elements relate to each other.

As explained above, every decision, input data, and business knowledge model at the DRG level is associated with a variable used at the decision logic level. Each variable that is referenced by a decision's expression must be associated with a required decision, required input data, or required knowledge. Also, each variable associated with the required decisions, required input data, and required knowledge **SHOULD** be referenced in the decision's expression.

- If a decision requires another decision, the value expression of the required decision assigns the value to the variable for use in evaluating the requiring decision. This is the generic mechanism in **DMN** for composing decisions at the decision logic level.
- If a decision requires an input data, the value of the variable is assigned the value of the data source attached to the input data at execution time. This is the generic mechanism in **DMN** for instantiating the data requirements for a decision.

The input variables of a decision's decision logic must not be used outside that value expression or its component value expressions: the decision element defines the lexical scope of the input variables for its decision logic. To avoid name collisions and ambiguity, the name of a variable must be unique within its scope. When DRG elements are mapped to FEEL, the name of a variable is the same as the (possibly qualified) name of its associated input data or decision, which guarantees its uniqueness.

When DRG elements are mapped to FEEL, all the decisions and input data in a DRG define a *context*, which is the literal expression that represents the logic associated with the decision element and that represents that scope (see 10.3.2.6). The information requirement elements in a decision are *context entries* in the associated context, where the *key* is the name of the variable that the information requirement defines, and where the *expression* is the *context* that is associated with the required decision or input data element that the information requirement references. The value expression that is associated with the decision as its decision logic is the *expression* in the *context entry* that specifies what is the result of the *context*.

In the same way, a business knowledge model element defines the lexical scope of its parameters, that is, of the input variables for its body.

In FEEL, the literal expression and scoping construct that represents the logic associated with a business knowledge model element is a *function definition* (see 10.3.2.13), where the *formal parameters* are the names of the parameters in the business knowledge model element, and the expression is the value expression that is the body of the business knowledge model element.

If a business knowledge model element requires one or more other business knowledge models, it **SHOULD** have an explicit value expression that describes how the required business knowledge models are invoked and their results combined or otherwise elaborated.

7.2 Notation

7.2.1 Expressions

We define a graphical notation for decision logic called **boxed expressions**. This notation serves to decompose the decision logic model into small pieces that can be associated with DRG artifacts. The DRD plus the boxed expressions form a complete, mostly graphical language that completely specifies Decision Models.

In addition to the generic notation of **boxed expression**, this section specifies two kinds of boxed expressions:

- **boxed literal expression**
- **boxed invocation**

The boxed expression for a decision table is defined in clause 8. Further types of boxed expressions are defined for FEEL, in clause 10.

Boxed expressions are defined recursively, *i.e.*, boxed expressions can contain other boxed expressions. The top-level boxed expression corresponds to the decision logic of a single DRG artifact. This boxed expression SHALL have a name box that contains the name of the DRG artifact. The name box may be attached in a single box on top, as shown in Figure 7-1:

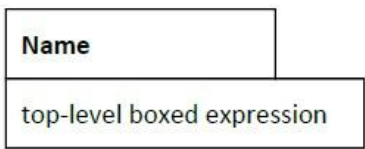


Figure 7-1: Boxed Expression

Alternatively, the name box and expression box can be separated by white space and connected on the left side with a line, as shown in Figure 7-2:

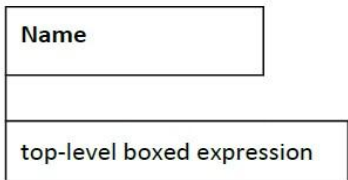


Figure 7-2: Boxed expression with separated name and expression boxes

Name is the only visual link defined between DRD elements and boxed expressions. Graphical tools are expected to support appropriate graphical links, for example, clicking on a decision shape opens a decision table. How the boxed expression is visually associated with the DRD element is left to the implementation.

7.2.2 Boxed literal expression

In a boxed expression, a literal expression is represented by its text. However, two notational conventions are provided to improve the readability of boxed literal expressions: typographical string literals and typographical date and time literals.

7.2.2.1 Typographical string literals

A string literal such as "DECLINED" can be represented alternatively as the italicized literal *DECLINED*. For example, Figure 7-3 is equivalent to Figure 7-4:

Credit contingency factor table		
U	Risk Category	Credit Contingency Factor
1	<i>HIGH, DECLINE</i>	0.6
2	<i>MEDIUM</i>	0.7
3	<i>LOW, VERY LOW</i>	0.8

Figure 7-3: Decision table with italicized literals

Credit contingency factor table		
U	Risk Category	Credit Contingency Factor
1	"HIGH", "DECLINE"	0.6
2	"MEDIUM"	0.7
3	"LOW", "VERY LOW"	0.8

Figure 7-4: Decision table with string literals

To avoid having to discern whether (e.g.) *HIGH, DECLINE* is "HIGH," "DECLINE," or "HIGH, DECLINE," typographical string literals SHALL be free of commas (", " characters). FEEL typographical string literals SHALL conform to grammar rule 22 (name).

7.2.2.2 Typographical date and time literals

A date, time, date and time, or duration expression such as `date("2013-08-09")` can be represented alternatively as the bold italicized literal ***2013-08-09***. The literal SHALL obey the syntax specified in clauses 10.3.2.3.4, 10.3.2.3.5, and 10.3.2.3.7.

7.2.3 Boxed invocation

An invocation is a container for the parameter bindings that provide the context for the evaluation of the body of a business knowledge model.

The representation of an invocation is the name of the business knowledge model with the parameters' bindings explicitly listed.

As a boxed expression, an invocation is represented by a box containing the name of the business knowledge model to be invoked, and boxes for a list of bindings, where each binding is represented by two boxed expressions on a row: the box on the left contains the name of a parameter, and the box on the right contains the binding expression, that is the expression whose value is assigned to the parameter for the purpose of evaluating the invoked business knowledge model (see Figure 7-5).

Name	
invoked business knowledge model	
parameter 1	Binding expression 1
...	
parameter 2	Binding expression 2
parameter <i>n</i>	Binding expression <i>n</i>

Figure 7-5: Boxed invocation

The invoked business knowledge model is represented by the name of the business knowledge model. Any other visual linkage is left to the implementation.

7.3 Metamodel

An important characteristic of decisions and business knowledge models is that they may contain an expression that describes the logic by which a modeled decision shall be made, or pieces of that logic.

The class `Expression` is the abstract superclass for all expressions that are used to describe complete or parts of decision logic in **DMN** models and that return a single value when interpreted (clause 7.3.1). Here “single value” possibly includes structured data, such as a decision table with multiple output clauses.

DMN defines three concrete kinds of `Expression`: `LiteralExpression`, `DecisionTable` (see 8) and `Invocation`.

An expression may reference variables, such that the value of the expression, when interpreted, depends on the values assigned to the referenced variables. The class `InformationItem` is used to model variables in expressions.

The value of an expression, like the value assigned to a variable, may have a structure and a range of allowable values. The class `ItemDefinition` is used to model data structures and ranges.

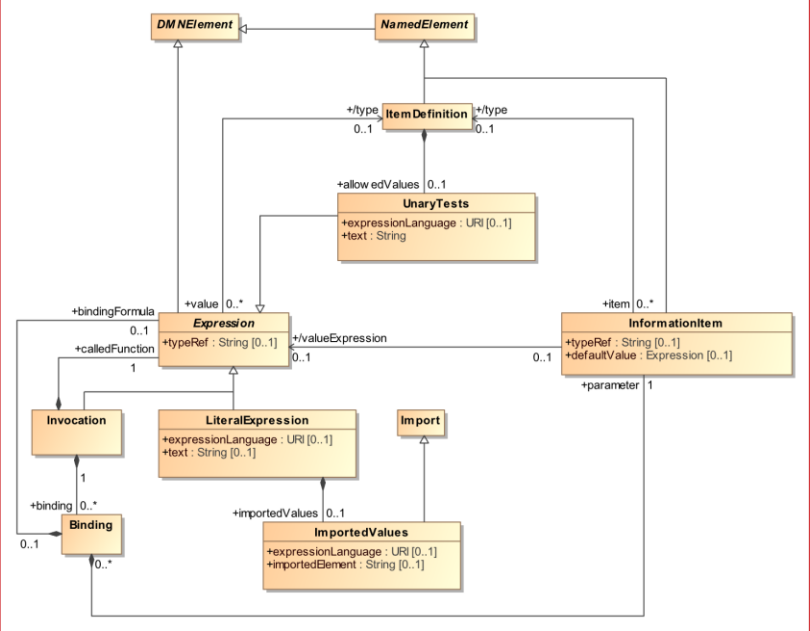


Figure 7-6: Expression class diagram

7.3.1 Expression metamodel

An important characteristic of decisions and business knowledge models is that they may contain an expression that describes the logic by which a modeled decision shall be made, or pieces of that logic.

Expression is an abstract specialization of DMNElement, from which it inherits the optional `id`, `description`, and `label` attributes.

An instance of Expression is a component of a Decision element, of a BusinessKnowledgeModel element, or of an ItemDefinition element, or it is a component of another instance of Expression, directly or indirectly.

An Expression references zero or more variables implicitly by using their names in its expression text. These variables, which are instances of InformationItem, are lexically scoped, depending on the Expression type. If the Expression is the logic of a Decision, the scope includes that Decision's requirements. If the Expression is the body of the encapsulatedLogic of a BusinessKnowledgeModel, the scope includes the FunctionDefinition's parameters and the BusinessKnowledgeModel's requirements. If the Expression is the value of a ContextEntry, the scope includes the previous entries in the Context. An instance of Expression references an optional typeRef, which points to either a base type in the default typeLanguage, a custom type specified by an ItemDefinition, or an imported type. The referenced type specifies the Expression's range of possible values. If an instance of Expression that defines the output of a Decision element includes a typeRef, the referenced type SHALL be the same as the type of the containing Decision element.

An instance of Expression can be interpreted to derive a single value from the values assigned to its variables. How the value of an Expression element is derived from the values assigned to its variables depends on the concrete kind of the Expression. The ItemDefinition element specializes NamedElement and it inherits its attributes and model associations. Table 26 presents the additional attributes and model associations of the ItemDefinition element.

Expression inherits from the attributes and model associations of DMNElement.

7.3.2 UnaryTests Metamodel

The class `UnaryTests` is used to model a boolean test where the argument to be tested is implicit or denoted with a `?`, and whose value is specified by text in some specified expression language.

`UnaryTests` is a concrete subclass of `Expression`.

An instance of `UnaryTests` inherits an optional `typeRef` from `Expression`, which SHALL NOT be used. An instance of `UnaryTests` also has an optional `text`, which is a `String`, and an optional `expressionLanguage`, which is a `String` that identifies the expression language of the text. If no `expressionLanguage` is specified, the expression language of the text is the `expressionLanguage` that is associated with the containing instance of `Definitions`. The `expressionLanguage` SHALL be specified in a URI format. The default expression language is FEEL. When the expression language is FEEL, the text must conform to grammar rule 15 in section 10.3.1.2.

A `UnaryTests` is satisfied if and only if one of the following alternatives is true:

- a) One of the expressions in the `UnaryTests` evaluates to a value, and the implicit value is equal to that value.
- b) One of the expressions in the `UnaryTests` evaluates to a list of values, and the implicit value is equal to at least one of the values in that list.
- c) One of the expressions in the `UnaryTests` evaluates to true when the implicit value is applied to it.
- d) One of the expressions in the `UnaryTests` is a boolean expression using the special `'?'` variable and that expression evaluates to true when the implicit value is assigned to `'?'`.

Table 25 presents additional attributes and model associations of the `UnaryTests` element.

Table 25: `UnaryTests` attributes and model associations

Attribute	Description
<code>text: string[0..1]</code>	The text of this <code>UnaryTests</code> . It SHALL be a valid expression in the <code>expressionLanguage</code>
<code>expressionLanguage: anyURI[0..1]</code>	This attribute identifies the expression language used in this <code>UnaryTests</code> . This value overrides the expression language specified for the containing instance of <code>DecisionRequirementDiagram</code> . The language SHALL be specified in a URI format.

7.3.3 ItemDefinition metamodel

The inputs and output of decisions, business knowledge models, and decision services, and the output of input data (all `DRGElements`) are data items whose value, at the logic level, is assigned to variables or represented by `Expressions`.

An important characteristic of data items in decision models is their structure. **DMN** does not require a particular format for this data structure, but it does designate a subset of FEEL as its default.

The class `ItemDefinition` is used to model the structure and the range of values of the input and the outcome of decisions.

As a concrete specialization of `NamedElement`, an instance of `ItemDefinition` has a `name` and an optional `id` and `description`. The name of an `ItemDefinition` element SHALL be distinct from the names of other `ItemDefinitions` and `Imports` within the same model.

The default type language for all elements can be specified in the `Definitions` element using the `typeLanguage` attribute. For example, a `typeLanguage` value of <http://www.w3.org/2001/XMLSchema> indicates that the data structures used by elements within that `Definitions` are in the form of XML Schema types. If unspecified, the default is FEEL.

Notice that the data types that are built-in in the `typeLanguage` that is associated with an instance of `Definitions` need not be redefined by `ItemDefinition` elements contained in that `Definitions` element: they are considered imported and can be referenced in **DMN** elements within the `Definitions` element.

The type language can be overridden locally using the `typeLanguage` attribute in the `ItemDefinition` element.

Notice, also, that the data types and structures that are defined at the top level in a data model that is imported using an `Import` element that is associated with an instance of `Definitions` need not be redefined by `ItemDefinition` elements contained in that `Definitions` element: they are considered imported and can be referenced in **DMN** elements within the `Definitions` element.

An `ItemDefinition` element MAY have a `typeRef`, which is a string that references, as a qualified name, either an `ItemDefinition` in the current instance of `Definitions` or a built-in type in the specified `typeLanguage` or a type defined in an imported DMN, XSD, or other document. In the latter case, the external document SHALL be imported in the `Definitions` element that contains the instance of `ItemDefinition`, using an `Import` element specifying both the namespace value and its name when used as a qualifier. For example, in the case of data structures contributed by an XML schema, an `Import` would be used to specify the file location of that schema, and the `typeRef` attribute would reference the type or element definition in the imported schema. If the type language is FEEL the built-in types are the FEEL built-in data types: *number*, *string*, *boolean*, *days and time duration*, *years and months duration*, *date*, *time*, *date and time* and *Any*. A `typeRef` referencing a built-in type SHALL omit the prefix.

An `ItemDefinition` element may restrict the values that are allowed from `typeRef`, using the `allowedValues` attribute. `allowedValues` is an instance of `UnaryTests` that constrains the domain of the `typeRef`. If an `ItemDefinition` element does not contain `allowedValues`, its range of allowed values is the full range of the referenced `typeRef`. When an `ItemDefinition` has sibling `itemComponents`, their values are available in the evaluation context of the `UnaryTests` of the `allowedValues`. In cases where the `isCollection` attribute of an `ItemDefinition` is true, each element of the collection must satisfy the `UnaryTests` of the `allowedValues`, i.e. the `allowedValues` are projected onto the collection elements. The default value of `isCollection` is false. The `allowedValues` attribute has been deprecated as of DMN 1.5 and replaced with the `typeConstraint` attribute. The `typeConstraint` attribute differs from `allowedValues` by not projecting onto collection elements but directly constraining the collection.

An alternative way to define an instance of `ItemDefinition` is as a composition of `ItemDefinition` elements. An instance of `ItemDefinition` may contain zero or more `itemComponent`, which are themselves `ItemDefinitions`. Each `itemComponent` may be defined by either a `typeRef`, `allowedValues`, and `typeConstraint` or a nested `itemComponent`. In this way, complex types may be defined within DMN. The name of an `itemComponent` (nested `ItemDefinition`) must be unique within its containing `ItemDefinition` or `itemComponent`.

An alternative way to define an instance of `ItemDefinition` is by specifying a `FunctionItem` element, which defines the signature of a function: the parameters and the output of the function. An instance of `ItemDefinition` may contain at most one `FunctionItem`. A `FunctionItem` may contain zero or more parameters defined as `InformationItems` and one output type defined as a `typeRef`. The names of the parameters of a `FunctionItem` are unique.

An `ItemDefinition` element SHALL be defined using only one of the alternative ways:

- reference to a built-in or imported `typeRef`, possibly restricted with `allowedValues`
- composition of `ItemDefinition` elements

- function signature element.

The `ItemDefinition` element specializes `NamedElement` and it inherits its attributes and model associations. Table 26 presents the additional attributes and model associations of the `ItemDefinition` element.

Table 26: ItemDefinition attributes and model associations

Attribute	Description
typeRef: String [1]	This attribute identifies by namespace-prefixed name the base type of this <code>ItemDefinition</code> .
typeLanguage: String [0..1]	This attribute identifies the type language used to specify the base type of this <code>ItemDefinition</code> . This value overrides the type language specified in the <code>Definitions</code> element. The language SHALL be specified in a URI format.
allowedValues: <code>UnaryTests</code> [0..1]	This attribute is a <code>UnaryTests</code> that restricts the values in the base type that are allowed in this <code>ItemDefinition</code> . In case of a collection, it is projected on the collection elements. (deprecated)
itemComponent: <code>ItemDefinition</code> [*]	This attribute defines zero or more nested <code>ItemDefinitions</code> that compose this <code>ItemDefinition</code> .
isCollection: Boolean [0..1]	Setting this flag to <i>true</i> indicates that the actual values defined by this <code>ItemDefinition</code> are collections of allowed values. The default is <i>false</i> .
isRange: Boolean [0..1]	Setting this flag to <i>true</i> indicates that the actual values defined by this <code>ItemDefinition</code> are ranges of allowed values. The default is <i>false</i> . The attributes <code>isCollection</code> and <code>isRange</code> are mutually exclusive.
functionItem: <code>FunctionItem</code> [0..1]	This attribute describes an optional <code>FunctionItem</code> that compose this <code>ItemDefinition</code> .
typeConstraint: <code>UnaryTests</code> [0..1]	This attribute is a <code>UnaryTests</code> that restricts the values in the base type that are allowed in this <code>ItemDefinition</code> . In case of a collection, it directly constrains the collection and is not projected on the collection elements.

Commented [AF10]: DMN17-74

Commented [AF11]: DMN17-74

Table 27: FunctionItem attributes and model associations

Attribute	Description
outputTypeRef: String [0..1]	Reference to output type of function
parameters: <code>InformationItem</code> [0..*]	Function parameters as <code>InformationItems</code>

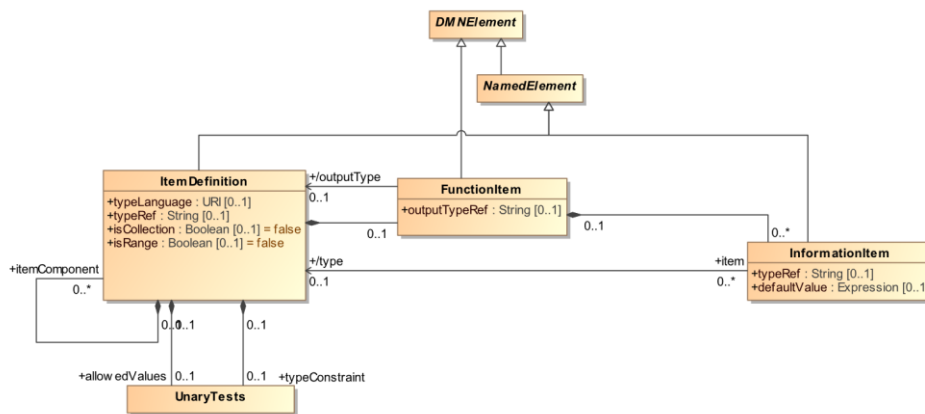


Figure 7-7: ItemDefinition class diagram

7.3.4 InformationItem metamodel

The class `InformationItem` is used to model variables at the decision logic level in decision models.

`InformationItem` is a concrete subclass of `NamedElement`, from which it inherits the `id`, and optional name, description, and label attributes, except that an `InformationItem` element SHALL have a name attribute, which is the name that is used to represent it in other `Expression` elements. The name of an `InformationItem` element SHALL be unique within its scope.

Variables represent values that result from a decision, are assigned to input data by an external data source or are passed to a module of decision logic that is defined as a function (and that is represented by a business knowledge model element). In the first or second case, a variable may be referenced by other dependent decisions by means of their information requirements. In the third case, a variable is one of the parameters of the function that is the realization, at the decision logic level, of a business knowledge model element.

A variable representing an instance of `Decision` or `InputData` referenced by a `InformationRequirement` SHOULD be referenced by the value expression of the decision logic in the `Decision` element that contains the `InformationRequirement` element. A parameter in an instance of `BusinessKnowledgeModel` SHOULD be a variable in the value expression of that `BusinessKnowledgeModel` element.

The `ItemInformation` elements are assigned values as follows:

- An `InformationItem` element contained in a `Decision` is assigned the value of the `Decision`'s value expression.
- An `InformationItem` element that is a parameter in a `FunctionDefinition` is assigned a value by a `Binding` element as part of an instance of `Invocation`.
- An `InformationItem` element contained in an `InputData` is assigned a value by an external data source that is attached at runtime. If no value is provided, **null** is assigned to the `InformationItem`. When an `InputData` is imported several times via transitive imports, the contained `InformationItem` is assigned only once and holds the same value.
- An `InformationItem` element contained in a `ContextEntry` is assigned a value by the `ContextEntry`'s value expression.
- An `InformationItem` element contained in an `Invocable` can never be assigned.

- An `InformationItem` element contained in an `Invocable` cannot have a `defaultValue`.
- If the value to be assigned is not specified or is null, and the `InformationItem` has a `defaultValue` (see Table 28), then the corresponding `InformationItem` is assigned the value of Expression specified in the `defaultValue` attribute.

An `InformationItem` element contained in a `Decision` is assigned the value of the `Decision`'s value expression:

- An `InformationItem` element that is a parameter in a `FunctionDefinition` is assigned a value by a `Binding` element as part of an instance of `Invocation`.
- An `InformationItem` element contained in an `InputData` is assigned a value by an external data source that is attached at runtime. When an `InputData` is imported several times via transitive imports, the contained `InformationItem` is assigned only once and holds the same value.
- An `InformationItem` element contained in a `ContextEntry` is assigned a value by the `ContextEntry`'s value expression.

Commented [AF13]: Editorial: spelling corrected from type in DMN17-105 ("InfiomationItem")

Commented [AF14]: DMN17-105

In any case, the datatype indicated by the `typeRef` that is associated with an instance of `InformationItem` SHALL be compatible with the datatype that is associated with the **DMN** model element from which it takes its value. `InformationItem` inherits all of the attributes and model associations of `NamedElement`. Table 28 presents the additional attributes and model associations of the `InformationItem` element.

Table 28: `InformationItem` attributes and model associations

Attribute	Description
<code>/valueExpression</code> : Expression [0..1]	The Expression whose value is assigned to this <code>InformationItem</code> . This is a derived attribute.
<code>typeRef</code> : String [1]	Qualified name of the type of this <code>InformationItem</code> .
<code>defaultValue</code> : Expression [0..1]	This attribute specifies an instance of Expression that is selected when the value of the associated variable is not provided. The Expression must be a constant expression

7.3.5 Literal expression metamodel

The class `LiteralExpression` is used to model a value expression whose value is specified by text in some specified expression language.

`LiteralExpression` is a concrete subclass of `Expression`, from which it inherits the `id` and `typeRef` attributes.

An instance of `LiteralExpression` has an optional text, which is a `String`, and an optional `expressionLanguage`, which is a `String` that identifies the expression language of the text. If no `expressionLanguage` is specified, the expression language of the text is the `expressionLanguage` that is associated with the containing instance of `Definitions`. The `expressionLanguage` SHALL be specified in a URI format. The default expression language is FEEL.

As a subclass of `Expression`, each instance of `LiteralExpression` has a value. The text in an instance of `LiteralExpression` determines its value, according to the semantics of the `LiteralExpression`'s `expressionLanguage`. The semantics of **DMN** decision models as described in this specification applies only if the text of all the instances of `LiteralExpression` in the model are valid expressions in their associated expression language.

An instance of `LiteralExpression` may include `importedValues`, which is an instance of a subclass `Import` that identifies where the text of the `LiteralExpression` is located. `importedValues` is an

expression that selects text from an imported document. An instance of `LiteralExpression` SHALL NOT have both a `text` and `importedValues`. The `importType` of the `importedValues` identifies the type of document containing the imported text and SHALL be consistent with the `expressionLanguage` of the `LiteralExpression` element. The `expressionLanguage` of the `importedValues` element identifies how the imported text is selected from the imported document. For example, if the `importType` indicates an XML document, the `expressionLanguage` of `importedValues` could be XPATH 2.0.

`LiteralExpression` inherits of all the attributes and model associations of `Expression`. Table 29 presents the additional attributes and model associations of the `LiteralExpression` element.

Table 29: `LiteralExpression` attributes and model associations

Attribute	Description
text: string [0..1]	The text of this <code>LiteralExpression</code> . It SHALL be a valid expression in the <code>expressionLanguage</code> .
expressionLanguage: anyURI [0.. 1]	This attribute identifies the expression language used in this <code>LiteralExpression</code> . This value overrides the expression language specified for the containing instance of <code>DecisionRequirementDiagram</code> . The language SHALL be specified in a URI format.
importedValues: <code>ImportedValues</code> [0..1]	The instance of <code>ImportedValues</code> that specifies where the text of this <code>LiteralExpression</code> is located.

7.3.6 Invocation metamodel

Invocation is a mechanism that permits the evaluation of one value expression – the invoked expression – inside another value expression – the invoking expression – by binding locally the input variables of the invoked expression to values inside the invoking expression. In an invocation, the input variables of the invoked expression are usually called: *parameters*. Invocation permits the same value expression to be re-used in multiple expressions, without having to duplicate it as a sub-expression in all the using expressions.

The class `Invocation` is used to model invocations as a kind of `Expression`: `Invocation` is a concrete specialization of `Expression`.

An instance of `Invocation` is made of zero or more `binding`, which are instances of `Binding`, and model how the `bindingFormulas` are bound to the `formalParameters` of the invoked function. The `formalParameters` of a `FunctionDefinition` are `InformationItems` and the `parameters` of the `Bindings` are `InformationItems`. The binding is by matching the `InformationItem` names.

An `Invocation` contains a `calledFunction`, an `Expression`, which must evaluate to a function. Most commonly, it is a `LiteralExpression` naming a `BusinessKnowledgeModel`.

The value of an instance of `Invocation` is the value of the associated `calledFunction`'s body, with its `formalParameters` assigned values at runtime per the bindings in the `Invocation`.

`Invocation` MAY be used to model invocations in decision models, when a `Decision` element has exactly one `knowledgeRequirement` element, and when the `decisionLogic` in the `Decision` element consists only in invoking the `BusinessKnowledgeModel` element that is referenced by that `requiredKnowledge` and a more complex value expression is not required.

Using `Invocation` instances as the `decisionLogic` in `Decision` elements permits the re-use of the `encapsulatedLogic` of a `BusinessKnowledgeModel` as the logic for any instance of `Decision` that requires that `BusinessKnowledgeModel`, where each requiring `Decision` element specifies its own bindings for the `encapsulatedLogic`'s parameters.

The `calledFunction` that is associated with the `Invocation` element SHALL BE the `encapsulatedLogic` of the `BusinessKnowledgeModel` element that is required by the `Decision` element that contains the `Invocation`. The `Invocation` element SHALL have exactly one binding for each parameter in the `BusinessKnowledgeModel`'s `encapsulatedLogic`.

`Invocation` inherits of all the attributes and model associations of `Expression`. Table 30 presents the additional attributes and model associations of the `Invocation` element.

Table 30: Invocation attributes and model associations

Attribute	Description
calledFunction: <code>Expression</code> [1]	An expression whose value is a function.
binding: <code>Binding</code> [*]	This attribute lists the instances of <code>Binding</code> used to bind the formalParameters of the <code>calledFunction</code> in this <code>Invocation</code> .

7.3.7 Binding metamodel

The class `Binding` is used to model, in an `Invocation` element, the binding of the `calledFunction`'s `formalParameters` to values.

A `Binding` is made of one `bindingFormula`, which is an `Expression`, and of one parameter, which is an `InformationItem`.

The parameter names in the `Binding` elements SHALL be a subset of the `formalParameters` of the `calledFunction`.

When the `Invocation` element is executed, each `InformationItem` element that is referenced as a parameter by a binding in the `Invocation` element is assigned, at runtime, the value of the `bindingFormula`.

Table 31 presents the attributes and model associations of the `Binding` element.

Table 31: Binding attributes and model associations

Attribute	Description
parameter: <code>InformationItem</code>	The <code>InformationItem</code> on which the <code>calledFunction</code> of the owning instance of <code>Invocation</code> depends that is bound by this <code>Binding</code> .
bindingFormula: <code>Expression</code> [0..1]	The instance of <code>Expression</code> to which the parameter in this <code>Binding</code> is bound when the owning instance of <code>Invocation</code> is evaluated.

7.3.8 Error Handling

When the evaluation of a DMN expression (see section 7.2.1) encounters a semantic error (e.g. type mismatch or duplicate keys in a context), the evaluation MUST report or log diagnostic information and SHALL return **null**.

There are two modes for error handling in DMN: lenient and strict. The error handling mode is configured during the initiation of the decision model evaluation. The default error handling mode is lenient. A given DMN model can be evaluated in lenient or strict mode, i.e. the error handling mode is not a property of a DMN model.

In lenient error mode, if an error is detected, it is collected, and the execution continues. For instance, errors detected in child DRG Elements are accumulated for the parent element.

When the error mode is set to strict, the model evaluation halts upon detecting the first error, reports the error, and returns **null**.

The configured error handling mode also applies to the handling of errors during the evaluation of literal expressions (e.g. FEEL expressions).

This page intentionally left blank.

8 Decision Table

8.1 Introduction

One of the ways to express the decision logic corresponding to the DRD decision artifact is as a decision table. A decision table is a tabular representation of a set of related input and output expressions, organized into rules indicating which output entry applies to a specific set of input entries. The decision table contains all (and only) the inputs required to determine the output. Moreover, a complete table contains all possible combinations of input values (all the rules).

Decision tables and decision table hierarchies have a proven track record in decision logic representation. It is one of the purposes of **DMN** to standardize different forms and types of decision tables.

A decision table consists of:

- An information item name: the name of an InformationItem, if any, for which the decision table is its value expression. This will usually be the name of the Decision or Business Knowledge Model for which the decision table provides the decision logic.
- A list of input clauses (zero or more). Each input clause is made of an input expression and optional allowed values for the input entries that correspond to the clause. The input entries are contained in the rules, and the i^{th} input entry corresponds to the i^{th} input clause.
- A list of output clauses (one or more). Each output clause is made of a name and optional allowed values for the output entries that correspond to the clause. The output entries are contained in the rules, and the i^{th} output entry corresponds to the i^{th} output clause. A single output clause has no name. Two or more output clauses describe a decision table that returns a context for each hit with an entry for each output clause. Each of the multiple output clauses SHALL be named.
- A set of outputs (one or more). A single output has no name, only a value. Two or more outputs are called output components. Each output component SHALL be named. Each output (component) SHALL specify an output entry for each rule. The specification of output component name (if multiple outputs) and all output entries is referred to as an output clause.
- A list of annotation clauses (zero or more). Each annotation clause is made of a name. Each annotation SHALL be named as part of a rule annotation clause. The annotation entries are contained in the rules, and the i^{th} annotation entry corresponds to the i^{th} annotation clause.
- A list of rules (one or more) in rows or columns of the table (depending on orientation), where each rule is composed of the specific input entries, output entries and optional rule annotations of the table row (or column). If the rules are expressed as rows, the columns are clauses, and vice versa.

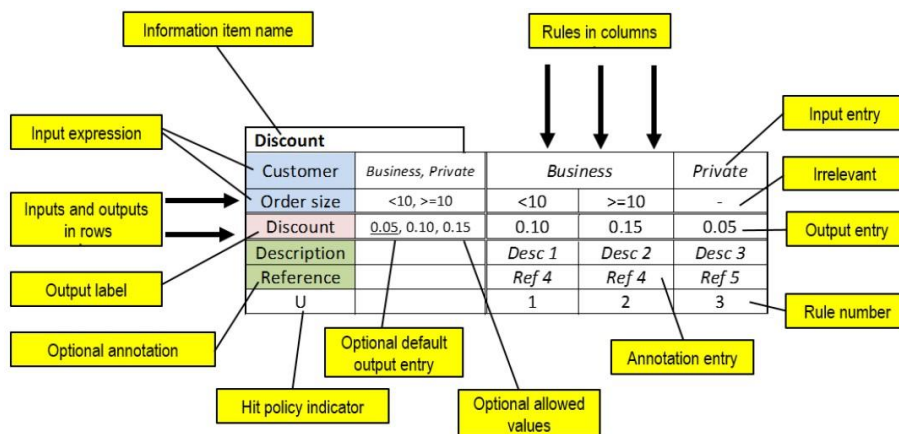


Figure 8-1: Decision table example (vertical orientations: rules as columns)

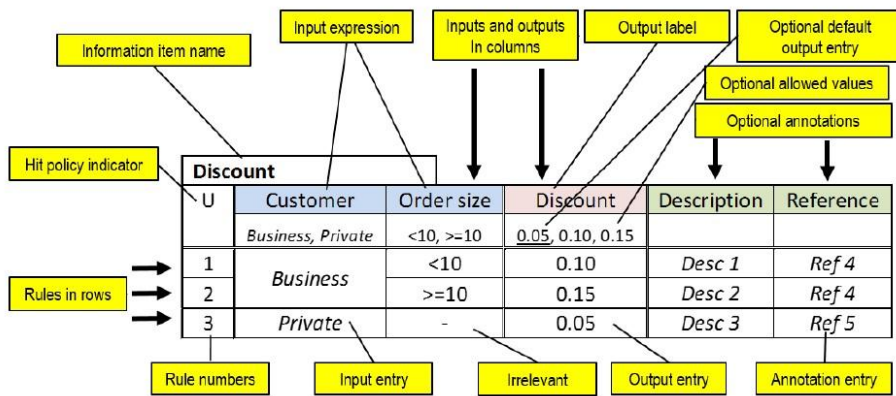


Figure 8-2: Decision table example (horizontal orientation: rules as rows)

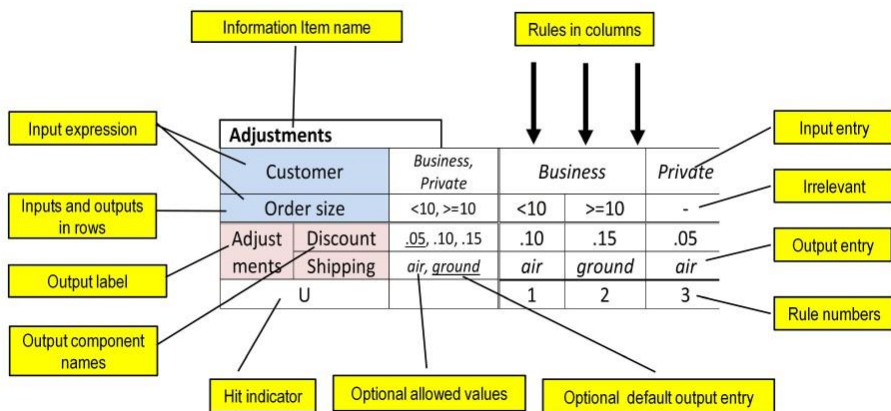


Figure 8-3: Decision table example (vertical orientation, multiple output components)

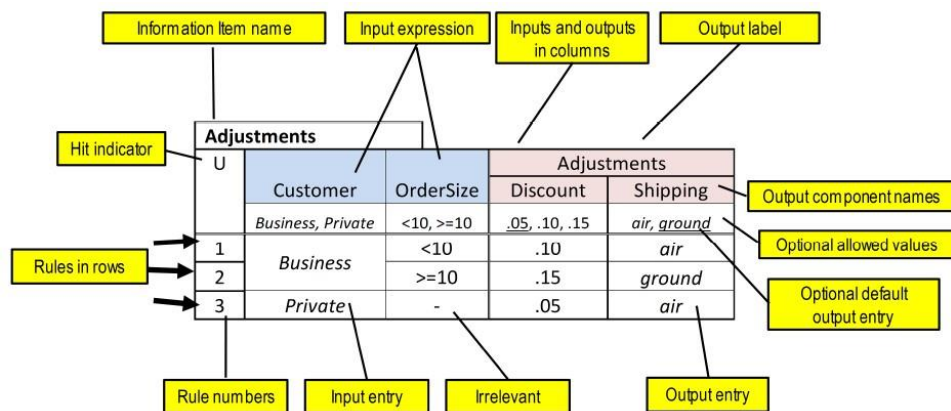


Figure 8-4: Decision table example (horizontal orientation, multiple output components)

The decision table shows the rules in a shorthand notation by arranging the entries in table cells. This shorthand notation shows all inputs in the same order in every rule and therefore has a number of readability and verification advantages.

For example:

Customer	OrderSize	Discount
Business	<10	0.10

reads as:

If Customer = "Business" **and** OrderSize < 10 **then** Discount = 0.10 In general, this is expressed as:

input expression 1	input expression 2	Output label
input entry a	input entry b	output entry c

The three highlighted cells in the decision table fragment above represent the following rule:

If the value of input expression 1 satisfies input entry a

and the value of input expression 2 satisfies input entry b

then the rule *matches* and the result of the decision table is output entry c.

An input expression value *satisfies* an input entry if the value is equal to the input entry or belongs to the list of values indicated by the input entry (e.g., a list or a range), or one of the expressions in the input entry evaluates to true. For the complete specification of the input entry satisfaction conditions, please refer to section 8.3.3. If the input entry is '-' (meaning *irrelevant*), every value of the input expression satisfies the input entry, and that particular input is irrelevant in the specified rule.

A rule *matches* if the value of every input expression satisfies the corresponding input entry. If there are no input entries, any rule matches.

The list of rules expresses the logic of the decision. For a given set of input values, the matching rule (or rules) indicate the resulting value for the output name. If rules *overlap*, multiple rules can match, and a *hit policy* indicates how to handle the multiple matches.

If two input entries of the same input expression share no values, the entries (cells) are called *disjoint*. If there is an intersection, the entries are called *overlapping* (or even equal). 'Irrelevant' ('-') overlaps with any input entry of the input expression.

Two rules are overlapping if all corresponding input entries are *overlapping*. A specific configuration of input data may then match the two rules.

Two rules are *disjoint* (non-overlapping) if at least one pair of corresponding input expressions is disjoint. No specific configuration of input data will match the two rules.

If tables are allowed to contain overlapping rules, the table hit policy indicates how overlapping rules have to be handled and which is the resulting value(s) for the output name, in order to avoid inconsistency.

8.2 Notation

This section builds on the generic notation for decision logic and boxed expressions defined in clause 7.2. A decision table representation standardizes:

- The orientation (rules as rows, columns, or crosstab), as shown by the table.
- Placement of inputs, outputs and (optional) allowed values in standard locations on a grid of cells. Each input expression is optionally associated with unary tests restricting the allowed input values. In this text the optional cells with allowed values are indicated in Each output (component) is optionally associated with allowed values. In this text the optional allowed output values are indicated in
- Line style and optional use of color.
- The contents of specific rule input and output entry cells.
- The hit policy, indicating how to interpret overlapping input combinations.
- Placement of information item name, hit policy (H) and rule numbers as indicated in Figure 8-5, Figure 8-7 and Figure 8-9. Rule numbers are consecutive natural numbers starting at 1. Rule numbering is required for tables with hit indicator F (first) or R (rule order) because the meaning depends on rule sequence. Crosstab tables have no rule numbers. Rule numbering is optional for other table types.

Input expressions, input values, output values, input entries and output entries can be any text (e.g., natural language, formal language, pseudo-code). Implementations claiming level 2 or 3 conformance SHALL support (S-)FEEL syntax. Implementations claiming level 1 conformance are not required to interpret the expressions. To avoid misinterpretation (e.g., when expressions are not meant to be valid (S-)FEEL but may conflict with the look and feel of (S-)FEEL syntax), conformant implementations SHOULD indicate when the input expression is not meant to be interpreted by using the URI: "https://www.omg.org/spec/DMN/uninterpreted/20140801".

8.2.1 Line style and color

Line style is normative. There is a double line between the input clauses and output clauses, continuing between the input entries and the output entries. There is also a double line between the output clauses and the annotation clauses, continuing between the output entries and the annotation entries. These two double lines are parallel to each other. There is a third double line, that intersects at right angles with the previous two, between input clauses and the input entries, continuing between the output clauses and the output entries, and continuing between the annotation clauses and the annotation entries. All other cells are separated by a single line.

Color is suggested but does not influence the meaning. It is considered good practice to use different colors for the input clauses, the output clauses, and the annotation clauses, and another (or no) color for the input, output, and annotation entries.

8.2.2 Table orientation

Depending on size, a decision table can be presented horizontally (rules as rows), vertically (rules as columns), or crosstab (rules composed from two input dimensions). Crosstab tables can only have the default hit policy (see later).

Decision table inputs and outputs should not be mixed. In a horizontal table, all input columns SHALL be represented on the left of all output columns. In a vertical table, all the input rows SHALL be represented above all output rows. In a crosstab, all the output cells SHALL be in the bottom-right part of the table.

The table SHALL be arranged in one of the following ways (see Figure 8-5, Figure 8-7, Figure 8-9). Cells indicated in **inverse** are optional.

The input cell entry ‘-’ means ‘irrelevant’. HC is a placeholder for hit policy indicator (e.g., U, A, F, ...).

information item name			
H	input expression 1	input expression 2	Output label
	input value 1a, input value 1b	input value 2a, input value 2b	output value 1a, output value 1b
1	input entry 1.1	input entry 2.1	output entry 1.1
2		input entry 2.2	output entry 1.2
3	input entry 1.2	-	output entry 1.3

Figure 8-5: Rules as rows - schematic layout

Discount				
U	Customer	OrderSize	Delivery	Discount
	Business, Private, Government	<10, >=10	sameday, slow	0, 0.05, 0.10, 0.15
1	Business	<10	-	0.05
2		>=10	-	0.10
3	Private	-	sameday	0
4		-	slow	0.05
5	Government	-	-	0.15

Figure 8-6: Rules as rows - example

information item name				
input expression 1	value 1a, value 1b	input entry 1.1		input entry 1.2
input expression 2	value 2a, value 2b	input entry 2.1	input entry 2.2	-
Output label	value 1a, value 1b	output entry 1.1	output entry 1.2	output entry 1.3
H		1	2	3

Figure 8-7: Rules as columns - schematic layout

Discount						
Customer	Business, Private, Government	Business		Private		Government
Ordersize	<10, >=10	<10	>=10	-		-
Delivery	sameday, slow	-	-	sameday	slow	-
Discount	0, 0.05, 0.10, 0.15	0.05	0.10	0	0.05	0.15
U		1	2	3	4	5

Figure 8-8: Rules as columns - example

information item name			
Output label		input expression 1	
		input entry 1.1	input entry 1.2
input expression 2	input entry 2.1	output entry 1.1	output entry 1.3
	input entry 2.2	output entry 1.2	output entry 1.4

Figure 8-9: Rules as crosstab - schematic layout (optional input and output values not shown)

Discount				
Discount		Customer		
		Business	Private	Government
Ordersize	<10	0.05	0	0.15
	>=10	0.10	0	0.15

Figure 8-10: Rules as crosstab - simplified example with only two inputs

Discount					
Discount		Customer, Delivery			
		Business	Private		Government
		-	sameday	slow	-
Ordersize	<10	0.05	0	0.05	0.15
	>=10	0.10	0	0.05	0.15

Figure 8-11: Rules as crosstab - example with three inputs

Crosstab tables with more than two inputs are possible (as shown in Figure 8-11).

8.2.3 Input expressions

Input expressions are usually simple, for example, a name (e.g., CustomerStatus) or a test (e.g., Age<25). The order of input expressions is not related to any execution order in implementation.

8.2.4 Input values

Input expressions may be expected to result in a limited number or a limited range of values. It is important to model these expected input values because a decision table will be considered complete if its rules cover all combinations of expected input values for all input expressions.

Regardless of how the expected input values are modeled, input values SHOULD be exclusive and complete. Exclusive means that input values are disjoint. Complete means that all relevant input values from the domain are present.

For example, the following two input value ranges overlap: <5, <10. The following two ranges are incomplete: <5, >5. The list of input values is optional. If provided, it is a list of unary tests that must be satisfied by the corresponding input.

8.2.5 Information Item names, output labels, and output component names

A decision table with multiple output components SHALL specify a name for each output component.

A decision table that is the value expression of an InformationItem (e.g., a Decision's logic or a boxed Invocation's binding formula) SHALL specify the name of the InformationItem as its Information Item name. A decision table that is not contained in another boxed expression shall place the Information Item name in a name box just above and adjoining the table.

A decision table that is contained in another boxed expression may use the containing expression for its Information Item name. For example, the Information Item name for a decision table bound to a function parameter is the name of the function parameter. Or, to save space, the Information Item name box may be omitted, and the Output label used instead.

8.2.6 Output values

The output entries of a decision table are often drawn from a list of output values.

The list of output values is optional. If provided, it is a list restricting output entries to the given list of values.

When the hit policy is P (priority), meaning that multiple rules can match, but only one hit should be returned, the ordering of the list of output values is used to specify the (decreasing) priority.

The ordering of the list of output values is also used when the hit policy is output order.

8.2.7 Multiple outputs

The decision table can show a compound output (see Figure 8-12, Figure 8-13, and Figure 8-14).

information item name				
H	input expression 1	input expression 2	output label	
	input value 1a, input value 1b	input value 2a, input value 2b	output component 1	output component 2
			output value 1a, output value 1b	output value 2a, output value 2b
1	input entry 1a	input entry 2a	output entry 1.1	output entry 2.1
2		input entry 2b	output entry 1.2	output entry 2.2
3	input entry 1b	-	output entry 1.3	output entry 2.3

Figure 8-12: Horizontal table with multiple output components

information item name					
input expression 1		input value 1a, input value 1b	input entry 1a		input entry 1b
input expression 2		input value 2a, input value 2b	input entry 2a	input entry 2b	-
output label	output component 1	output value 1a, output value 1b	output entry 1.1	output entry 1.2	output entry 1.3
	output component 2	output value 2a, output value 2b	output entry 2.1	output entry 2.2	output entry 2.3
H			1	2	3

Figure 8-13: Vertical table with multiple output components

information item name			
output label		input expression 1	
output component 1, output component 2		input entry 1a	input entry 1b
input expression 2	input entry 2a	output entry 1.1, output entry 2.1	output entry 1.3 output entry 2.3
	input entry 2b	output entry 1.2, output entry 2.2	output entry 1.4, output entry 2.4

Figure 8-14: Crosstab with multiple output components

8.2.8 Input entries

Rule input entries are unary tests (grammar rule 15).

A dash symbol ('-') can be used to mean any input value, *i.e.*, the input is irrelevant for the containing rule.

The input entries in a unary test SHOULD be '-' or a subset of the input values specified. For example, if the input values for input 'Age' are specified as $[0..120]$, then an input entry of <0 SHOULD be reported as invalid.

Tables containing at least one '-' input entry are called *contracted* tables. The others are called *expanded*.

Tables where every input entry is *true*, *false*, or '-' are historically called *limited-entry* tables, but there is no need to maintain this restriction.

Evaluation of the input expressions in a decision table does not produce side-effects that influence the evaluation of other input expressions. This means that evaluating an expression or executing a rule should not change the evaluation of other expressions or rules of the same table. This is particularly important in first hit tables where the rules are evaluated in a predefined sequence: evaluating or executing a rule should not influence other rules.

8.2.9 Merged input entry cells

Adjacent input entry cells from different rules, with the same content and same (or no) prior cells can be merged, as shown in Figure 8-15 and Figure 8-16. Rule output cells cannot be merged (except in crosstabs).

information item name			
H	input expression 1	input expression 2	Output label
	input value 1a, input value 1b	input value 2a, input value 2b	output value 1a, output value 1b
1	input entry 1a	input entry 2a	output entry 1.1
2		input entry 2b	output entry 1.2
3	input entry 1b	-	output entry 1.3

Figure 8-15: Merged rule input cells allowed

information item name			
H	input expression 1	input expression 2	Output label
	input value 1a, input value 1b	input value 2a, input value 2b	output value 1a, output value 1b
1	input entry 1a	input entry 2a	output entry 1.1
2		input entry 2b	output entry 1.2
3	input entry 1b	input entry 2b	output entry 1.3
4		input entry 2a	output entry 1.4

Figure 8-16: Merged rule input cells not allowed

8.2.10 Output entry

A rule output entry is an expression.

Rule output cells cannot be merged (except in crosstabs, where adjacent output cells with the same content can be merged).

8.2.10.1 Shorthand notation

In vertical (rules as columns) tables with a single output name (equal to the information item name), a shorthand notation may be used to indicate: output value applies ('X') or does not apply ('-'), as is common practice in decision tables.

Because there can be only one output entry for an output name, every rule must indicate no more than one 'X'. The other output entries must contain '-'.

The table in Figure 8-17 is shorthand notation for the table in Figure 8-18. It is called shorthand, because the output entries need not be (re-)written in every column but are indicated with a one-character notation ('X' or '-'), thereby saving space in vertical tables, which tend to expand in width as the number of rules increases. The output values are written only once, before the rules, in the output expression part.

If an information item name is provided, and there is only one output name (which has to be equal to the information item name), the output name is optional.

Applicant Risk Rating					
Applicant Age	< 25		[25..60]	> 60	
Medical History	good	bad	-	good	bad
Low	X	-	-	-	-
Medium	-	X	X	X	-
High	-	-	-	-	X
U	1	2	3	4	5

Figure 8-17: Shorthand notation for vertical tables (rules as columns)

Applicant Risk Rating					
Applicant Age	< 25		[25..60]	> 60	
Medical History	good	bad	-	good	bad
Applicant Risk Rating	Low	Medium	Medium	Medium	High
U	1	2	3	4	5

Figure 8-18: Full notation for vertical tables (rules as columns)

8.2.11 Hit policy

A decision table normally has several rules. As a default, rules do not overlap. If rules overlap, meaning that more than one rule may match a given set of input values, the hit policy indicator is required in order to recognize the table type and unambiguously understand the decision logic. The hit policy can be used to check correctness at design-time.

The hit policy specifies what the result of the decision table is in cases of overlapping rules, i.e., when more than one rule matches the input data. For clarity, the hit policy is summarized using a single character in a particular decision table cell. In horizontal tables this is the top-left cell (Figure 8-2) and in vertical tables this is the bottom-left cell (Figure 8-1).

The character is the initial letter of the defined hit policy (Unique, Any, Priority, First, Collect, Output order or Rule order). Crosstab tables are always Unique and need no indicator.

The hit policy SHALL default to Unique, in which case the hit indicator is optional. Decision tables with the Unique hit policy SHALL NOT contain overlapping rules.

Tools may support only a nonempty subset of hit policies, but the table type SHALL be clear and therefore the hit policy indication is mandatory, except for the default unique tables. Unique tables SHALL always be supported.

8.2.11.1 Single and multiple hit tables

A single hit table shall return the output of one rule only; a multiple hit table may return the output of multiple rules (or a function of the outputs, e.g., sum of values). If rules are allowed to overlap, the hit policy indicates how overlapping rules have to be interpreted.

The initial letter for hit policy also identifies if a table is single hit or multiple hits.

A single hit table may or may not contain overlapping rules but returns the output of one rule only. In case of overlapping rules, the hit policy indicates which of the matching rules to select. Some restrictions apply to tables with compound outputs.

Regardless of whether a single or multiple hit policy is used, some columns in a decision table may be designated as *rule annotations*. Rule Annotations contain text that is not returned as part of the expression results, and they are ignored for purposes of the hit policy validations described below. Although there is no standard mechanism to access the annotations of the matched rules in a decision table at execution time, implementations may use the annotations for auditing, debugging, logging, documentation, analytics, consumption by down-stream systems, or for other purposes.

Single hit policies for single output decision tables are:

1. **Unique:** no overlap is possible, and all rules are disjoint. Only a single rule can be matched. This is the default.
2. **Any:** there may be overlap, but all the matching rules show equal output entries for each output (ignoring rule annotations), so any match can be used. If the output entries are non-equal (ignoring rule annotations), the hit policy is incorrect, and the result is undefined.
3. **Priority:** multiple rules can match, with different output entries. This policy returns the matching rule with the highest output priority. Output priorities are specified in the ordered list of output values, in decreasing order of priority. Note that priorities are independent from rule sequence.
4. **First:** multiple (overlapping) rules can match, with different output entries. The first hit by rule order is returned (and evaluation can halt). This is still a common usage because it resolves inconsistencies by forcing the first hit. However, first hit tables are not considered good practice because they do not offer a clear overview of the decision logic. It is important to distinguish this type of table from others because the meaning depends on the order of the rules. The last rule is often the catch-remainder. Because of this order, the table is hard to validate manually and therefore has to be used with care.

A multiple hit table may return output entries from multiple rules. The result will be a list of rule outputs or a simple function of the outputs.

Multiple hit policies for single output decision tables can be:

5. **Output order:** returns all hits in decreasing output priority order. Output priorities are specified in the ordered list of output values in decreasing order of priority.
6. **Rule order:** returns all hits in rule order. Note: the meaning may depend on the sequence of the rules.
7. **Collect:** returns either all hits in arbitrary order, or the result of applying a simple function to them. An operator ('+', '<', '>', '#') can be added. If no operator is present, the result is the list of the output entries of all the rules matched.

If an operator is present, the result is a singleton value resulting from applying the function denoted by the selected operator to the list of the output entries of all the rules matched. Collect operators are:

- a) + (sum): the result of the decision table is the sum of all the outputs.
- b) < (min): the result of the decision table is the smallest value of all the outputs.
- c) > (max): the result of the decision table is the largest value of all the outputs.
- d) # (count): the result of the decision table is the number of outputs.

Other policies, such as more complex manipulations on the outputs, can be performed by post-processing the output list (outside the decision table).

Decision tables with compound outputs support only the following hit policies: Unique, Any, Priority, First, Output order, Rule order and Collect without operator, because the collect operator is undefined over multiple outputs. This restriction ignores rule annotations of which there may be multiple regardless of the hit policy specified.

For the Priority and Output order hit policies, priority is decided in compound output tables over all the outputs for which output values have been provided (ignoring rule annotations). The priority for each output is specified in the ordered list of output values in decreasing order of priority, and the overall priority is established by considering the ordered outputs from left to right in horizontal tables (i.e., columns to the left take precedence over columns to the right), or from top to bottom in vertical tables. Outputs for which no output values are provided are not considered in the ordering, although their output entries are included in the ordered compound output.

So, for example, if called with Age = 17, Risk category = "HIGH" and Debt review = true, the Routing rules table in Figure 8-19 would return the outputs of all four rules, in the order 2, 4, 3, 1.

Routing rules						
O	Age	Risk category	Debt review	Routing	Review level	Reason
		LOW, MEDIUM, HIGH		DECLINE, REFER, ACCEPT	LEVEL 2, LEVEL 1, NONE	
1	-	-	-	ACCEPT	NONE	Acceptable
2	< 18	-	-	DECLINE	NONE	Applicant too young
3	-	HIGH	-	REFER	LEVEL 1	High risk application
4	-	-	true	REFER	LEVEL 2	Applicant under debt review

Figure 8-19: Output order with compound output

Note 1

Crosstab tables are unique and complete by definition and therefore do not need a hit policy.

Note 2

The sequence of the rules in a decision table does not influence the meaning, except in First tables (single hit) and Rule order tables (multiple hit). These tables should be used with care.

8.2.12 Default output values

Tables may specify a default output. The default value is underlined in the list of output values.

8.3 Metamodel

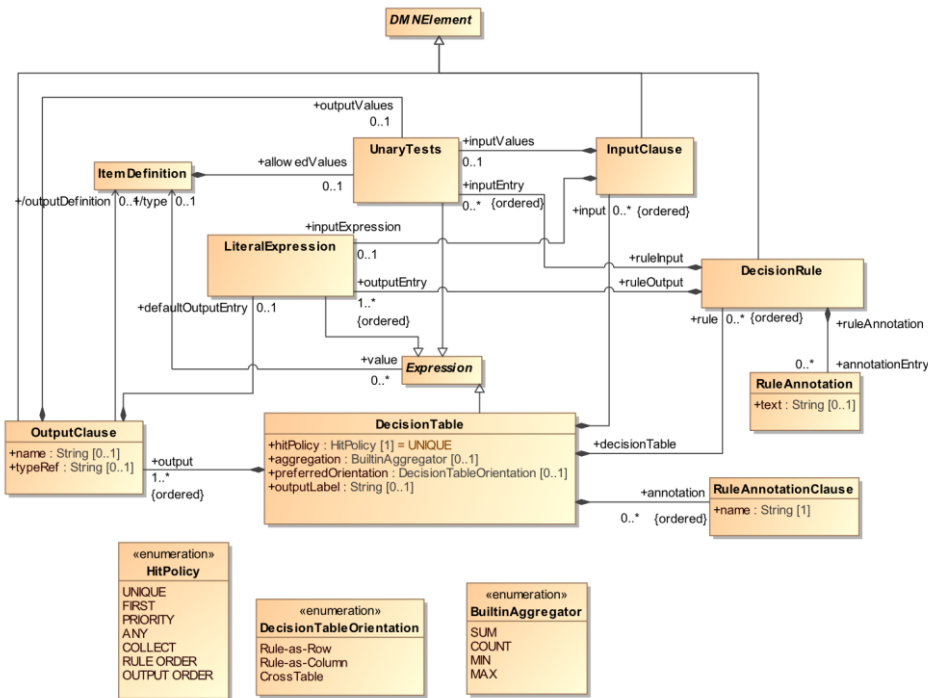


Figure 8-20: DecisionTable class diagram

8.3.1 Decision Table metamodel

The class `DecisionTable` is used to model a decision table.

`DecisionTable` is a concrete specialization of `Expression`.

An instance of `DecisionTable` contains a list of rules which are instances of `DecisionRule`, a list of inputs which are instances of `InputClause`, a list of outputs which are instances of `OutputClause`, and a list of annotations which are instances of `RuleAnnotationClause`.

It has a `preferredOrientation`, which SHALL be one of the enumerated `DecisionTableOrientation`: `Rule-as-Row`, `Rule-as-Column` or `CrossTable`. An instance of `DecisionTable` SHOULD BE represented as specified by its `preferredOrientation`, as defined in clause 8.2.2.

An instance of `DecisionTable` has an associated `hitPolicy`, which SHALL be one of the enumerated `HitPolicy`: `UNIQUE`, `FIRST`, `PRIORITY`, `ANY`, `COLLECT`, `RULE ORDER`, `OUTPUT ORDER`. The default value for the `hitPolicy` attribute is: `UNIQUE`. In the diagrammatic representation of an instance of `DecisionTable`, the `hitPolicy` is represented as specified in clause 8.2.11.

The semantics that is associated with an instance of `DecisionTable` depends on its associated `hitPolicy`, as specified below and in clause 8.2.11. The `hitPolicy` attribute of an instance of `DecisionTable` is represented as specified in clause 8.2.11.

If the `hitPolicy` associated with an instance of `DecisionTable` is `FIRST` or `RULE ORDER`, the rules that are associated with the `DecisionTable` SHALL be ordered. The ordering is represented by the explicit numbering of the rules in the diagrammatic representation of the `DecisionTable`.

If the `hitPolicy` associated with an instance of `DecisionTable` is `PRIORITY` or `OUTPUT ORDER`, the `outputValue s` determine the result as specified in clause 8.2.11.

If the `hitPolicy` that is associated with an instance of `DecisionTable` is `COLLECT`, the `DecisionTable` MAY have an associated aggregation, which is one of the enumerated `BuiltinAggregator` (see clause 8.2.11).

As a kind of `Expression`, an instance of `DecisionTable` has a value, which depends on the outputs of the associated rules, the associated `hitPolicy` and the associated `aggregation`, if any. The value of an instance of `DecisionTable` is determined according to the specification in clause 10.3.2.10.

`DecisionTable` inherits all the attributes and model associations from `Expression`. Table 32 presents the additional attributes and model associations of the `DecisionTable` element.

Commented [AF15]: DMN17-195

Table 32: `DecisionTable` attributes and model associations

Attribute	Description
input: <code>InputClause</code> [*]	This attributes lists the instances of <code>InputClause</code> that compose this <code>DecisionTable</code> .
output: <code>OutputClause</code> [*]	This attributes lists the instances of <code>OutputClause</code> that compose this <code>DecisionTable</code> .
annotation: <code>RuleAnnotationClause</code> [*]	This attribute lists the instances of <code>RuleAnnotationClause</code> that compose this <code>DecisionTable</code> .
rule: <code>DecisionRule</code> [*]	This attributes lists the instances of <code>DecisionRule</code> that compose this <code>DecisionTable</code> .
hitPolicy: <code>HitPolicy</code>	The hit policy that determines the semantics of this <code>DecisionTable</code> . Default is: <code>UNIQUE</code> .
aggregation: <code>BuiltinAggregator</code>	If present, this attribute specifies the aggregation function to be applied to the unordered set of values of the applicable rules to determine the value of this <code>DecisionTable</code> when the <code>hitPolicy</code> is <code>COLLECT</code> .
preferredOrientation: <code>DecisionTableOrientation</code> [0.. 1]	The preferred orientation for the diagrammatic representation of this <code>DecisionTable</code> . This <code>DecisionTable</code> SHOULD BE represented as specified by this attribute.
outputLabel: <code>string</code> [0..1]	This attribute gives a description of the decision table output and is often the same as the name of the <code>InformationItem</code> for which the decision table is the value expression.

8.3.2 Decision Table Input and Output metamodel

In a `DecisionTable`, an input specifies an `inputExpression` (the subject) and a number of `inputEntries`. An output specifies the name and the domain of definition of an output value, a number of `outputEntries`.

The class `InputClause` is used to model a decision table input, and the class `OutputClause` is used to model a decision table output, and the class `RuleAnnotationClause` is used to model a decision table annotation.

An instance of `InputClause` is made of an optional `inputExpression` and an ordered list of `inputEntry`, which are instances of `UnaryTests`. An instance of `OutputClause` optionally references a `typeRef`, specifying its datatype, and it is made of an ordered list of `outputEntry`, which are instances of `LiteralExpression`, and an optional `defaultOutputEntry`, which is also an instance of `LiteralExpression`. An instance of `RuleAnnotationClause` contains a name of type `String`.

When a `DecisionTable` contains more than one `OutputClause`, each `OutputClause` SHALL have a name. When a `DecisionTable` has a single `OutputClause`, the `OutputClause` SHALL NOT have a name. A `RuleAnnotationClause` SHALL have a name.

Table 33, Table 34 and Table 35 present the attributes and model associations of `InputClause`, `OutputClause` and `RuleAnnotationClause` respectively.

Table 33: InputClause attributes and model associations

Attribute	Description
inputExpression: <code>Expression</code> [0..1]	The subject of this <code>InputClause</code> .
inputValues: <code>UnaryTests</code> [0..1]	This attribute contains <code>UnaryTests</code> that constrain the result of the <code>inputExpression</code> of this <code>InputClause</code> .

Table 34: OutputClause attributes and model associations

Attribute	Description
typeRef: <code>String</code> [1]	The <code>OutputClause</code> of a single output decision table SHALL NOT specify a <code>typeRef</code> . <code>OutputClauses</code> of a multiple output decision table MAY specify a <code>typeRef</code> . A <code>typeRef</code> is the name of the datatype of the output, either an <code>ItemDefinition</code> , a base type in the specified <code>expressionLanguage</code> , or an imported type.
name: <code>string</code> [0..1]	The <code>OutputClause</code> of a single output decision table SHALL NOT specify a name. <code>OutputClauses</code> of a multiple output decision table SHALL specify a name.
outputValues: <code>UnaryTests</code> [0..1]	This attribute contains <code>UnaryTests</code> that constrain the result of the <code>outputEntries</code> of the <code>DecisionRules</code> corresponding to this <code>OutputClause</code> .
defaultOutputEntry: <code>LiteralExpression</code> [0..1]	In an Incomplete table, this attribute lists an instance of <code>LiteralExpression</code> that is selected when no rules match for the decision table.

Commented [AF16]: DMN17-110

Table 35: RuleAnnotationClause attributes and model associations

Attribute	Description
name: <code>string</code> [1]	<code>RuleAnnotationClause</code> SHALL specify a name that is used as the name of the rule annotation column of the containing decision table.

8.3.3 Decision Rule metamodel

The class `DecisionRule` is used to model the rules in a decision table (see 8.2).

An instance of `DecisionRule` has an ordered list of `inputEntry` instances which are instances of `UnaryTests`, an ordered list of `outputEntry` instances, which are instances of `LiteralExpression`, and an ordered list of `ruleAnnotations`.

In a tabular representation of the containing instance of `DecisionTable`, the representation of an instance of `DecisionRule` depends on the orientation of the decision table. For instance, if the decision table is represented horizontally (rules as row, see 8.2.2), instances of `DecisionRule` are represented as rows, with all the `inputEntry`s represented on the left of all the `outputEntry`s, and all the `ruleAnnotations` represented to their right.

By definition, a `DecisionRule` element that has no `inputEntry`s is always applicable. Otherwise, an instance of `DecisionRule` is said to be *applicable* if and only if, all of the `DecisionTable`'s `inputExpression` values satisfy their corresponding `inputEntry`.

The `inputEntry`s are matched in arbitrary order.

The `inputEntry` elements SHALL be in the same order as the containing `DecisionTable`'s inputs.

The i^{th} `inputExpression` must satisfy the i^{th} `inputEntry` for all `inputEntry`s in order for the `DecisionRule` to *match*, as defined in section 8.1.

The `outputEntry` elements SHALL be in the same order as the containing `DecisionTable`'s outputs. The i^{th} `outputEntry` SHALL be consistent with the `typeRef` of the i^{th} `OutputClause`.

The `ruleAnnotation` elements SHALL be in the same order as the containing `DecisionTable`'s annotations. The i^{th} `ruleAnnotation` refers to the i^{th} `RuleAnnotationClause`.

Table 36 presents the attributes and model associations of the `DecisionRule` element; Table 36 presents the attributes and model associations of the `RuleAnnotation` element.

Table 36: `DecisionRule` attributes and model associations

Attribute	Description
inputEntry: <code>UnaryTests</code> [0..*]	The instances of <code>UnaryTests</code> that specify the input conditions that this <code>DecisionRule</code> must match for the corresponding (by index) <code>inputExpression</code> .
outputEntry: <code>LiteralExpression</code> [1..*]	A list of the instances of <code>LiteralExpression</code> that compose the output components of this <code>DecisionRule</code> .
annotationEntry: <code>RuleAnnotation</code> [0..*]	A list of the instances of <code>RuleAnnotation</code> that compose the annotations of this <code>DecisionRule</code> and match the corresponding (by index) instances of <code>RuleAnnotationClause</code> .

Table 37: `RuleAnnotation` attributes and model associations

Attribute	Description
text: <code>string</code> [0..1]	The text of the <code>RuleAnnotation</code>

8.4 Examples

Table 38 provides examples for the various types of decision table discussed in this section. Further examples may be found in clause 12.1.4, in the context of a complete example of a **DMN** decision model.

Table 38: Examples of decision tables

Single Hit
Unique

Table 30: Examples of decision tables

Applicant Risk Rating			
U	Applicant Age	Medical History	Applicant Risk Rating
1	> 60	good	Medium
2		bad	High
3	[25..60]	-	Medium
4	< 25	good	Low
5		bad	Medium

Applicant Risk Rating						
Applicant Age		< 25		[25..60]	> 60	
Medical History		good	bad	-	good	bad
Applicant Risk Rating		Low	Medium	Medium	Medium	High
U		1	2	3	4	5

Applicant Risk Rating						
Applicant Age		< 25		[25..60]	> 60	
Medical History		good	bad	-	good	bad
Low		X	-	-	-	-
Medium			X	X	X	
High						X
U		1	2	3	4	5

Single Hit
Any

Person Loan Compliance				
A	Persons Credit Rating from Bureau	Person Credit Card Balance	Person Education Loan Balance	Person Loan Compliance
1	A	< 10000	< 50000	Compliant
2	Not(A)	-	-	Not Compliant
3	-	>= 10000	-	Not Compliant
4	-	-	>= 50000	Not Compliant

Single Hit
Priority

Applicant Risk Rating			
P	Applicant Age	Medical History	Applicant Risk Rating
			High, Medium, Low
1	>= 25	good	Medium
2	> 60	bad	High
3	-	bad	Medium
4	< 25	good	Low

Single Hit

First

Special Discount

F	Type of Order	Customer Location	Type of Customer	Special Discount %
1	Web	US	Wholesaler	10
2	Phone	-	-	0
3	-	Non-US	-	0
4	-	-	Retailer	5

Multiple Hit

No order

Special Discount

Type of Order	Web		-
Customer Location	US		-
Type of Customer	Wholesaler	Retailer	-
Special Discount %	10	5	0
F	1	2	3

Multiple Hit

Output order

Holidays

O	Age	Years of Service	Holidays
			22, 5, 3, 2
1	-	-	22
2	>= 60	-	3
3	-	>= 30	3
4	< 18	-	5
5	>= 60	-	5
6	-	>= 30	5
7	[18..60)	[15..30)	2
8	[45..60)	< 30	2

Multiple Hit

Rule order

Student Financial Package Eligibility

R	Student GPA	Student Extra-Curricular Activities Count	Student National Honor Society Membership	Student Financial Package Eligibility List
1	> 3.5	>= 4	Yes	20% Scholarship
2	> 3.0	-	Yes	30% Loan
3	> 3.0	>= 2	No	20% Work-On-Campus
4	<= 3.0	-	-	5% Work-On-Campus

This page intentionally left blank.

9 Simple Expression Language (S-FEEL)

9.1 Introduction

DMN defines the friendly enough expression language (FEEL) for the purpose of giving standard executable semantics to many kinds of expressions in decision model (see 10).

This section defines a simple subset of FEEL, S-FEEL, for the purpose of giving standard executable semantics to decision models that use only simple expressions: in particular, decision models where the decision logic is modeled mostly or only using decision tables.

Experience with DMN since its release has shown that few if any complete decision models can be defined using S-FEEL. Individual decision tables can be defined using only S-FEEL but within a decision model there is generally at least one decision that requires FEEL. Developers and users are therefore encouraged to use and implement the full FEEL specification rather than the S-FEEL subset.

9.2 S-FEEL syntax

The syntax for the S-FEEL expressions used in this section is specified in the EBNF below: it is a subset of the FEEL syntax specified in clause 10.3.1.2.

Grammar rules:

1. expression = simple expression ;
2. arithmetic expression =
 - 2.a addition | subtraction |
 - 2.b multiplication | division |
 - 2.c exponentiation |
 - 2.d arithmetic negation ;
- 3 simple expression = arithmetic expression | simple value | comparison ;
- 4 simple expressions = simple expression , { ",", simple expression } ;
- 5 simple positive unary test =
 - 5.a ["<" | "<=" | ">" | ">="] , endpoint |
 - 5.b interval ;
- 6 interval = (open interval start | closed interval start) , endpoint , ".." , endpoint , (open interval end | closed interval end) ;
- 7 open interval start = "(" | "[" ;
- 8 closed interval start = "[" ;
- 9 open interval end = ")" | "]" ;
- 10 closed interval end = "]" ;
- 11 simple positive unary tests = simple positive unary test , { ",", simple positive unary test } ;

12 simple unary tests =

12.a simple positive unary tests |

12.b "not", "(", simple positive unary tests, ")" |

12.c "-";

13 endpoint = simple value ;

14 simple value = qualified name | simple literal ;

15 qualified name = name , { ".", name } ;

16 addition = expression , "+", expression ;

17 subtraction = expression , "-", expression ;

18 multiplication = expression , "*", expression ;

19 division = expression , "/", expression ;

20 exponentiation = expression , "**", expression ;

21 arithmetic negation = "-", expression ;

22 name = name start , { name part | additional name symbols } ;

23 name start = name start char , { name part char } ;

24 name part = name part char , { name part char } ;

25 name start char = "?" | [A-Z] | " " | [a-z] | [\u00-\u00D6] | [\u00D8-\u00F6] | [\u00F8-\u02FF] | [\u0370-\u037D] | [\u037F\u01FFF] | [\u0200C-\u0200D] | [\u02070-\u0218F] | [\u02C00-\u02FEF] | [\u03001-\u0D7FF] | [\u0F900-\u0FDCF] | [\u0FDF0-\u0FFFD] | [\u01 0000-\u0EFFFF] ;

26 name part char = name start char | digit | \u00B7 | [\u00300-\u0036F] | [\u00203F-\u002040] ;

27 additional name symbols = "." | "/" | "-" | "" | "+" | "*" ;

28 simple literal = numeric literal | string literal | boolean literal | date time literal ;

29 string literal = """, { character – ("" | vertical space) | string escape sequence}, """;

30 boolean literal = "true" | "false" ;

31 numeric literal = ["-"] , (digits , [".", digits] | ".", digits) ;

32 digit = [0-9] ;

33 digits = digit , {digit} ;

34 date time literal = ("date" | "time" | "duration" | "days and time duration" | "years and months duration") ,
"(", string literal , ")" ; date time literal = ("date" | "time" | "duration") , "(", string literal , ")" ;

35 comparison = expression , ("=" | "!=" | "<" | "<=" | ">" | ">=") , expression ;

36 white space = vertical space | \u0009 | \u0020 | \u0085 | \u00A0 | \u00680 | \u0080E | [\u02000-\u0200B] |
\u02028 | \u02029 | \u0202F | \u0205F | \u03000 | \u0FEFF ;

37 vertical space = [\u0000A-\u0000D];

Commented [AF17]: DMN17-72

[3.3.7](#)string escape sequence = "\\" | "\"" | "\\n" | "\\r" | "\\t" | "\\u", hex digit, hex digit, hex digit, hex digit;

9.3 S-FEEL data types

S-FEEL supports all FEEL data types: *number*, *string*, *boolean*, *days and time duration*, *years and months duration*, *time*, and *date*, although with a simplified definition for some of them.

S-FEEL *number* has the same literal and values spaces as the XML Schema decimal datatype. Implementations are allowed to limit precision to 34 decimal digits and to round toward the nearest neighbor with ties favoring the even neighbor. Notice that “*precision is not reflected in this value space: the number 2.0 is not distinct from the number 2.00*” [XML Schema]. Notice, also, that this value space is totally ordered. The definition of S-FEEL *number* is a simplification over the definition of FEEL *number*.

S-FEEL supports FEEL *string* and FEEL *Boolean*: FEEL *string* has the same literal and values spaces as the Java String and XML Schema string datatypes. FEEL *boolean* has the same literal and values spaces as the Java Boolean and XML Schema Boolean datatypes.

S-FEEL supports the FEEL *time* data type. The lexical and value spaces of FEEL *time* are the literal and value spaces of the XML Schema time datatype. Notice that “*since the lexical representation allows an optional time zone indicator, time values are partially ordered because it may not be able to determine the order of two values one of which has a time zone and the other does not. Pairs of time values with or without time zone indicators are totally ordered*” [XSD].

S-FEEL does not support FEEL date and time. However, it supports the *date* type, which is like FEEL *date and time* with hour, minute, and second required to be absent. The lexical and value spaces of FEEL *date* are the literal and value spaces of the XML Schema date datatype.

S-FEEL supports the FEEL *days and time duration* and *years and months duration* datatypes. FEEL *days and time duration* and *years and months duration* have the same literal and value spaces as the XPath Data Model `dayTimeDuration` and `yearMonthDuration` datatypes, respectively. That is, FEEL *days and time duration* is derived from the XML Schema duration datatype by restricting its lexical representation to contain only the days, hours, minutes, and seconds components, and FEEL *years and months duration* is derived from the XML Schema duration datatype by restricting its lexical representation to contain only the year and month components.

The FEEL data types are specified in detail in clause 10.3.2.2.

9.4 S-FEEL semantics

S-FEEL contains only a limited set of basic features that are common to most expression and programming languages, and on the semantics of which most expression and programming languages agree.

The semantics of S-FEEL expressions are defined in this section, in terms of the semantics of the XML Schema datatypes and the XQuery [3.14.0](#) and XPath [3.12.0](#) Data Model datatypes, and in terms of the corresponding functions and operators defined by XQuery [3.14.0](#) and XPath [3.12.0](#) Functions and Operators (prefixed by “op:” below). A complete standalone specification of the semantics is to be found in clause 10.3.2, as part of the definition of FEEL. Within the scope of S-FEEL, the two definitions are equivalent and equally normative.

Arithmetic addition and subtraction (grammar rule 2.a) have the same semantics as:

- op:numeric-add and op:numeric-subtract, when its two operands are numbers.
- op:add-yearMonthDurations and op:subtract-yearMonthDurations, when the two operands are years and months durations.
- op:add-dayTimeDuration and subtract:dayTimeDurations, when the two operands are days and time durations.
- op:add-yearMonthDuration-to-date and op:subtract-yearMonthDuration-from-date, when the first operand is a years and months duration, and the second operand is a date.
- op:add-dayTimeDuration-to-date and op:subtract-dayTimeDuration-from-date, when the first operand is a days and time duration, and the second operand is a date.
- op:add-dayTimeDuration-to-time and op:subtract-dayTimeDuration-from-time, when the first operand is a days and time duration, and the second operand is a time.

In addition, arithmetic subtraction has the semantics of op:subtract-dates or op:subtract-times, when the two operands are dates or times, respectively.

Commented [AF18]: DMN17-73

Arithmetic addition and subtraction are not defined in other cases.

Arithmetic multiplication and division (grammar rule 2.b) have the same semantics as defined for `op:numeric-multiply` and `op:numeric-divide`, respectively, when the two operands are numbers. They are not defined otherwise.

Arithmetic exponentiation (grammar rule 2.c) is defined as the result of raising the first operand to the power of the second operand, when the two operands are numbers. It is not defined in other cases.

Arithmetic negation (grammar rule 2.d) is defined only when its operand is a number: in that case, its semantics is according to the specification of `op:numeric-unary-minus`.

Comparison operators (grammar rule 35) between numbers are defined according to the specification of `op:numeric-equal`, `op:numeric-less-than` and `op:numeric-greater-than`; comparisons between dates are defined according to the specification of `op:date-equal`, `op:date-less-than` and `op:date-greater-than`; comparisons between times are defined according to the specification of `op:time-equal`, `op:time-less-than` and `op:time-greater-than`; comparisons between years and months durations are defined according to the specification of `op:duration-equal`, `op:yearMonthDuration-less-than` and `op:year-MonthDuration-greater-than`; comparisons between days and time durations are defined according to the specification of `op:duration-equal`, `op:dayTimeDuration-less-than` and `op:dayTimeDuration-greater-than`.

String and Booleans can only be compared for equality: the semantics of strings and Booleans equality is as defined in the specification of `fn:codepoint-equal` and `op:Boolean-equal`, respectively.

Comparison operators are defined only when the two operands have the same type, except for years and months duration and days and time duration, which can be compared for equality. Notice, however, that “*with the exception of the zero-length duration, no instance of `xs:dayTimeDuration` can ever be equal to an instance of `xs:yearMonthDuration`.*” [XFO].

Given an expression `o` to be tested and two endpoint `e1` and `e2`:

- is in the interval $(e1..e2)$, also notated $]e1..e2[$, if and only if $o > e1$ and $o < e2$
- is in the interval $[e1..e2)$, also notated $]e1..e2]$, if and only if $o > e1$ and $o \leq e2$
- is in the interval $[e1..e2]$ if and only if $o \geq e1$ and $o \leq e2$
- is in the interval $]e1..e2]$, also notated $[e1..e2[$, if and only if $o \geq e1$ and $o < e2$

An expression to be tested satisfies an instance of simple unary tests (grammar rule 12) if and only if, either the expression is a list and the expression satisfies at least one simple unitary test in the list, or the simple unitary tests is “_”.

9.5 Use of S-FEEL expressions

This section summarizes which kinds of S-FEEL expressions are allowed in which role when the expression language is S-FEEL.

9.5.1 Item definitions

The expression that defines an **allowed value** SHALL be an instance of *simple unary tests* (grammar rule 12), where only the values in the defined or referenced type that satisfy the test are allowed values.

9.5.2 Invocations

In the bindings of an invocation, the **binding formula** SHALL be a *simple expression* (grammar rule 3).

9.5.3 Decision tables

Each **input expression** SHALL be a *simple expression* (grammar rule 3).

Each list of **input values** SHALL be an instance of *simple unary tests* (grammar rule 12).

Each list of **output values** SHALL be an instance of *simple unary tests* (grammar rule 12). Each **input entry** SHALL be an instance of *simple unary tests* (grammar rule 12).

Each **output entry** SHALL be a *simple expression* (grammar rule 3).

This page intentionally left blank.

10 Expression Language (FEEL)

10.1 Introduction

In **DMN**, all decision logic is represented as *boxed expressions*. Clause 7.2 introduced the concept of the boxed expression and defined two simple kinds: boxed literal expressions and boxed invocations. Clause 8 defined decision tables, a very important kind of boxed expression. This section completes the graphical notation for decision logic, by defining other kinds of boxed expressions.

The expressions 'in the boxes' are FEEL expressions. FEEL stands for Friendly Enough Expression Language, and it has the following features:

- Side-effect free
- Simple data model with numbers, dates, strings, lists, and contexts
- Simple syntax designed for a wide audience
- Three-valued logic (**true**, **false**, **null**)

This section also completely specifies the syntax and semantics of FEEL. The syntax is specified as a grammar (10.3.1). The subset of the syntax intended to be rendered graphically as a boxed expression is also specified as a meta-model (10.5).

FEEL has two roles in **DMN**:

1. As a textual notation in the boxes of boxed expressions such as decision tables.
2. As a slightly larger language to represent the logic of expressions and DRGs for the main purpose of composing the semantics in a simple and uniform way.

10.2 Notation

10.2.1 Boxed Expressions

This section builds on the generic notation for decision logic and boxed expressions defined in clause 7.2.

We define a graphical notation for decision logic called **boxed expressions**. This notation serves to decompose the decision logic model into small pieces that can be associated with DRG artifacts. The DRG plus the boxed expressions form a complete, mostly graphical language that completely specifies Decision Models.

A boxed expression is either:

- a decision table
- a boxed FEEL expression
- a boxed invocation
- a boxed context
- a boxed list
- a relation
- a boxed function
- a boxed conditional
- a boxed filter, or
- a boxed iterator

Boxed expressions are defined recursively, i.e., boxed expressions can contain other boxed expressions. The top-level boxed expression corresponds to the decision logic of a single DRG artifact. This boxed expression **SHALL** have a name box that contains the name of the DRG artifact. The name box may be attached in a single box on top, as shown in Figure 10-1:

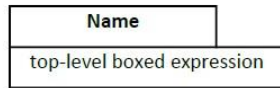


Figure 10-1: Boxed expression

Alternatively, the name box and expression box can be separated by white space and connected on the left side with a line, as shown in Figure 10- 2:

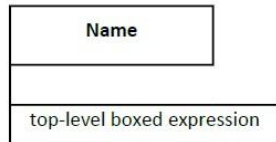


Figure 10- 2: Boxed expression with separated name and expression boxes

Graphical tools are expected to support appropriate graphical links, for example, clicking on a decision shape opens a decision table.

10.2.1.1 Decision Tables

The executable decision tables defined here use the same notation as the decision tables defined in Clause 8. Their execution semantics is defined in clause 10.3.2.10.

10.2.1.2 Boxed FEEL expression

A **boxed FEEL expression** is any FEEL expression e , as defined by the FEEL grammar (clause 10.3.1), in a table cell, as shown in Figure 10-3:



Figure 10-3: Boxed FEEL expression

The meaning of a boxed expression containing e is **FEEL**(e , s), where s is the scope. The scope includes the context derived from the containing DRD as described in 10.4, and any boxed contexts containing e .

It is usually good practice to make e relatively simple and compose small boxed expressions into larger boxed expressions.

10.2.1.3 Boxed Invocation

The syntax for boxed invocation is described in clause 7.2.3. This syntax may be used to invoke any function (e.g., business knowledge model, FEEL built-in function, boxed function definition).

The box labeled 'invoked business knowledge model' can be any boxed expression whose value is a function, as shown in Figure **10-4**:

Name	
function-valued expression	
parameter 1	binding expression 1
parameter 2	binding expression 2
...	
parameter n	binding expression n

Figure 10-4: Boxed invocation

The boxed syntax maps to the textual syntax defined by grammar rules 38, 39, 40, 41. Boxed invocation uses named parameters. Positional invocation can be achieved using a boxed expression containing a textual positional invocation.

The boxed syntax requires at least one parameter. A parameterless function must be invoked using the textual syntax, e.g., as shown in Figure 10-5.

function-valued expression()

Figure 10-5: Parameterless function

Formally, the meaning of a boxed invocation is given by the semantics of the equivalent textual invocation, *e.g.*, **function-valued expression** (parameter₁: **binding expression**₁, parameter₂: **binding expression**₂, ...).

10.2.1.4 **Boxed Context**

A **boxed context** is a collection of n (name, value) pairs with an optional result value. The names SHALL be distinct within a context. Each pair is called a context entry. Context entries may be separated by whitespace and connected with a line on the left (top). The intent is that all the entries of a context should be easily identified by looking down the left edge of a vertical context or across the top edge of a horizontal context. Cells SHALL be arranged in one of the following ways (see Figure 10-6, Figure 10-7):

Name 1	Value 1
Name 2	Value 2
Name n	Value n
Result	

Figure 10-6: Vertical context

Name 1	Name 2	Name <i>n</i>		Result
Value 1	Value 2	Value <i>n</i>		

Figure 10-7: Horizontal context

The context entries in a context are often used to decompose a complex expression into simpler expressions, each with a name. These context entries may be thought of as intermediate results. For example, contexts without a final Result box are useful for representing case data (see Figure 10-8).

Applicant Data		
Age	51	
MaritalStatus	"M"	
EmploymentStatus	"EMPLOYED"	
ExistingCustomer	false	
Monthly	Income	10000.00
	Repayments	2500.00
	Expenses	3000.00

Figure 10-8: Use of context entries

Contexts with a final result box are useful for representing calculations (see Figure 10-9).

Eligibility	
Age	Applicant. Age
Monthly Income	Applicant. Monthly. Income
Pre-Bureau Risk Category	Affordability. Pre-Bureau Risk Category
Installment Affordable	Affordability. Installment Affordable
if Pre-Bureau Risk Category = "DECLINE" or Installment Affordable = false or Age < 18 or Monthly Income < 100 then "INELIGIBLE" else "ELIGIBLE"	

Figure 10-9: Use of final result box

When decision tables are (non-result) context entries, the output cell can be used to name the entry, thus saving space. Any format decision table can be used in a vertical context. A jagged right edge is allowed. Whitespace between context entries may be helpful. See Figure 10-10.

Name 1	Value 1
	Name 2
Name n	Value n
Result	

Figure 10-10: Vertical context with decision table entry

Color is suggested. The names SHALL be legal FEEL names. The values and optional result are boxed expressions.

Boxed contexts may have a decision table as the result and use the named context entries to compute the inputs and give them names. For example (see Figure 10-11):

Post-Bureau Risk Category				
Existing Customer		Applicant. ExistingCustomer		
Credit Score		Report. CreditScore		
Application Risk Score		Affordability Model(Applicant, Product). Application Risk Score		
U	Existing Customer	Application Risk Score	Credit Score	Post-Bureau Risk Category
1	true	<=120	<590	“HIGH”
2			[590..610]	“MEDIUM”
3			>610	“LOW”
4		>120	<600	“HIGH”
5			[600..625]	“MEDIUM”
6			>625	“LOW”
7	false	<=100	<580	“HIGH”
8			[580..600]	“MEDIUM”
9			>600	“LOW”
10		>100	<590	“HIGH”
11			[590..615]	“MEDIUM”
12			>615	“LOW”

Figure 10-11: Use of boxed expressions with a decision table

Formally, the meaning of a boxed context is { "Name 1": Value 1, "Name 2": Value 2, ..., "Name *n*": Value *n* } if no Result is specified. Otherwise, the meaning is { "Name 1": Value 1, "Name 2": Value 2, ..., "Name *n*": Value *n*, "result": Result }. Recall that the bold face indicates elements in the FEEL Semantic Domain. The scope includes the context derived from the containing DRG as described in 10.4.

Boxed context entries for contexts that do not have a result box are accessible outside the context (as QNs), subject to the scope rules defined in clause 10.3.2.11. Boxed context entries for contexts that have a result box are not accessible outside the context.

10.2.1.5 **Boxed List**

A **boxed list** is a list of n items. Cells SHALL be arranged in one of the following ways (see Figure 10-12, Figure 10-13):

Item 1
Item 2
Item n

Figure 10-12: Vertical list

Item 1, Item 2, Item n

Figure 10-13: Horizontal list

Line styles are normative. The items are boxed expressions. Formally, the meaning of a boxed list is just the meaning of the list, i.e., [**Item 1**, **Item 2**, ..., **Item n**]. The scope includes the context derived from the containing DRG as described in 10.4.

10.2.1.6 **Relation**

A vertical list of homogeneous horizontal contexts (with no result cells) can be displayed with the names appearing just once at the top of the list, like a relational table, as shown in Figure 10-14:

Name 1	Name 2	Name n
Value 1a	Value 2a	Value na
Value 1b	Value 2b	Value nb
Value 1 m	Value 2 m	Value nm

Figure 10-14: Relation

10.2.1.7 **Boxed Function**

A Boxed Function Definition is the notation for parameterized boxed expressions.

The boxed expression associated with a Business Knowledge Model SHALL be a boxed function definition or a decision table whose input expressions are assumed to be the parameter names.

A boxed function has 3 cells:

1. **Kind**, containing the initial letter of one of the following:
 - **FEEL**
 - **ONNX**
 - **PMML**
 - **Java**

The **Kind** box can be omitted for FEEL functions, including decision tables.

2. Parameters: 0 or more comma-separated names, in parentheses
3. Body: a boxed expression

The 3 cells SHALL be arranged as shown in Figure 10-15:

K	(Parameter1, Parameter2, ...)
Body	

Figure 10-15: Boxed function definition

For FEEL functions, denoted by **Kind** *FEEL* or by omission of **Kind**, the Body SHALL be a FEEL expression that references the parameters. For externally defined functions denoted by **Kind** *Java*, the Body SHALL be a context as described in 10.3.2.13.3 and the form of the mapping information SHALL be the *java* form. For externally defined functions denoted by **Kind** *PMML*, the Body SHALL be a context as described in 10.3.2.13.3 and the form of the mapping information SHALL be the *pmml* form. For externally defined functions denoted by **Kind** *ONNX*, the Body SHALL be a context as described in 10.3.2.13.3 and the form of the mapping information SHALL be the *onnx* form.

Formally, the meaning of a boxed function is just the meaning of the function, *i.e.*, $FEEL(function(Parameter1, Parameter2, ...) \text{ Body})$ if the **Kind** is *FEEL*, and $FEEL(function(Parameter1, Parameter2, ...) \text{ external Body})$ otherwise. The scope includes the context derived from the containing DRG as described in 10.4.

10.2.1.8 Boxed conditional

Boxed conditional offers a visual representation of an **if** statement using three rows. The first one is labelled “if”; the second one is labelled “then” and the last one is labelled “else”. In the right part, another FEEL expression is expected. The expression in the “if” part **MUST** resolve to a boolean.

if	FEEL expression
then	FEEL expression
else	FEEL expression

Figure 10-16: Boxed conditional

Color is suggested.

if	U	Credit Score Rating	if
		"Poor", "Bad", "Fair", "Good", "Excellent"	
	1	"Good", "Excellent"	true
	2	"Poor", "Bad", "Fair"	false
then	Calculate interest rate		
	customer info		Customer Info
else	Calculate risky interest rate		
	customer info		Customer Info

Figure 10-17: Use of conditional expression with decision table and invocation

10.2.1.9 Boxed filter

Boxed filter offers a visual representation of collection filtering. The top part is an expression that is the collection to be filtered. The bottom part, between the square brackets, holds the filter expression. The expression in the top part MUST resolve to a collection including implicit conversion to *singleton list* as defined in section 10.3.2.9.4. The expression in the bottom part MUST resolve to a Boolean.

FEEL Collection Expression	
[	FEEL Filter Expression]

Figure 10-18: Filter expression

Color is suggested but it is considered a good practice to have a different color for the square brackets, so the filtering expression is easier to see.

1		
2		
3		
22		
31		
[	even(item)	]

Figure 10-19: Use of filter expression with a list expression

10.2.1.10 Boxed iterator

Boxed iterator offers a visual representation of an iterator statement. There are three flavors to it: **for** loop and quantified expression **some** and **every**.

For the **for** loop, the three rows are labelled “for”, “in” and “return”. The right part of the “for” displays the iterator variable name. The second row holds an expression representing the collection that will be iterated over. The expression in the in row **MUST** resolve to a collection including implicit conversion to *singleton list* as defined in section 10.3.2.9.4. The last row contains the expression that will process each element of the collection.

for	Iterator variable name
in	FEEL Collection Expression
return	FEEL Expression

Figure 10-20: For expression

for	letter	
in	["a", "b", "c", "d", "e"]	
return	Upper case	upper case(letter)
	Is it a vowel?	list contains(["a", "e", "i", "o", "u"], letter)

Figure 10-21: Use of for expression that returns a context

Every and **some** expression have a similar structure. The only difference between the two is the caption on the first line which is “every” or “some”. The second line is labelled “in” and the last one “satisfies”. The right part of the first line is the iterator variable name. The expression defined in the second row is the collection that will be tested. The expression in the in row MUST resolve to a collection including implicit conversion to *singleton list* as defined in section 10.3.2.9.4. The last line is an expression that will be evaluated on each item. The expression defined in the satisfies MUST resolve to a boolean.

every	Iterator variable name
in	FEEL Collection Expression
satisfies	FEEL Expression

Figure 10-22: Every expression

every	num	
in	1	
	2	
	3	
	4	
	5	
	6	
satisfies	num > 5	

Figure 10-23: Use of every with a list expression

some	Iterator variable name
in	FEEL Collection Expression
satisfies	FEEL Expression

Figure 10-24: Some expression

some	customer		
in	Name	Age	
	<i>Text</i>	<i>Number</i>	
	"Georges"	55	
	"Henry"	69	
	"Alexander"	10	
	"Emma"	5	
	"Jane"	39	
satisfies	U	customer.Age	satisfies
		<i>Number</i>	<i>Boolean</i>
	1	<18	false
	2	>=18	true

Figure 10-25: Use of some with a relation and a decision table

10.2.2 FEEL

A subset of FEEL, defined in the next section, serves as the notation "in the boxes" of boxed expressions. A FEEL object is a number, a string, a date, a time, a duration, a function, a context, or a list of FEEL objects (including nested lists).

Note: A JSON object is a number, a string, a context (JSON calls them maps) or a list of JSON objects. So, FEEL is an extension of JSON in this regard. In addition, FEEL provides friendlier syntax for literal values and does not require context keys to be quoted.

Here we give a "feel" for the language by starting with some simple examples.

10.2.2.1 Comparison of ranges

Ranges and lists of ranges appear in decision table input entry, input value, and output value cells. In the examples in Table 39, this portion of the syntax is shown underlined. Strings, dates, times, and durations also may be compared, using typographical literals defined in section 7.2.2.1.

FEEL Expression	Value
5 in (<u><=5</u>)	true
5 in ((<u>5..10</u>])	false
5 in ([<u>5..10</u>])	true
5 in (<u>4, 5, 6</u>)	true
5 in (<5, >5)	false
<u>2012-12-31</u> in ((<u>2012-12-25..2013-02-14</u>))	true

FEEL numbers and calculations are exemplified in Table 40.

[illegible]

Clause 9 introduced a subset of FEEL sufficient to support decision tables for Conformance Level 2 (see clause 2). The full **DMN** friendly-enough expression language (FEEL) required for Conformance Level 3 is specified here.

FEEL is a simple language with inspiration drawn from Java, JavaScript, XPath, SQL, PMML, Lisp, and many others.

The syntax is defined using grammar rules that show how complex expressions are composed of simpler expressions. Likewise, the semantic rules show how the meaning of a complex expression is composed from the meaning of constituent simpler expressions.

DMN completely defines the meaning of FEEL expressions that do not invoke externally-defined functions. There are no implementation-defined semantics. FEEL expressions (that do not invoke externally-defined functions) have no side- effects and have the same interpretation in every conformant implementation. Externally-defined functions SHOULD be deterministic and side-effect free.

10.3.1 Syntax

FEEL syntax is defined as grammar here and equivalently as a UML Class diagram in the meta-model (10.5)

10.3.1.1 Grammar notation

The grammar rules use the [ISO EBNF](#) notation. Each rule defines a non-terminal symbol *S* in terms of some other symbols *S*₁, *S*₂, ... The following table summarizes the EBNF notation.

Table 41: EBNF notation

Example	Meaning
$S = S_1 ;$	Symbol <i>S</i> is defined in terms of symbol <i>S</i> ₁
$S_1 S_2$	Either <i>S</i> ₁ or <i>S</i> ₂
$S_1 S_2$	<i>S</i> ₁ followed by <i>S</i> ₂
$[S_1]$	<i>S</i> ₁ occurring 0 or 1 time
$\{S_1\}$	<i>S</i> ₁ repeated 0 or more times
$k * S_1$	<i>S</i> ₁ repeated k times
"and"	literal terminal symbol

We extend the ISO notation with character ranges for brevity, as follows:

A character range has the following EBNF syntax:

character range = "[" , low character , "-" , high character , "]" ; low
character = unicode character ; high character = unicode
character ; unicode character = simple character | code point ;
code point = "\u", 4 * hexadecimal digit | "\U", 6 * hexadecimal
digit; hexadecimal digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
| "8" | "9" |
"a" | "A" | "b" | "B" | "c" | "C" | "d" | "D" | "e" | "E" | "f" | "F" ;

A simple character is a single Unicode character, *e.g.*, a, 1, \$, *etc.* Alternatively, a character may be specified by its hexadecimal code point value, prefixed with `\u`.

Every Unicode character has a numeric code point value. The low character in a range must have numeric value less than the numeric value of the high character.

For example, hexadecimal digit can be described more succinctly using character ranges as follows:

hexadecimal digit = `[0-9] | [a-i] | [A-F]` ;

Note that the character range that includes all Unicode characters is `[\u00-\u10FFFF]`.

10.3.1.2 Grammar rules

The complete FEEL grammar is specified below. Grammar rules are numbered, and in some cases, alternatives are lettered, for later reference. Boxed expression syntax (rule 53) is used to give execution semantics to boxed expressions.

1. expression =
 - a. boxed expression |
 - b. textual expression ;
2. textual expression =
 - a. for expression | if expression | quantified expression |
 - b. disjunction |
 - c. conjunction |
 - d. comparison |
 - e. arithmetic expression |
 - f. instance of |
 - g. path expression | descendant expression | filter expression | function invocation |
 - h. literal | simple positive unary test | name | "(" , expression , ")" ;
3. textual expressions = textual expression , { " , " , textual expression } ;
4. arithmetic expression =
 - a. addition | subtraction |
 - b. multiplication | division |
 - c. exponentiation |
 - d. arithmetic negation ;
5. simple expression = arithmetic expression | simple value ;
6. simple expressions = simple expression , { " , " , simple expression } ;
7. simple positive unary test =
 - a. ("<" | "<=" | ">" | ">=" | "=" | "!=") , endpoint |
 - b. interval ;

8. interval = (open interval start | closed interval start) , endpoint , ".." , endpoint , (open interval end | closed interval end) ;

9. open interval start = "(" | "]" ;

10. closed interval start = "[" ;

11. open interval end = ")" | "]" ;

12. closed interval end = "]" ;

13. positive unary test = expression ;

14. positive unary tests = positive unary test , { "," , positive unary test } ;

15. unary tests =

- a. positive unary tests |
- b. "not", " (" , positive unary tests, ")" |
- c. "-"

16. endpoint = expression ;

17. simple value = qualified name | simple literal ;

18. qualified name = name , { "." , name } ;

19. addition = expression , "+" , expression ;

20. subtraction = expression , "-" , expression ;

21. multiplication = expression , "*" , expression ;

22. division = expression , "/" , expression ;

23. exponentiation = expression , "**" , expression ;

24. arithmetic negation = "-" , expression ;

25. name = name start , { name part | additional name symbols } ;

26. name start = name start char , { name part char } ;

27. name part = name part char , { name part char } ;

28. name start char = "?" | [A-Z] | "_" | [a-z] | [\uC0-\uD6] | [\uD8-\uF6] | [\uF8-\u2FF] | [\u370-\u37D] | [\u37F-\u1FFF] | [\u200C-\u200D] | [\u2070-\u218F] | [\u2C00-\u2FEF] | [\u3001-\uD7FF] | [\uF900-\uFDCF] | [\uFDF0-\uFFFD] | [\u10000-\uEFFFE] ;

29. name part char = name start char | digit | \uB7 | [\u0300-\u036F] | [\u203F-\u2040] ;

30. additional name symbols = "." | "/" | "-" | "'" | "+" | "*" ;

31. literal = simple literal | "null" ;

32. simple literal = numeric literal | string literal | boolean literal | date time literal ;

```

33.string literal = """, { character – ("" | vertical space) | string escape sequence}, """;

34.boolean literal = "true" | "false";

35.numeric literal = [ "-" ], ( digits, [ ".", digits ] | ".", digits ), [ ( "e" | "E" ), [ "+" | "-" ], digits ] ;

36.digit = [ 0-9 ];

37.digits = digit, {digit} ;

38.function invocation = expression, parameters ;

39.parameters = "(", ( named parameters | positional parameters ), ")" ;

40.named parameters = parameter name, ":", expression, { ",", parameter name, ":", expression } ;

41.parameter name = name ;

42.positional parameters = [ expression, { ",", expression } ] ;

43.path expression = expression, ".", name ;

44.for expression = "for", name, "in", iteration context { ",", name, "in", iteration context }, "return",
    expression
;

45.if expression = "if", expression, "then", expression, "else" expression ;

46.quantified expression = ("some" | "every"), name, "in", iteration context, { ",", name, "in", iteration
    context }, "satisfies", expression ; quantified expression = ("some" | "every"), name, "in", expression, { ",",
    name, "in", expression }, "satisfies", expression ;

47.disjunction = expression, "or", expression ;

48.conjunction = expression, "and", expression ;

49.comparison =
    a. expression, ( "=" | "!=" | "<" | "<=" | ">" | ">=" ), expression |
    b. expression, "between", expression, "and", expression |
    c. expression, "in", positive unary test |
    d. expression, "in", "(", positive unary tests, ")" ;

50.filter expression = expression, "[", expression, "]" ;

51.instance of = expression, "instance", "of", type, expression, "as", type ;

52.type =
    qualified name |
    "range" "<" type ">" |
    "list" "<" type ">" |
    "context" "<" [ name ":", type { ":", name ":", type } ] ">" "context" "<" name ":", type { ":", name ":", type
    } ">" | "function" "<" [ type { ":", type } ] ">" ">" type
;

```

Commented [AF19]: DMN17-107

Commented [AF20]: DMN17-94

Commented [AF21]: DMN17-86

53. boxed expression = list | function definition | context ;

54. list = "[" , [expression , { ",", expression }] , "]" ;

55. function definition = "function" , "(" , [formal parameter { ",", formal parameter }] , ")" , ["external"] , expression ;

56. formal parameter = parameter name [":" type] ["?" "..."] ;

57. context = "{" , [context entry , { ",", context entry }] , "}" ;

58. context entry = key , ":" , expression ;

59. key = name | string literal ;

60. date time literal = at literal | function invocation ;

61. white space = vertical space | \u0009 | \u0020 | \u0085 | \u00A0 | \u1 680 | \u1 80E | [\u2000-\u200B] | \u2028 | \u2029 | \u202F | \u205F | \u3000 | \uFEFF ;

62. vertical space = [\u000A-\u000D]

63. iteration context = expression , [".." , expression] ;

64. string escape sequence = "\" | "\\\" | "\n\" | \"r\" | \"t\" | code point ;

65. at literal = "@" , string literal

66. range literal =

a. (open range start | closed range start) , range endpoint , ".." , range endpoint (open range end | closed range end) |

b. open range start , ".." , range endpoint (open range end | closed range end) |

c. (open range start | closed range start) , range endpoint , ".." , open range end ;

67. range endpoint = numeric literal | string literal | date time literal ;

68. descendant expression = expression , "..." , name ;

Additional syntax rules:

- Operator precedence is given by the order of the alternatives in grammar rules 1, 2 and 4, in order from lowest to highest. *E.g.*, (boxed) invocation has higher precedence than multiplication, multiplication has higher precedence than addition, and addition has higher precedence than comparison. Addition and subtraction have equal precedence, and like all FEEL infix binary operators, are left associative. Note that FEEL's order of operations regarding arithmetic negation and exponentiation differs from standard mathematical precedence, e.g., the FEEL expression $-4 ** 2$ is interpreted as $(-4) * (-4)$ and evaluates to 16. In standard mathematics, $-4 ** 2$ is interpreted as $-(4 * 4)$ and evaluates to -16 instead. To avoid any ambiguity, users are recommended to use explicit parentheses, e.g., instead of $-4 ** 2$ specify $-(4 ** 2) = -16$ or $(-4) ** 2 = 16$ as appropriate. Tools MAY present a warning to users to inform about the potentially unexpected precedence of the combination of these two operators.
- Java-style comments can be used, *i.e.*, `/*` to end of line and `/* ... */`.
- In rule 60 ("date time literal"), for the "function invocation" alternative, the only permitted functions are the builtins `date`, `time`, `date and time`, `duration`, `days and time duration`, and `years and months duration`. In rule 60 ("date time literal"), for the "function invocation" alternative, the only permitted functions are the builtins `date`, `time`, `date and time`, and `duration`.

Commented [AF22]: DMN17-37

Commented [AF23]: DMN17-72

- The string in rule 65 must follow the date string, time string, date and time string or duration string syntax, as detailed in section 10.3.4.1.
- In rule 56 ("formal parameter"), the symbols "?" and "..." are used to define type qualifiers. The symbol "?" denotes an optional parameter and "..." that the enclosing function definition has a variable number of parameters. Parameters with these qualifiers must come after the ones without qualifiers. A function definition can have at most one optional parameter and at most one variable argument parameter. A function definition cannot have both optional and variable argument qualifiers.

Commented [AF24]: DMN17-37

10.3.1.3 Literals, data types, built-in functions

FEEL supports literal syntax for numbers, strings, booleans, date, time, date and time, duration, and *null*. (See grammar rules, clause 10.3.1.2). Literals can be mapped directly to values in the FEEL semantic domain (clause 10.3.2.1).

FEEL supports the following datatypes:

- number
- string
- boolean
- days and time duration
- years and months duration
- date
- time
- date and time
- list
- range
- context
- function

10.3.1.4 Tokens, Names and White space

A FEEL expression consists of a sequence of tokens, possibly separated with white space (grammar rule 63). A token is a sequence of Unicode characters, either:

- A literal terminal symbol in any grammar rule other than grammar rule 30. Literal terminal symbols are enclosed in double quotes in the grammar rules, e.g., "and", "+", "=", or
- A sequence conforming to grammar rule 28, 29, 35, or 37

For backward compatibility reasons, "list", "context" and "range" from grammar rule 52 are not considered literal terminal symbols.

White space (except inside strings) acts as token separators. Most grammar rules act on tokens and thus ignore white space (which is not a token).

A name (grammar rule 27) is defined as a sequence of tokens. I.e., the name `IncomeTaxesAmount` is defined as the list of tokens [**Income**, **Taxes**, **Amount**]. The name `Income+Expenses` is defined as the list of tokens [**Income**, **+**, **Expenses**]. A consequence of this is that a name like `Phone Number` with one space in between the tokens is the same as `Phone Number` with several spaces in between the tokens.

A name start (grammar rule 26) SHALL NOT be a literal terminal symbol.

A name part (grammar rule 27) MAY be a literal terminal symbol.

10.3.1.5 Contexts, Lists, Qualified Names, and Context Lists

A context is a map of key-value pairs called context entries and is written using curly braces to delimit the context, commas to separate the entries, and a colon to separate key and value (grammar rule 57). The key can be a string or a name. The value is an expression.

A list is written using square brackets to delimit the list, and commas to separate the list items (grammar rule 54).

Contexts and lists can reference other contexts and lists, giving rise to a directed acyclic graph. Naming is path based. The *qualified name* (QN) of a context entry is of the form $N_1.N_2 \dots N_n$ where N_i is the name of an in-scope context.

Nested lists encountered in the interpretation of $N_1.N_2 \dots N_n$ are preserved. *E.g.*,

$$\begin{aligned} & \{a: \{b: [1]\}, \{a: \{b: [2.1, 2.2]\}, \{a: \{b: [3]\}, \{a: \{b: [4, 5]\}\}.a.b = \\ & \{b: [1]\}, \{b: [2.1, 2.2]\}, \{b: [3]\}, \{b: [4, 5]\}.b = \\ & [[1], [2.1, 2.2], [3], [4, 5]] \end{aligned}$$

Nested lists can be flattened using the *flatten()* built-in function (10.3.4).

10.3.1.6 Ambiguity

FEEL expressions reference InformationItems by their qualified name (QN), in which name parts are separated by a period. For example, variables containing components are referenced as `[varName].[componentName]`. Imported elements such as InformationItems and ItemDefinitions are referenced by namespace-qualified name, in which the first name part is the name specified by the Import element importing the element. For example, an imported variable containing components is referenced as `[import name].[varName].[componentName]`.

Because names are a sequence of tokens, and some of those tokens can be FEEL operators and keywords, context is required to resolve ambiguity. For example, the following could be names or other expressions:

- a-b
- a – b
- what if?
- Profit and loss

Ambiguity is resolved using the scope. Name tokens are matched from left to right against the names in-scope, and the longest match is preferred. In the case where the longest match is not desired, parenthesis or other punctuation (that is not allowed in a name) can be used to disambiguate a FEEL expression. For example, to subtract b from a if a-b is the name of an in-scope context entry, one could write `(a)-(b)`. Notice that it does not help to write `a - b`, using space to separate the tokens, because the space is not part of the token sequence and thus not part of the name.

10.3.2 Semantics

FEEL semantics is specified by mapping syntax -fragments to values in the FEEL semantic domain. Literals (clause 10.3.1.3) can be mapped directly. Expressions composed of literals are mapped to values in the semantic domain using simple logical and arithmetic operations on the mapped literal values. In general, the semantics of any FEEL expression are composed from the semantics of its sub-expressions.

10.3.2.1 Semantic Domain

The FEEL semantic domain **D** consists of an infinite number of typed values. The types are organized into a lattice called **L**.

The types include:

- simple datatypes such as number, boolean, string, date, time, and duration
- constructed datatypes such as functions, lists, and contexts
- the Null type, which includes only the **null** value
- the special type Any, which includes all values in **D**

A function is a lambda expression with lexical closure or is externally defined by Java, ONNX or PMML. A list is an ordered collection of domain elements, and a context is a partially ordered collection of (string, value) pairs called context entries.

We use *italics* to denote syntactic elements and **boldface** to denote semantic elements. For example, `FEEL((1 + 1, 2 + 2))` is **[2, 4]**.

Note that we use bold **[]** to denote a list in the FEEL semantic domain, and bold numbers **2, 4** to denote those decimal values in the FEEL semantic domain.

10.3.2.2 Equality, Identity and Equivalence

The semantics of equality are specified in the semantic mappings in clause 10.3.2.15. In general, the values to be compared must be of the same kind, for example, both numbers, to obtain a non-null result.

Identity simply compares whether two objects in the semantic domain are the same object. We denote the test for identity using infix **is**, and its negation using infix **is not**. For example, FEEL("I" = I) **is null**. Note that **is** never results in **null**.

Every FEEL expression *e* in scope *s* can be mapped to an element **e** in the FEEL semantic domain. This mapping defines the meaning of *e* in *s*. The mapping may be written **e is FEEL(*e*,*s*)**. Two FEEL expressions *e*₁ and *e*₂ are equivalent in scope *s* if and only if FEEL(*e*₁,*s*) **is** FEEL(*e*₂,*s*). When *s* is understood from context (or not important), we may abbreviate the equivalence as **e₁ is e₂**.

10.3.2.3 Semantics of literals and datatypes

FEEL datatypes are described in the following sub-sections. The meaning of the datatypes includes:

1. A mapping from a literal form (which in some cases is a string) to a value in the semantic domain.
2. A precise definition of the set of semantic domain values belonging to the datatype, and the operations on them.

Each datatype describes a (possibly infinite) set of values. The sets for the datatypes defined below are disjoint. We use *italics* to indicate a literal and **boldface** to indicate a value in the semantic domain.

10.3.2.3.1 number

FEEL Numbers are based on [IEEE 754-2008](#) Decimal128 format, with 34 decimal digits of precision and rounding toward the nearest neighbor with ties favoring the even neighbor.

Grammar rule 35 defines literal numbers. Literals consist of base 10 digits, an optional decimal point, and an optional exponent. -INF, +INF, and NaN literals are not supported. There is no distinction between -0 and 0. The number(from, grouping separator, decimal separator) built-in function supports a richer literal format. E.g., FEEL(number("1.000.000,01 ", ".", ",")) = **1000000.01**.

FEEL supports literal scientific notation, e.g., 1.2e3, which is equivalent to 1.2*10**3.

A FEEL number is represented in the semantic domain as a pair of integers (**p,s**) such that **p** is a signed 34 digit integer carrying the precision information, and **s** is the scale, in the range [-611 1..6176]. Each such pair represents the number **p/10^s**. To indicate the numeric value, we write **value(p,s)**. E.g., **value(100,2) = 1**. If precision is not of concern, we may write the value as simply **1**. Note that many different pairs have the same value. For example, **value(1,0) = value(10,1) = value(100,2)**.

There is no value for notANumber, positiveInfinity, or negativeInfinity. Use **null** instead.

10.3.2.3.2 string

Grammar rule 33 defines literal strings as a double-quoted sequence of Unicode characters (see <https://unicode.org/glossary/#character>), e.g., "abc". The supported Unicode character range is [\u00-\u10FFFF]. The string literals are described by rule 33. The corresponding Unicode code points are used to encode a string literal.

The literal string "abc" is mapped to the semantic domain as a sequence of three Unicode characters **a**, **b**, and **c**, written **"abc"**. The literal "\ U01F4 0E" is mapped to a sequence of one Unicode character written **"ð"** corresponding to the code point U+1F40E. ■

10.3.2.3.3 boolean

The Boolean literals are given by grammar rule 34. The values in the semantic domain are **true** and **false**.

10.3.2.3.4 time

Times in FEEL can be expressed using either a time literal (see grammar rule 65) or the *time()* built-in function (See 10.3.4.1). We use boldface time literals to represent values in the semantic domain.

A time in the semantic domain is a value of the [XML Schema](#) time datatype. It can be represented by a sequence of numbers for the hour, minute, second, and an optional time offset from Universal Coordinated Time (UTC). If a time offset is specified, including time offset = 00:00, the time value has a UTC form and is comparable to all time values that have UTC forms. If no time offset is specified, the time is interpreted as a local time of day at some location, whose relationship to UTC time is dependent on time zone rules for that location and may vary from day to day. A local time of day value is only sometimes comparable to UTC time values, as described in [XML Schema](#) Part 2 Datatypes.

A time **t** can also be represented as the number of seconds since midnight. We write this as **value_t(t)**. *E.g.*, **value_t(01:01:01) = 3661**.

The **value_t** function is one-to-one, but its range is restricted to [0..86400]. So, it has an inverse function **value_t⁻¹(x)** that returns: the corresponding time value for x, if x is in [0..86400]; and **value_t⁻¹(y)**, where $y = x - \text{floor}(x/86400) * 86400$, if x is not in [0..86400].

Note: That is, **value_t⁻¹(x)** is always actually applied to x modulo 86400. For example, **value_t⁻¹(3600)** will return the time of day that is "01:00:00", **value_t⁻¹(90000)** will also return "T01:00:00", and **value_t⁻¹(-3600)** will return the time of day that is "23:00:00", treating -3600 seconds as one hour *before* midnight.

10.3.2.3.5 date

Dates in FEEL can be expressed using either a date literal (see grammar rule 65) or the *date()* built-in function (See 10.3.4.1). A date in the semantic domain is a sequence of numbers for the year, month, day of the month. The year must be in the range [-999,999,999..999,999,999]. We use boldface date literals to represent values in the semantic domain.

When a date value is subject to implicit conversions (10.3.2.9.4) it is considered to be equivalent to a date time value in which the time of day is UTC midnight (00:00:00).

10.3.2.3.6 date-time

Date and time in FEEL can be expressed using either a *date time literal* (see grammar rule 65) or the *date and time()* built-in function (See 10.3.2.3.6). We use boldface *date and time literals* to represent values in the semantic domain.

A date and time in the semantic domain is a sequence of numbers for the year, month, day, hour, minute, second, and optional time offset from Universal Coordinated Time (UTC). The year must be in the range [-999,999,999..999,999,999]. If there is an associated time offset, including 00:00, the date-time value has a UTC form and is comparable to all other date-time values that have UTC forms. If there is no associated time offset, the time is taken to be a local time of day at some location, according to the time zone rules for that location. When the time zone is specified, e.g., using the IANA tz form (see 10.3.4.1), the date-time value may be converted to a UTC form using the time zone rules for that location, if applicable.

Note: projecting time zone rules into the future may only be safe for near-term date-time values.

A date and time **d** that has a UTC form can be represented as a number of seconds since a reference date and time (called the epoch). We write **value_{dt}(d)** to represent the number of seconds between **d** and the epoch. The **value_{dt}** function is one-to-one and so it has an inverse function **value_{dt}⁻¹**. *E.g.*, **value_{dt}(value_{dt}(d)) = d**. **value_{dt}⁻¹** returns **null** rather than a date with a year outside the legal range.

10.3.2.3.7 days and time duration

Days and time durations in FEEL can be expressed using either a duration literal (see grammar rule 65) or the **duration()** and **days and time duration()** builtin functions ~~duration()-built-in-function~~ (See 10.3.4.1). We use boldface days and time duration literals to represent values in the semantic domain. The literal format of the characters within the quotes of the string literal is defined by the lexical space of the [XPath Data Model](#) dayTimeDuration datatype. A

Commented [AF25]: DMN17-72

days and time duration in the semantic domain is a sequence of numbers for the days, hours, minutes, and seconds of duration, normalized such that the sum of these numbers is minimized. For example,
 $FEEL(duration("P0DT25H")) = P1DT1H$.

The value of a days and time duration can be expressed as a number of seconds. *E.g.*, $value_{dtd}(P1DT1H) = 90000$. The $value_{dtd}$ function is one-to-one and so it has an inverse function $value_{dtd^{-1}}$. *E.g.*, $value_{dtd^{-1}}(90000) = P1DT1H$.

10.3.2.3.8 years and months duration

Years and months durations in FEEL can be expressed using either a duration literal (see grammar rule 65) or the ~~duration() and years and months duration() builtin functions~~ ~~duration() built-in function~~ (See 10.3.4.1). We use boldface years and month duration literals to represent values in the semantic domain. The literal format of the characters within the quotes of the string literal is defined by the lexical space of the [XPath Data Model](#) yearMonthDuration datatype. A years and months duration in the semantic domain is a pair of numbers for the years and months of duration, normalized such that the sum of these numbers is minimized. For example,
 $FEEL(duration("P0Y13M")) = P1Y1M$.

The value of a years and months duration can be expressed as a number of months. *E.g.*, $value_{ymd}(P1Y1M) = 13$. The $value_{ymd}$ function is one-to-one and so it has an inverse function $value_{ymd^{-1}}$. *E.g.*, $value_{ymd^{-1}}(13) = P1Y1M$.

10.3.2.4 Ternary logic

FEEL, like SQL and PMML, uses of ternary logic for truth values. This makes **and** and **or** complete functions from $D \times D \rightarrow D$. Ternary logic is used in [Predictive Modeling Markup Language](#) to model missing data values.

10.3.2.5 Lists and filters

Lists are immutable and may be nested. The *first* element of a list L can be accessed using $L[1]$ and the *last* element can be accessed using $L[-1]$. The n^{th} element from the beginning can be accessed using $L[n]$, and the n^{th} element from the end can be accessed using $L[-n]$.

If $FEEL(L) = \mathbf{L}$ is a list in the FEEL semantic domain, the first element is $FEEL(L[1]) = \mathbf{L}[1]$. If $\mathbf{L}$ does not contain n items, then $\mathbf{L}[n]$ is **null**.

$\mathbf{L}$ can be filtered with a Boolean expression in square brackets. The expression in square brackets can reference a list element using the name *item*, unless the list element is a context that contains the key "**item**". If the list element is a context, then its context entries may be referenced within the filter expression without the '*item*.' prefix. For example: $[1, 2, 3, 4][item > 2] = [3, 4]$

$$[\{x:1, y:2\}, \{x:2, y:3\}][x=1] = [\{x:1, y:2\}]$$

The filter expression is evaluated for each item in list, and a list containing only items where the filter expression is **true** is returned. *E.g.*:

$$[\{x:1, y:2\}, \{x:null, y:3\}][x < 2] = [\{x:1, y:2\}]$$

The expression to be filtered is subject to implicit conversions (10.3.2.9.4) before the entire expression is evaluated.

For convenience, a selection using the "." operator with a list of contexts on its left hand side returns a list of selections, *i.e.* $FEEL(e, \mathbf{c}) = [FEEL(f, \mathbf{c}'), FEEL(f, \mathbf{c}''), \dots]$ where $FEEL(e) = [\mathbf{e}', \mathbf{e}'', \dots]$ and $\mathbf{c}'$ is $\mathbf{c}$ augmented with the context entries of $\mathbf{e}'$, $\mathbf{c}''$ is $\mathbf{c}$ augmented with the context entries of $\mathbf{e}''$, etc. For example:

$$[\{x:1, y:2\}, \{x:2, y:3\}].y = [2, 3]$$

$$[\{x:1, y:2\}, \{x:2\}].y = [2, null]$$

10.3.2.6 Context

A FEEL context is a partially ordered collection of (key, expression) pairs called context entries. In the syntax, keys can be either names or strings. Keys are mapped to strings in the semantic domain. These strings are distinct within

Commented [AF26]: DMN17-72

a context. A context in the domain is denoted using bold FEEL syntax with string keys, e.g. { "key₁" : expr₁, "key₂" : expr₂, ... }.

The syntax for selecting the value of the entry named *key₁* from context-valued expression *m* is *m.key₁*.

If *key₁* is not a legal name or for whatever reason one wishes to treat the key as a string, the following syntax is allowed: *get value(m, "key₁")*. Selecting a value by key from context **m** in the semantic domain is denoted as **m.key₁** or **get value(m, "key₁")**

To retrieve a list of key, value pairs from a context *m*, the following built-in function may be used: *get entries(m)*. For example, the following is true: *get entries({key₁: "value₁"})[key= "key₁"].value = "value₁"*

An expression in a context entry may not reference the key of the same context entry but may reference keys (as QNs) from previous context entries in the same context, as well as other values (as QNs) in scope.

These references SHALL be acyclic and form a partial order. The expressions in a context SHALL be evaluated consistent with this partial order.

10.3.2.7 Ranges

FEEL supports a compact syntax for a range of values, useful in decision table test cells and elsewhere. Ranges can be syntactically represented either:

- as a comparison operator and a single endpoint (grammar rule 7.a.)
- or a pair of endpoints and endpoint inclusivity flags that indicate whether one or both endpoints are included in the range (grammar rule 7.b.); on this case, endpoints must be of equivalent types (see section 10.3.2.9.1 for the definition of type equivalence) and the endpoints must be ordered such that range start <= range end.

Endpoints can be expressions (grammar rule 16) of the following types: number, string, date, time, date and time, or duration. The following are examples of valid ranges:

- < 10
- >= date("2019-03-31")
- >= @"2019-03-31"
- <= duration("PT01H")
- <= @"PT01H"
- [5 .. 10]
- (birthday .. @"2019-01-01")

Ranges are mapped into the semantic domain as a typed instance of the *range* type. If the syntax with a single endpoint and an operator is used, then the other endpoint is undefined, and the inclusivity flag is set to false. E.g.:

Table 42: Examples of range properties values

range	start included	start	end	end included
[1..10]	true	1	10	true
(1..10]	false	1	10	true
<= 10	false	undefined	10	true
> 1	false	1	undefined	false

The semantics of comparison expressions involving ranges (grammar rules 49c and 49d) is defined in Table 56, Table 55, Table 53, and Table 51. The same rules apply when ranges are created programmatically, e.g., using the range function.

10.3.2.8 Functions

The FEEL function literal is given by grammar rule 55. Functions can also be specified in **DMN** via Function Definitions (see 6.3.9). The constructed type $(T1, \dots, Tn) \rightarrow U$ contains the function values that take arguments of types $T1, \dots, Tn$ and yield results of type U , regardless of the way the function syntax (e.g., FEEL literal or **DMN** Function Definition). In the case of exactly one argument type $T \rightarrow U$ is a shorthand for $(T) \rightarrow U$.

10.3.2.9 Relations between types

Every FEEL expression executed in a certain context has a value in **D**, and every value has a type. The FEEL types are organized as a lattice (see Figure 10-26), with upper type *Any* and lower type *Null*. The lattice determines the conformance of the different types to each other. For example, because comparison is defined only between values with conforming types, you cannot compare a number with a boolean or a string.

The FEEL lattice contains several inner abstract types (e.g. *comparable*, *temporal*, *duration* and *context*) to represent unions of concrete types (e.g. *number* and *string*). For example, *duration* is the union of the *years* and *months* *duration* and *days* and *time duration* types. The *context* type is the union of all context types and is equivalent to *context*.

Commented [AF27]: DMN17-86

We define **type(e)** as the type of the domain element **FEEL(e, c)**, where *e* is an expression defined by grammar rule 1. Literals for numbers, strings, booleans, null, date, time, date and time and duration literals are mapped to the corresponding node in lattice **L**. Complex expression such as list, contexts and functions are mapped to the corresponding parameterized nodes in lattice **L**. For example, see Table 43.

Table 43: Examples of types of domain elements

e	type(e)
123	number
true	boolean
"abc"	string
date("2017-01-01 ")	date
["a", "b", "c"]	list<string>
["a", true, 123]	list<Any>
[1..10)	range<number>
>= @"201 9-01-01"	range<date>
e	type(e)
{"name": "Peter", age: 30}	context<"age": number, "name":string>
function f(x: number, y: number) x + y	(number, number) → number
DecisionA	context<"id":number, "name":string>
BkmA	(number, number) → number

A type expression e defined by grammar rule 54 is mapped to the nodes in the lattice **L** by function **type**(e) as follows: primitive data type names are mapped to the node with the same name (e.g., *string* is mapped the **string** node)

- *Any* is mapped to the node **Any**
- *Null* is mapped to the node **Null**
- *list*< T > is mapped to the **list** node with the parameter **type**(T)
- *context*($k_1:T_1, \dots, k_n:T_n$) where $n \geq 1$ is mapped to the **context** node with parameters $k_1: \mathbf{type}(T_1), \dots, k_n: \mathbf{type}(T_n)$
- *function*< $T_1, \dots, T_n \rightarrow T$ > is mapped to the **function** node with signature **type**(T_1), ..., **type**(T_n) $\rightarrow$ **type**(T)
- Type names defined in the *itemDefinitions* section are mapped similarly to the context types (see rule above).

If none of the above rules can be applied (e.g., type name does not exist in the decision model) the type expression is semantically incorrect.

We define two relations between types:

- Equivalence ($T \equiv S$): Types T and S are interchangeable in all contexts.

Conformance ($T \leq S$): An instance of type T can be substituted at each place where an instance of type S is expected.

10.3.2.9.1 Type Equivalence

The equivalence relationship ($\equiv$) between types is defined as follows:

- Primitive datatypes are equivalent to themselves, e.g., *string* $\equiv$ *string*.
- Two list types *list*< T > and *list*< S > are equivalent iff T is equivalent to S . For example, the types of ["a", "b"] and ["c"] are equivalent.
- Two context types *context*< $k_1: T_1, \dots, k_n: T_n$ > and *context*< $l_1: S_1, \dots, l_m: S_m$ > are equivalent iff $n = m$ and for every $k_i: T_i$ there is a unique $l_j: S_j$ such that $k_i = l_j$ and $T_i \equiv S_j$ for $i = 1, n$. Context types are the types defined via ItemDefinitions or the types associated to FEEL context literals such as { "name": "John", "age": 25 }.
- Two function types $(T_1, \dots, T_n) \rightarrow U$ and $(S_1, \dots, S_m) \rightarrow V$ are equivalent iff $n = m$, $T_i \equiv S_j$ and T_i and S_j have the same type qualifiers for $i = 1, n$ and $U \equiv V$.
- Two range types *range*< T > and *range*< S > are equivalent iff T is equivalent to S . For example, the types of [1..10] and [30..40] are equivalent.

Commented [AF28]: DMN17-37

Type equivalence is transitive: if *type1* is equivalent to *type2*, and *type2* is equivalent to *type3*, then *type1* is equivalent to *type3*.

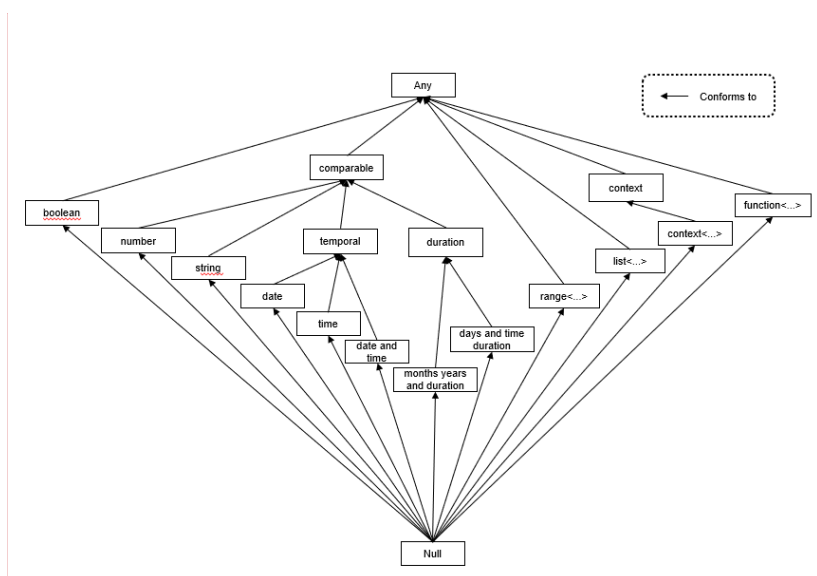
10.3.2.9.2 Type Conformance

The conformance relation ($\leq$) is defined as follows:

- Conformance includes equivalence. If $T \equiv S$ then $T \leq S$
- For every type T , $Null \leq T \leq Any$, where *Null* is the lower type in the lattice and *Any* the upper type in the lattice.
- The list type *list*< T > conforms to *list*< S > iff T conforms to S .
- The context type *context*< $k_1: T_1, \dots, k_n: T_n$ > conforms to *context*< $l_1: S_1, \dots, l_m: S_m$ > iff $n \geq m$ and for every $l_i: S_i$ there is a unique $k_j: T_j$ such that $l_i = k_j$ and $T_j \leq S_i$ for $i = 1, m$
- The function type $(T_1, \dots, T_n) \rightarrow U$ conforms to type $(S_1, \dots, S_m) \rightarrow V$ iff $n = m$, $S_i \leq T_i$ for $i = 1, n$ and $U \leq V$. The FEEL functions follow the "contravariant function argument type" and "covariant function return type" principles to provide type safety.

- The function type $(T_1, \dots, T_n) \rightarrow U$ conforms to type $(S_1, \dots, S_m) \rightarrow V$ if T_i has no type qualifiers for $i = 1, n$ and one of the following is satisfied:
 - if $n = m$, $S_i <: T_i$ for $i = 1, n$ and $U <: V$;
 - if $n = m - 1$ and second function type has an optional parameter, $S_i <: T_i$ for $i = 1, m - 1$ and $U <: V$;
 - if $n < m$ and second function type has a variable argument parameter, $S_i <: T_i$ for $i = 1, n - 1$, $S_n <: T_i$ for $i = n, m$ and $U <: V$;
- The FEEL functions follow the “contravariant function argument type” and “covariant function return type” principles to provide type safety.
- The range type $\text{range} < T >$ conforms to $\text{range} < S >$ iff T conforms to S . Type conformance is transitive: if type1 conforms to type2 , and type2 conforms to type3 , then type1 conforms to type3 .

Commented [AF29]: DMN17-37



Commented [AF30]: DMN17-86

Figure 10-26: FEEL type lattice type

Commented [AF31]: DMN17-86

10.3.2.9.3 Examples

Let us consider the following ItemDefinitions:

```
<itemDefinition name="Employee1">
  <itemComponent name="id">
    <typeRef>number</typeRef>
  </itemComponent>
  <itemComponent name="name">
    <typeRef>string</typeRef>
  </itemComponent>
</itemDefinition>

<itemDefinition name="Employee2">
```

```

    <itemComponent name="name">
      <typeRef>string</typeRef>
    </itemComponent>
    <itemComponent name="id">
      <typeRef>number</typeRef>
    </itemComponent>
  </itemDefinition>

  <itemDefinition name="Employee3">
    <itemComponent name="id">
      <typeRef>number</typeRef>
    </itemComponent>
    <itemComponent name="name">
      <typeRef>string</typeRef>
    </itemComponent>
    <itemComponent name="age">
      <typeRef>number</typeRef>
    </itemComponent>
  </itemDefinition>

  <itemDefinition isCollection="true" name="Employee3List">
    <itemComponent name="id">
      <typeRef>number</typeRef>
    </itemComponent>
    <itemComponent name="name">
      <typeRef>string</typeRef>
    </itemComponent>
    <itemComponent name="age">
      <typeRef>number</typeRef>
    </itemComponent>
  </itemDefinition>

```

and the decisions *Decision1*, *Decision2*, *Decision3* and *Decision4* with corresponding *typeRefs* *Employee1*, *Employee2*, *Employee3* and *Employee3List*.

Table 44 provides examples for *equivalence to* and *conforms to* relations.

Table 44: Examples of equivalence and conformance relations

type1	type2	equivalent to	conforms to
number	number	True	True
string	string	True	True
string	date	False	False

date	date and time	False	False
type (<i>Decision 1</i>)	type (<i>Decision2</i>)	True	True
type (<i>Decision1</i>)	type (<i>Decision3</i>)	False	False
type (<i>Decision3</i>)	type (<i>Decision1</i>)	False	True
type (<i>Decision 1</i>)	type ({"id": 1, "name ": "Peter"})	True	True
type ({"id": 1, "name ": "Peter"})	type (<i>Decision3</i>)	False	False
type ({"id": 1, "name": "Peter", "age": 45})	type (<i>Decision1</i>)	False	True
type ({"id": 1, "name": "Peter", "age": 45})	type (<i>Decision3</i>)	True	True
type ([1, 2, 3])	type (["1", "2", "3"])	False	False
type ([1, 2, 3])	type (<i>Decision3</i>)	False	False
type ({"id": 1, "name": "Peter", "age": 45})	type (<i>Decision4</i>)	True	True
type (<i>Decision4</i>)	type (<i>Decision3</i>)	False	False
type (function(x:Employee 1) → Employee1)	type (function(x:Employee 1) → Employee1)	True	True
type (function(x:Employee 1) → Employee1)	type (function(x:Employee 1) → Employee2)	True	True
type (function(x:Employee 1) → Employee3)	type (function(x:Employee 1) → Employee1)	False	True
type (function(x:Employee 1) → Employee1)	type (function(x:Employee 1) → Employee1)	False	False
type ([1..10])	type ((20..100))	True	True
type 1	type 2	equivalent to	conforms to
type ([1..10])	type (["a".. "x"])	False	False

10.3.2.9.4 Type conversions

The type of a FEEL expression e is determined from the value $e = \text{FEEL}(e, s)$ in the semantic domain, where s is a set of variable bindings (see 10.3.2.11 and 10.3.2.12). When an expression appears in a certain context it must be compatible with a type expected in that context, called the *target type*. After the type of the expression is deduced, an implicit conversion from the type of the expression to the target type can be performed sometimes. If an implicit conversion is mandatory but it cannot be performed the result is **null**.

In implicit type conversions, the data type is converted automatically without loss of information. There are several possible implicit type conversions:

- *to singleton list*:
When the type of the expression is T and the target type is $\text{List}\langle T \rangle$ the expression is converted to a singleton list.
- *from singleton list*:
When the type of the expression is $\text{List}\langle T \rangle$, the value of the expression is a singleton list and the target type is T , the expression is converted by unwrapping the first element.
- *from date to date and time*:
When the type of the expression is date and the target type is date and time, the expression is converted to a date time value in which the time of day is UTC midnight (00:00:00).
- *from decimal to integer*:
When the type of an expression is number and the expected value in the context is an integer (e.g. arguments of built-in functions substring and decimal) any fractional part of this number will be discarded.

There is one type of conversion to handle semantic errors:

- *conforms to* (as defined in 10.3.2.9.2 Type Conformance):
When the type of the expression is S , the target type is T , and S conforms to T the value of expression remains unchanged. Otherwise, the result is **null**.

There are several kinds of contexts in which conversions may occur:

- Filter context (10.3.2.5) in which a filter expression is present. The expression to be filtered is subject to implicit conversion *to singleton list*.
- Invocation context (Table 64) in which an actual parameter is bound to a formal parameter of a function. The actual parameter is subject to implicit conversions.
- Binding contexts in which the result of a DRG Element's logic is bound to the output variable. If after applying the implicit conversions the converted value and the target type do not conform, the *conforms to* conversion is applied.

10.3.2.9.4.1 Examples

The table below contains several examples for singleton list conversions.

Table 45: Examples of singleton list conversions

Expression	Conversion	Result
$3[\text{item} > 2]$	3 is converted to [3] as this a filter context, and an <i>to singleton list</i> is applied	[3]
$\text{contains}(["\text{foobar}"], "\text{of}')$	$["\text{foobar}"]$ is converted to "foobar", as this is an invocation context and <i>from singleton list</i> is applied	false

In the example below, before binding variable *decision_003* to value "123" the conversion to the target type (number) fails, hence the variable is bound to *null*.

```
<decision name="decision_003" id="_decision_003">
  <variable name="decision_003" typeRef="number"/>
  <literalExpression>
    <text>"123"</text>
  </literalExpression>
</decision>
```

10.3.2.10 Decision Table

The normative notation for decision tables is specified in Clause 8. Each input expression SHALL be a textual expression (grammar rule 2). Each list of input values SHALL be an instance of unary tests (grammar rule 15). The value that is tested is the value of the input expression of the containing InputClause. Each list of output values SHALL be an instance of unary tests (grammar rule 15). The value that is tested is the value of a selected output entry of the containing OutputClause. Each input entry SHALL be an instance of unary tests (grammar rule 15). Rule annotations are ignored in the execution semantics.

The decision table components are shown in Figure 8-5: Rules as rows – schematic layout, and also correspond to the metamodel in clause 8.3 For convenience, Figure 8-5 is reproduced here.

information item name			
H	input expression 1	input expression 2	Output label
	input value 1a, input value 1b	input value 2a, input value 2b	output value 1a, output value 1b
1	input entry 1.1	input entry 2.1	output entry 1.1
2		input entry 2.2	output entry 1.2
3	input entry 1.2	-	output entry 1.3

The semantics of a decision table is specified by first composing its literal expressions and unary tests into Boolean expressions that are mapped to the semantic domain and composed into rule matches then rule hits. Finally, some of the decision table output expressions are mapped to the semantic domain and comprise the result of the decision table interpretation. Decision table components are detailed in Table 46.

Table 46: Semantics of decision table

Component name (* means optional)	Description
input expression	One of the N>=0 input expressions, each a literal expression
input values*	One of the N input values, corresponding to the N input expressions. Each is a unary tests literal (see below).
output values*	A unary tests literal for the output. (In the event of M>1 output components (see Figure 8-12), each output component may have its own output values)
rules	a list of R>0 rules. A rule is a list of N input entries followed by M output entries. An input entry is a

	unary tests literal. An output entry is a literal expression.
hit policy*	one of: "U", "A", "P", "F", "R", "O", "C", "C+", "C#", "C<", "C>" (default is "U")
default output value*	The default output value is one of the output values. If $M > 1$, then default output value is a context with entries composed of output component names and output values.

Unary tests (grammar rule 15) are used to represent both input values and input entries. An input expression e is said to *satisfy* an input entry t (with optional input values v), depending on the syntax of t , as follows:

- grammar rule 15.a: $\text{FEEL}(e \text{ in } (t)) = \text{true}$
- grammar rule 15.b: $\text{FEEL}(e \text{ in } (t)) = \text{false}$
- grammar rule 15.c when v is not provided: $e \neq \text{null}$
- grammar rule 15.c when v is provided: $\text{FEEL}(e \text{ in } (v)) = \text{true}$

A rule with input entries $t_1, t_2, \dots, t_N$ is said to *match* the input expression list $[e_1, e_2, \dots, e_N]$ (with optional input values list $[v_1, v_2, \dots, v_N]$) if e_i *satisfies* t_i (with optional input values v_i) for all i in $1..N$.

A rule is *hit* if it is matched, and the hit policy indicates that the matched rule's output value should be included in the decision table result. Each hit results in one output value (multiple outputs are collected into a single context value). Therefore, multiple hits require aggregation.

The hit policy is specified using the initial letter of one of the following boldface policy names.

Single hit policies:

- **Unique** – only a single rule can be matched.
- **Any** – multiple rules can match, but they all have the same output,
- **Priority** – multiple rules can match, with different outputs. The output that comes first in the supplied *output values* list is returned,
- **First** – return the first match in rule order,

Multiple hit policies:

- **Collect** – return a list of the outputs in arbitrary order,
- **Rule order** – return a list of outputs in rule order,
- **Output order** – return a list of outputs in the order of the *output values* list

The Collect policy may optionally specify an *aggregation*, as follows:

- **C+** – return the sum of the outputs
- **C#** – return the count of the outputs
- **C<** – return the minimum-valued output
- **C>** – return the maximum-valued output

The *aggregation* is defined using the following built-in functions specified in clause 10.3.4.4: *sum*, *count*, *minimum*, *maximum*. To reduce complexity, decision tables with compound outputs do not support aggregation and support only the following hit policies: *Unique*, *Any*, *Priority*, *First*, *Collect without operator*, and *Rule order*.

A decision table may have no rule hit for a set of input values. In this case, the result is given by the default output value, or **null** if no default output value is specified. A complete decision table SHALL NOT specify a default output value.

The semantics of a decision table invocation **DTI** are as follows:

1. Every rule in the rule list is matched with the input expression list. Matching is unordered.
2. If no rules match,
 - a) if a default output value d is specified, **DTI**=FEEL(d)
 - b) else **DTI**=**null**.
3. Else let m be the sublist of rules that match the input expression list. If the hit policy is "First" or "Rule order", order m by rule number.
 - a) Let o be a list of output expressions, where the expression at index i is the output expression from rule $m[i]$. The output expression of a rule in a single output decision table is simply the rule's output entry. The output expression of a multiple output decision table is a context with entries composed from the output names and the rule's corresponding output entries. If the hit policy is "Output order", the decision table SHALL be single output, and o is ordered consistent with the order of the *output values*. Rule annotations are ignored for purposes of determining the expression value of a decision table.
 - b) If a multiple hit policy is specified, **DTI**=FEEL(aggregation(o)), where aggregation is one of the built-in functions *sum*, *count*, *minimum* as specified in clause 10.3.4.4.
 - c) else **DTI**=FEEL($o[1]$).

10.3.2.11 Scope and context stack

A FEEL expression e is always evaluated in a well-defined set of name bindings that are used to resolve QNs in e . This set of name bindings is called the scope of e . Scope is modeled as a list of contexts. A scope s contains the contexts with entries that are in scope for e . The last context in s is the *built-in* context. Next to last in s is the *global* context. The first context in s is the context immediately containing e (if any). Next are enclosing contexts of e (if any).

The QN of e is the QN of the first context in s appended with .N, where N is the name of entry in the first context of s containing e . QNs in e are resolved by looking through the contexts in s from first to last.

10.3.2.11.1 Local context

If e denotes the value of a context entry of context m , then m is the local context for e , and m is the first element of s . Otherwise, e has no local context and the first element of s is the global context, or in some cases explained later, the first element of s is a special context.

All of the entries of m are in-scope for e , but the *depends on* graph SHALL be acyclic. This provides a simple solution to the problem of the confusing definition above: if m is the result of evaluating the context expression m that contains e , how can we know it in order to evaluate e ? Simply evaluate the context entries in *depends on* order.

10.3.2.11.2 Global context

The global context is a context created before the evaluation of e and contains names and values for the variables defined outside expression e that are accessible in e . For example, when e is the body of a decision D , the global context contains entries for the information requirements and knowledge requirements of D (i.e., names and logic of the business knowledge models, decisions and decision services required by D).

10.3.2.11.3 Built-in context

The built-in context contains all the built-in functions.

10.3.2.11.4 Special context

Some FEEL expressions are interpreted in a *special* context that is pushed on the front of **s**. For example, a filter expression is repeatedly executed with special first context containing the name 'item' bound to successive list elements. A function is executed with a special first context containing argument name->value mappings.

Qualified names (QNs) in FEEL expressions are interpreted relative to **s**. The meaning of a FEEL expression *e* in scope **s** is denoted as **FEEL**(*e*, **s**). We can also say that *e* evaluates to **e** in scope **s**, or **e** = **FEEL**(*e*, **s**). Note that **e** and **s** are elements of the FEEL domain. **s** is a list of contexts.

10.3.2.12 Mapping between FEEL and other domains

A FEEL expression *e* denotes a value **e** in the semantic domain. Some kinds of values can be passed between FEEL and external Java methods, between FEEL and external PMML or ONNX models, and between FEEL and XML, as summarized in Table 47. An empty cell means that no mapping is defined.

Table 47: Mapping between FEEL and other domains

FEEL value	Java	XML	PMML	ONNX
number	java.math.BigDecimal	decimal	decimal, PROB-NUMBER, PERCENTAGE-NUMBER	FLOAT, UINT8, INT8, UINT16, INT16, INT32, INT64, FLOAT16, DOUBLE, UINT32, UINT64, COMPLEX64, COMPLEX128, BFLOAT16, FLOAT8E4M3FN, FLOAT8E4M3FNUZ, FLOAT8E5M2, FLOAT8E5M2FNUZ, UINT4, INT4
		integer	integer , INT-NUMBER	
		double	double, REAL-NUMBER	
string	java.lang.String	string	string, FIELD-NAME	STRING
date, time, date and time	javax.xml.datatype.XMLGregorianCalendar	date, dateTime, time, dateTimeStamp	date, dateTime, time conversion required for dateDaysSince, <i>et. al.</i>	
duration	javax.xml.datatype.Duration	yearMonthDuration, dayTimeDuration		
boolean	java.lang.Boolean	boolean	boolean	BOOL
list	java.util.List	contain multiple child elements	array (homogeneous)	contiguous array
context	java.util.Map	contain attributes and child elements		Tensor

For ONNX, each tensor is a context consisting of a string containing an ONNX type name, a list containing the dimension(s) of the tensor and a list of values.

Some kinds of values can also be passed between FEEL and JSON, as summarized in Table 48:

Table 48: Mapping between FEEL and JSON domains

<i>FEEL type</i>	<i>JSON type</i>	<i>Notes</i>
number	number	
string	string	
date, time, date and time	string	A string representation conforming to an ISO 8601 Date, Time or Date and Time combination. If the FEEL date and time contains an IANA timezone id, the ISO 8601 Date and Time is suffixed by the IANA timezone id in rectangular brackets, e.g. 2007-12-03T10:15:30+01:00[Europe/Paris]
years and months duration, days and time duration	string	A string representation conforming to an ISO 8601 Duration
boolean	The JSON literal "true" or the JSON literal "false"	
list	array	
context	object	
range	string	A string conforming to grammar rule 66 "range literal" as defined in chapter 10.3.1.2.
null	The JSON literal "null"	

Sometimes we do not want to evaluate a FEEL expression e , we just want to know the type of e . Note that if e has QNs, then a context may be needed for type inference. We write $\mathbf{type}(e)$ as the type of the domain element $\mathbf{FEEL}(e, c)$.

10.3.2.13 Functions Semantics

FEEL functions can be:

- built-in, e.g., sum (see clause 10.3.4.4), or
- user-defined, e.g., $\mathit{function}(\mathit{age}) \mathit{age} < 21$, or
- externally defined, e.g., $\mathit{function}(\mathit{angle}) \mathit{external} \{$
 $\mathit{java}: \{$
 $\mathit{class}: \text{"java.lang.Math"},$
 $\mathit{method signature}: \text{"cos(double)"} \}$
 $\}$

10.3.2.13.1 Built-in Functions

The built-in functions are described in detail in section 10.3.4. In particular, function signatures and parameter domains are specified. Some functions have more than one signature.

Built-in functions are invoked using the same syntax as other functions (grammar rule 40). The actual parameters must conform to the parameter domains in at least one signature before or after applying implicit conversions, or the result of the invocation is **null**.

10.3.2.13.2 User-defined functions

User-defined functions (grammar rule 55) have the form

$\mathit{function}(X1, \dots, Xn) \mathit{body}$

The terms $X_1, \dots, X_n$ are formal parameters. Each formal parameter has the form n_i or $n_i : t_i$, where the n_i are the parameter names and t_i are their types. If the type isn't specified, *Any* is assumed. The meaning of $\text{FEEL}(\text{function}(X_1, \dots, X_n) \text{ body}, \mathbf{s})$ is an element in the FEEL semantic domain that we denote as **function(argument list: $[X_1, \dots, X_n]$, body: body , scope: $\mathbf{s}$)** (shortened to **f** below). FEEL functions are lexical closures, i.e., the *body* is an expression that references the formal parameters and any other names in scope $\mathbf{s}$.

User-defined functions are invoked using the same syntax as other functions (grammar rule 38). The meaning of an invocation $f(n_1:e_1, \dots, n_n:e_n)$ in scope $\mathbf{s}$ is $\text{FEEL}(f, \mathbf{s})$ applied to arguments $n_1:\text{FEEL}(e_1, \mathbf{s}), \dots, n_n:\text{FEEL}(e_n, \mathbf{s})$. This can also be written as **f(n:e₁... , n_n:e_n)**.

The arguments $n_1:e_1, \dots, n_m:e_m$ conform to the argument list $[X_1, \dots, X_n]$ if the function type $(\text{type}(e_1), \dots, \text{type}(e_m)) \rightarrow U$ conforms to $(X_1, \dots, X_n) \rightarrow U$ where U is the return type of function f (see 10.3.2.9.2). The arguments $n_1:e_1, \dots, n_m:e_m$ conform to the argument list $[X_1, \dots, X_n]$ if $\text{type}(e_i)$ conforms to t_i before or after applying implicit conversions or t_i is not specified in X_i for all i in $1 \dots m$. The result of applying **f** to the interpreted arguments $n_1:e_1, \dots, n_m:e_m$ is determined as follows. If **f** is not a function, or if the arguments do not conform to the argument list, the result of the invocation is **null**. Otherwise, let $\mathbf{c}$ be a context with entries $n_1:e_1, \dots, n_m:e_m$. The result of the invocation is $\text{FEEL}(\text{body}, \mathbf{s}')$, where $\mathbf{s}' = \text{insert before}(\mathbf{s}, 1, \mathbf{c})$ (see 10.3.4.4).

Commented [AF32]: DMN17-37

Invocable elements (Business Knowledge Models or Decision Services) are invoked using the same syntax as other functions (grammar rule 38). An Invocable is equivalent to a FEEL function whose parameters are the invocable's inputs (see 10.4)

10.3.2.13.3 Externally-defined functions

FEEL externally-defined functions have the following form
`function ( $X_1, \dots, X_n$ ) external mapping-information`

Mapping-information is a context that SHALL have one of the following forms:

```
{
  java: {class: class-name, method signature: method-signature}
}
```

or

```
{
  onnx: {file: IRI, function signature: function-signature}
}
```

or

```
{
  pmml: {document: IRI, model: model-name}
}
```

The meaning of an externally defined function is an element in the semantic domain that we denote as **function(argument list: $[X_1, \dots, X_n]$, external: mapping-information)**.

The *java* form of the mapping information indicates that the external function is to be accessed as a method on a Java class. The *class-name* SHALL be the string name of a Java class on the classpath. Classpath configuration is implementation-defined. The *method-signature* SHALL be a string consisting of the name of a public static method in the named class, followed by an argument list containing only Java argument type names. The argument type information SHOULD be used to resolve overloaded methods and MAY be used to detect out-of-domain errors before runtime.

The *pmml* form of the mapping information indicates that the external function is to be accessed as a PMML model. The *IRI* SHALL be the resource identifier for a PMML document. The *model-name* is optional. If the *model-name* is

specified, it SHALL be the name of a model in the document to which the *IRI* refers. If no *model-name* is specified, the external function SHALL be the first model in the document.

The *omnx* form of the mapping information indicates that the external function is to be accessed as a ONNX model. The *IRI* SHALL be the resource identifier for a ONNX file. The *function-signature* SHALL be a string containing only one or more tensor definitions, each consisting of a ONNX type and the tensor dimensions in the form [a,b,c]. The tensor information SHOULD be passed to the ONNX implementation at runtime along with the data and MAY be used to detect errors before runtime.

When an externally-defined function is invoked, actual argument values and result value are converted when possible, using the type mapping table for Java, ONNX or PMML (see Table 47). When a conversion is not possible, **null** is substituted. If a result cannot be obtained, *e.g.*, an exception is thrown, the result of the invocation is **null**. If the externally-defined function is of type PMML or ONNX, and invocation results in a single predictor output, the result of the externally-defined function is the single predictor output's value.

Passing parameter values to the external method or model requires knowing the expected parameter types. For Java, this information is obtained using reflection. For PMML, this information is obtained from the mining schema and data dictionary elements associated with independent variables of the selected model. For ONNX this is determined by analysis of the protobuf data structure which contains a list of all the inputs and their (tensor) types.

Note that **DMN** does not completely define the semantics of a Decision Model that uses externally-defined functions. Externally-defined functions SHOULD have no side-effects and be deterministic.

10.3.2.13.4 Function name

To name a function, define it as a context entry. For example:

```
{ isPositive : function(x) x
  > 0,
  isNotNegative : function(x) isPositive(x+
    1), result: isNotNegative(0)
}
```

10.3.2.13.5 Positional and named parameters

An invocation of any FEEL function (built-in, user-defined, or externally-defined) can use positional parameters or named parameters. If positional, all parameters SHALL be supplied. If named, unsupplied parameters are bound to **null**.

10.3.2.14 For loop expression

The *for loop expression* iterates over lists of elements or ranges of numbers or dates. The general syntax is:

```
for  $i_1$  in  $ic_1$  [,  $i_2$  in  $ic_2$  [, ...]] return  $e$ 
```

where:

- $ic_1, ic_2, \dots, ic_n$ are iteration contexts
- $i_1, i_2, \dots, i_n$ are variables bound to each element in the iteration context
- e is the **return** expression

An *iteration context* may either be an expression that returns a list of elements, or two expressions that return integers connected by “..”. Examples of valid iteration contexts are:

- [1, 2, 3]
- a list
- 1..10

- 50..40
- x..x+10
- @”2021-01-01”..@”2022-01-01”

A *for loop expression* will iterate over each element in the *iteration context*, binding the element to the corresponding variable i_n and evaluating the expression e in that scope.

When the *iteration context* is a range of numbers, the *for loop expression* will iterate over the range incrementing or decrementing the value of i_n by 1, depending if the range is ascendant (when the resulting integer from the first expression is lower than the second) or descendant (when the resulting integer from the first expression is higher than the second).

When the *iteration context* is a range of dates, the *for loop expression* will iterate over the range incrementing or decrementing the value of i_n by 1 day, depending if the range is ascendant (when the resulting date from the first expression is lower than the second) or descendant (when the resulting date from the first expression is higher than the second).

The result of the *for loop expression* is a list containing the result of the evaluation of the expression e for each individual iteration in order.

The expression e may also reference an implicitly defined variable called “*partial*” that is a list containing all the results of the previous iterations of the expression. The variable “*partial*” is immutable. E.g.: to calculate the factorial list of numbers, from 0 to N, where N is a non-negative integer, one may write:

*for i in 0..N return if i = 0 then 1 else i * partial[-1]*

When multiple *iteration contexts* are defined in the same *for loop expression*, the resulting iteration is a cross product of the elements of the *iteration contexts*. The iteration order is from the inner *iteration context* to the outer *iteration context*.

E.g., the result of the following *for loop expression* is:

for i in [i₁,i₂], j in [j₁,j₂] return e = [r₁, r₂, r₃, r₄]

Where:

$r_1 = FEEL(e, \{ i: i_1, j: j_1, partial: [], \dots \})$

$r_2 = FEEL(e, \{ i: i_1, j: j_2, partial: [r_1], \dots \})$

$\dots r_3 = FEEL(e, \{ i: i_2, j: j_1, partial: [r_1, r_2], \dots \})$

$r_4 = FEEL(e, \{ i: i_2, j: j_2, partial: [r_1, r_2, r_3], \dots \})$

10.3.2.15 Semantic mappings

The meaning of each substantive grammar rule is given below by mapping the syntax to a value in the semantic domain. The value may depend on certain input values, themselves having been mapped to the semantic domain. The input values may have to obey additional constraints. The input domain(s) may be a subset of the semantic domain. Inputs outside of their domain result in a **null** value unless the implicit conversion *from singleton list* (10.3.2.9.4) can be applied.

Table 49: Semantics of FEEL functions

Grammar Rule	FEEL Syntax	Mapped to Domain
55	<i>function($n_1, \dots, n_N$) e</i>	function(argument list: [$n_1, \dots, n_N$], body: e, scope: s)
55	<i>function($n_1, \dots, n_N$) external e</i>	function(argument list: [$n_1, \dots, n_N$], external: e)

See 10.3.2.7.

Table 50: Semantics of other FEEL expressions

Grammar Rule	FEEL Syntax	Mapped to Domain
44	<i>for i_1 in ic_1, i_2 in ic_2, ... return e</i>	[FEEL(e, s'), FEEL(e, si, ...]
45	<i>if e_1 then e_2 else e_3</i>	if FEEL(e_1) is true then FEEL(e_2) else FEEL(e_3)
46	<i>some n_1 in e_1, n_2 in e_2, ... satisfies e</i>	false or FEEL(e, s') or FEEL(e, s'') or ...
46	<i>every n_1 in e_1, n_2 in e_2, ... satisfies e</i>	true and FEEL(e, s') and FEEL(e, s'') and ...
47	<i>e_1 or e_2 or ...</i>	FEEL(e_1) or FEEL(e_2) or ...
48	<i>e_1 and e_2 and ...</i>	FEEL(e_1) and FEEL(e_2) and ...
49.a	<i>e = null</i>	FEEL(e) is null
49.a	<i>null = e</i>	FEEL(e) is null
49.a	<i>e != null</i>	FEEL(e) is not null
49.a	<i>null != e</i>	FEEL(e) is not null

Notice that we use bold syntax to denote contexts, lists, conjunctions, disjunctions, conditional expressions, true, false, and null in the FEEL domain.

The meaning of the conjunction **a and b** and the disjunction **a or b** is defined by ternary logic. Because these are total functions, the input can be **true**, **false**, or **otherwise** (meaning any element of **D** other than **true** or **false**).

A conditional **if a then b else c** is equal to **b** if **a** is **true**, and equal to **c** otherwise.

s' is the scope **s** with a special first context containing keys n_1, n_2 , etc. bound to the first element of the Cartesian product of FEEL(e_1) x FEEL(e_2) x ..., **s''** is **s** with a special first context containing keys bound to the second element of the Cartesian product, etc. When the Cartesian product is empty, the *some ... satisfies* quantifier returns **false** and the *every ... satisfies* quantifier returns **true**.

Table 51: Semantics of conjunction and disjunction

a	b	a and b	a or b
true	true	true	true

true	false	false	true
true	otherwise	null	true
false	true	false	true
false	false	false	false
false	otherwise	false	null
otherwise	true	null	true
otherwise	false	false	null
otherwise	otherwise	null	null

Negation is accomplished using the built-in function **not**. The ternary logic is as shown in Table 52.

Table 52: Semantics of negation

a	not(a)
true	false
false	true
otherwise	null

Equality and inequality map to several kind- and datatype-specific tests, as shown in Table 53, Table 54 and Table 55. By definition, $FEEL(e_1 \neq e_2)$ is $FEEL(not(e_1 = e_2))$. The other comparison operators are defined only for the datatypes listed in Table 55. Note that Table 55 defines only '<'; '>' is similar to '<' and is omitted for brevity; $e_1 \leq e_2$ is defined as $e_1 < e_2$ or $e_1 = e_2$.

Table 53: General semantics of equality and inequality

Grammar Rule	FEEL Syntax	Input Domain	Result
49.a	$e_1 = e_2$	e1 and e2 must both be of the same kind/datatype – both numbers, both strings, etc.	<i>See below</i>
49.a	$e_1 < e_2$	e1 and e2 must both be of the same kind/datatype – both numbers, both strings, etc.	<i>See below</i>

Table 54: Specific semantics of equality

kind/datatype	$e_1 = e_2$
---------------	-------------

list	lists must be same length N and $e_1[i] = e_2[i]$ for $1 \leq i \leq N$.
context	contexts must have same set of keys K and $e_1.k = e_2.k$ for every k in K
range	the ranges must specify the same endpoint(s) and the same comparison operator or endpoint inclusivity flag.
function	internal functions must have the same parameters, body, and scope. Externally defined functions must have the same parameters and external mapping information.
number	value(e_1) = value(e_2) . Value is defined in 10.3.2.3.1. Precision is not considered.
string	e_1 is the same sequence of characters as e_2
date	value(e_1) = value(e_2) . Value is defined in 10.3.2.3.5
date and time	value(e_1) = value(e_2) . Value is defined in 10.3.2.3.6
time	value(e_1) = value(e_2) . Value is defined in 10.3.2.3.4.
days and time duration	value(e_1) = value(e_2) . Value is defined in 10.3.2.3.7
years and months duration	value(e_1) = value(e_2) . Value is defined in 10.3.2.3.8.
boolean	e_1 and e_2 must both be true or both be false

Table 55: Specific semantics of inequality

datatype	$e_1 < e_2$
number	value(e_1) < value(e_2) . value is defined in 10.3.2.3.1. Precision is not considered.
string	sequence of characters e_1 is lexicographically less than the sequence of characters e_2 . <i>I.e.</i> , the sequences are padded to the same length if needed with <code>\0</code> characters, stripped of common prefix characters, and then the first character in each sequence is compared.
date	$e_1 < e_2$ if the year value of $e_1 <$ the year value of e_2 $e_1 < e_2$ if the year values are equal and the month value of $e_1 <$ the month value of e_2 $e_1 < e_2$ if the year and month values are equal and the day value of $e_1 <$ the day value of e_2
date and time	value_{dt}(e_1) < value_{dt}(e_2) . value_{dt} is defined in 10.3.2.3.5. If one input has a null timezone offset, that input uses the timezone offset of the other input.
time	value_t(e_1) < value_t(e_2) . value_t is defined in 10.3.2.3.4. If one input has a null timezone offset, that input uses the timezone offset of the other input.

days and time duration	$\text{value}_{\text{dttd}}(e_1) < \text{value}_{\text{dttd}}(e_2)$. $\text{value}_{\text{dttd}}$ is defined in 10.3.2.3.7.
years and months duration	$\text{value}_{\text{ymd}}(e_1) < \text{value}_{\text{ymd}}(e_2)$. $\text{value}_{\text{ymd}}$ is defined in 10.3.2.3.8.

FEEL supports additional syntactic sugar for comparison. Note that Grammar Rules (clause 10.3.1.2) are used in decision table condition cells. These decision table syntaxes are defined in Table 56.

Table 56: Semantics of decision table syntax

Grammar Rule	FEEL Syntax	Equivalent FEEL Syntax	applicability
49.b	$e_1 \text{ between } e_2 \text{ and } e_3$	$e_1 \geq e_2 \text{ and } e_1 \leq e_3$	
49.c	$e_1 \text{ in } [e_2, e_3, \dots]$	$e_1 = e_2 \text{ or } e_1 = e_3 \text{ or } \dots$	e_2 and e_3 are endpoints
49.c	$e_1 \text{ in } [e_2, e_3, \dots]$	$e_1 \text{ in } e_2 \text{ or } e_1 \text{ in } e_3 \text{ or } \dots$	e_2 and e_3 are ranges
49.c	$e_1 \text{ in } \leq e_2$	$e_1 \leq e_2$	
49.c	$e_1 \text{ in } < e_2$	$e_1 < e_2$	
49.c	$e_1 \text{ in } \geq e_2$	$e_1 \geq e_2$	
49.c	$e_1 \text{ in } > e_2$	$e_1 > e_2$	
49.c	$e_1 \text{ in } (e_2..e_3)$	$e_1 > e_2 \text{ and } e_1 < e_3$	
49.c	$e_1 \text{ in } (e_2..e_3]$	$e_1 > e_2 \text{ and } e_1 \leq e_3$	
49.c	$e_1 \text{ in } [e_2..e_3)$	$e_1 \geq e_2 \text{ and } e_1 < e_3$	
49.c	$e_1 \text{ in } [e_2..e_3]$	$e_1 \geq e_2 \text{ and } e_1 \leq e_3$	
49.c	$e_1 \text{ in } e_2$	$e_1 = e_2$	e_2 is a qualified name that does <i>not</i> evaluate to a list
49.c	$e_1 \text{ in } e_2$	$\text{list contains}(e_2, e_1)$	e_1 is a simple value that is not a list and e_2 is a qualified name that evaluates to a list
49.c	$e_1 \text{ in } e_2$	$\{ ? : e_1, r : e_2 \}.r$	e_2 is a boolean expression that uses the special

Addition and subtraction are defined in Table 57 and Table 58. Note that if input values are not of the listed types, the result is **null**.

Table 57: General semantics of addition and subtraction

Grammar Rule	FEEL	Input Domain and Result
19	$e_1 + e_2$	<i>See below</i>
20	$e_1 - e_2$	<i>See below</i>

Table 58: Specific semantics of addition and subtraction

type(e1)	type(e2)	e1 + e2, e1 – e2	result type
number	number	Let $e1=(p1,s1)$ and $e2=(p2,s2)$ as defined in 10.3.2.3.1. If $value(p1,s1) +/- value(p2,s2)$ requires a scale outside the range of valid scales, the result is null. Else the result is (p,s) such that $value(p,s) = value(p1,s1) +/- value(p2,s2) + \epsilon$ $s \leq \max(s1,s2)$ s is maximized subject to the limitation that p has 34 digits or less ϵ is a possible rounding error.	number
date and time	date and time	Addition is undefined. Subtraction is defined as $valuedtj^1 (valuedt(e1)-valuedt(e2))$, where $valuedt$ is defined in 10.3.2.3.5 and $valuedtj^1$ is defined in 10.3.2.3.7. In case either value is of type date, it is implicitly converted into a date and time with time of day of UTC midnight ("00:00:00") as defined in 10.3.2.3.6. Subtraction requires either both values to have a timezone or both not to have a timezone. Subtraction is undefined for the case where only one of the values has a timezone.	days and time duration
time	time	Addition is undefined. Subtraction is defined as $valuedtd^{-1} (valuedt(e1)-valuedt(e2))$ where $valuedt$ is defined in 10.3.2.3.4 and $valuedtd^{-1}$ is defined in 10.3.2.3.7.	days and time duration
years and months duration	years and months duration	$valueymd^{-1}(valueymd(e1) +/- valueymd(e2))$ where $valueymd$ and $valueymd^{-1}$ is defined in 10.3.2.3.8.	years and months duration
days and time duration	days and time duration	$valuedtd^{-1}(valuedtd(e1) +/- valuedtd(e2))$ where $valuedtd$ and $valuedtd^{-1}$ is defined in 10.3.2.3.7.	days and time duration
date and time	years and months duration	date and time ($date(e1.year +/- e2.years + floor((e1.month +/- e2.months)/12)$, $e1.month +/- e2.months - floor((e1.month +/- e2.months)/12) * 12$, $e1.day$), $time(e1)$), where the named properties are as defined in Table 66 below, and the date, date and time, time and floor functions are as defined in 10.3.4, $valuedt$ and $valuedt^{-1}$ is defined in 10.3.2.3.5 and $valueymd$ is defined in 10.3.2.3.8.	date and time
years and months duration	date and time	Subtraction is undefined. Addition is commutative and is defined by the previous rule.	date and time
date and time	days and time duration	$valuedt^{-1}(valuedt(e1) +/- valuedtd(e2))$ where $valuedt$ and $valuedt^{-1}$ is defined in 10.3.2.3.5 and $valuedtd$ is defined in 10.3.2.3.7.	date and time
days and time duration	date and time	Subtraction is undefined. Addition is commutative and is defined by the previous rule.	date and time

time	days and time duration	value ⁻¹ (value (e1) +/- valuedtd (e2)) where value and value ⁻¹ are defined in 10.3.2.3.4 and valuedtd is defined in 10.3.2.3.7.	time
days and time duration	time	Subtraction is undefined. Addition is commutative and is defined by the previous rule.	time
string	string	Subtraction is undefined. Addition concatenates the strings. The result is a string containing the sequence of characters in e1 followed by the sequence of characters in e2.	string
date	years and months duration	date (e1.year +/- e2.years + floor((e1.month +/- e2.months)/12), e1.month +/- e2.months – floor((e1.month +/- e2.months)/12) * 12, e1.day), where the named properties are as defined in Table 66 below, and the date and floor functions are as defined in 10.3.4.	date
years and months duration	date	Subtraction is undefined. Addition is commutative and is defined by the previous rule.	date
date	days and time duration	date(value ⁻¹ (valuedt(e1) +/- valuedtd(e2))) where valuedt and value ⁻¹ is defined in 10.3.2.3.5 and valuedtd is defined in 10.3.2.3.7.	date
days and time duration	date	Subtraction is undefined. Addition is commutative and is defined by the previous rule.	date

Multiplication and division are defined in Table 59 and Table 60. Note that if input values are not of the listed types, the result is **null**.

Table 59: General semantics of multiplication and division

Grammar Rule	FEEL	Input Domain and Result
21	$e_1 * e_2$	See below
22	e_1 / e_2	See below

Table 60: Specific semantics of multiplication and division

type(e ₁)	type(e ₂)	$e_1 * e_2$	e_1 / e_2	result type
-----------------------	-----------------------	-------------	-------------	-------------

number $e_1=(p_1,s_1)$	number $e_2=(p_2,s_2)$	If $\text{value}(p_1,s_1) * \text{value}(p_2,s_2)$ requires a scale outside the range of valid scales, the result is null. Else the result is (p,s) such that $\text{value}(p,s) = \text{value}(p_1,s_1) * \text{value}(p_2,s_2) + \epsilon$ $s \leq s_1+s_2$ s is maximized subject to the limitation that p has 34 digits or less ϵ is a possible rounding error	If $\text{value}(p_2,s_2)=0$ or $\text{value}(p_1,s_1) / \text{value}(p_2,s_2)$ requires a scale outside the range of valid scales, the result is null. Else the result is (p,s) such that $\text{value}(p,s) = \text{value}(p_1,s_1) / \text{value}(p_2,s_2) + \epsilon$ $s \leq s_1-s_2$ s is maximized subject to the limitation that p has 34 digits or less	number
years and months duration	number	$\text{value}_{\text{ymd}}(\text{value}_{\text{ymd}}(e_1) * \text{value}(e_2))$ where $\text{value}_{\text{ymd}}$ and $\text{value}_{\text{ymd}-1}$ are defined in 10.3.2.3.8	If $\text{value}(e_2)=0$, the result is null. Else the result is $\text{value}_{\text{ymd}}(\text{value}_{\text{ymd}}(e_1) / \text{value}(e_2))$ where $\text{value}_{\text{ymd}}$ and $\text{value}_{\text{ymd}-1}$ are defined in 10.3.2.3.8.	years and months duration
number	years and months duration	<i>See above, reversing e_1 and e_2</i>	<i>Not allowed</i>	years and months duration
years and months duration	years and months duration	<i>Not allowed</i>	If $\text{value}_{\text{ymd}}(e_2)=0$, the result is null. Else the result is $\text{value}_{\text{ymd}}(e_1) / \text{value}_{\text{ymd}}(e_2)$ where $\text{value}_{\text{ymd}}$ is defined in 10.3.2.3.8.	number
days and time duration	number	$\text{value}_{\text{dtd}}(\text{value}_{\text{dtd}}(e_1) * \text{value}(e_2))$ where $\text{value}_{\text{dtd}}$ and $\text{value}_{\text{dtd}-1}$ are defined in 10.3.2.3.7.	If $\text{value}(e_2)=0$, the result is null. Else the result is $\text{value}_{\text{dtd}}(\text{value}_{\text{dtd}}(e_1) * \text{value}(e_2))$ where $\text{value}_{\text{dtd}}$ and $\text{value}_{\text{dtd}-1}$ are defined in 10.3.2.3.7.	days and time duration
number	days and time duration	<i>See above, reversing e_1 and e_2</i>	<i>Not allowed</i>	days and time duration
days and time duration	days and time duration	<i>Not allowed</i>	If $\text{value}_{\text{dtd}}(e_2)=0$, the result is null. Else the result is $\text{value}_{\text{dtd}}(e_1) / \text{value}_{\text{dtd}}(e_2)$ where $\text{value}_{\text{dtd}}$ is defined in 10.3.2.3.7.	number

Table 61: Semantics of exponentiation

Grammar Rule	FEEL Syntax	Input Domain	Result
23	$e_1 ** e_2$	type(e_1) is number. value(e_2) is a number in the range $[-999,999,999..999,999,999]$.	If value(e_1)^{value(e_2)} requires a scale that is out of range, the result is null. Else the result is (p,s) such that value(p,s)= value(e_1)^{value(e_2)} + ϵ p is limited to 34 digits ϵ is rounding error

Type-checking is defined in Table 62. Note that *type* is not mapped to the domain, and **null** is the only value in the Null type (see 10.3.2.1).

Before evaluating the *instance of* operator both operands are mapped to the type lattice **L** (see 10.3.2.9).

Table 62: Semantics of type-checking

Grammar Rule	FEEL Syntax	Mapped to Domain	Examples
51	e_1 instance of e_2	If e_2 cannot be mapped to a node in the lattice L, the result is null. If e_1 is null and type(e_2) is <i>Null</i>, the result is true. If type(e_1) conforms to type(e_2) (see section 10.3.2.9) and e_1 is not null, the result is true. Otherwise, the result is false.	<i>[123] instance of list<number></i> is true <i>"abc" instance of string</i> is true <i>123 instance of string</i> is false <i>123 instance of list</i> is null as a list type requires parameters (see rule 54).
<u>51</u>	<u>e_1 as e_2</u>	<u>If e_1 instance of e_2 is false or null, the result is null.</u> <u>Otherwise the result is e_1 and type(e_1 instance of e_2) is type(e_2)</u>	<u><i>"abc" as string</i> = "abc"</u> <u><i>123 as string</i> = null</u> <u><i>[1, 2, 3] as list<number></i> = [1, 2, 3]</u> <u><i>abs(x as number)</i> where x is a variable of type Any. If x is a number, the function is invoked with that number as argument; otherwise, the function is invoked with null as argument.</u>

Commented [AF33]: DMN17-94

Negative numbers and negation of durations are defined in Table 63.

Table 63: Semantics of negative numbers and negation of durations

Grammar Rule	FEEL Syntax	Equivalent FEEL Syntax
24	-e	$e*-1$

Invocation is defined in Table 64. An invocation can use positional arguments or named arguments. If positional, all arguments must be supplied. If named, unsupplied arguments are bound to **null**. Note that **e** can be a user-defined function, a user-defined external function, or a built-in function. The arguments are subject to implicit conversions (10.3.2.9.4). If the argument types before or after conversion do not conform to the corresponding parameter types, the result of the invocation is **null**.

Table 64: Semantics of invocation

Grammar Rule	FEEL	Mapped to Domain	Applicability
38, 39, 42	$e(e_1, \dots)$	$e(e_1, \dots)$	e is a function with matching arity and conforming parameter types
38, 39, 40, 41	$e(n_1; e_1, \dots)$	$e(n_1; e_1, \dots)$	e is a function with matching parameter names and conforming parameter types

Properties are defined in Table 65 and Table 66. If **type(e)** is date and time, time, or duration, and **name** is a property name, then the meaning is given by Table 66 and Table 67. For example, $\text{FEEL}(\text{date and time}("2012-0307Z").\text{year}) = 2012$.

Table 65: General semantics of properties

Grammar Rule	FEEL	Mapped to Domain	Applicability
18	$e.\text{name}$	$e.\text{"name"}$	type(e) is a context
18	$e.\text{name}$	<i>see below</i>	type(e) is a date/time/duration

Table 66: List of properties per type

type(e)	e . name	name =
date	result is the named component of the date object e . Valid names are shown to the right.	year, month, day, weekday, value
date and time	result is the named component of the date and time object e . Valid names are shown to the right.	year, month, day, weekday, hour, minute, second, time offset, timezone, value
time	result is the named component of the time object e . Valid names are shown to the right	hour, minute, second, time offset, timezone, value
years and months duration	result is the named component of the years and months duration object e . Valid names are shown to the right.	years, months, value
days and time duration	result is the named component of the days and time duration object e . Valid names are shown to the right.	days, hours, minutes, seconds, value
range	result is the named component of the range object e . Valid names are shown to the right.	start, end, start included, end included

Table 67: Specific semantics of date, time, and duration properties

name	type(name)	description
year	number	The year number as an integer in the interval [-999,999,999 .. 999,999,999]

month	number	The month number as an integer in the interval [1..12], where 1 is January and 12 is December
day	number	The day of the month as an integer in the interval [1..31]
weekday	number	The day of the week as an integer in the interval [1..7] where 1 is Monday and 7 is Sunday (compliant with the definition in ISO 8601)
hour	number	The hour of the day as an integer in the interval [0..23]
minute	number	The minute of the hour as an integer in the interval [0..59]
second	number	The second of the minute as a decimal in the interval [0..60)
time offset	days and time duration	The duration offset corresponding to the timezone the date or date and time value represents. The time offset duration must be in the interval [duration("-PT14H")..duration("PT14H")] as per the XML Schema Part 2 dateTime datatype. The time offset property returns null when the object does not have a time offset set.
timezone	string	The timezone identifier as defined in the IANA Time Zones database. The timezone property returns null when the object does not have an IANA timezone defined.
name	type(name)	description
years	number	The normalized years component of a years and months duration value as an integer. This property returns null when invoked on a days and time duration value.
months	number	The normalized months component of a years and months duration value. Since the value is normalized, this property must return an integer in the interval [0..11]. This property returns null when invoked on a days and time duration value.
days	number	The normalized days component of a days and time duration value as an integer. This property returns null when invoked on a years and months duration value.
hours	number	The normalized hours component of a days and time duration value. Since the value is normalized, this property must return an integer in the interval [0..23]. This property returns null when invoked on a years and months duration value.
minutes	number	The normalized minutes component of a days and time duration value. Since the value is normalized, this property must return an integer in the interval [0..59]. This property returns null when invoked on a years and months duration value.
seconds	number	The normalized minutes component of a days and time duration value. Since the value is normalized, this property must return a decimal in the interval [0..60). This property returns null when invoked on a years and months duration value.

value	number	The value returned by the value function corresponding to the type as defined in 10.3.2.3.4, 10.3.2.3.5, 10.3.2.3.6, 10.3.2.3.7 and 10.3.2.3.8.
-------	--------	---

Table 68: Specific semantics of range properties

name	type(name)	description
start	Type of the start endpoint of the range	the start endpoint of the range
end	Type of the end endpoint of the range	the end endpoint of the range
start included	boolean	true if the start endpoint is included in the range
end included	boolean	true if the end endpoint is included in the range

In the case of nested contexts, the descendant expression can be used to access a property name recursively throughout the nested context. For example:

```
{ a: { b: { b: 1 } } }...b
```

is evaluated to:

```
[ { b: 1 }, 1 ]
```

because each key contained in the context and all of its nested contexts are returned as a list of associated values.

Grammar Rule	FEEL	Mapped to Domain	Applicability
68	<i>e...name</i>	e..."name"	type e is a context

Lists are defined in Table 69.

Table 69: Semantics of lists

Grammar Rule	FEEL Syntax	Mapped to Domain (scope s)	Applicability
54	<i>e₁[e₂]</i>	e₁[e₂]	e₁ is a list and e₂ is an integer (0 scale number)
54	<i>e₁[e₂]</i>	e₁	e₁ is not a list and not null and value(e₂) = 1

54	$e_1[e_2]$	list of items e such that i is in e iff i is in e_1 and $\text{FEEL}(e_2, s')$ is true , where s' is the scope s with a special first context containing the context entry (" item ", i) and if i is a context, the special context also contains all the context entries of i .	e_1 is a list and type ($\text{FEEL}(e_2, s')$) is boolean
54	$e_1[e_2]$	[e_1] if $\text{FEEL}(e_2, s')$ is true , where s' is the scope s with a special first context containing the context entry (" item ", e_1) and if e_1 is a context, the special context also contains all the context entries of e_1 . Else [] .	e_1 is not a list and not null and type ($\text{FEEL}(e_2, s')$) is boolean

Contexts are defined in Table 70.

Table 70: Semantics of contexts

Grammar Rule	FEEL Syntax	Mapped to Domain (scope s)
57	$\{n_1 : e_1, n_2 : e_2, \dots\}$	{ "n₁": $\text{FEEL}(e_1, s_1)$, "n₂": $\text{FEEL}(e_2, s_2)$, ... } such that the s_i are all s with a special first context c_i containing a subset of the entries of this result context. If c_i contains the entry for n_i , then c_j does not contain the entry for n_i .
	$\{ "n_1" : e_1, "n_2" : e_2, \dots \}$	
54	$[e_1, e_2, \dots]$	[$\text{FEEL}(e_1)$, $\text{FEEL}(e_2)$, ...]

10.3.2.16 Error Handling

Errors in FEEL expressions are handled according to section 7.3.8.

10.3.3 XML Data

FEEL supports XML Data in the FEEL context by mapping XML Data into the FEEL Semantic Domain. Let $\text{XE}(e, p)$ be a function mapping an XML element e and a parent FEEL context p to a FEEL context, as defined in the following tables. XE makes use of another mapping function, $\text{XV}(v)$, that maps an XML value v to the FEEL semantic domain.

XML namespace semantics are not supported by the mappings. For example, given the namespace prefix declarations $\text{xmlns:p1} = \text{"http://example.org/foobar"}$ and $\text{xmlns:p2} = \text{"http://example.org/foobar"}$, the tags $p1:\text{myElement}$ and $p2:\text{myElement}$ are the same element using XML namespace semantics but are different using XML without namespace semantics.

10.3.3.1 Semantic mapping for XML elements (XE)

Table 71, e is the name of an XML element, a is the name of one of its attributes, c is a child element, and v is a value. The parent context p is initially empty.

Table 71: Semantics of XML elements

XML	context entry in p	Remark
$\langle e \rangle$	" e " : null	empty element $\rightarrow$ null -valued entry in p

<code><q:e /></code>	"e" : null	namespaces are ignored.
<code><e>v</e></code>	"e":XV(v)	unrepeated element without attributes
<code><e>v₁</e> <e>v₂</e></code>	"e": [XV(v ₁), XV(v ₂)]	repeating element without attributes
<code><e a="v"/></code> <code><c₁>v₁</c₁></code>	"e": { "a": XV(v), "c ₁ ": XV(v ₁),	An element containing attributes or child elements → context
<code><e a="v₁">v₂</e></code>	"e": { "@a": XV(v ₁), "\$content": XV(v ₂) }	v ₂ is contained in a generated \$content entry

An entry in the **context entry in p** column such as "e" : null indicates a context entry with string key "e" and value **null**. The context entries are contained by context **p** that corresponds to the containing XML element, or to the XML document itself.

The mapping does not replace namespace prefixes with the namespace IRIs. FEEL requires only that keys within a context be distinct, and the namespace prefixes are sufficient.

10.3.3.2 Semantic mapping for XML values (XV)

If an XML document was parsed with a schema, then some atomic values may have a datatype other than string. Table 72 defines how a typed XML value *v* is mapped to FEEL.

Table 72: Semantics of XML values

Type of <i>v</i>	FEEL Semantic Domain
number	FEEL(<i>v</i>)
string	FEEL("v")
date	FEEL(date("v"))
dateTime	FEEL(date and time("v"))
time	FEEL(time("v"))
duration	FEEL(duration("v"))
list, e.g. "v1 v2"	[XV(v1), XV(v2)]
element	XE(v)

10.3.3.3 XML example

The following schema and instance are equivalent to the following FEEL:

10.3.3.3.1 schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://www.example.org"
  targetNamespace="http://www.example.org" elementFormDefault="qualified">
  <xsd:element name="Context">
    <xsd:complexType> <xsd:sequence>
      <xsd:element name="Employee">
        <xsd:complexType> <xsd:sequence>
          <xsd:element name="salary" type="xsd:decimal"/>
        </xsd:sequence> </xsd:complexType>
      </xsd:element>
      <xsd:element name="Customer" maxOccurs="unbounded">
        <xsd:complexType> <xsd:sequence>
          <xsd:element name="loyalty_level" type="xsd:string"/>
          <xsd:element name="credit_limit" type="xsd:decimal"/>
        </xsd:sequence> </xsd:complexType>
      </xsd:element>
    </xsd:sequence> </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

10.3.3.3.2 instance

```
<Context xmlns:tns="http://www.example.org" xmlns="http://www.example.org">
  <tns:Employee>
    <tns:salary>13000</tns:salary>
  </tns:Employee>
  <Customer>
    <loyalty_level>gold</loyalty_level>
    <credit_limit>10000</credit_limit>
  </Customer>
  <Customer>
    <loyalty_level>gold</loyalty_level>
    <credit_limit>20000</credit_limit>
  </Customer>
  <Customer>
    <loyalty_level>silver</loyalty_level>
    <credit_limit>5000</credit_limit>
  </Customer>
</Context>
```

10.3.3.3.3 equivalent FEEL boxed context

Context		
Employee	salary	13000
Customer	loyalty_level	credit_limit

	gold	10000
	gold	20000
	silver	5000

When a decision model is evaluated, its input data described by an item definition such as an XML Schema element (clause 7.3.2) is bound to case data mapped to the FEEL domain. The case data can be in various formats, such as XML. We can notate case data as an equivalent boxed context, as above. Decision logic can reference entries in the context using expressions such as *Context.ins\$Employee.ins\$salary*, which has a value of 13000.

10.3.4 Built-in functions

To promote interoperability, FEEL includes a library of built-in functions. The syntax and semantics of the built-ins are required for a conformant FEEL implementation.

In all of the tables in this section, a superscript refers to an additional domain constraint stated in the corresponding footnote to the table. Whenever a parameter is outside its domain, the result of the built-in is **null**.

10.3.4.1 Conversion functions

FEEL supports many conversions between values of different types. Of particular importance is the conversion from strings to dates, times, and durations. There is no literal representation for date, time, or duration. Also, formatted numbers such as *1,000.00* must be converted from a string by specifying the grouping separator and the decimal separator.

Built-ins are summarized in Table 73. The first column shows the name and parameters. A question mark (?) denotes an optional parameter. The second column specifies the domain for the parameters. The parameter domain is specified as one of:

- a type, e.g., number, string
- any – any element from the semantic domain, including **null**
- not null – any element from the semantic domain, excluding **null**
- date string – a string value in the lexical space of the date datatype specified by XML Schema Part 2 Datatypes
- time string – a string value that is the extended form of a local time representation as specified by ISO 8601, followed by the character "@", followed by a string value that is a time zone identifier in the IANA Time Zones Database (<http://www.iana.org/time-zones>). The usage of the IANA Time Zones identifiers has been deprecated as of DMN 1.6 for time literals only. The FEEL function `date and time(date, time, timezone)` SHOULD be used instead of `date and time(date, time)`. The use of IANA Time Zones identifiers for date and time literals is NOT deprecated.
- date time string – a string value consisting of a date string value, as specified above, optionally followed by the character "T" followed by a time string value as specified above.
- duration string – a string value in the lexical space of the `xs:dayTimeDuration` or `xs:yearMonthDuration` datatypes specified by the ~~XQuery 3.1+0~~ and ~~XPath 3.12+0~~ Data Model.
- range string – a string value conforming to grammar rule 66 “range literal” as defined in chapter 10.3.1.2.

Commented [AF34]: DMN17-73

Table 73: Semantics of conversion functions

Name(parameters)	Parameter Domain	Description	Example

<code>date(from)</code>	date string	convert <i>from</i> to a date	<code>date("2012-12-25") - date("2012-12-24") = duration("P1D ")</code>
<code>date(from)</code>	date and time	convert <i>from</i> to a date (set time components to null)	<code>date(date and time("2012-12-25T11:00:00Z")) = date("2012-12-25")</code>
<code>date(year, month, day)</code>	<i>year, month, day</i> are numbers ²	creates a date from year, month, day component values	<code>date (2012, 12, 25) = date("2012-12-25")</code>
<code>date and time(date, time)</code>	<i>date</i> is a date or date time; <i>time</i> is a time	creates a date time from the given date (ignoring any time component) and the given time	<code>date and time ("2012-12-24T23:59:00") = date and time (date("2012-12-24"), time ("23:59:00"))</code>
<code>date and time(date, time, timezone)</code>	<i>date</i> is a date or date time; <i>time</i> is a time without timezone; <i>timezone</i> is a string denoting a timezone offset or a IANA zone identifier	creates a date time from the given date, time and timezone	<code>date and time (date("2024-12-24"), time("23:59:00"), "Z") = date and time ("2024-12-24T23:59:00Z")</code> <code>date and time (date("2024-12-24"), time("23:59:00"), "America/Costa_Rica") = date and time ("2024-12-24T23:59:00@America/Costa_Rica")</code>
<code>date and time(from)</code>	date time string	convert <i>from</i> to a date and time	<code>date and time("2012-12-24T23:59:00") + duration("PT1M") = date and time("2012-12-25T00:00:00")</code>
<code>time(from)</code>	time string	convert <i>from</i> to time	<code>time("23:59:00z") + duration("PT2M") = time("00:01:00@Etc/UTC")</code>
<code>time(from)</code>	time, date and time	convert <i>from</i> to time (ignoring date components)	<code>time(date and time("2012-12-25T11:00:00Z")) = time("11:00:00Z")</code>
<code>time(hour, minute, second, offset?)</code>	<i>hour, minute, second</i> , are numbers ² , <i>offset</i> is a days and time duration, or null	creates a time from the given component values	<code>time ("23:59:00z") = time (23, 59, 0, duration("PT0H"))</code>
<code>number(from, grouping separator, decimal separator)</code>	string ¹ , string ¹ , string ¹	convert <i>from</i> to a number	<code>number("1 000.0", " ", ".") = number("1,000.0", " ", ".")</code>
<code>number(from)</code>	string ¹	convert <i>from</i> to a number	<code>number("1.1") = number("1.1", null, null) = 1.1</code>
<code>number(from)</code>	number	return <i>from</i>	<code>number(5) = 5</code>

string(<i>from</i>)	non-null	convert <i>from</i> to a string	<i>string</i> (1.1) = "1.1" <i>string</i> (null) = null
duration(<i>from</i>)	duration string	convert <i>from</i> to a days and time or years and months duration	<i>date and time</i> ("2012-12-24T23:59:00") - <i>date and time</i> ("2012-12-22T03:45:00") = <i>duration</i> ("P2DT20H14M") <i>duration</i> ("P2Y2M") = <i>duration</i> ("P26M")
<u>days and time duration(<i>from</i>)</u>	<u>duration string</u>	<u>convert <i>from</i> to a days and time duration</u>	<u><i>date and time</i>("2012-12-24T23:59:00") - <i>date and time</i>("2012-12-22T03:45:00") = <i>days and time duration</i>("P2DT20H14M")</u>
<u>years and months duration(<i>from</i>)</u>	<u>duration string</u>	<u>convert <i>from</i> to a years and months duration</u>	<u><i>years and months duration</i>("P2Y2M") = <i>years and months duration</i>("P26M")</u>
years and months duration(<i>from</i> , <i>to</i>)	both are date or both are date and time	return years and months duration between <i>from</i> and <i>to</i>	<i>years and months duration</i> (<i>date</i> ("2011-12-22"), <i>date</i> ("2013-08-24")) = <i>duration</i> ("P1Y8M")
range (<i>from</i>)	range string	Convert from a range string to a range, according to the definitions of chapter 10.3.2.7 "Ranges". Please notice that in range string, only literal range endpoints are allowed as defined in grammar rule 67 "range endpoint" in chapter 10.3.1.2. If range string does not conform with grammar rule 66, the result is null .	<i>range</i> ("[18..21]") is [18..21] <i>range</i> ("[2..)") is >=2 <i>range</i> ("(.2]") is <2 <i>range</i> ("") is null <i>range</i> ("[.]") is null

Commented [AF35]: DMN17-72

1. *grouping separator* SHALL be one of space (' '), comma (','), period ('.'), or null.

Decimal separator SHALL be one of period, comma, or null, but SHALL NOT be the same as the grouping separator unless both are null.

from SHALL conform to grammar rule 37, after removing all occurrences of the grouping separator, if any, and after changing the decimal separator, if present, to a period.

2. If year, month, day, hour, minute or second are decimal numbers, the implicit conversion *from decimal to integer* is applied.

3. The *duration()* function has been deprecated as of DMN 1.7 and replaced with *days and time duration()* and *years and months duration()*.

Commented [AF36]: DMN17-72

10.3.4.2 Boolean function

Table 74 defines Boolean functions.

Table 74: Semantics of Boolean functions

Name(parameters)	Parameter Domain	Description	Example
<i>not(negand)</i>	boolean	logical negation	<i>not(true) = false</i> <i>not(null) = null</i>

10.3.4.3 String functions

Table 75 defines string functions.

Table 75: Semantics of string functions

Name(parameters)	Parameter Domain	Description	Example
<i>substring(string, start position, length?)</i>	string, number ¹	return <i>length</i> (or all) characters in <i>string</i> , starting at <i>start position</i> . 1 st position is 1, last position is -1	<i>substring("foobar",3) = "obar"</i> <i>substring("foobar",3,3) = "oba"</i> <i>substring("foobar", -2, 1) = "a"</i> <i>substring("\U01F40Eab", 2) = "ab"</i> where "\U01F40Eab" is the representation of 🐘 ab
<i>string length(string)</i>	string	return number of characters (or code points) in string.	<i>string length("foo") = 3</i> <i>string length("\U01F40Eab") = 3</i>
<i>upper case(string)</i>	string	return uppercased string	<i>upper case("aBc4") = "ABC4"</i>
<i>lower case(string)</i>	string	return lowercased string	<i>lower case("aBc4") = "abc4"</i>
<i>substring before (string, match)</i>	string, string	return substring of string before the match in string	<i>Substring before("foobar", "bar") = "foo"</i> <i>substring before("foobar", "xyz") = ""</i>
<i>substring after (string, match)</i>	string, string	return substring of string after the match in string	<i>substring after("foobar", "ob") = "ar"</i> <i>substring after("", "a") = ""</i>

replace(input, pattern, replacement, flags?)	string2	regular expression pattern matching and replacement	<i>replace("banana", "a", "o") = "bonono"</i> <i>replace("abcd", "(ab) (a)", "[1=\$1][2=\$2]") = "[1=ab][2=]cd"</i>
contains(string, match)	string	does the string contain the match?	<i>contains("foobar", "of") = false</i>
starts with(string, match)	string	does the string start with the match?	<i>starts with("foobar", "fo") = true</i>
ends with(string, match)	string	does the string end with the match?	<i>ends with("foobar", "r") = true</i>
matches(input, pattern, flags?)	string2	does the input match the regexp pattern?	<i>matches("foobar", "^fo*b") = true</i>
split(string, delimiter)	<i>string</i> is a string, <i>delimiter</i> is a pattern2	Splits the string into a list of substrings, breaking at each occurrence of the delimiter pattern.	<i>split("John Doe", "\s") = ["John", "Doe"]</i> <i>split("a;b;c;;", ";") = ["a", "b", "c", "", ""]</i>

string join(list, delimiter)	<i>list</i> is a list of strings, <i>delimiter</i> is a string	return a string which is composed by joining all the string elements from the list parameter, separated by the delimiter. The delimiter can be an empty string. Null elements in the list parameter are ignored. If <i>list</i> is empty, the result is the empty string. If <i>delimiter</i> is null, the string elements are joined without a separator.	<pre>string join(["a","b","c"], "_and_") = "a_and_b_and_c"</pre> <pre>string join(["a","b","c"], "") = "abc"</pre> <pre>string join(["a","b","c"], null) = "abc"</pre> <pre>string join(["a"], "X") = "a"</pre> <pre>string join(["a",null,"c"], "X") = "aXc"</pre> <pre>string join([], "X") = ""</pre>
string join(list)	<i>list</i> is a list of strings	return a string which is composed by joining all the string elements from the list parameter. Null elements in the list parameter are ignored. If <i>list</i> is empty, the result is the empty string.	<pre>string join(["a","b","c"]) = "abc"</pre> <pre>string join(["a",null,"c"]) = "ac"</pre> <pre>string join([]) = ""</pre>
<u>trim(string)</u>	<u>string</u>	<u>return a string with white spaces (see rule 61 of the grammar) removed from the beginning and the end</u>	<pre>trim(" abc ") = "abc"</pre> <pre>trim("x ") = "x"</pre> <pre>trim("") = ""</pre> <pre>trim(null) = null</pre>

Commented [AF37]: DMN17-96

1. *start position* must be a non-zero integer (0 scale number) in the range [-L..L], where L is the length of the string. *length* must be in the range [1..E], where E is L – *start position* + 1 if *start position* is positive, and –*start position* otherwise.
2. *pattern*, *replacement*, and *flags* SHALL conform to the syntax and constraints specified in clause 7.6 of [XQuery 3.1+0](#) and [XPath 3.12-0](#) Functions and Operators. Note that where XPath specifies an error result, FEEL specifies a null result.

Commented [AF38]: DMN17-73

10.3.4.4 List functions

Table 76 defines list functions.

Table 76: Semantics of list functions

Name(parameters)	Parameter Domain	Description	Example
list contains(<i>list</i> , <i>element</i>)	list, any element of the semantic domain including null	does the <i>list</i> contain the <i>element</i> ?	<i>list contains</i> ([1,2,3], 2) = true

count(<i>list</i>)	list	return size of <i>list</i> , or zero if <i>list</i> is empty	<i>count</i> ([1,2,3]) = 3 <i>count</i> ([]) = 0 <i>count</i> ([1,[2,3]]) = 2
min(<i>list</i>) min(<i>c</i> ₁ ,..., <i>c</i> _{<i>N</i>}), <i>N</i> > 0 max(<i>list</i>) max(<i>c</i> ₁ ,..., <i>c</i> _{<i>N</i>}), <i>N</i> > 0	non-empty list of comparable items or argument list of one or more comparable items	return minimum(maximum) item, or null if <i>list</i> is empty	<i>min</i> ([1,2,3]) = 1 <i>max</i> (1,2,3) = 3 <i>min</i> (1) = <i>min</i> ([1]) = 1 <i>max</i> ([]) = <i>null</i>
sum(<i>list</i>) sum(<i>n</i> ₁ ,..., <i>n</i> _{<i>N</i>}), <i>N</i> > 0	list of 0 or more numbers or argument list of one or more numbers	return sum of numbers, or null if <i>list</i> is empty	<i>sum</i> ([1,2,3]) = 6 <i>sum</i> (1,2,3) = 6 <i>sum</i> (1) = 1 <i>sum</i> ([]) = <i>null</i>
mean(<i>list</i>) mean(<i>n</i> ₁ ,..., <i>n</i> _{<i>N</i>}), <i>N</i> > 0	non-empty list of numbers or argument list of one or more numbers	return arithmetic mean (average) of numbers	<i>mean</i> ([1,2,3]) = 2 <i>mean</i> (1,2,3) = 2 <i>mean</i> (1) = 1 <i>mean</i> ([]) = <i>null</i>
all(<i>list</i>) all(<i>b</i> ₁ ,..., <i>b</i> _{<i>N</i>}), <i>N</i> > 0	list of Boolean items or argument list of one or more Boolean items	return <i>false</i> if any item is <i>false</i> , else <i>true</i> if empty or all items are <i>true</i> , else <i>null</i>	<i>all</i> ([<i>false</i> , <i>null</i> , <i>true</i>]) = <i>false</i> <i>all</i> (<i>true</i>) = <i>all</i> ([<i>true</i>]) = <i>true</i> <i>all</i> ([]) = <i>true</i> <i>all</i> (0) = <i>null</i>
any(<i>list</i>) any(<i>b</i> ₁ ,..., <i>b</i> _{<i>N</i>}), <i>N</i> > 0	list of Boolean items or argument list of one or more Boolean items	return <i>true</i> if any item is <i>true</i> , else <i>false</i> if empty or all items are <i>false</i> , else <i>null</i>	<i>any</i> ([<i>false</i> , <i>null</i> , <i>true</i>]) = <i>true</i> <i>any</i> (<i>false</i>) = <i>false</i> <i>any</i> ([]) = <i>false</i> <i>any</i> (0) = <i>null</i>
sublist(<i>list</i> , <i>start position</i> , <i>length</i> ?)	list, number ¹ , number ²	return list of <i>length</i> (or all) elements of <i>list</i> , starting with <i>list</i> [<i>start position</i>]. 1 st position is 1, last position is -1	<i>sublist</i> ([4,5,6], 1, 2) = [4,5]
append(<i>list</i> , <i>item</i> ...)	list, any element including null	return new <i>list</i> with <i>items</i> appended	<i>append</i> ([1], 2, 3) = [1,2,3]
concatenate(<i>list</i> ...)	list	return new <i>list</i> that is a concatenation of the arguments	<i>concatenate</i> ([1,2],[3]) = [1,2,3]
insert before(<i>list</i> , <i>position</i> , <i>newItem</i>)	list, number ¹ , any element including null	return new <i>list</i> with <i>newItem</i> inserted at <i>position</i>	<i>insert before</i> ([1,3], 1,2) = [2,1,3]
<u>insert after(<i>list</i>, <i>position</i>, <i>newItem</i>)</u>	<u>list, number¹, any element including null</u>	<u>return new <i>list</i> with <i>newItem</i> inserted after <i>position</i></u>	<u><i>insert after</i>([1,3],1,2) = [1,2,3] <i>insert after</i>([1,3], <i>null</i>, 2) = <i>null</i> <i>insert after</i>([1,3], 1, <i>null</i>) = [1, <i>null</i>, 3] <i>insert after</i>([1,3], 1, "2") = [1,"2",3]</u>
remove(<i>list</i> , <i>position</i>)	list, number ¹	<i>list</i> with item at <i>position</i> removed	<i>remove</i> ([1,2,3], 2) = [1,3]
list replace(<i>list</i> , <i>position</i> , <i>newItem</i>) list replace(<i>list</i> , <i>match</i> , <i>newItem</i>)	list, number ¹ or boolean function(item, <i>newItem</i>), any	return new list with <i>newItem</i> replaced at <i>position</i> (if <i>position</i> is a number) or return a new list where <i>newItem</i> replaced at all positions	<i>list replace</i> ([2, 4, 7, 8], 3, 6) = [2, 4, 6, 8] <i>list replace</i> ([2, 4, 7, 8], <i>function</i> (item, <i>newItem</i>) item < <i>newItem</i> , 5) = [5, 5, 7, 8]

Commented [AF39]: DMN17-100

	element including null	where the <i>match</i> function returned <i>true</i>	
<i>reverse(list)</i>	list	reverse the <i>list</i>	<i>reverse ([1,2,3]) = [3,2,1]</i>
<i>index of(list, match)</i>	list, any element including null	return ascending list of <i>list</i> positions containing <i>match</i>	<i>index of([1,2,3,2],2) = [2,4]</i>
<i>union(list...)</i>	list	return a list containing all the elements in the lists in no particular order with duplicate removal concatenate with duplicate removal	<i>union ([1,2],[2,3]) = [1,2,3]</i>
<i>intersection(list...)</i>	list	return the common elements of the lists in no particular order with duplicate removal	<i>intersection([1,2],[2,3],[2,3,4]) = [2]</i> <i>intersection(["a","b","c"],["a","b"]) = ["a","b"]</i> <i>intersection([1],[2]) = []</i> <i>intersection([1,2,3],null) = null</i>
<i>distinct values(list)</i>	list	duplicate removal	<i>distinct values([1,2,3,2,1]) = [1,2,3]</i>
<i>flatten(list)</i>	list	flatten nested lists	<i>flatten ([[1,2],[3]], 4) = [1,2,3,4]</i>
<i>product(list)</i> <i>product(n₁, ..., n_n)</i>	<i>list</i> is a list of numbers. <i>n₁ ... n_n</i> are numbers.	Returns the product of the numbers	<i>product([2, 3, 4]) = 24</i> <i>product([]) = null</i> <i>product(2, 3, 4) = 24</i>
<i>median(list)</i> <i>median(n₁, ..., n_n)</i>	<i>list</i> is a list of number. <i>n₁ ... n_n</i> are numbers.	Returns the median element of the list of numbers. I.e., after sorting the list, if the list has an odd number of elements, it returns the middle element. If the list has an even number of elements, returns the average of the two middle elements. If the list is empty, returns null.	<i>median(8, 2, 5, 3, 4) = 4</i> <i>median([6, 1, 2, 3]) = 2.5</i> <i>median([]) = null</i>
<i>stddev(list)</i> <i>stddev(n₁, ..., n_n)</i>	<i>list</i> is a list of number. <i>n₁ ... n_n</i> are numbers.	Returns the sample standard deviation of the list of numbers. If the <i>list</i> is empty or if the <i>list</i> contains only one element, the function returns null.	<i>stddev(2, 4, 7, 5) = 2.081665999466132735282297706979931</i> <i>stddev([47]) = null</i> <i>stddev(47) = null</i>
<i>mode(list)</i> <i>mode(n₁, ..., n_n)</i>	<i>list</i> is a list of number. <i>n₁ ... n_n</i> are numbers.	Returns the mode of the list of numbers. If the result contains multiple elements, they are returned in ascending order. If the list is empty, an empty list is returned.	<i>mode(6, 3, 9, 6, 6) = [6]</i> <i>stddev([]) = null</i> <i>mode([6, 1, 9, 6, 1]) = [1, 6]</i> <i>mode([]) = []</i>

Commented [AF40]: DMN17-103

Commented [AF41]: DMN17-98

Commented [AF42R41]: Corrected by DMN17-195

1. *position* must be a non-zero integer (0 scale number) in the range [-L..L], where L is the length of the list
2. *length* must be in the range [1..E], where E is L – *start position* + 1 if *start position* is positive, and –*start position* otherwise.

3. If position or length are decimal numbers, the implicit conversion *from decimal to integer* is applied.

10.3.4.5 Numeric functions

Table 77 defines numeric functions.

Table 77: Semantics of numeric functions

Name(parameters)	Parameter Domain	Description	Example
<code>decimal(<i>n</i>, <i>scale</i>)</code>	number, number ¹	return <i>n</i> with given <i>scale</i>	<i>decimal(1/3, 2) = .33</i> <i>decimal(1.5, 0) = 2</i> <i>decimal(2.5, 0) = 2</i>
<code>floor(<i>n</i>)</code> <code>floor(<i>n</i>, <i>scale</i>)</code>	number number, number1	Return <i>n</i> with given <i>scale</i> and rounding mode flooring. If at least one of <i>n</i> or <i>scale</i> is null the result is null .	<i>floor(1.5) = 1</i> <i>floor(-1.56, 1) = -1.6</i>
<code>ceiling(<i>n</i>)</code> <code>ceiling(<i>n</i>, <i>scale</i>)</code>	number number, number1	Return <i>n</i> with given <i>scale</i> and rounding mode ceiling. If at least one of <i>n</i> or <i>scale</i> is null the result is null .	<i>ceiling(1.5) = 2</i> <i>ceiling(-1.56, 1) = -1.5</i>
<code>round up(<i>n</i>)</code> <code>round up(<i>n</i>, <i>scale</i>)</code>	number, number1	Return <i>n</i> with given <i>scale</i> and rounding mode round up. If at least one of <i>n</i> or <i>scale</i> is null the result is null .	<i>round up(5.5) = 6</i> <i>round up(-5.5, 0) = -6</i> <i>round up(1.121, 2) = 1.13</i> <i>round up(-1.126, 2) = -1.13</i>
<code>round down(<i>n</i>)</code> <code>round down(<i>n</i>, <i>scale</i>)</code>	number, number1	Return <i>n</i> with given <i>scale</i> and rounding mode round down. If at least one of <i>n</i> or <i>scale</i> is null the result is null .	<i>round down(5.5) = 5</i> <i>round down (-5.5, 0) = -5</i> <i>round down (1.121, 2) = 1.12</i> <i>round down (-1.126, 2) = -1.12</i>
<code>round half up(<i>n</i>)</code> <code>round half up(<i>n</i>, <i>scale</i>)</code>	number, number1	Return <i>n</i> with given <i>scale</i> and rounding mode round half up. If at least one of <i>n</i> or <i>scale</i> is null the result is null .	<i>round half up(5.5) = 6</i> <i>round half up(-5.5, 0) = -6</i> <i>round half up(1.121, 2) = 1.12</i> <i>round half up(-1.126, 2) = -1.13</i>
<code>round half down(<i>n</i>)</code> <code>round half down(<i>n</i>, <i>scale</i>)</code>	number, number1	Return <i>n</i> with given <i>scale</i> and rounding mode round up. If at least one of <i>n</i> or <i>scale</i> is null the result is null .	<i>round half down (5.5) = 5</i> <i>round half down (-5.5, 0) = -5</i> <i>round half down (1.121, 2) = 1.12</i> <i>round half down (-1.126, 2) = -1.13</i>
<code>abs(<i>n</i>)</code>	<i>n</i> is a number, a days and time duration or a year and month duration	Returns the absolute value of <i>n</i> .	<i>abs(10) = 10</i> <i>abs(-10) = 10</i> <i>abs(@ "PT5H") = @ "PT5H"</i> <i>abs(@ "-PT5H") = @ "PT5H"</i>

<code>modulo(dividend, divisor)</code>	<i>dividend</i> and <i>divisor</i> are numbers, where <i>divisor</i> must not be 0 (zero). Returns the remainder of the division of <i>dividend</i> by <i>divisor</i>. In case either <i>dividend</i> or <i>divisor</i> is negative, the result has the same sign of the <i>divisor</i>. The <code>modulo</code> function can be expressed as follows: <pre>modulo (dividend, divisor) = dividend - divisor*floor (dividend/divisor).</pre>	Returns the remainder of the division of dividend by divisor.	<i>modulo</i>(12, 5) = 2 <i>modulo</i>(-12,5)= 3 <i>modulo</i>(12,-5)= -3 <i>modulo</i>(-12,-5)= -2 <i>modulo</i>(10. 1, 4.5)= 1.1 <i>modulo</i>(-10.1, 4.5)= 3.4 <i>modulo</i>(10.1, -4.5)= -3.4 <i>modulo</i>(-10.1, -4.5)= -1.1
<code>sqrt(number)</code>	<i>number</i> is a number.	Returns the square root of the given number. If <i>number</i> is negative it returns null.	<code>sqrt(16) = 4</code>
<code>log(number)</code>	<i>number</i> is a number	Returns the natural logarithm (base <i>e</i>) of the <i>number</i> parameter.	<code>log(10) = 2.30258509299</code>
<code>exp(number)</code>	<i>number</i> is a number	Returns the Euler's number <i>e</i> raised to the power of <i>number</i> .	<code>exp(5) = 148.413159102577</code>
<code>odd(number)</code>	<i>number</i> is a number	Returns true if <i>number</i> is odd, false if it is even.	<code>odd(5) = true</code> <code>odd(2) = false</code>
<code>even(number)</code>	<i>number</i> is a number	Returns true if <i>number</i> is even, false if it is odd.	<code>even(5) = false</code> <code>even (2) = true</code>

Commented [AF43]: DMN17-195

1. Scale is in the range [-6111..6176]
2. If scale is decimal numbers, the implicit conversion from decimal to integer is applied.

10.3.4.6 Date and time functions

Table 78 defines date and time functions.

Table 78: Semantics of date and time functions

Name(parameters)	Parameter Domain	Description	Example
------------------	------------------	-------------	---------

is(<i>value</i> ₁ , <i>value</i> ₂)	Both are elements of the D	Returns true if both values are the same element in the FEEL semantic domain D (see 10.3.2.2)	<i>is(date("2012-12-25"), time("23:00:50"))</i> is false <i>is(date("2012-12-25"), date("2012-12-25"))</i> is true <i>is(time("23:00:50z"), time("23:00:50"))</i> is false <i>is(time("23:00:50z"), time("23:00:50+00:00"))</i> is true
---	-----------------------------------	---	--

10.3.4.7 Range Functions

The following set of functions establish relationships between single scalar values and ranges of such values. All functions in this list take two arguments and return True if the relationship between the argument holds, or False otherwise.

The specification of these functions is heavily inspired by the equivalent functions in the HL7 CQL (Clinical Quality Language) standard version 1.4.

The following table intuitively depicts the relationships defined by the functions in this chapter, but the full semantics of the functions are listed in Table 79.

	Point-Point	Point-Interval	Interval-Interval	Interval-Point
<code>before(<i>A</i>, <i>B</i>)</code>				
<code>after(<i>A</i>, <i>B</i>)</code>				
<code>meets(<i>A</i>, <i>B</i>)</code>				
<code>met by(<i>A</i>, <i>B</i>)</code>				
<code>overlaps before(<i>A</i>, <i>B</i>)</code>				
<code>overlaps after(<i>A</i>, <i>B</i>)</code>				
<code>finishes(<i>A</i>, <i>B</i>)</code>				
<code>finished by(<i>A</i>, <i>B</i>)</code>				
<code>includes(<i>A</i>, <i>B</i>)</code>				
<code>during(<i>A</i>, <i>B</i>)</code>				
<code>starts(<i>A</i>, <i>B</i>)</code>				
<code>started by(<i>A</i>, <i>B</i>)</code>				
<code>coincides(<i>A</i>, <i>B</i>)</code>				

Table 79: Semantics of range functions

Name(parameters)	Evaluates to true if and only if (for each signature, respectively)	Example
(a) before(<i>point1</i> , <i>point2</i>)	(a) point1 < point2	before(1, 10) = true before(10, 1) = false

(b) before(<i>point</i> , <i>range</i>)	(b) point < range.start or (point = range.start and not(range.start included))	before(1, [1..10]) = false before(1, (1..10]) = true before(1, [5..10]) = true
(c) before(<i>range</i> , <i>point</i>)	(c) range.end < point or (range.end = point and not(range.end included))	before([1..10], 10) = false before([1..10], 10) = true before([1..10], 15) = true
(d) before(<i>range1</i> , <i>range2</i>)	(d) range 1 .end < range2.start or ((not(range1 .end included) or not(range2.start included)) and range 1 .end = range2.start)	before([1..10], [15..20]) = true before([1..10], [10..20]) = false before([1..10], [10..20]) = true before([1..10], (10..20]) = true
(a) after(<i>point1</i> , <i>point2</i>)	(a) point1 > point2	after(10, 5) = true after(5, 10) = false
(b) after(<i>point</i> , <i>range</i>)	(b) point > range.end or (point = range.end and not(range.end included))	after(12, [1..10]) = true after(10, [1..10]) = true after(10, [1..10]) = false
(c) after(<i>range</i> , <i>point</i>)	(c) range.start > point or (range.start = point and not(range.start included))	after([11..20], 12) = false after([11..20], 10) = true after((11..20], 11) = true after([11..20], 11) = false
(d) after(<i>range1</i> , <i>range2</i>)	(d) range 1 .start > range2.end or ((not(range1 .start included) or not(range2.end included)) and range 1 .start = range2.end)	after([11..20], [1..10]) = true after([1..10], [11..20]) = false after([11..20], [1.. 11]) = true after((11..20], [1..11]) = true
(a) meets(<i>range1</i> , <i>range2</i>)	(a) range1.end included and range2.start included and range 1 .end = range2.start	meets([1..5], [5..10]) = true meets([1..5], [5..10]) = false meets([1..5], (5..10]) = false meets([1..5], [6..10]) = false

(a) met by(<i>range1</i> , <i>range2</i>)	(a) range1.start included and range2.end included and range1.start = range2.end	met by([5..10], [1..5]) = true met by([5..10], [1..5]) = false met by([5..10], [1..5]) = false met by([6..10], [1..5]) = false
(a) overlaps(<i>range1</i> , <i>range2</i>)	(a) (range1.end > range2.start or (range1.end = range2.start and range1.end included and range2.start included)) and (range1.start < range2.end or (range1.start = range2.end and range1.start included and range2.end included))	overlaps([1..5], [3..8]) = true overlaps([3..8], [1..5]) = true overlaps([1..8], [3..5]) = true overlaps([3..5], [1..8]) = true overlaps([1..5], [6..8]) = false overlaps([6..8], [1..5]) = false overlaps([1..5], [5..8]) = true overlaps([1..5], [5..8]) = false overlaps([1..5], [5..8]) = false overlaps([1..5], [5..8]) = false overlaps([1..5], [5..8]) = false overlaps([5..8], [1..5]) = true overlaps([5..8], [1..5]) = false overlaps([5..8], [1..5]) = false overlaps([5..8], [1..5]) = false

(a) overlaps before(<i>range1</i> , <i>range2</i>)	(a) (range1.start < range2.start or range1.start = range2.start and range1.start included and not(range2.start included))) and (range1.end > range2.start or (range1.end = range2.start and range1.end included and range2.start included)) and (range1.end < range2.end or (range1.end = range2.end and (not(range1.end included) or range2.end included)))	overlaps before([1..5], [3..8]) = true overlaps before([1..5], [6..8]) = false overlaps before([1..5], [5..8]) = true overlaps before([1..5], [5..8]) = false overlaps before([1..5], [5..8]) = false overlaps before([1..5], [1..5]) = true overlaps before([1..5], [1..5]) = true overlaps before([1..5], [1..5]) = false overlaps before([1..5], [1..5]) = false
--	---	---

(a) overlaps after(<i>range1</i> , <i>range2</i>)	(a) (range2.start < range1.start or (range2.start = range1.start and range2.start included and not(range 1.start included))) and (range2.end > range 1.start or (range2.end = range 1.start and range2.end included and range 1.start included)) and (range2.end < range1.end or (range2.end = range1.end and (not(range2.end included) or range1.end included)))	overlaps after([3..8], [1..5]) = true overlaps after([6..8], [1..5]) = false overlaps after([5..8], [1..5]) = true overlaps after([5..8], [1..5]) = false overlaps after([5..8], [1..5]) = false overlaps after([1..5], [1..5]) = true overlaps after([1..5], [1..5]) = true overlaps after([1..5], [1..5]) = false overlaps after([1..5], [1..5]) = false
(a) finishes(<i>point</i> , <i>range</i>) (b) finishes(<i>range1</i> , <i>range2</i>)	(a) range.end included and range.end = point (b) range1.end included = range2.end included and range1.end = range2.end and (range1.start > range2.start or (range1.start = range2.start and (not(range1.start included) or range2.start included)))	finishes(10, [1..10]) = true finishes(10, [1..10]) = false finishes([5..10], [1..10]) = true finishes([5..10], [1..10]) = false finishes([5..10], [1..10]) = true finishes([1..10], [1..10]) = true finishes([1..10], [1..10]) = true
(a) finished by(<i>range</i> , <i>point</i>) (b) finished by(<i>range1</i> , <i>range2</i>)	(a) range.end included and range.end = point (b) range1.end included = range2.end included and range1.end = range2.end and (range1.start < range2.start or (range1.start = range2.start and (range1.start included or not(range2.start included)))	finished by([1..10], 10) = true finished by([1..10], 10) = false finished by([1..10], [5..10]) = true finished by([1..10], [5..10]) = false finished by([1..10], [5..10]) = true finished by([1..10], [1..10]) = true finished by([1..10], [1..10]) = true

(a) starts(<i>point</i> , <i>range</i>)	(a) range.start = point and range.start included	starts(1, [1..10]) = true starts(1, (1..10]) = false starts(2, [1..10]) = false
(b) starts(<i>range1</i> , <i>range2</i>)	(b) range1.start = range2.start and range1.start included = range2.start included and (range1.end < range2.end or (range1.end = range2.end and (not(range1.end included) or range2.end included)))	starts([1..5], [1..10]) = true starts((1..5], (1..10]) = true starts((1..5], [1..10]) = false starts([1..5], (1..10]) = false starts([1..10], [1..10]) = true starts([1..10], [1..10]) = true starts((1..10), (1..10)) = true
(a) started by(<i>range</i> , <i>point</i>)	(a) range.start = point and range.start included	started by([1..10], 1) = true started by((1..10], 1) = false started by([1..10], 2) = false
(b) started by(<i>range1</i> , <i>range2</i>)	(b) range1.start = range2.start and range1.start included = range2.start included and (range2.end < range1.end or (range2.end = range1.end and (not(range2.end included) or range1.end included)))	started by([1..10], [1..5]) = true started by((1..10], (1..5]) = true started by([1..10], (1..5]) = false started by((1..10], [1..5]) = false started by([1..10], [1..10]) = true started by([1..10], [1..10]) = true started by((1..10), (1..10)) = true
(a) coincides(<i>point1</i> , <i>point2</i>)	(a) point1 = point2	coincides(5, 5) = true coincides(3, 4) = false
(b) coincides(<i>range1</i> , <i>range2</i>)	(b) range1.start = range2.start and range1.start included = range2.start included and range1.end = range2.end and range1.end included = range2.end included	coincides([1..5], [1..5]) = true coincides((1..5], [1..5]) = false coincides([1..5], [2..6]) = false

10.3.4.8 Temporal built-in functions

The following set of functions provide common support utilities when dealing with date or date and time values; listed in Table 80.

Table 80: Temporal built-in functions

Name(parameters)	Parameter Domain	Description	Example
day of year(date)	date or date and time	returns the Gregorian number of the day within the year	day of year(date(2019, 9, 17)) = 260
day of week(date)	date or date and time	returns the day of the week according to the Gregorian calendar enumeration: "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"	day of week(date(2019, 9, 17)) = "Tuesday"
month of year(date)	date or date and time	returns the month of the year according to the Gregorian calendar enumeration: "January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"	month of year(date(2019, 9, 17)) = "September"
week of year(date)	date or date and time	returns the Gregorian number of the week within the year, accordingly to ISO 8601	week of year(date(2019, 9, 17)) = 38 week of year(date(2003, 12, 29)) = 1 week of year(date(2004, 1, 4)) = 1 week of year(date(2005, 1, 1)) = 53 week of year(date(2005, 1, 3)) = 1 week of year(date(2005, 1, 9)) = 1

10.3.4.9 Sort

Sort a list using an ordering function. The initial order of equal items is not guaranteed to be preserved i.e., no stability guarantee. For example, `sort(list: [3,1,4,5,2], precedes: function(x,y) x < y) = [1,2,3,4,5]`

Commented [AF44]: DMN17-63

Table 81: Semantics of sort functions

Parameter name (* means optional)	Domain
list	list of any element, be careful with nulls
<code>precedes</code>	boolean function of 2 arguments defined on every pair of list elements. <u>This function should return true if the first argument</u>

	precedes the second and false if they are equal or the second precedes the first argument.
--	---

Commented [AF45]: DMN17-63

10.3.4.10 Context function

Table 82: Context functions

Name(parameters)	Parameter domain	Description	Example
get value(m, key)	context, string	select the value of the entry named key from context m	<i>get value</i> ({key1 : "value1"}, "key1 ") = "value1" <i>get value</i> ({key1 : "value 1"}, "unexistent-key") = null
get entries(m)	context	produces a list of key,value pairs from a context m	<i>get entries</i> ({key1 : "value 1", key2 : "value2"}) = [{ key : "key1 ", value : "value 1" }, {key : "key2", value : "value2"}]
context(entries)	<i>entries</i> is a list of contexts; each context item SHALL have two entries having keys: "key" and "value", respectively.	Returns a new context that includes all specified entries. If a context item contains additional entries beyond the required "key" and "value" entries, the additional entries are ignored. If a context item is missing the required "key" and "value" entries, the final result is null. See also: <i>get entries()</i> builtin function.	<i>context</i> ({key:"a", value:1}, {key:"b", value:2}) = {a:1, b:2} <i>context</i> ({key:"a", value:1}, {key:"b", value:2, something:"else"}) = {a:1, b:2} <i>context</i> ({key:"a", value:1}, {key:"b"}) = null
(a) context put(context, key, value)	(a) <i>context</i> is a context, <i>key</i> is a string, <i>value</i> is Any type	(a) Returns a new context that includes the new entry or overriding the existing value if an entry for the same key already exists in the supplied context parameter. A new entry is added as the last entry of the new context. If overriding an existing entry, the order of the keys maintains the same order as in the original context.	<i>context put</i> ({x:1}, "y", 2) = {x:1, y:2} <i>context put</i> ({x:1, y:0}, "y", 2) = {x:1, y:2} <i>context put</i> ({x:1, y:0, z:0}, "y", 2) = {x:1, y:2, z:0} <i>context put</i> ({x:1}, ["y"], 2) = <i>context put</i> ({x:1}, "y", 2) = {x:1, y:2}

(b) context put(context, keys, value)	(b) context is a context, keys is a list of string, value is Any type	(b) Returns the composite of nested invocations to <i>context put()</i> for each item in <i>keys</i> hierarchy in <i>context</i>. If <i>keys</i> is a list of 1 element, this is equivalent to <i>context put(context, key', value)</i>, where <i>key'</i> is the only element in the list <i>keys</i>. If <i>keys</i> is a list of 2 or more elements, this is equivalent of calling <i>context put(context, key', value')</i>, with: <i>key'</i> is the head element in the list <i>keys</i>, <i>value'</i> is the result of invocation of <i>context put(context', keys', value)</i>, where: <i>context'</i> is the result of <i>context.key'</i>, <i>keys'</i> is the remainder of the list <i>keys</i> without the head element <i>key'</i>. If <i>keys</i> is an empty list or null, the result is null.	<pre>context put({x:1, y: {a: 0} }, ["y", "a"], 2) = context put({x:1, y: {a: 0} }, "y", context put({a: 0}, ["a"], 2)) = {x:1, y: {a: 2} } context put({x:1, y: {a: 0} }, [], 2) = null</pre>
context merge(contexts)	contexts is a list of contexts	Returns a new context that includes all entries from the given contexts; if some of the keys are equal, the entries are overridden. The entries are overridden in the same order as specified by the supplied parameter, with new entries added as the last entry in the new context.	<pre>context merge([{{x:1}, {y:2}}]) = {x:1, y:2} context merge([{{x:1, y:0}, {y:2}}]) = {x:1, y:2}</pre>

10.3.4.11 Miscellaneous functions

The following set of functions provide support utilities for several miscellaneous use-cases. For example, when a decision depends on the current date, like deciding the support SLA over the weekends, additional charges for weekend delivery, etc.

It is important to note that the functions in this section are intended to be side-effect-free, but they are not deterministic and not idempotent from the perspective of an external observer.

Vendors are encouraged to guide end-users in ensuring deterministic behavior of the DMN model during testing, for example, through specific configuration.

Users are encouraged to isolate decision logic that uses these functions in specific DRG elements, such as Decisions. This encapsulation enables them to be overridden with synthetic values that remain constant across executions of the DMN model's test cases.

Table 83: Miscellaneous functions

Name(parameters)	Parameter domain	Description
now()	(none)	returns current date and time
today()	(none)	returns current date

10.3.5 FEEL Libraries

A library has a name, is defined in a namespace, and contains a list of function declarations. The syntax of a library is defined below:

```
library =
  "namespace" , qualified name , ","
  name , "{" , functions , "}"
  ;

functions =
  function , { ";" , function }

function =
  "function" , name , "(" , [ formal parameter { "," , formal parameter } ] , ")" , ":" ,
  type
  ;
```

where

- name is described in section 10.3.1.2, rule 25
- qualified name is described in section 10.3.1.2, rule 18
- formal parameter is described in section 10.3.1.2, rule 56
- type is described in section 10.3.1.2, rule 52
- name, qualified name and formal parameter contain only symbols that are classified as letters and digits in the Unicode standard
- the libraries are uniquely identified by the namespace and library name pair

The definitions of the FEEL libraries are platform-independent, they do not contain any information about the execution platform (e.g. Java). The discovery mechanism of the definitions of the libraries and the artifacts needed to execute the functions (e.g. Java jars or Python modules) is vendor-specific. The mapping of the FEEL types to the native platforms (e.g. Java) is defined in Table 47: Mapping between FEEL and other domains.

The functions defined in a library become visible in the scope of the evaluation (see 10.3.2.11) once they are imported in a DMN file (see 6.3.3) with the import type equal to the FEEL namespace (e.g. <https://www.omg.org/spec/DMN/20240513/FEEL/>).

The functions defined in a library are invoked in the same way as the imported BKMs or Decision Services (e.g. `prefix.f(a, b, c)`).

For example, for the following library

```
namespace org.omg.feel.stringUtil;
```

```

stringLib {
  // Checks if str is empty ("" ) or null.
  function isEmpty(str: string) : boolean ;
  ...
  // The capitalized str, null if str is null
  function capitalize(str: string) : string
}

```

once the library is imported with

```

<import namespace="org.omg.feel.stringUtil"
      name="myLib"
      importType="https://www.omg.org/spec/DMN/20240513/FEEL/"
/>

```

the function isEmpty can be invoked by myLib.isEmpty("abc").

Commented [AF46]: DMN17-54

10.4 Execution Semantics of Decision Services

FEEL gives execution semantics to decision services defined in decision models where FEEL is the expression language. A decision service is semantically equivalent to a FEEL function whose parameters are the decision service inputs, and whose logic is a context assembled from the decision service's decisions and knowledge requirements.

Decision service implementations SHALL return a result as described above, and MAY return additional information such as intermediate results, log records, debugging information, error messages, rule annotations, etc. The format of any additional information is left unspecified.

Every FEEL expression in a decision model has execution semantics. `LiteralExpression` (FEEL text) semantics is defined in 10.3. Boxed expressions described in 10.2.2 can be mapped to FEEL text and thus also have execution semantics.

Recall that a `DecisionService` is defined by four lists: `inputData`, `inputDecisions`, `outputDecisions`, and `encapsulatedDecisions`. The lists are not independent and thus not all required to be specified, e.g., each required decision (direct and indirect) of the `outputDecisions` must be an `encapsulatedDecision`, an `inputDecision`, or required by an `inputDecision`. For simplicity in the following, we assume that all four lists are correctly and completely specified.

A `DecisionService` is given execution semantics by mapping it to a FEEL function F . Let S be a `DecisionService` with input data $id_1, id_2, \dots$, input decisions $di_1, di_2, \dots$, encapsulated decisions $de_1, de_2, \dots$, and output decisions $do_1, do_2, \dots$. Each input data id_i has a qualified name n_{id_i} . Each decision d_i has a qualified name n_{d_i} and a decision logic expression e_d . The decisions may have knowledge requirements. In particular the decisions may require `BusinessKnowledgeModels` $bkm_1, bkm_2, \dots$ and `DecisionServices` $s_1, s_2, \dots$. `BusinessKnowledgeModels` have qualified names n_{bkm_i} and `encapsulatedLogic` f_{bkm_i} . `DecisionServices` have qualified names n_{s_i} and equivalent logic f_{s_i} , where the equivalent logic is defined recursively, binding s_i to S .

The syntax for FEEL function F is $function(n_{id_1}, n_{id_2}, \dots, n_{di_1}, n_{di_2}, \dots) C.result$, where C is the context $\{$

$$\begin{aligned}
 &n_{s_1} : f_{s_1}, n_{s_2} : f_{s_2}, \dots, \\
 &n_{bkm_1} : f_{bkm_1}, n_{bkm_2} : f_{bkm_2}, \dots, \\
 &n_{de_1} : e_{de_1}, n_{de_2} : e_{de_2}, \dots, \\
 &result: \{ n_{do_1} : e_{do_1}, n_{do_2} : e_{do_2}, \dots \}
 \end{aligned}$$

such that s_i , bkm_i , de_i and do_i are partially ordered by requirements (e.g., the context entry for a required decision comes before a decision that requires it).

The qualified name of an element named E (decision, input data, decision service, or BKM) that is defined in the same decision model as S is simply E. Otherwise, the qualified name is I.E, where I is the name of the import element that refers to the model where E is defined.

The execution semantics of S is $FEEL(F)$: a function that when invoked with values from the FEEL semantic domain bound to the parameters representing input data and input decisions, returns:

- In the case of a single output decision(s), the single decision's output value.
- In the case of multiple output decisions, a context consisting of all the output decisions' output values.

XML elements SHALL map to the FEEL semantic domain as specified in section 10.3.3. Otherwise, details of the syntax of input/output data values and mapping to/from FEEL are undefined.

10.5 Metamodel

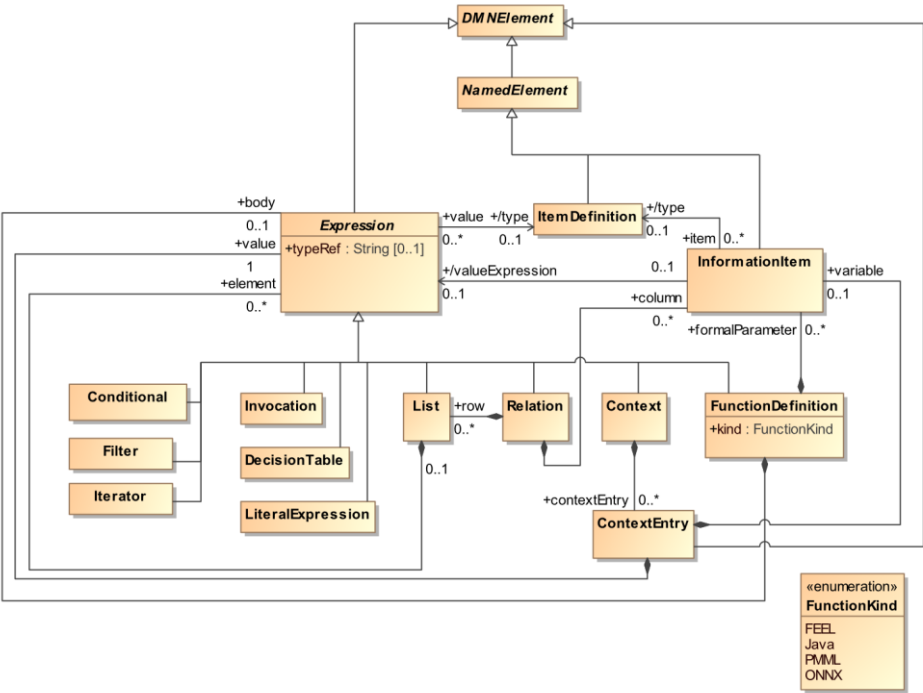


Figure 10-27: Expression class diagram

The class **Expression** is extended to support the four new kinds of boxed expressions introduced by FEEL, namely: **Context**, **FunctionDefinition**, **Relation** and **List**.

Boxed expressions are Expressions that have a standard diagrammatic representation (see clauses 7.2.1 and 10.2.1). FEEL contexts, function definitions, relations and lists SHOULD be modeled as **Context**, **FunctionDefinition**, **Relation** and **List** elements, respectively, and represented as a boxed expression

whenever possible; that is, when they are top-level expressions, since an instance of `LiteralExpression` cannot contain another `Expression` element.

10.5.1 Context metamodel

A `Context` is composed of any number of `contextEntries`, which are instances of `ContextEntry`.

A `Context` element is represented diagrammatically as a **boxed context** (clause 10.2.1.4). A FEEL *context* (grammar rule 57 and clause 10.3.2.6) SHOULD be modeled as a `Context` element whenever possible.

`Context` inherits all the attributes and model associations from `Expression`. Table 84 presents the additional attributes and model associations of the `Context` element.

Table 84: Context attributes and model association

Attribute	Description
contextEntry: <code>ContextEntry</code> [*]	This attributes lists the instances of <code>ContextEntry</code> that compose this <code>Context</code> .

10.5.2 ContextEntry metamodel

The class `ContextEntry` is used to model FEEL *context entries* when a *context* is modeled as a `Context` element. `ContextEntry` is a specialization of `DMNElement`, from which it inherits the optional `id`, `description`, and `label` attributes.

An instance of `ContextEntry` is composed of an optional variable, which is an `InformationItem` element whose name is the *key* in the *context entry*, and of a value, which is the instance of `Expression` that models the *expression* in the *context entry*.

Table 85 presents the attributes and model associations of the `ContextEntry` element.

Table 85: ContextEntry attributes and model associations

Attribute	Description
variable: <code>InformationItem</code> [0..1]	The instance of <code>InformationItem</code> that is contained in this <code>ContextEntry</code> , and whose name is the <i>key</i> in the modeled <i>context entry</i>
value: <code>Expression</code>	The instance of <code>Expression</code> that is the <i>expression</i> in this <code>ContextEntry</code>

10.5.3 FunctionDefinition metamodel

A `FunctionDefinition` has `formalParameters` and a body. A `FunctionDefinition` element is represented diagrammatically as a **boxed function**, as described in clause . A FEEL *function definition* (grammar rule 55 and clause 10.3.2.15) SHOULD be modeled as a `FunctionDefinition` element whenever possible.

`FunctionDefinition` inherits all the attributes and model associations from `Expression`. Table 86 presents the additional attributes and model associations of the `FunctionDefinition` element.

Table 86: FunctionDefinition attributes and model associations

Attribute	Description
FormalParameter: InformationItem [*]	This attributes lists the instances of InformationItem that are the parameters of this Context.
body: Expression [0..1]	The instance of Expression that is the body in this FunctionDefinition
kind: FunctionKind = FEEL { FEEL Java ONNX PMML }	The kind attribute defines the type of the FunctionDefinition. The default value is FEEL. Supported values also include Java, ONNX and PMML

10.5.4 List metamodel

A *List* is simply a list of element, which are instances of Expressions. A *List* element is represented diagrammatically as a **boxed list**, as described in clause 10.2.1.5. A FEEL *list* (grammar rule 54 and clause 10.3.2.15) SHOULD be modeled as a *List* element whenever possible.

List inherits all the attributes and model associations from *Expression*. Table 87 presents the additional attributes and model associations of the *List* element.

Table 87: List attributes and model associations

Attribute	Description
element: Expression [*]	This attributes lists the instances of Expression that are the elements in this List.

10.5.5 Relation metamodel

A *Relation* is convenient shorthand for a list of similar contexts. A *Relation* has a *column* instead of repeated ContextEntry's, and a *List* is used for every row, with one of the *List*'s expression for each column value.

Relation inherits all the attributes and model associations from *Expression*. Table 88 presents the additional attributes and model associations of the *Relation* element.

Table 88: Relation attributes and model associations

Attribute	Description
row: List [*]	This attributes lists the instances of List that compose the rows of this Relation.
column: InformationItem [*]	This attributes lists the instances of InformationItem that define the columns in this Relation.

10.5.6 Conditional metamodel

A `Conditional` is a visual way to express an if statement.

`Conditional` inherits all the attributes and model associations from `Expression`. Table 89 presents the additional attributes and model associations of the `Conditional` element.

Table 89: Conditional attributes and model associations

Attribute	Description
if: ChildExpression	This attribute holds the expression that is evaluate by the conditional expression.
then: ChildExpression	This attribute holds the expression that will be evaluated when the condition in the if statement evaluates to true .
else: ChildExpression	This attribute holds the expression that will be evaluated when the condition in the if statement evaluates to false .

10.5.7 ChildExpression metamodel

A `ChildExpression` is used to hold an expression inside a node. Table 90 presents the attributes of a `ChildExpression`.

Table 90: ChildExpression attributes and model associations

Attribute	Description
id: ID[0..1]	Optional identifier for this element. SHALL be unique within its containing <code>Definitions</code> element.
value: Expression	The instance of <code>Expression</code> that is the expression in this <code>ChildExpression</code>

10.5.8 Filter metamodel

A `Filter` is a visual way to express list filtering.

`Filter` inherits all the attributes and model associations from `Expression`. Table 90 presents the additional attributes and model associations of the `Filter` element.

Table 91: Filter attributes and model associations

Attribute	Description
in: ChildExpression	This attribute holds the expression that is evaluate as the collection to be filtered.
match: ChildExpression	This attribute holds the expression that is used to filter the collection.

10.5.9 Iterator metamodel

An `Iterator` is the abstract class for all boxed iteration.

`Iterator` inherits all the attributes and model associations from `Expression`. Table 92 presents the additional attributes and model associations of the `Iterator` element.

Table 92: `Iterator` attributes and model associations

Attribute	Description
iteratorVariable: String	This attribute holds name of the iterator variable that will be populated at each iteration.
in: TypedChildExpression	This attribute holds the expression that is evaluated as the collection to be processed.

10.5.10 For metamodel

A `For` is a representation of a loop.

`For` inherits all the attributes and model associations from `Iterator`. Table 93 presents the additional attributes and model associations of the `For` element.

Table 93: `For` attributes and model associations

Attribute	Description
return: ChildExpression	This attribute holds the expression that is evaluated to create the new collection that will be returned.

10.5.11 Quantified metamodel

A `Quantified` is an abstraction of an expression that is evaluated on each item of a collection.

`Quantified` inherits all the attributes and model associations from `Iterator`. Table 93 presents the additional attributes and model associations of `Quantified`.

Table 94: `Quantified` attributes and model associations

Attribute	Description
satisfies: ChildExpression	This attribute holds the expression that is evaluated to determine if the current item satisfies a condition.

10.5.12 Every metamodel

`Every` is an expression where all “satisfies” needs to be true for it to return true.

`Every` inherits all the attributes and model associations of `Quantified`.

10.5.13 Some metamodel

Some is an expression where at least one of the “satisfies” needs to be true for it to return true.

Some inherits all the attributes and model associations of Quantified.

10.6 Examples

A good way to get a quick overview of FEEL is by example.

FEEL expressions may reference other FEEL expressions by name. Named expressions are contained in a context. Expressions are evaluated in a scope, which is a list of contexts in which to resolve names. The result of the evaluation is an element in the FEEL semantic domain.

10.6.1 Context

Figure 10-28 shows the boxed context used for the examples. Such a context could arise in several ways. It could be part of the decision logic for a single, complex decision. Or it could be a context that is equivalent to part of a DRG as defined in clause 10.4, where applicant, requested product, and credit history are input data instances, monthly income and monthly outgoings are the results of other decisions linked through information requirements, and PMT is a business knowledge model.

applicant	age	51	
	maritalStatus	"M"	
	existingCustomer	false	
	monthly	income	10000
		repayments	2500
		expenses	3000
requested product	product type		"STANDARD LOAN"
	rate		0.25
	term		36
	amount		100000.00
monthly income	applicant.monthly.income		
monthly outgoings	applicant.monthly.repayments, applicant.monthly.expenses		
credit history	record date	event	weight
	date("2008-03-12")	"home mortgage"	100
	date("2011-04-01")	"foreclosure warning"	150
PMT	(rate, term, amount)		
	(amount *rate/12) / (1 - (1 + rate/12)**-term)		

Figure 10-28: Example context

Notice that there are 6 top-level context entries, represented by the six rows of the table. The value of the context entry named 'applicant' is itself a context, and the value of the context entry named 'monthly' is itself a context. The

value of the context entry named 'monthly outgoings' is a list, the value of the context entry named 'credit history' is a relation, *i.e.*, a list of two contexts, one context per row. The value of the context entry named 'PMT' is a function with parameters 'rate', 'term', and 'amount'.

The following examples use the above context. Each example has a pair of equivalent FEEL expressions separated by a horizontal line. Both expressions denote the same element in the semantic domain. The second expression, the 'answer', is a literal value.

10.6.2 Calculation

```
monthly income * 12  
120000
```

The context defines *monthly income* as *applicant.monthly.income*, which is also defined in the context as 10,000. Twelve times the *monthly income* is 120,000.

10.6.3 If, In

```
if applicant.maritalStatus in ("M", "S") then "valid" else "not valid"  
"valid"
```

The *in* test determines if the left-hand side expression satisfies the list of values or ranges on the right-hand side. If satisfied, the *if* expression returns the value of the *then* expression. Otherwise, the value of the *else* expression is returned.

10.6.4 Sum entries of a list

```
sum (monthly outgoings)  
5500
```

Monthly outgoings is computed in the context as the list [*applicant.monthly.repayments*, *applicant.monthly.expenses*], or [2500, 3000]. The square brackets are not required to be written in the boxed context.

10.6.5 Invocation of user-defined PMT function

The PMT function defined in the context computes the monthly payments for a given interest rate, number of months, and loan amount.

```
PMT (requested product . rate,  
      requested product . term,  
      requested product . amount)  
3975.982590125552338278440100112431
```

A function is invoked textually using a parenthesized argument list after the function name. The arguments are defined in the context, and are 0.25, 36, and 100,000, respectively.

10.6.6 Sum weights of a recent credit history

```
sum (credit history[record date > date ("2011-01-01")].weight)  
150
```

This is a complex "one-liner" that will be useful to expand into constituent sub-expressions:

- built-in: *sum*
- path expression ending in *.weight*
- filter: *[record date > date("2011-01-01 ")]*
- name resolved in context: *credit history*

An expression in square brackets following a list expression filters the list. *Credit history* is defined in the context as a relation, that is, a list of similar contexts. Only the last item in the relation satisfies the filter. The first item is too old. The path expression ending in *.weight* selects the value of the *weight* entry from the context or list of contexts satisfied by the filter. The *weight* of the last item in the credit history is 150. This is the only item that satisfies the filter, so the sum is 150 as well.

10.6.7 Determine if credit history contain a bankruptcy event

```
Some ch in credit history satisfies ch.event = "bankruptcy"  
false
```

The *some* expression determines if at least one element in a list or relation satisfies a test. There are no bankruptcy events in the credit history in the context.

This page intentionally left blank.

11 B-FEEL

11.1 Introduction

DMN defines the friendly enough expression language (FEEL) for the purpose of giving standard executable semantics to many kinds of expressions in a decision model (see chapter 10).

This chapter defines a dialect of FEEL: B-FEEL (Business Friendly Enough Expression Language). B-FEEL shares the same grammar as FEEL but alters the semantics to be friendlier and more intuitive toward non-IT users.

In FEEL, the **null** value is used to both represent missing data or an execution error. In B-FEEL, **null** is used only to represent missing data. All operations and built-in functions that returns null in FEEL when an error occurs have their semantics modified in B-FEEL to return a non-**null** value. A warning message should still be produced when an error occurs.

To use B-FEEL instead of FEEL, the expression language must be set to: ["https://www.omg.org/spec/DMN/B-FEEL/"](https://www.omg.org/spec/DMN/B-FEEL/) If FEEL is used in other languages, e.g. as an expression language in a BPMN process model, a version number MUST be added to the URL, i.e. ["https://www.omg.org/spec/DMN/B-FEEL/1.7"](https://www.omg.org/spec/DMN/B-FEEL/1.7). ["https://www.omg.org/spec/DMN/20240513/B-FEEL/"](https://www.omg.org/spec/DMN/20240513/B-FEEL/)

The following sections present the semantics of B-FEEL and compare it to the semantics of FEEL. Anything not covered in this chapter has the behavior described for FEEL in Chapter 10.

11.2 Operator and built-in functions returning a Boolean

Removing **null** as an error from the result of operators and built-in functions for boolean values makes B-FEEL a two-value logic (**true** and **false**) compared to the three-value logic of FEEL (**true**, **false** and **null**).

In B-FEEL boolean operators (=, <=, <, >, >=, not(), and, or, in, between) always return a true or false result (never **null**) even when incompatible types are used in their expression.

In B-FEEL an incompatible type in a boolean expression is considered false with the exception of the not equal (!=) where it is considered true.

Expression	FEEL	B-FEEL
"a" = 1	null	false
"a" < 1	null	false
"a" <= null	null	false
"a" > 1	null	false
null >= 1	null	false
not("a")	null	false
true and "x"	null	false
false or "x"	null	false
"a" in [1..100]	null	false

Commented [AF47]: DMN17-65 (interpreted by editor as applying to page 169 (PDF 179) rather than page 269 (PDF 279)).

null between 1 and 100	null	false
“a” != 1	null	true

Several FEEL built-in functions return a boolean result. In B-FEEL, those functions’ semantics are modified to return **false** everywhere FEEL would return **null** for them.

Expression	FEEL	B-FEEL
matches("bad pattern","[0-9")	null	false
before(date("2021-01-01"), null)	null	false
all(true,"x",true)	null	false
any(null)	null	false

11.3 Built-in functions returning a number

Several FEEL built-in functions return a numeric result. In B-FEEL, those functions’ semantics are modified to return 0 everywhere FEEL would return **null** for them.

In addition, the list functions that return a number (mean(), median(), product(), stddev(), sum()) except count() ignore non-numeric parameters passed in their input list in B-FEEL.

Expression	FEEL	B-FEEL
decimal(“a”, 0)	null	0
round up(“5.5”, 0)	null	0
string length(22)	null	0
day of year(“a”)	null	0
count([1,null,3])	3	3
sum([1, null, 3])	null	4
sum([1, ”1” ,3])	null	4
sum([])	null	0
mean([“a”])	null	0
mean([1, “a”, 3])	null	2

The C+ decision table policy being defined as the sum of the outputs yields a slightly different result in B-FEEL because the sum function semantics are altered.

11.4 Built-in functions returning a string

Several FEEL built-in functions return a string result. Those methods' semantics in B-FEEL are modified to return an empty string ("") everywhere FEEL would return **null** for them.

Expression	FEEL	B-FEEL
lowercase(12)	null	""
string(null)	null	""
day of week("a")	null	""
substring("a", "z")	null	""

11.5 Built-in functions returning a date and time, date and time

Several FEEL built-in functions return a date and time, date or time result. In B-FEEL, those functions' semantics are modified to return January 1st of year 1970 (1970-01-01T00:00:00+00:00) value (epoch) everywhere FEEL would return **null** for them.

The default values are based on the return type:

Type	Default Value
Date and time	date and time("1970-01-01T00:00:00+00:00")
Date	date("1970-01-01")
Time	time("00:00:00+00:00")

Expression	FEEL	B-FEEL
time("a")	null	time("00:00:00+00:00")
date(null)	null	date("1970-01-01")

11.6 Built-in functions returning a duration

Several FEEL built-in functions return a duration result. In B-FEEL, those functions' semantics are modified to return a duration of 0 months (years and months durations) or 0 seconds (days and time duration) everywhere FEEL would return **null** for them.

Expression	FEEL	B-FEEL
duration("a")	null	duration("P0M")
years and months duration(null, null)	null	duration("P0M")

days and time duration("a")	null	days and time duration("POS")
years and months duration("a")	null	years and months duration("POM")

Commented [AF48]: DMN17-72

11.7 Built-in functions returning a collection and expressions using a collection

Several FEEL built-in functions return collection results. In B-FEEL, those functions' semantics are modified to return an empty collection everywhere FEEL would return **null** for them.

The *mode* function additionally ignores non-numeric parameters passed in their input list in B-FEEL.

Some specific FEEL grammar expression can reference collections that could be **null**. In B-FEEL the execution semantic is altered as follow:

- for expression (grammar rule 44), when the iteration context is evaluated to **null**, the expression returns an empty collection.
- for quantified expression (grammar rule 46), when one of the expression is evaluated to **null**, the expression returns **false**.
- for comparison using "in" (grammar rules 49c, 49d), when the positive unary test is evaluated to **null**, the expression returns **false**.

Commented [AF50]: DMN17-112

Expression	FEEL	B-FEEL
split("abc", 22)	null	[]
mode([null,null,null, 1, 1,2])	null	[1]
for x in null return x	null	[]
some x in null satisfies x>0	null	[]
1 in null	null	false

Commented [AF51]: DMN17-112

11.8 Built-in functions returning a range

The FEEL built-in function range() returns a range result. In B-FEEL, that function's semantics is modified to return an empty range that does not match anything ((0..0)) where in FEEL it would return **null**.

Expression	FEEL	B-FEEL
range("x")	null	range("(0..0)")

11.9 Semantics of addition and subtraction

In B-FEEL, the semantics of addition and subtraction are modified when the types of e1 and e2 in Table 57 do not match.

The following rules are added after the rules in Table 58 in order of precedence:

If type(e_1) or type(e_2) is ...	$e_1 + e_2 / e_1 - e_2$
string	The non-string value is converted to a string using the string B-FEEL function and Table 58 applies. Subtraction returns an empty string.
number	The non-number value is converted to a number using the number B-FEEL function and Table 58 applies.
date and time	The non-date and time value is converted to a duration using the duration B-FEEL function and Table 58 applies.
date	The non-date value is converted to a duration using the duration B-FEEL function and Table 58 applies.
time	The non-time value is converted to a duration using the duration B-FEEL function and Table 58 applies.
years and months duration	The non-years and months duration value is converted to a duration using the duration B-FEEL function and Table 58 applies.
days and time duration	The non-days and time duration value is converted to a duration using the duration B-FEEL function and Table 58 applies.

Expression	FEEL	B-FEEL
"Today is " + today()	null	"Today is 2020-01-01"
"The result is: " + 1	null	"The result is: 1"
5 + " minutes"	null	"5 minutes"
"This is " + null	null	"This is "
1 + null	null	1
null - 6	null	-6
date("2021-01-01") + 7	null	7
"abc" - 2	null	""

11.10 Semantics of multiplication and division

In B-FEEL, the semantics of multiplication and division are modified when the types of e_1 and e_2 in Table 59 do not match.

The following rules are added after the rules in Table 60 in order of precedence:

If type(e_1) or type(e_2) is ...	$e_1 * e_2$ and e_1 / e_2
number	The non-number type is converted to a number using the number B-FEEL function and Table 60 applies.

years and months duration	The non-years and months duration type is converted to a number using the number B-FEEL function and Table 60 applies.
days and time duration	The non-days and time duration type is converted to a number using the number B-FEEL function and Table 60 applies.

Expression	FEEL	B-FEEL
22 * "a"	null	0
null / 22	null	0
duration("P1Y") * null	null	duration("P0M")

11.11 Semantics of exponentiation

The FEEL semantics of exponentiation described in Table 61 (grammar rule 23) are used and B-FEEL further specifies that each operand is converted to a number using the number B-FEEL function.

This page intentionally left blank.

12 DMN Examples

12.1 Example 1: Originations

12.1.1 Introduction

In this clause we present an example of the use of **DMN** to model and execute decision-making in a simple business process modeled in **BPMN**, including decisions to be automated in decision services called from the business process management system.

12.1.2 The business process model

Figure 12-1 shows a simple process for loan originations, modeled in **BPMN 2.0**. The process handles as application for a loan, obtaining data from a credit bureau only if required for the case, and automatically deciding whether the application should be accepted, declined, or referred for human review. If referred, documents are collected from the applicant and a credit officer adjudicates the case. It consists of the following components:

- The **Collect application data** task collects data describing the Requested product and the Applicant (e.g., through an on-line application form).
- The **Decide bureau Strategy** task calls a decision service, passing Requested product and Applicant data. The service returns two decisions: Strategy and Bureau call type.
- A **gateway** uses the value of Strategy to route the case to Decline application, Collect bureau data or Decide routing.
- The **Collect bureau data** task collects data from a credit bureau according to the Bureau call type decision, then the case is passed to Decide routing.
- The **Decide routing** task calls a decision service, passing Requested product, Applicant data and Bureau data (if the Collect bureau data task was not performed, the Bureau data are set to null). The service returns a single decision: Routing.
- A **gateway** uses the value of Routing to route the case to Accept application, Review application or Decline application.
- The **Collect documents** task requests and uploads documents from the applicant in support of their application.
- The **Review application** task allows a credit officer to review the case and decide whether it should be accepted or declined.
- A **gateway** uses the credit officer's Adjudication to route the case to Accept application or Decline application.
- The **Accept application** task informs the applicant that their application is accepted and initiates the product.
- The **Decline application** task informs the applicant that their application is declined.

Note that in this example two decision points (automated as calls to decision services) are represented in **BPMN 2.0** as business rule tasks; the third decision point (which is human decision-making) is represented as a user task.

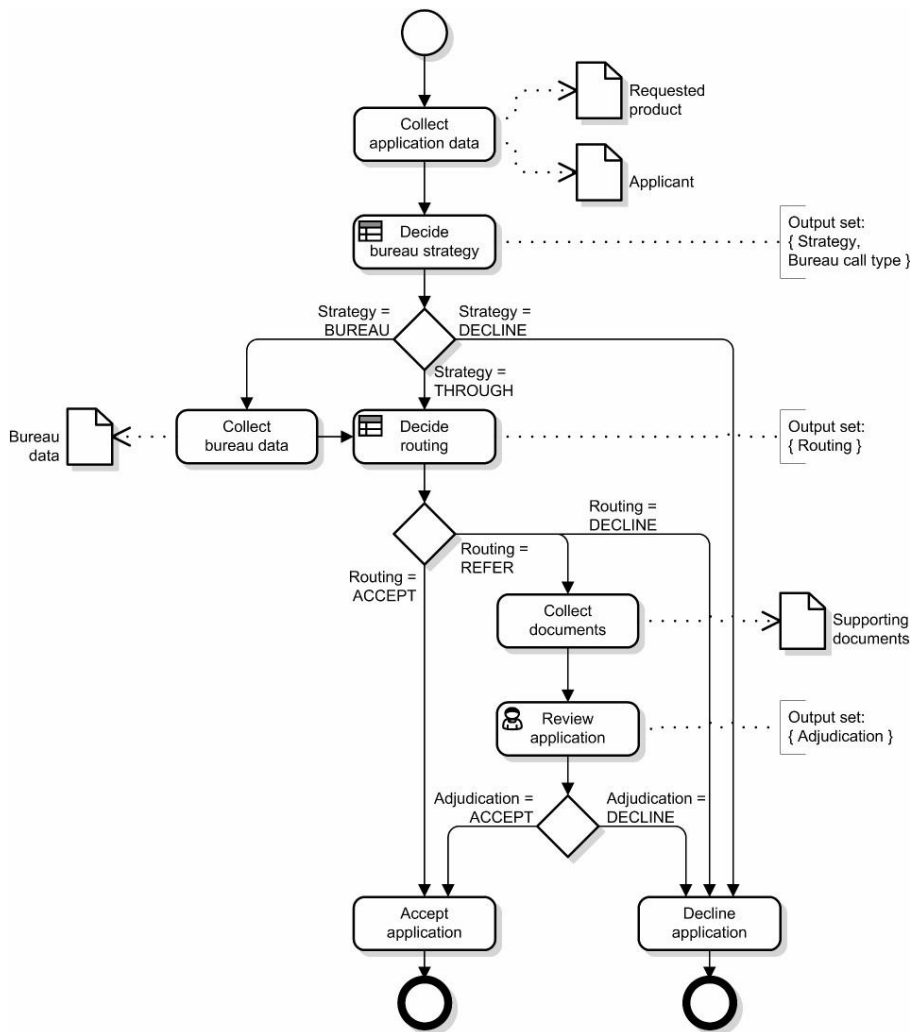


Figure 12-1: Example business process

Commented [AF52]: Editorial: corrected figure numbering throughout section 12.

12.1.3 The decision requirements level

The examples in this chapter were developed using a software that adds icons to the elements. Although adding these icons is allowable by this document it is not normative.

12.1.3.1 Decision Requirements Diagrams

Figure 12-2 shows a DRD of all the decision-making in this business process. There are four sources of input data for the decision-making (Requested product, Applicant data, Bureau data and Supporting documents), and four decisions whose results are used in the business process (Strategy, Bureau call type, Routing and Adjudication). Between the two are intermediate decisions: evaluations of risk, affordability, and eligibility.

Notable features of this DRD include:

- It covers both automated and human decision-making.
- Some decisions (e.g., Pre-bureau risk category) and input data (e.g., Applicant data) are required by multiple decisions, i.e., the information requirements network is not a tree.
- Business knowledge models (see Affordability calculation) may be invoked by multiple decisions.
- Business knowledge models (see Credit contingency factor) may be invoked by other business knowledge models.
- Some decisions do not have associated business knowledge models.

Knowledge sources may provide authority for multiple decisions and/or business knowledge models.

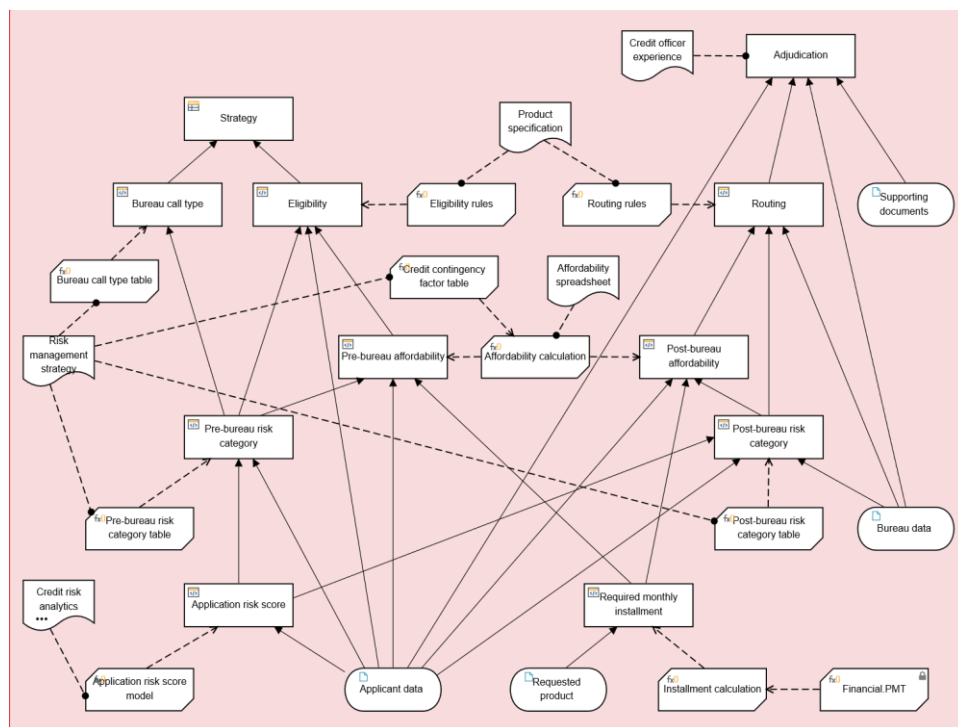


Figure 12-2: DRD of all automated decision-making

It might be considered more convenient to draw separate (but overlapping) DRDs for the three decision points:

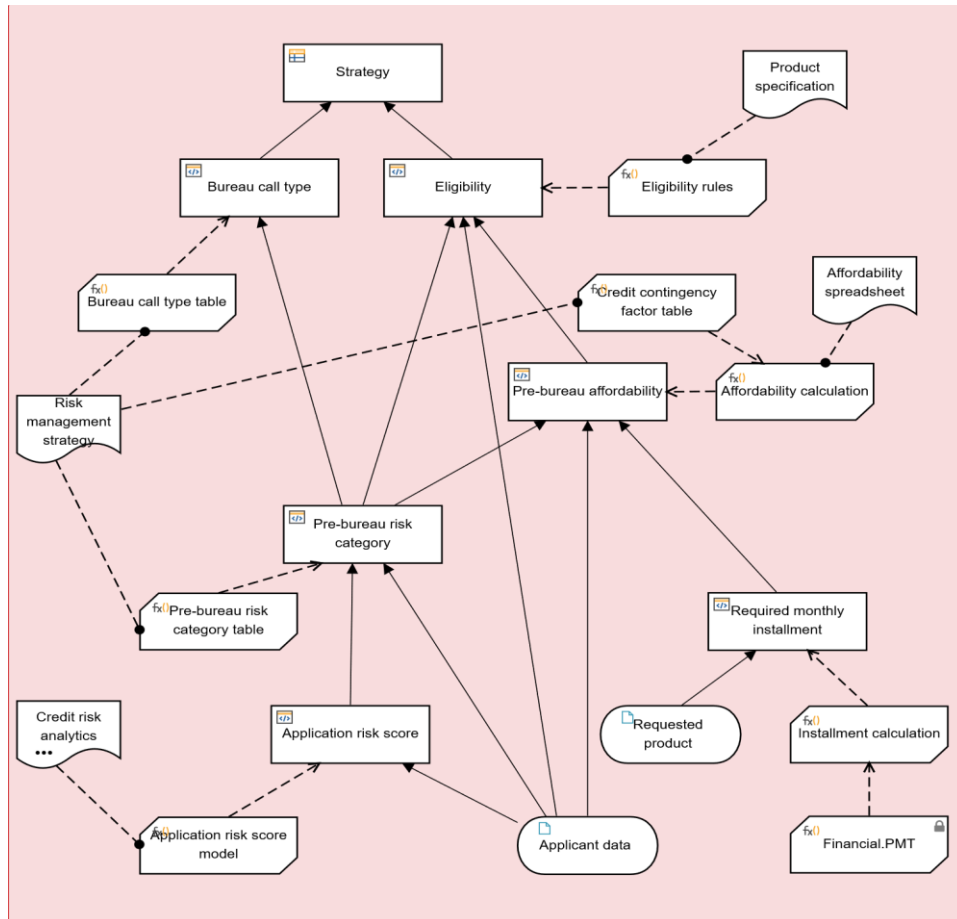
- Figure 12-3 shows the DRD of the decisions required for the Decide bureau strategy decision point, i.e., the requirements subgraph of the Strategy and Bureau call type decisions. These are decisions to be automated through encapsulation in a decision service called at this point and therefore need their logic to be specified completely.
- Figure 12-4 shows the DRD for the Decide routing decision point, i.e., the requirements subgraph of the Routing decision. These are also decisions automated with a decision service and therefore need their logic to be specified completely. Note that some elements appear in both Figure 12-3 and Figure 12-4.
- Figure 12-5 shows the DRD for the Review application decision point, i.e., the requirements subgraph of the Adjudication decision. This is a human decision and has no associated specification of decision logic, but the DRD indicates that the Credit officer takes into account the results of the automated Routing

Commented [AF53]: DMN17-197

decision along with the case data, including the Supporting documents. (The requirements subgraph of the Routing decision has been hidden in this DRD as shown by the ellipsis (...) marker.)

- Figure 12-6 shows an additional DRD for the Credit Risk Analytics Knowledge Source i.e., the requirements linking this Knowledge Source to other elements. DRDs can be used to provide views other than for a specific decision.

All four DRDs – Figure 12-2, Figure 12-3, Figure 12-4, Figure 12-5 and Figure 12-6 – are views of the same DRG.



Commented [AF54]: DMN17-197

Figure 12-3: DRD for Decide bureau strategy decision point

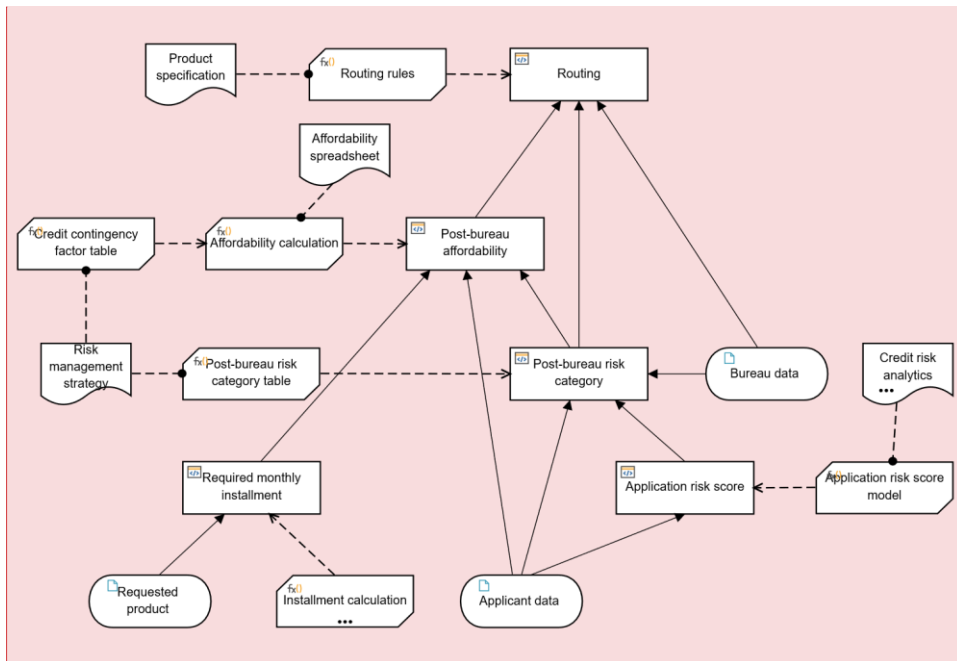


Figure 12-4: DRD for Decide routing decision point

Commented [AF55]: DMN17-197

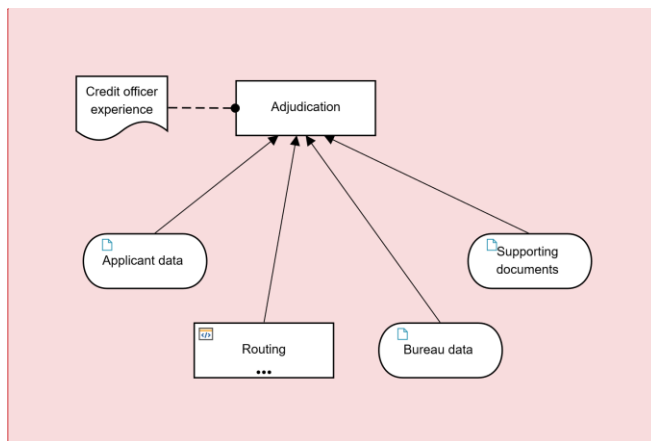


Figure 12-5: DRD for Review application decision point

Commented [AF56]: DMN17-197

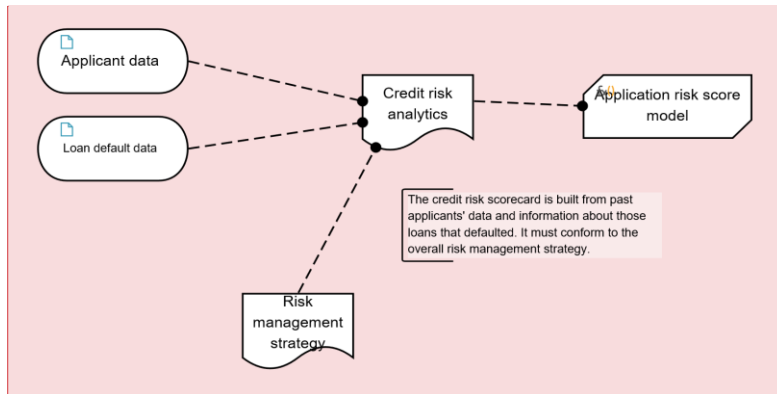


Figure 12-6: DRD for Credit Risk Analytics Knowledge Source

12.1.3.2 DRG Elements

12.1.3.2.1 Decisions

The DRG depicted in these DRDs shows dependencies between the following decisions:

- The **Strategy** decision, requiring the Bureau call type and Pre-bureau eligibility decisions, invokes the Strategy table shown in Figure 12-9 (without that table being encapsulated in a business knowledge model).
- The **Bureau call type** decision, requiring the Pre-bureau risk category decision, invokes the Bureau call type table shown in Figure 12-11.
- The **Eligibility** decision, requiring Applicant data and the Pre-bureau risk category and Pre-bureau affordability decisions, invokes the Eligibility rules shown in Figure 12-13.
- The **Pre-bureau affordability** decision, requiring Applicant data and the Pre-bureau risk category and Required monthly installment decisions, invokes the Affordability calculation boxed expression shown in Figure 12-24, which in turn invokes the Credit contingency factor table shown in Figure 12-25.
- The **Pre-bureau risk category** decision, requiring Applicant data and the Application risk score decision, invokes the Pre-bureau risk category table shown in Figure 12-15.
- The **Application risk score** decision, requiring Applicant data, invokes the Application risk score model shown in Figure 12-17.
- The **Routing** decision, requiring Bureau data and the Post-bureau affordability and Post-bureau risk category decisions, invokes the Routing rules shown in Figure 12-19.
- The **Post-bureau affordability** decision, requiring Applicant data and the Post-bureau risk score and Required monthly installment decisions, invokes the Affordability calculation boxed expression shown in Figure 12-24, which in turn invokes the Credit contingency factor table shown in Figure 12-25.
- The **Post-bureau risk category** decision, requiring Applicant and Bureau data and the Application risk score decision, invokes the Post-bureau risk category table shown in Figure 12-21.
- The **Required monthly installment** decision, requiring Requested product data, invokes the Installment calculation boxed expression shown in Figure 12-27.
- The **Adjudication** decision, requiring Applicant data, Bureau data, Supporting documents, and the Routing decision, has no associated decision logic.

Commented [AF57]: DMN17-197

Questions and allowed answers are specified for these decisions. These are typically used when modeling decisions for which no logic will be specified and for other decisions before it is appropriate to describe the decision logic in detail. The description and Question/Allowed Answers for each decision follow.

Adjudication

Question: Should this application that has been referred for adjudication be accepted? Allowed Answers: Yes/No

Description: Determine if an application requiring adjudication should be accepted or declined given the available application data and supporting documents.

Application risk score

Question: What is the risk score for this applicant?

Allowed Answers: A number greater than 70 and less than 150

Description: The **Application Risk Score** decision logic invokes the Application risk score model business knowledge model, passing Applicant data.Age as the Age parameter, Applicant data.MaritalStatus as the Marital Status parameter and Applicant data.EmploymentStatus as the Employment Status parameter.

Bureau call type

Question: How much data should be requested from the credit bureau for this application? Allowed Answers: A value from the explicit list "Full", "Mini", "None"

Description: The **Bureau call type** decision logic invokes the Bureau call type table, passing the output of the Prebureau risk category decision as the Pre-Bureau Risk Category parameter.

Eligibility

Question: Does this applicant appear eligible for the loan they applied for given only their application data?

Allowed Answers: Value from the explicit list "Eligible", "Not Eligible"

Description: The **Eligibility** decision logic invokes the Eligibility rules business knowledge model, passing Applicant data.Age as the Age parameter, the output of the Pre-bureau risk category decision as the Pre-Bureau Risk Category parameter, and the output of the Pre-bureau affordability decision as the Pre-Bureau Affordability parameter.

Pre-bureau affordability

Question: Can the applicant afford the loan they applied for given only their application data?

Allowed Answers: Yes/No

Description: The **Pre-bureau affordability** decision logic invokes the Affordability calculation business knowledge model, passing Applicant data.Monthly.Income as the Monthly Income parameter, Applicant data.Monthly.Repayments as the Monthly Repayments parameter, Applicant data.Monthly.Expenses as the Monthly Expenses parameter, the output of the Pre-bureau risk category decision as the Risk Category parameter, and the output of the Required monthly installment decision as the Required Monthly Installment parameter.

Post-bureau affordability

Question: Can the applicant afford the loan they applied for given all available data?

Allowed Answers: Yes/No

Description: The **Post-bureau affordability** decision logic invokes the Affordability calculation business knowledge model, passing Applicant data.Monthly.Income as the Monthly Income parameter, Applicant

data.Monthly.Repayments as the Monthly Repayments parameter, Applicant data.Monthly.Expenses as the Monthly Expenses parameter, the output of the Post-bureau risk category decision as the Risk Category parameter, and the output of the Required monthly installment decision as the Required Monthly Installment parameter.

Pre-bureau risk category

Question: Which risk category is most appropriate for this applicant given only their application data?

Allowed Answers: Value from explicit list "Decline", "High Risk", "Medium Risk", "Low Risk", "Very Low Risk"

Description: The Pre-Bureau Risk Category decision logic invokes the Pre-bureau risk category table business knowledge model, passing Applicant data.ExistingCustomer as the Existing Customer parameter and the output of the Application risk score decision as the Application Risk Score parameter.

Post-bureau risk category

Question: Which risk category is most appropriate for this applicant given all available data?

Allowed Answers: A value from the explicit list "Decline", "High Risk", "Medium Risk", "Low Risk", "Very Low Risk"

Description: The **Post-bureau risk category** decision logic invokes the Post-bureau risk category business knowledge model, passing Applicant data.ExistingCustomer as the Existing Customer parameter, Bureau data.CreditScore as the Credit Score parameter, and the output of the Application risk score decision as the Application Risk Score parameter. Note that if Bureau data is null (due to the THROUGH strategy bypassing the Collect bureau data task) the Credit Score parameter will be null.

Required monthly installment

Question: What is the minimum monthly installment payment required for this loan product? Allowed Answers: A dollar amount greater than zero

Description: The **Required monthly installment** decision logic invokes the Installment calculation business knowledge model, passing Requested product.ProductType as the Product Type parameter, Requested product.Rate as the Rate parameter, Requested product.Term as the Term parameter, and Requested product.Amount as the Amount parameter.

Routing

Question: How this should this applicant be routed given all available data?

Allowed Answers: A value from the explicit list "Decline", "Refer for Adjudication", "Accept without Review"

Description: The **Routing** decision logic invokes the Routing rules business knowledge model, passing Bureau data.Bankrupt as the Bankrupt parameter, Bureau data.Credit Score as the Credit Score parameter, the output of the Post-bureau risk category decision as the Post-Bureau Risk Category parameter, and the output of the Post-bureau affordability decision as the Post-Bureau Affordability parameter. Note that if Bureau data is null (due to the THROUGH strategy bypassing the Collect bureau data task) the Bankrupt and Credit Score parameters will be null.

Strategy

Question: What is the appropriate handling strategy for this application?

Allowed Answers: A value from the explicit list "Decline", "Bureau", "Through"

Description: The **Strategy** decision logic defines a complete, unique-hit decision table deriving Strategy from Eligibility and Bureau call type.

12.1.3.2.2 Knowledge Sources

The DRG contains the following Knowledge Sources:

Affordability spreadsheet

Description: Internal spreadsheet showing the relationship of income, payments, expenses, risk, and affordability.

Type: Policy

Credit officer experience

Description: The collected wisdom of the credit officers as collected in their best practice wiki.

Type: Expertise

Credit risk analytics

Description: Credit risk scorecard analysis to determine the relevant factors for application risk scoring

Type: Analytic Insight

Product specification

Description: Definitions of the products, their cost structure and eligibility criteria.

Type: Policy

Risk management strategy

Description: Overall risk management approach for the financial institution including its approach to application risk, credit contingencies and credit risk scoring.

Type: Policy

12.1.3.2.3 Input Data

The DRG contains the following Input Data:

Applicant data

Description: Information about the applicant including personal information, marital status, and household income/expenses.

Bureau data

Description: External credit score and bankruptcy information provided by a bureau.

Loan default data

Description: Information about historical loan defaults.

Requested product

Description: Details of the loan the applicant has applied for.

Supporting documents

Description: Documents associated with a loan that are not processed electronically but are available for manual adjudication.

12.1.3.2.4 Business Knowledge Models

Finally, the DRG contains the following Business Knowledge Models:

Eligibility rules

Description: The Eligibility rules decision logic defines a complete, priority-ordered single hit decision table deriving Eligibility from Pre-Bureau Risk Category, Pre-Bureau Affordability and Age.

Routing rules

Description: The Routing Rules decision logic defines a complete, priority-ordered single hit decision table deriving Routing from Post-Bureau Risk Category, Post-Bureau Affordability, Bankrupt and Credit Score.

Bureau call type table

Description: The Bureau call type table decision logic defines a complete, unique-hit decision table deriving Bureau Call Type from Pre-Bureau Risk Category.

Credit contingency factor table

Description: The Credit contingency factor table decision logic defines a complete, unique-hit decision table deriving Credit contingency factor from Risk Category.

Affordability calculation

Description: The Affordability calculation decision logic defines a boxed function deriving Affordability from Monthly Income, Monthly Repayments, Monthly Expenses and Required Monthly Installment. One step in this calculation derives Credit contingency factor by invoking the Credit contingency factor table business.

Pre-bureau risk category table

Description: The Pre-bureau risk category table decision logic defines a complete, unique-hit decision table deriving Pre-bureau risk category from Existing Customer and Application Risk Score.

Post-bureau risk category table

Description: The Post-bureau risk category table decision logic defines a complete, unique-hit decision table deriving Post-Bureau Risk Category from Existing Customer, Application Risk Score and Credit Score.

Application risk score model

Description: The Application risk score model decision logic defines a complete, no-order multiple-hit table with aggregation, deriving Application risk score from Age, Marital Status and Employment Status, as the sum of the Partial scores of all matching rows (this is therefore a predictive scorecard represented as a decision table).

Installment calculation

Description: The Installment calculation decision logic defines a boxed function deriving monthly installment from Product Type, Rate, Term and Amount.

Financial.PMT

Description: Standard calculation of monthly installment from Rate, Term and Amount.

12.1.3.3 Business Context

In addition to the information represented in the DRD, the business context of the decision-making can be specified. The Performance Indicators used to track the effectiveness of decision-making, Objectives the organization seeks to meet through its decision-making approach, and the Organizational Units that make decisions or own the decision making approach may all be specified. Decisions are cross-referenced to the performance indicators and objectives they impact and to the organizational units that either make the decision or own the definition of how the decision should be made.

Performance indicators

Monthly bureau costs	The total cost charged by the bureau for all Bureau Data requested while originating Loans in a calendar month.
Monthly loan accept rate	The percentage of loans accepted in a calendar month.
Monthly auto-adjudication rate	The percentage of loans that did not require a credit officer to review the case in a calendar month.
Monthly value of loans written	The total value of Loans written in a calendar month
Auto adjudication rate 90%	By end of the current year, have an auto-adjudication rate of at least 90 percent

Decisions are mapped to the Performance Indicators and Goals that they impact as follows:

	Monthly Loan Accept Rate	Monthly Value of Loans Written	Monthly Bureau Costs	Auto-adjudication rate 90%	Monthly Auto-adjudication Rate
Adjudication	Yes	Yes			
Application risk score			Yes		
Bureau Call Type			Yes		
Routing	Yes	Yes		Yes	Yes
Strategy	Yes	Yes		Yes	Yes

Organizations

Credit officers	Individuals in the Retail Banking Organization responsible for manual adjudication of loans.
Product management	Organization responsible for defining loan and other banking products, how those products are priced, sold and tracked for profitability.
Credit risk analytics group	Organization responsible for credit risk models and the use of data to predict credit risk for customers and loan applicants.
Retail banking	Overall Organization focused on banking products for consumers.
Credit risk	Organization within the bank responsible for defining credit risk strategies and policies and providing tools for managing against these.

Credit officers are likely to be part of the Retail Banking organization, Credit risk analytic and Risk management are part of the Credit risk organization, although these relationships are not managed in DMN.

These organizations own decisions, make decisions and own knowledge sources as follows:

	Owns Decisions	Makes Decisions	Knowledge Sources
--	----------------	-----------------	-------------------

Credit officers		Adjudication	Credit officer experience
Credit risk analytics group	Application risk score		Credit risk analytics
Credit risk	Adjudication Bureau call type Eligibility Pre-bureau risk category Post-bureau risk category Routing		Risk management strategy

12.1.3.4 Decision Services

The two decision services required by the business process model are defined against the decision model. The **Bureau Strategy Decision Service**, called by the **Decide bureau strategy** task, has output decisions {Bureau call type, Strategy}, and is shown in Figure 12-7. The **Routing Decision Service**, called by the **Decide routing** task, has output decisions {Routing}, and is shown in Figure 12-8.

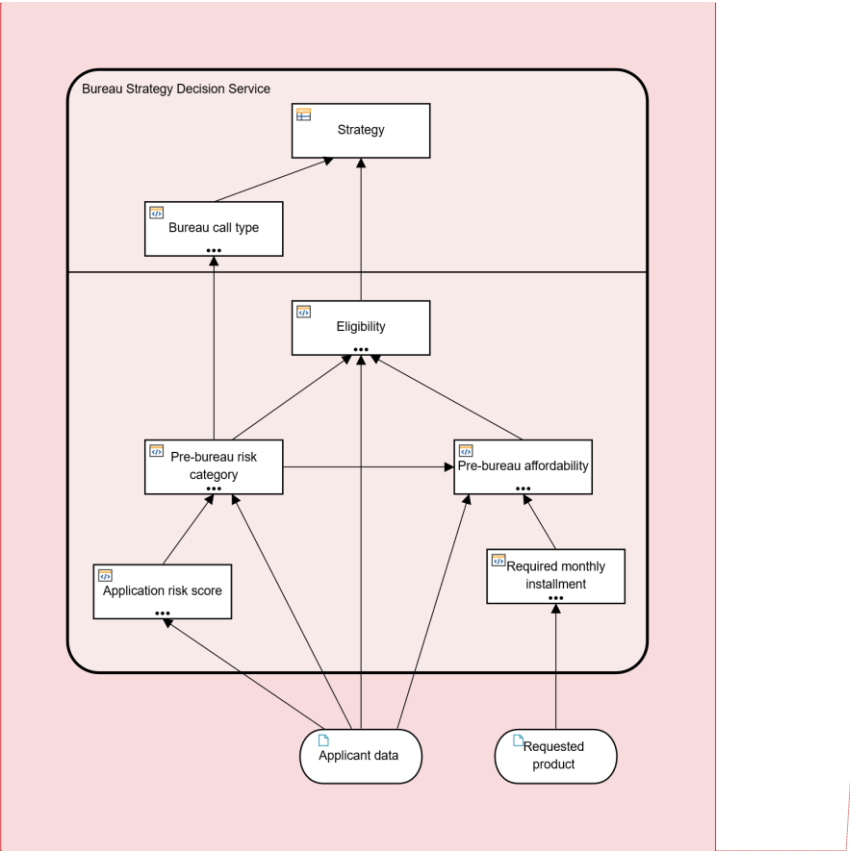


Figure 12-7: Bureau Strategy Decision Service

Commented [AF58]: DMN17-197

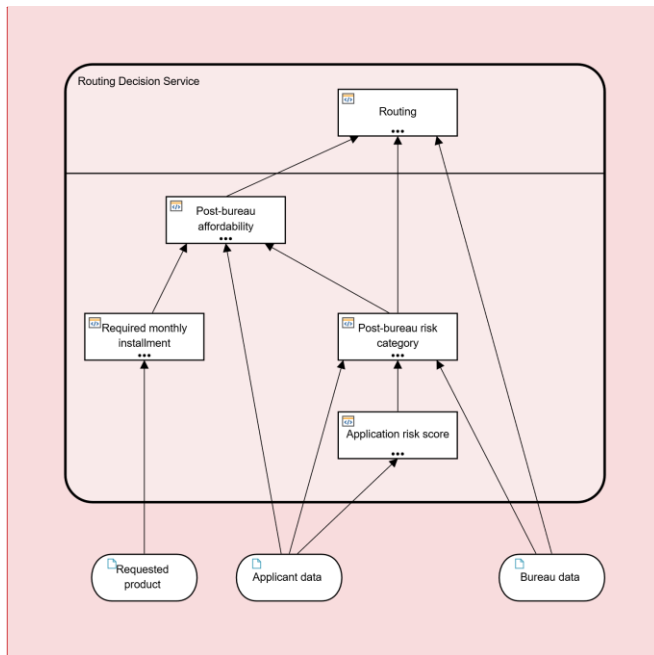


Figure 12-8: Routing Decision Service

12.1.4 The decision logic level

The DRG in Figure 12-2 is defined in more detail in the following specifications of the value expressions associated with decisions and business knowledge models:

- The **Strategy** decision logic (Figure 12-9) defines a complete, unique-hit decision table deriving Strategy from Eligibility and Bureau call type.
- The **Bureau call type** decision logic (shown as a boxed invocation in Figure 12-10) invokes the Bureau call type table, passing the output of the Pre-bureau risk category decision as the Pre-Bureau Risk Category parameter.
- The **Bureau call type table** decision logic (Figure 12-11) defines a complete, unique-hit decision table deriving Bureau Call Type from Pre-Bureau Risk Category.
- The **Eligibility** decision logic (shown as a boxed invocation in Figure 12-12) invokes the Eligibility rules business knowledge model, passing Applicant data. Age as the Age parameter, the output of the Pre-bureau risk category decision as the Pre-Bureau Risk Category parameter, and the output of the Pre-bureau affordability decision as the Pre-Bureau Affordability parameter.
- The **Eligibility rules** decision logic (Figure 12-13) defines a complete, priority-ordered single hit decision table deriving Eligibility from Pre-Bureau Risk Category, Pre-Bureau Affordability and Age.
- The **Pre-bureau risk category** decision logic (shown as a boxed invocation in Figure 12-14) invokes the Pre-bureau risk category table business knowledge model, passing Applicant data. ExistingCustomer as the Existing Customer parameter and the output of the Application risk score decision as the Application Risk Score parameter.
- The **Pre-bureau risk category table** decision logic (Figure 12-15) defines a complete, unique-hit decision table deriving Pre-Bureau Risk Category from Existing Customer and Application Risk Score.
- The **Application risk score** decision logic (shown as a boxed invocation in Figure 12-16) invokes the Application risk score model business knowledge model, passing Applicant data. Age as the Age

Commented [AF59]: DMN17-197

parameter, Applicant data. MaritalStatus as the Marital Status parameter and Applicant data. EmploymentStatus as the Employment Status parameter.

- The **Application Risk Score Model** decision logic (Figure 12-17) defines a complete, no-order multiple-hit table with aggregation, deriving Application risk score from Age, Marital Status and Employment Status, as the sum of the Partial scores of all matching rows (this is therefore a predictive scorecard represented as a decision table).
- The **Routing** decision logic (shown as a boxed invocation in Figure 12-18) invokes the Routing rules business knowledge model, passing Bureau data. Bankrupt as the Bankrupt parameter, Bureau data. CreditScore as the Credit Score parameter, the output of the Post-bureau risk category decision as the Post-Bureau Risk Category parameter, and the output of the Post-bureau affordability decision as the Post-Bureau Affordability parameter. Note that if Bureau data is null (due to the THROUGH strategy bypassing the Collect bureau data task) the Bankrupt and Credit Score parameters will be null.
- The **Routing rules** decision logic (Figure 12-19) defines a complete, priority-ordered single hit decision table deriving Routing from Post-Bureau Risk Category, Post-Bureau Affordability, Bankrupt and Credit Score.
- The **Post-bureau risk category** decision logic (shown as a boxed invocation in Figure 12-20) invokes the Post- bureau risk category business knowledge model, passing Applicant data. ExistingCustomer as the Existing Customer parameter, Bureau data. CreditScore as the Credit Score parameter, and the output of the Application risk score decision as the Application Risk Score parameter. Note that if Bureau data is null (due to the THROUGH strategy bypassing the Collect bureau data task) the Credit Score parameter will be null.
- The **Post-bureau risk category table** decision logic (Figure 12-21) defines a complete, unique-hit decision table deriving Post-Bureau Risk Category from Existing Customer, Application Risk Score and Credit Score.
- The **Pre-bureau affordability** decision logic (shown as a boxed invocation in Figure 12-22) invokes the Affordability calculation business knowledge model, passing Applicant data. Monthly. Income as the Monthly Income parameter, Applicant data. Monthly. Repayments as the Monthly Repayments parameter, Applicant data. Monthly. Expenses as the Monthly Expenses parameter, the output of the Pre-bureau risk category decision as the Risk Category parameter, and the output of the Required monthly installment decision as the Required Monthly Installment parameter.
- The **Post-bureau affordability** decision logic (shown as a boxed invocation in Figure 12-23) invokes the Affordability calculation business knowledge model, passing Applicant data. Monthly. Income as the Monthly Income parameter, Applicant data. Monthly. Repayments as the Monthly Repayments parameter, Applicant data. Monthly. Expenses as the Monthly Expenses parameter, the output of the Post-bureau risk category decision as the Risk Category parameter, and the output of the Required monthly installment decision as the Required Monthly Installment parameter.
- The **Affordability calculation** decision logic (Figure 12-24) defines a boxed function deriving Affordability from Monthly Income, Monthly Repayments, Monthly Expenses and Required Monthly Installment. One step in this calculation derives Credit contingency factor by invoking the Credit contingency factor table business knowledge model, passing the output of the Risk category decision as the Risk Category parameter.
- The **Credit contingency factor table** decision logic (Figure 12-25) defines a complete, unique-hit decision table deriving Credit contingency factor from Risk Category.
- The **Required monthly installment** decision logic (shown as a boxed invocation in Figure 12-26) invokes the Installment calculation business knowledge model, passing Requested product. ProductType as the Product Type parameter, Requested product. Rate as the Rate parameter, Requested product. Term as the Term parameter and Requested product. Amount as the Amount parameter.
- The **Installment calculation** decision logic (Figure 12-27) defines a boxed function deriving monthly installment from Product Type, Rate, Term and Amount. One step in this calculation invokes an external

function PMT, imported from a DMN XML file as “Financial”. Figure 12-28 shows the decision logic of PMT function.

Strategy			
U	Eligibility	Bureau call type	Strategy
	"INELIGIBLE", "ELIGIBLE"	"FULL", "MINI", "NONE"	"DECLINE", "BUREAU", "THROUGH"
1	"INELIGIBLE"	-	"DECLINE"
2	"ELIGIBLE"	"FULL", "MINI"	"BUREAU"
3		"NONE"	"THROUGH"

Figure 12-9: Strategy decision logic

Bureau call type	
Bureau call type table	
Pre-Bureau Risk Category	Pre-bureau risk category

Figure 12-10: Bureau call type decision logic

Bureau call type table		
U	Pre-Bureau Risk Category	Bureau call type table
	"DECLINE", "HIGH", "MEDIUM", "LOW", "VERY LOW"	"FULL", "MINI", "NONE"
1	"HIGH", "MEDIUM"	"FULL"
2	"LOW"	"MINI"
3	"VERY LOW", "DECLINE"	"NONE"

Figure 12-11: Bureau call type table decision logic

Eligibility	
Eligibility rules	
Age	Applicant data.Age
Pre-Bureau Risk Category	Pre-bureau risk category
Pre-Bureau Affordability	Pre-bureau affordability

Figure 12-12: Eligibility decision logic

Eligibility rules				
P	Pre-Bureau Risk Category	Pre-Bureau Affordability	Age	Eligibility rules
	"DECLINE", "HIGH", "MEDIUM", "LOW", "VERY LOW"			"INELIGIBLE", "ELIGIBLE"
1	"DECLINE"	-	-	"INELIGIBLE"
2	-	false	-	"INELIGIBLE"
3	-	-	< 18	"INELIGIBLE"
4	-	-	-	"ELIGIBLE"

Figure 12-13: Eligibility rules decision logic

Pre-bureau risk category	
Pre-bureau risk category table	
Existing Customer	Applicant data.ExistingCustomer
Application Risk Score	Application risk score

Figure 12-14: Pre-bureau risk category decision logic

Pre-bureau risk category table			
U	Existing Customer	Application Risk Score	Pre-bureau risk category table
			"DECLINE", "HIGH", "MEDIUM", "LOW", "VERY LOW"
1	false	< 100	"HIGH"
2		[100..120)	"MEDIUM"
3		[120..130]	"LOW"
4		> 130	"VERY LOW"
5	true	< 80	"DECLINE"
6		[80..90)	"HIGH"
7		[90..110]	"MEDIUM"
8		> 110	"LOW"

Figure 12-15: Pre-bureau risk category table decision logic

Application risk score	
Application risk score model	
Age	Applicant data.Age
Marital Status	Applicant data.MaritalStatus
Employment Status	Applicant data.EmploymentStatus

Figure 12-16: Application risk score decision logic

Application risk score model				
C+	Age	Marital Status	Employment Status	Application risk score model
	[18..120]	"S", "M"	"UNEMPLOYED", "STUDENT", "EMPLOYED", "SELF-EMPLOYED"	
1	[18..22]	-	-	32
2	[22..26]	-	-	35
3	[26..36]	-	-	40
4	[36..50]	-	-	43
5	>=50	-	-	48
6	-	"S"	-	25
7	-	"M"	-	45
8	-	-	"UNEMPLOYED"	15
9	-	-	"STUDENT"	18
10	-	-	"EMPLOYED"	45
11	-	-	"SELF-EMPLOYED"	36

Figure 12-17: Application risk score model decision logic

Routing	
Routing rules	
Bankrupt	Bureau data.Bankrupt
Credit score	Bureau data.CreditScore
Post-bureau risk category	Post-bureau risk category
Post-bureau affordability	Post-bureau affordability

Figure 12-18: Routing decision logic

Routing rules					
P	Post-bureau risk category	Post-bureau affordability	Bankrupt	Credit score	Routing rules
	"DECLINE", "HIGH", "MEDIUM", "LOW", "VERY LOW"			null, [0..999]	"DECLINE", "REFER", "ACCEPT"
1	-	false	-	-	"DECLINE"
2	-	-	true	-	"DECLINE"
3	"HIGH"	-	-	-	"REFER"
4	-	-	-	< 580	"REFER"
5	-	-	-	-	"ACCEPT"

Figure 12-19: Routing rules decision logic

Post-bureau risk category	
Post-bureau risk category table	
Existing Customer	Applicant data.ExistingCustomer
Credit Score	Bureau data.CreditScore
Application Risk Score	Application risk score

Figure 12-20: Post-bureau risk category decision logic

Post-bureau risk category table				
U	Existing Customer	Application Risk Score	Credit Score	Post-bureau risk category table
				"DECLINE", "HIGH", "MEDIUM", "LOW", "VERY LOW"
1	false	< 120	< 590	"HIGH"
2			[590..610]	"MEDIUM"
3			> 610	"LOW"
4		[120..130]	< 600	"HIGH"
5			[600..625]	"MEDIUM"
6			> 625	"LOW"
7		> 130	-	"VERY LOW"
8	true	<= 100	< 580	"HIGH"
9			[580..600]	"MEDIUM"
10			> 600	"LOW"
11		> 100	< 590	"HIGH"
12			[590..615]	"MEDIUM"
13			> 615	"LOW"

Figure 12-21: Post-bureau risk category table decision logic

Pre-bureau affordability	
Affordability calculation	
Monthly Income	Applicant data.Monthly.Income
Monthly Repayments	Applicant data.Monthly.Repayments
Monthly Expenses	Applicant data.Monthly.Expenses
Risk Category	Pre-bureau risk category
Required Monthly Installment	Required monthly installment

Figure 12-22: Pre-bureau affordability decision logic

Post-bureau affordability	
Affordability calculation	
Monthly Income	Applicant data.Monthly.Income
Monthly Repayments	Applicant data.Monthly.Repayments
Monthly Expenses	Applicant data.Monthly.Expenses
Risk Category	Post-bureau risk category
Required Monthly Installment	Required monthly installment

Figure 12-23: Post-bureau affordability decision logic

Affordability calculation		
F	(Monthly Income , Monthly Repayments , Monthly Expenses , Risk Category , Required Monthly Installment)	
Disposable Income	Monthly Income - (Monthly Repayments + Monthly Expenses)	
Credit Contingency Factor	Credit contingency factor table	
	Risk Category	Risk Category
Affordability	if Disposable Income * Credit Contingency Factor > Required Monthly Installment then true else false	
Affordability		

Figure 12-24: Affordability calculation decision logic

Credit contingency factor table		
U	Risk Category	Credit contingency factor table
	"DECLINE", "HIGH", "MEDIUM", "LOW", "VERY LOW"	
1	"HIGH", "DECLINE"	0.6
2	"MEDIUM"	0.7
3	"LOW", "VERY LOW"	0.8

Figure 12-25: Credit contingency factor table decision logic

Required monthly installment	
Installment calculation	
Product Type	Requested product.ProductType
Rate	Requested product.Rate
Term	Requested product.Term
Amount	Requested product.Amount

Figure 12-26: Required monthly installment decision logic

Installment calculation	
F	(Product Type , Rate , Term , Amount)
Monthly Fee	if Product Type = "STANDARD LOAN" then 20.00 else if Product Type = "SPECIAL LOAN" then 25.00 else null
Monthly Repayment	Financial.PMT(Rate, Term, Amount)
Monthly Repayment + Monthly Fee	

Figure 12-27: Installment calculation decision logic

Financial.PMT	
F	(Rate , Term , Amount)
$(Amount * Rate / 12) / (1 - (1 + Rate / 12)^{-Term})$	

Figure 12-28: Financial.PMT decision logic

12.1.5 Executing the Decision Model

In order to execute a decision model (in this case, by calling two decision services), case data must be bound to the input data, much as an invocation binds arguments to function parameters. The binding of case data to input data, however, is not part of the decision model, unlike the invocation that specifies how a decision’s requirement inputs bind to the parameters of that decision’s required knowledge.

FEEL allows contexts and other expressions to be used to represent case data (see also clauses 0 and 10.6.1). Input data is associated with an item definition (clause 7.3.2) and the case data must have the same type and other constraints specified by the item definition. Case data must be mapped to the FEEL domain. For example, XML instance data is mapped to the FEEL domain as described in clause 10.3.3.

For convenience, we will specify case data using boxed expressions instead of XML. Figure 12-29, Figure 12-30, and Figure 12-31 show boxed contexts defining case data for Applicant data, Requested product and Bureau data.

Applicant data		
Age	51	
MartitalStatus	"M"	
EmploymentStatus	"EMPLOYED"	
ExistingCustomer	false	
Monthly	Income	10000
	Repayments	2500
	Expenses	100000

Figure 12-29: Applicant data input data sample

Bureau data	
Bankrupt	false
CreditScore	600

Figure 12-30: Requested Product input data sample

Strategy	"THROUGH"
Bureau call type	"NONE"

Figure 12-31: Bureau Data input data sample

When the Bureau Strategy Decision Service is called with the Applicant data and Requested product case data, it returns the context shown in Figure 12-32:

Routing	"ACCEPT"
---------	----------

Figure 12-32: Output of the Bureau Strategy Decision Service

When the Routing Decision Service is called with the Applicant data, Requested product and Bureau data case data, it returns the context shown in Figure 12-33.

Requested product	
ProductType	"STANDARD LOAN"
Rate	0.08
Term	36
Amount	100000

Figure 12-33: Output of the Routing decision Service

12.2 Example 2: Ranked Loan Products

The second example considers eligibility for various mortgage loan products based on the Borrower's income, assets, liabilities, and credit score, and ranks them based on specified sort criteria. It illustrates the wide variety of DMN expression types, including context, invocation, relation, and function definition, as well as some of the newer FEEL functions and operators, including import, service invocation, enhanced iteration, generalized unary tests, and Java binding. The logic represented here is just one of many different ways to model the scenario.

The DRD for the decision model is shown in Figure 12-34.

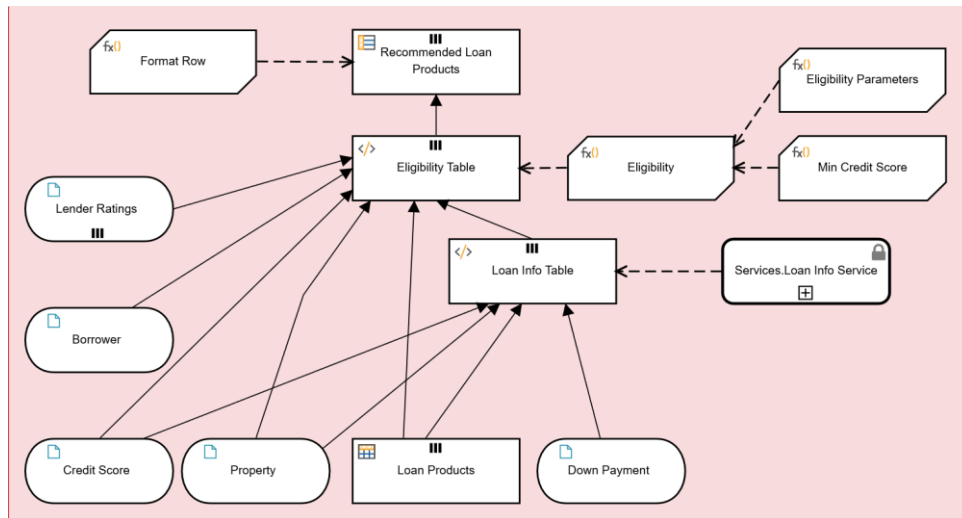


Figure 12-34: DRD for Recommended Loan Products

The input data elements include:

- **Credit Score**, a number from 300 to 850 inclusive
- **Down Payment**, a number
- **Property**, a structure of type *tProperty* (Figure 12-35)
- **Borrower**, a structure of type *tBorrower* (Figure 12-37), and
- **Lender Ratings**, a structure of type *tLenderRatings* (Figure 12-38)

The boxed expression format for the datatype definitions in Figure 12-35, Figure 12-37, and Figure 12-38 is non-normative. Figure 12-35, for example, is a visualization of the XML representation of Figure 12-36.

Commented [AF60]: DMN17-197

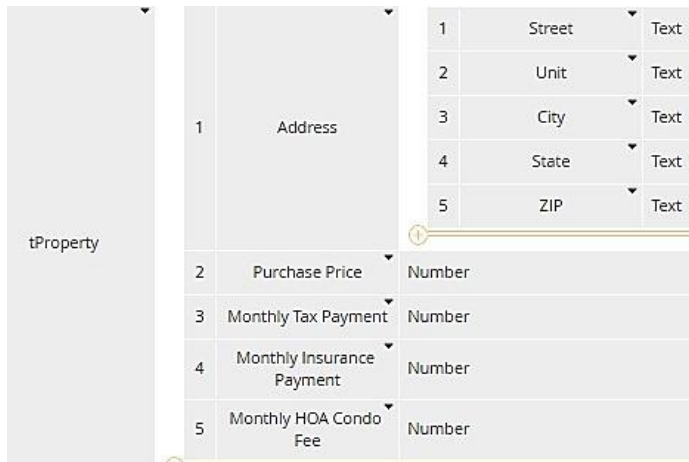


Figure 12-35: Type tProperty (non-normative representation)

```

<semantic:itemDefinition name="tProperty" label="tProperty">
  <semantic:itemComponent id="_5e820b14-1f14-44e2-bee1-a35fbedc477f" name="Address">
    <semantic:itemComponent id="_d40919e3-168d-46dc-a7da-ccfeead8a49" name="Street">
      <semantic:typeRef>string</semantic:typeRef>
    </semantic:itemComponent>
    <semantic:itemComponent id="_a03ae467-fb6a-46f0-ab1a-dc0992d81095" name="Unit">
      <semantic:typeRef>string</semantic:typeRef>
    </semantic:itemComponent>
    <semantic:itemComponent id="_f902cd87-2c95-4b90-95cf-a2c6b1b87a1e" name="City">
      <semantic:typeRef>string</semantic:typeRef>
    </semantic:itemComponent>
    <semantic:itemComponent id="_97f12b0d-be5c-4d42-abb5-d565599fee87" name="State">
      <semantic:typeRef>string</semantic:typeRef>
    </semantic:itemComponent>
    <semantic:itemComponent id="_2fdc92bc-55da-4ff7-8a9d-c5213b69a0a8" name="ZIP">
      <semantic:typeRef>string</semantic:typeRef>
    </semantic:itemComponent>
  </semantic:itemComponent>
  <semantic:itemComponent id="_cc0e8c3f-ae44-4080-88db-555d8a2f8560" name="Purchase Price">
    <semantic:typeRef>number</semantic:typeRef>
  </semantic:itemComponent>
  <semantic:itemComponent id="_ce17ee0b-f1e1-43cf-8a5e-4a18390fc6d6" name="Monthly Tax Payment">
    <semantic:typeRef>number</semantic:typeRef>
  </semantic:itemComponent>
  <semantic:itemComponent id="_338c3f84-8ff7-404d-9b61-d211a5cebe4b" name="Monthly Insurance Payment">
    <semantic:typeRef>number</semantic:typeRef>
  </semantic:itemComponent>
  <semantic:itemComponent id="_fe427d63-1cf3-4d2d-b268-f7e01dccad59" name="Monthly HOA Condo Fee">
    <semantic:typeRef>number</semantic:typeRef>
  </semantic:itemComponent>
</semantic:itemDefinition>

```

Figure 12-36: Type tProperty (XML representation)

tBorrower	1	Full Name	Text		
	2	Tax ID	Text		
	3	Employment Income	Number		
	4	Other Income	Number		
	5	Assets tAssets tAsset	1	Type	tAssetType Text "Checking Savings Brokerage account", "Real Estate", "Other Liquid", "Other Non-Liquid"
			2	Institution Account or Description	Text
			3	Value	Number
	6	Liabilities tLiabilities tLiability	1	Type	tLiabilityType Text "Credit card", "Auto loan", "Student loan", "Lease", "Lien", "Real estate loan", "Alimony child support", "Other"
			2	Payee	Text
			3	Monthly payment	Number
4			Balance	Number	
5			To be paid off	Boolean	

Figure 12-37: Type tBorrower

tLenderRatings III tLenderRating	1	Lender Name	Text
	2	Customer Rating	Number [1..5]

Figure 12-38: Type tLenderRatings, a collection of tLenderRating

In addition, the zero-input decision Loan Products, a structure of type tLoanProducts, is a relation (Figure 12-39). Cells in a relation are FEEL expressions but often contain literal values as a way to embed static data tables inside a decision model. In this case it represents a list of mortgage loan products available from various lenders, specifying the best interest rate offered to lowest risk borrowers and loan origination costs specified as “points”, a percentage of the loan amount, and “fees”, a constant value.

	Lender Name	Product Name	Type	Best Rate Pct	Points Pct	Fees Amount	Term
	Text	tProductName "Fixed30-NoPoints", "Fixed30-Standard", "Fixed15-NoPoints", "Fixed15-Standard", "ARM5/1-NoPoints", "ARM5/1-Standard"	tAmortizationType "Fixed rate", "Variable rate"	tPercent	tPercent	Number	Number
1	"Lender A"	"Fixed30-NoPoints"	"Fixed rate"	3.95	0	1925	360
2	"Lender C"	"Fixed30-Standard"	"Fixed rate"	3.75	0.972	1975	360
3	"Lender A"	"Fixed15-NoPoints"	"Fixed rate"	3.625	0	816	180
4	"Lender C"	"Fixed15-Standard"	"Fixed rate"	3.25	0.767	1975	180
5	"Lender B"	"ARM5/1-NoPoints"	"Variable rate"	3.875	0	1776	360
6	"Lender B"	"ARM5/1-Standard"	"Variable rate"	3.625	0.667	1975	360

Commented [AF61]: DMN17-197

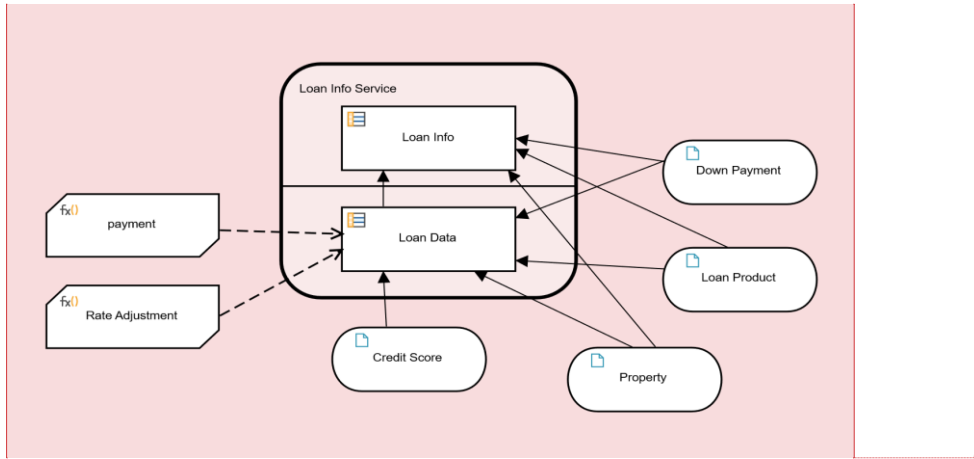
Figure 12-39: Loan Products

tLoanProducts III tLoanProduct	1	Lender Name	Text
	2	Product Name	tProductName Text "Fixed30-NoPoints", "Fixed30-Standard", "Fixed15-NoPoints", "Fixed15-Standard", "ARM5/1-NoPoints", "ARM5/1-Standard"
	3	Type	tAmortizationType Text "Fixed rate", "Variable rate"
	4	Best Rate Pct	tPercent Number
	5	Points Pct	tPercent Number
	6	Fees Amount	Number
	7	Term	Number

Figure 12-40: Type tLoanProducts, a collection of tLoanProduct

The **Recommended Loan Products** model imports another decision model **Loan Info**, with the DRD shown in Figure 12-41, defining a decision service **Loan Info Service**. Imported models are assigned a modeler-chosen prefix, here *Services*, to distinguish its namespace from that of the importing model. In the importing DRD (Figure 12-34), the imported service **Services.Loan Info Service** is depicted with the non-normative lock icon, indicating that its logic may not be edited within the importing model. The service parameters are the input data shown in Figure 12-41: **Credit Score**, **Property**, **Loan Product**, and **Down Payment**, with types identical to those defined in the importing model.

Services.Loan Info Service populates a row of the decision **Loan Info Table**, a collection of type **tLoanInfoRow** (Figure 12-39), calculating the details of the selected loan product for the given property value (purchase price) and down payment.



Commented [AF62]: DMN17-197

Figure 12-41: DRD of imported Loan Info Service

tLoanInfoTable III tLoanInfoRow	1	Product	tProductName Text "Fixed30-NoPoints", "Fixed30-Standard", "Fixed15-NoPoints", "Fixed15-Standard", "ARMS/1-NoPoints", "ARMS/1-Standard"
	2	Amortization Type	tAmortizationType Text "Fixed rate", "Variable rate"
	3	LTV	tPercent Number
	4	Note Amount	Number
	5	Initial Rate Pct	tPercent Number
	6	Qualifying Rate Pct	tPercent Number
	7	Initial Monthly Payment	Number
	8	Qualifying Monthly Payment	Number
	9	Points Amount	Number
	10	Fees Amount	Number
	11	Funds Toward Purchase	Number
	12	Down Payment	Number
	13	Closing Costs	Number
	14	Cash to Close	Number

Figure 12-42: Type tLoanInfoTable, a collection of tLoanInfoRow

Loan Data		
tLoanData		
1	Points Amount Number	decimal((Property.Purchase Price - Down Payment)*Loan Product.Points Pct/100,2)
2	Note Amount Number	Property.Purchase Price - Down Payment + Loan Product.Fees Amount + Points Amount
3	LTV tPercent	decimal(100*Note Amount/Property.Purchase Price,2)
4	Closing Costs Number	decimal(0.02*Note Amount,2)
5	Funds Toward Purchase Number	Note Amount - Loan Product.Fees Amount - Points Amount - Closing Costs
6	Interest Rate Percent tPercent	Loan Product.Best Rate Pct + Rate Adjustment(Credit Score, LTV)
7	Qualifying Rate Percent tPercent	if Loan Product.Type="Variable rate" then Interest Rate Percent+2 else Interest Rate Percent
8	Monthly Payment Number	payment
		1 p Number Note Amount
		2 r Number Interest Rate Percent/100
		3 n Number Loan Product.Term
9	Qualifying Payment Number	payment
		1 p Number Note Amount
		2 r Number Qualifying Rate Percent/100
		3 n Number Loan Product.Term
Result		

Figure 12-43: Loan Data

Within the service, **Loan Data** performs calculations used in the presentation decision, **Loan Info**. It is modeled as a context with no final result box, meaning every context entry creates a component of the result. (The text “Result” in the final result box is a tool artifact not in the spec, overwritten by a literal expression if the context has a final result box value.) A few things to note about the logic shown in Figure 12-43:

- FEEL arithmetic can create values with many digits following the decimal point. The function $\text{decimal}(x, 2)$ rounds value x to 2 decimal places.

- Context entry Interest Rate Percent invokes the BKM **Rate Adjustment** (Figure 12-44), a function of the borrower's **Credit Score** and the loan-to-value ratio **LTV**. This increments the Loan Product's interest rate by a small amount based on the loan risk.
- **Credit Score** values less than 620 are ineligible for a loan. In that case, **Rate Adjustment** could return null, but then all expressions using **Rate Adjustment** would also be null, complicating the logic. To simplify the downstream logic, it is better in this case to return a number, since ultimately the loan will not be approved if the **Credit Score** is less than 620.
- For loans with variable interest rate, the debt-to-income ratio uses a *Qualifying Payment* amount based on an interest rate 2 percent higher than the rate used in the initial *Monthly Payment*.
- *Monthly Payment* and *Qualifying Payment* are modeled as boxed invocations of the BKM **payment**, the amortization formula (Figure 12-45). The parameters of payment are the loan amount p , the interest rate r , and the term in months, n .

The decision **Loan Info** (Figure 12-46), the output of **Services.Loan Info**, returns a row of **Loan Info Table**. It is also modeled as a context with no final result box, meaning each context entry represents a column of **Loan Info Table**.

Rate Adjustment			
tPercent			
U	inputs		outputs
	Credit Score	LTV	Rate Adjustment
	tCreditScore [300..850]	tPercent	tPercent
1	>=660	<=60	0
2	[620..660)	<=60	0.125
3	>=700	>60	0.125
4	[660..700)	(60..70]	0.125
5	[620..660)	(60..70]	0.25
6	[680..700)	>70	0.25
7	[640..680)	>70	0.375
8	[620..640)	(70..80]	0.375
9	[620..640)	>80	0.5
10	<620	-	0.5

Figure 12-44: BKM Rate Adjustment

payment	
Number	
F	$\left(\begin{matrix} p \\ \text{Number} \end{matrix} , \begin{matrix} r \\ \text{Number} \end{matrix} , \begin{matrix} n \\ \text{Number} \end{matrix} \right)$
decimal(p*r/12/(1-(1+r/12)**-n),2)	

Figure 12-45: BKM payment

Loan Info <i>LoanInfo</i>		
1	Product <i>ProductName</i> "Fixed30-NoPoints", "Fixed30-Standard", "Fixed15-NoPoints", "Fixed15-Standard", "ARM5/1-NoPoints", "ARM5/1-Standard"	Loan Product.Name
2	Amortization Type <i>AmortizationType</i> "Fixed rate", "Variable rate"	Loan Product.Type
3	LTV <i>PPercent</i>	Loan Data.LTV
4	Note Amount <i>Number</i>	Loan Data.Note Amount
5	Initial Rate Pct <i>PPercent</i>	Loan Data.Interest Rate Percent
6	Qualifying Rate Pct <i>PPercent</i>	Loan Data.Qualifying Rate Percent
7	Initial Monthly Payment <i>Number</i>	Loan Data.Monthly Payment
8	Qualifying Monthly Payment <i>Number</i>	Loan Data.Qualifying Payment
9	Points Amount <i>Number</i>	Loan Data.Points Amount
10	Fees Amount <i>Number</i>	Loan Product.Fees Amount
11	Funds Toward Purchase <i>Number</i>	Loan Data.Funds Toward Purchase
12	Down Payment <i>Number</i>	Down Payment
13	Closing Costs <i>Number</i>	Loan Data.Closing Costs
14	Cash to Close <i>Number</i>	Property.Purchase Price - Funds Toward Purchase
Result		

Figure 12-46: Loan Info

In the importing model, the decision **Loan Info Table** (Figure 12-47) iterates invocation of **Loan Info** over rows of **Loan Products**. It is modeled as a literal expression using the FEEL *for . in .return* operator. Here *x* is a range variable meaning one item in a list – one **Loan Product** in **Loan Products** – producing an argument of the function call.

Loan Info Table

tLoanInfoTable

```
for x in Loan Products return Services.Loan Info(x,Down Payment,Property,Credit Score)
```

Figure 12-47: Loan Info Table

Loan Info Table now provides values for each **Loan Product** used to determine whether the Borrower's income, assets, liabilities, and credit score qualify for loan approval.

At the heart of the logic for determining eligibility for a particular loan is the BKM **Min Credit Score** (Figure 12-48), a decision table that calculates the minimum credit score required based on three parameters: *DTI*, the borrower's debt-to-income ratio; *LTV*, the loan-to-value ratio; and *Reserves*, a measure of the Borrower's liquid assets after closing in units of monthly *Housing Costs*. The table is modeled as hit policy *Collect* with aggregation *Minimum*, meaning when multiple rules match the lowest value output is returned. When *DTI* is greater than 95%, the loan is automatically ineligible. In that case, no rule matches and **Min Credit Score** returns the value null. Downstream logic referencing this variable must account for the possibility of null value.

Min Credit Score

tCreditScore
[300..850]

C<	inputs			outputs
	DTI	LTV	Reserves	Min Credit Score
	tPercent	tPercent	Number	tCreditScore [300..850]
1	<=36	<=75	>2	620
2	<=36	<=75	>0	640
3	<=36	(75..95]	>6	660
4	<=36	(75..95]	>0	680
5	(36..45]	<=75	>6	660
6	(36..45]	<=75	>0	680
7	(36..45]	(75..95]	>6	700
8	(36..45]	(75..95]	>0	720

Figure 12-48: Min Credit Score

Min Credit Score is called by the BKM **Eligibility**, which in turn calls the BKM **Eligibility Parameters** (Figure 12-49). **Eligibility Parameters** calculates the two key parameters of **Min Credit Score**, the debt-to-income ratio *DTI Pct*, and the liquid assets after closing, called *Reserves*. Note that context entry *Housing Expense*, which sums the loan payment, tax and insurance payments, and homeowner association/condo fee, must account for the possibility that the latter is left blank, i.e., null, in the input data **Property**, since adding null to a number gives null. To prevent this, instead of the + operator we use the *sum()* function on a list filtered by the condition *item != null*. We use this technique also on context entry *Income*.

Eligibility Parameters <i>tEligibilityParameters</i>		
F	$\left(\begin{array}{c} \text{Loan Product} \quad \text{Borrower} \quad \text{Loan Info} \quad \text{Property} \quad \text{Credit Score} \\ t\text{LoanProduct} \quad t\text{Borrower} \quad t\text{LoanInfoRow} \quad t\text{Property} \quad t\text{CreditScore} \\ [300..850] \end{array} \right)$	
1	Housing Expense <i>Number</i>	sum([Loan Info.Qualifying Monthly Payment, Property.Monthly Tax Payment, Property.Monthly Insurance Payment, Property.Monthly HOA Condo Fee][item != null])
2	Non-Housing Debt Payments <i>Number</i>	sum(Borrower.Liabilities[Type!="Real estate loan" and To be paid off =false].Monthly payment)
3	Income <i>Number</i>	sum([Borrower.Employment Income, Borrower.Other Income][item != null])
4	DTI Pct <i>tPercent</i>	decimal((Housing Expense+Non-Housing Debt Payments)/Income*100,2)
5	Liquid Assets Before Closing <i>Number</i>	sum(Borrower.Assets[Type="Checking Savings Brokerage account" or Type="Other Liquid"].Value)
6	Debts Paid Off By Closing <i>Number</i>	sum(Borrower.Liabilities[Type!="Real estate loan" and To be paid off=true].Balance[item!=null])
7	Liquid Assets After Closing <i>Number</i>	Liquid Assets Before Closing - Debts Paid Off By Closing - Loan Info.Cash to Close
8	Reserves <i>Number</i>	decimal(Liquid Assets After Closing/Housing Expense,2)
Result		

Figure 12-49: Eligibility Parameters

For legibility, the BKM **Eligibility** is shown in two pieces (Figure 12-50 and Figure 12-51). This BKM creates a row of type *tTableRow* for the decision **Eligibility Table**. It is modeled as a context, where the first four context entries (Figure 12-51) call BKM to determine values to populate the *Table Row* components.

- *Params* calls the BKM **Eligibility Parameters** for a given **Loan Product**.
- *Required Credit Score* uses *Params* to call the BKM **Min Credit Score**, returning the minimum credit score required by that **Loan Product** for the Borrower to be eligible.
- *Eligible* is a Boolean comparing the Borrower's credit score to **Min Credit Score**. *Recommendation* uses the input data **Lender Ratings** in combination with **Eligible** to return a recommendation value for the **Loan Product**. *Recommendation* illustrates an alternative decision table syntax introduced in DMN 1.2 called generalized unary test. With generalized unary tests, a decision table input entry may be any FEEL

expression, substituting ? for the input expression. For example, in the first column of this decision table the rules filter the **Lender Ratings** table for an item with *Lender Name* matching that of the **Loan Product** and *Customer Rating* in a specified range, returning true if that filter returns any values.

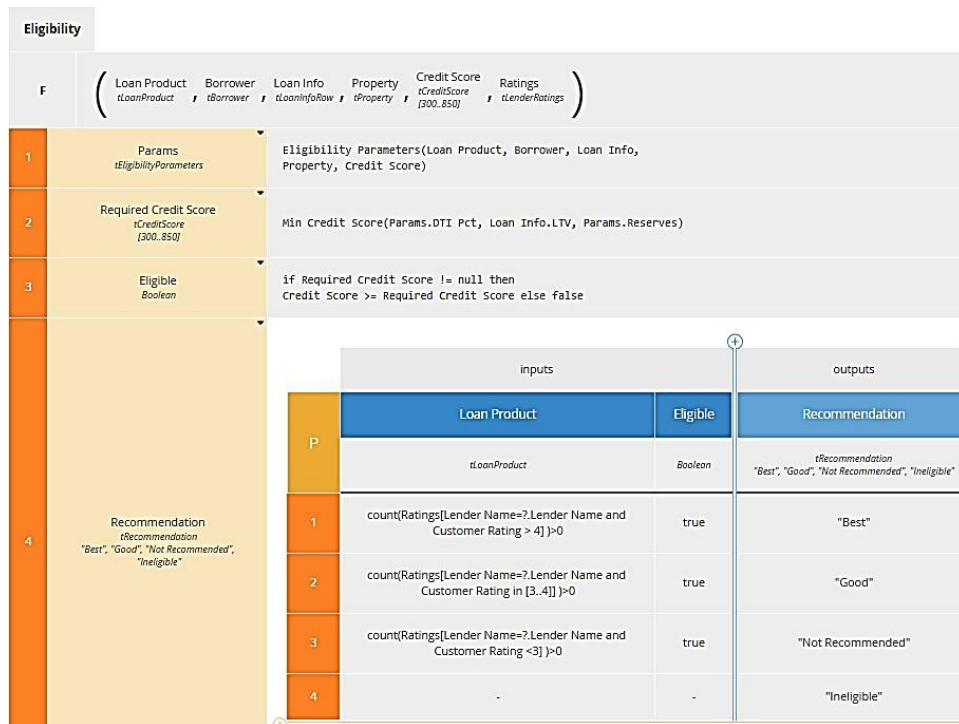


Figure 12-50: Eligibility (top)

The rest of **Eligibility** is shown in Figure 12-51.

- *Table Row* is a nested context with no final result box value. Each context entry represents a column in the row.
- The DMN spec allows the final result box to be a context, but in this example, we use a context entry to create the result value and return it in the result box. Here context entry *Table Row* creates the row structure, and the final result box simply selects this context entry.

5	Table Row <i>tTableRow</i>	1	Product <i>Text</i>	Loan Product.Lender Name + " - " + Loan Product.Product Name
		2	Note Amount <i>Number</i>	Loan Info.Note Amount
		3	Interest Rate Pct <i>Percent</i>	Loan Info.Initial Rate Pct
		4	Monthly Payment <i>Number</i>	Loan Info.Initial Monthly Payment
		5	LTV <i>Percent</i>	Loan Info.LTV
		6	DTI <i>Percent</i>	Params.DTI Pct
		7	Cash to Close <i>Number</i>	Loan Info.Cash to Close
		8	Liquid Assets After Closing <i>Number</i>	Params.Liquid Assets After Closing
		9	Reserves <i>Number</i>	Params.Reserves
		10	Required Credit Score <i>tCreditScore [900..850]</i>	Required Credit Score
		11	Recommendation <i>tRecommendation "Best", "Good", "Not Recommended", "Ineligible"</i>	Recommendation
		Result		

Table Row

Figure 12-51: Eligibility (bottom)

The decision **Eligibility Table** (Figure 12-52) uses an alternative form of the *for .in..return* operator to iterate over an index rather than iterate over list item values. This alternative format allows the returned expression to involve corresponding items in multiple lists, in this case **Loan Products** and **Loan Info Table**.

Eligibility Table <i>tEligibilityTable</i> <pre>for i in 1..count(Loan Products) return Eligibility(Loan Products[i], Borrower, Loan Info Table[i], Property, Credit Score)</pre>

Figure 12-52: Eligibility Table

The top-level decision **Recommended Loan Products** (Figure 12-53) first sorts **Eligibility Table** based on *Recommendation* and *Monthly Payment* and then calls a Java method to format number values as strings for final presentation.

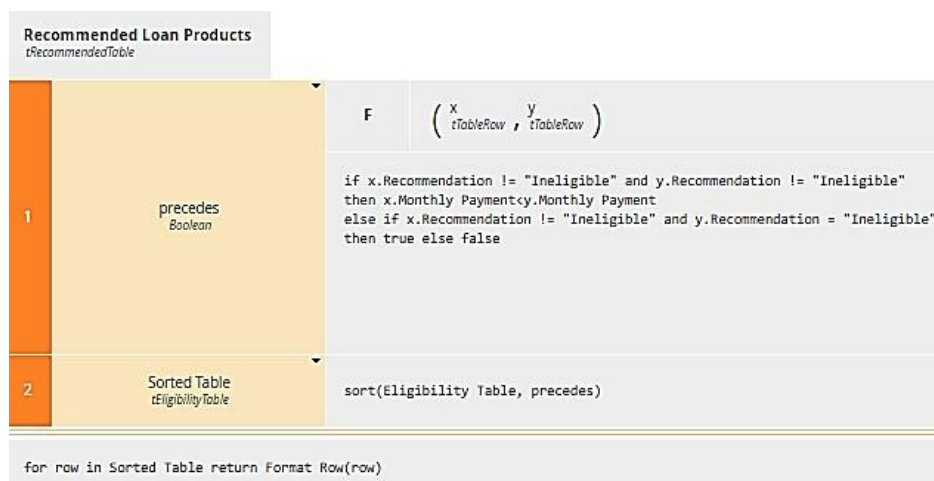


Figure 12-53: Recommended Loan Products

- The first context entry *precedes* is a function definition used by the FEEL *sort()* function. The second parameter of *sort()*, called the *precedes* function, is a Boolean function with two arguments representing list items. It returns true if the first argument precedes the second in the sorted list.
- The context entry *Sorted Table* performs the sort. With simple sort criteria, the *precedes* function is typically defined inline as an anonymous function using the keyword *function*, as in
`sort(myTable, function(x, y) x.Amount < y.Amount)`
which sorts the rows of *myTable* in ascending order of the column *Amount*. However, in **Recommended Loan Products** we instead use a named *precedes* function, the context entry *precedes*. In that case, the name of the function provides the second argument of *sort()*.
- The final result box iterates a call to the BKM **Format Row**, which executes a static Java method to format number values in *Sorted Table* as strings with a currency symbol and two digits following the decimal point.

Format Row (Figure 12-54) operates on a single row of *Sorted Table*. It is modeled as a context.

- The first context entry *string format* is a Java function definition, indicated by the code J. DMN specifies such a function definition as a context with two context entries, *class*, and *method signature*. This example applies a mask string to a number, returning a formatted number string.
- The second context entry *formatted row* generates a row of **Recommended Loan Products** in final presentation format, calling *string format* to format amount and percent values.
- The final result box returns *formatted row*.

Format Row formattedrow_1				
F	(row tableRow)			
1	string format Text	J	(mask , value Text , Number)	
		class	"java.lang.String"	
		method signature	"format(java.lang.String, [Ljava.lang.Object;)"	
2	formatted row formattedrow_1	1	Product Text	row.Product
		2	Note Amount Text	string -format("%\$,4.2f", row.Note Amount)
		3	Interest Rate Pct Text	string -format(" %\$,4.2f", row.Interest Rate Pct)
		4	Monthly Payment Text	string -format("%\$,4.2f", row.Monthly Payment)
		5	Cash to Close Text	string -format("%\$,4.2f", row.Cash to Close)
		6	Required Credit Score tCreditScore [300..850]	row.Required Credit Score
		7	Recommendation Text	row.Recommendation
		Result		
formatted row				

Figure 12-54: Format Row

Figure 12-55 shows the output of **Recommended Loan Products** based on the Test Case input data of Figure 12-56.

Product	Note Amount	Interest Rate Pct	Monthly Payment	Cash to Close	Required Credit Score	Recommendation
Lender B - ARM5/1- Standard	\$273,775.90	3.75	\$1,267.90	\$75,475.52	720	Good
Lender C - Fixed30- Standard	\$274,599.40	3.88	\$1,291.27	\$75,491.99	680	Best
Lender B - ARM5/1- NoPoints	\$271,776.00	4.00	\$1,297.50	\$75,435.52	720	Good
Lender A - Fixed30- NoPoints	\$271,925.00	4.08	\$1,310.00	\$75,438.50	680	Best
Lender C - Fixed15- Standard	\$274,045.90	3.38	\$1,942.33	\$75,480.92	720	Best
Lender A - Fixed15- NoPoints	\$270,816.00	3.75	\$1,969.43	\$75,416.32	720	Best

Figure 12-55: Test Case output of Recommended Loan Products

Decision Test

Page 1

Credit Score
[300..850]
735

Property

Address

Street
272 10th St.

Unit

City
Marina

State
CA

ZIP
93933

Purchase Price
340000

Monthly Tax Payment
350

Monthly Insurance Payment
100

Monthly HOA Condo Fee
0

Down Payment
70000

Borrower

Full Name
Ken Customer

Tax ID
111223333

Employment Income
100000

Assets

Type	Institution Account or Description	Value
Checking Savings Brokerage account	Chase	35,000
Checking Savings Brokerage account	Vanguard	45,000
Other Non-Liquid		17,000

Liabilities

Type	Payee	Monthly payment	Balance	To be paid off
Credit card	Chase	300	0	false
Lease	BMW Finance	450	0	false
Alimony child support		1,000	0	false
Lien	LA County	100	850	true

Figure 12-56: Test Case Input Data (partial)

This page intentionally left blank.

13 Exchange Formats

13.1 Interchanging Incomplete Models

It is common for **DMN** models to be interchanged before they are complete. This occurs frequently when doing iterative modeling, where one user (such as a knowledge source expert or business user) first defines a high-level model and then passes it on to another person to complete or refine the model.

Such "incomplete" models are ones in which not all of the mandatory model attributes have been filled in yet or the cardinality of the lower bound of attributes and associations has not been satisfied.

XMI allows for the interchange of such incomplete models. In **DMN**, we extend this capability to interchange of XML files based on the **DMN** XML-Schema. In such XML files, implementers are expected to support this interchange by:

- Disregarding missing attributes that are marked as "required" in the **DMN** XML-Schema.
- Reducing the lower bound of elements with "minOccurs" greater than 0.

13.2 Machine Readable Files

All machine-readable files, including XSD, XMI and XML files, can be found at <https://www.omg.org/spec/DMN>.

- For the **DMN** XMI Model, the main file is DMN.xmi.
- For the **DMN** XSD Interchange (supporting Conformance Levels 1, 2 and 3), the main file is DMN.xsd.
- XML serializations of the examples in clause 12 are provided as a non-normative zip file.

13.3 XSD

13.3.1 Document Structure

A domain-specific set of model elements is interchanged in one or more **DMN** files. The root element of each file SHALL be `<DMN.Definitions>`. The set of files SHALL be self-contained, i.e., all definitions that are used in a file SHALL be imported directly or indirectly using the `<DMN.Import>` element.

Each file SHALL declare a "name space" that MAY differ between multiple files of one model.

DMN files MAY import non-**DMN** files (such as XSDs and PMMLs) if the contained elements use external definitions.

Starting from DMN 1.7, the XML namespace URIs for backwards-compatible 1.x versions of DMN are fixed at:

- <https://www.omg.org/spec/DMN/> for the semantic model
- <https://www.omg.org/spec/DMN/DMNDI/> for the diagram interchange model
- <https://www.omg.org/spec/DMN/FEEL/> for the FEEL expression language
- <https://www.omg.org/spec/DMN/B-FEEL/> for the B-FEEL expression language

The `<dmn:definitions>` root element of a DMN XML file MUST indicate the concrete DMN version that is used in the file by setting the `dmnVersion` attribute, e.g.:

```
<dmn:definitions dmnVersion="1.7" ...
```

In addition, the `<dmn:definitions>` root element SHOULD include an `xsi:schemaLocation` attribute that points to the XML schema files for all namespaces, i.e. the URLs of the XSDs that are publicly hosted by OMG without dated version stamps, e.g.

```
<dmn:definitions dmnVersion="1.7" ...
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    https://www.omg.org/spec/DMN/
    https://www.omg.org/spec/DMN/DMN.xsd
    https://www.omg.org/spec/DMN/DMNDI/
    https://www.omg.org/spec/DMN/DMNDI.xsd
    https://www.omg.org/spec/SCE/DI/
    https://www.omg.org/spec/SCE/DI.xsd
    https://www.omg.org/spec/SCE/DC/
    https://www.omg.org/spec/SCE/DC.xsd
">
```

Commented [AF63]: DMN17-65

13.3.2 References within the DMN XSD

Many **DMN** elements that may need to be referenced contain IDs and within the **BPMN** XSD, references to elements are expressed via these IDs. The XSD IDREF type is the traditional mechanism for referencing by IDs, however it can only reference an element within the same file. **DMN** elements of type `DMNElementReference` support referencing by ID, across files, by utilizing an `href` attribute whose value must be a valid URI reference [RFC 3986] where the path components may be absolute or relative, the reference has no query component, and the fragment consists of the value of the `id` of the referenced **DMN** element.

Commented [AF64]: DMN17-77

For example, consider the following Decision:

```
<decision name="Pre-Bureau Risk Category"
id="prebureauriskDec01">...</decision>
```

When this Decision is referenced, e.g., by an `InformationRequirement` in a Decision that is defined in another file, the reference could take the following form:

```
<requiredDecision
href="http://www.example.org/Definitions01#prebureauriskDec01"/>
```

where “`http://www.example.org/Definitions01`” is a URI representing the namespace of the model specified in the DMN file. When this Decision is referenced, e.g., by an `InformationRequirement` in a Decision that is defined in another file, the reference could take the following form:

```
<requiredDecision
href="http://www.example.org/Definitions01.xml#prebureauriskDec01"/> where
“http://www.example.org/Definitions01.xml” is an URI reference to the XML document in
which the “PreBureau Risk Category” Decision is defined (e.g., the value of the locationURI attribute in the
corresponding Import element), and “prebureauriskDec01” is the value of the id attribute for the Decision.
```

Commented [AF65]: DMN17-77

When the Decision is referenced in the same file, the reference could take both of the following forms:

```
<requiredDecision
href="http://www.example.org/Definitions01.xml#prebureauriskDec01"/> or
<requiredDecision href="#prebureauriskDec01"/>
```

If the path component in the URI reference is relative, the base URI against which the relative reference is applied is determined as specified in [RFC 3986]. According to that specification, “if no base URI is embedded and the representation is not encapsulated within some other entity, then, if a URI was used to retrieve the representation, that URI shall be considered the base URI” ([RFC 3986], section 5.1.3). That is, if the reference is not in the scope of an `xml:base` attribute [XBASE], a value of the `href` attribute that contains only a fragment, and no path component references a **DMN** element that is defined in the same instance of XML file as the referencing element. In the example below, assuming that the `requiredDecision` element is not in the scope of an `xml:base` attribute, the **DMN** element whose `id` is “prebureauriskDec01” must be defined in the same XML document:

```
<requiredDecision href="#prebureauriskDec01"/>
```

Commented [AF66]: DMN17-77

Notice that the **BPMN** processes and tasks that use a decision are referenced using the `href` attribute as well; indeed, it is compatible with the system to reference external Process and Task instances in **BPMN 2.0** Definitions, which is also based on IDs.

Attribute `typeRef` references `ItemDefinitions` and built-in types by name not ID. In order to support imported types, `typeRef` uses the namespace-qualified name syntax `[qualifier].[local-name]`, where `qualifier` is specified by the `name` attribute of the `Import` element for the imported type. If the referenced type is not imported, the prefix SHALL be omitted.

This page intentionally left blank

14 DMN Diagram Interchange (DMN DI)

14.1 Scope

This chapter specifies the meta-model and schema for **DMN** Diagram Interchange (**DMN DI**). The **DMN DI** is meant to facilitate the interchange of **DMN** diagrams between tools rather than being used for internal diagram representation by the tools. The simplest interchange approach to ensure the unambiguous rendering of a **DMN** diagram was chosen for **DMN DI**. As such, **DMN DI** does not aim to preserve or interchange any “tool smarts” between the source and target tools (e.g., layout smarts, efficient styling, etc.).

DMN DI does not ascertain that the **DMN** diagram is syntactically or semantically correct.

This version of DMN DI focuses on the interchange of Decision Requirements Diagrams (DRDs). Diagram Interchange for boxed expressions and decision tables might be added in future versions.

14.2 Diagram Definition and Interchange

The **DMN DI** meta-model, similar to the DMN abstract syntax meta-model, is defined as a MOF-based meta-model. As such, its instances can be serialized and interchanged using XML. DMN DI is also defined by an XML schema. Thus, its instances can also be serialized and interchanged using XML.

Both DMN DI meta-model and schema are harmonized with the OMG Diagram Definition (DD) standard version 1.1. The referenced DD contains two main parts: the Diagram Commons (DC) and the Diagram Interchange (DI). The DC defines common types like bounds and points, while the DI provides a framework for defining domain-specific diagram models. As a domain-specific DI, DMN DI defines a few new meta-model classes that derive from the abstract classes from DI.

The focus of DMN DI is the interchange of laid out shapes and edges that constitute a **DMN** diagram. Each shape and edge reference a particular **DMN** model element. The referenced **DMN** model elements are all part of the actual **DMN** model. As such, DMN DI is meant to only contain information that is neither present nor derivable, from the **DMN** model whenever possible. Simply put, to render a **DMN** diagram both the DMN DI instance(s) and the referenced **DMN** model are REQUIRED.

From the DMN DI perspective, a DMN diagram is a particular snapshot of a **DMN** model at a certain point in time. Multiple **DMN** diagrams can be exchanged referencing model elements from the same **DMN** model. Each diagram may provide an incomplete or partial depiction of the content of the **DMN** model. As described in clause 12, a **DMN** model package consists of one or more files. Each file may contain any number of **DMN** diagrams. The exporting tool is free to decide how many diagrams are exported and the importing tool is free to decide if and how to present the contained diagrams to the user.

14.3 How to read this chapter

Clause 14.4 describes in detail the meta-model used to keep the layout and the look of **DMN** Diagrams. Clause 14.5 presents in tables a library of the **DMN** element depictions and an unambiguous resolution between a referenced **DMN** model element and its depiction.

14.4 DMN Diagram Interchange Meta-Model

14.4.1 Overview

The DMN DI is an instance of the OMG DI meta-model. The basic concept of DMN DI, as with DI in general, is that serializing a diagram [DMNDiagram] for interchange requires the specification of a collection of shapes [DMNShape] and edges [DMNEdge].

The DMN DI classes only define the visual properties used for depiction. All other properties that are REQUIRED for the unambiguous depiction of the **DMN** element are derived from the referenced **DMN** element [dmnElementRef].

DMN diagrams may be an incomplete or partial depiction of the content of the **DMN** model. Some **DMN** elements from a **DMN** model may not be present in any of the diagram instances being interchanged.

DMN DI does not directly provide for any containment concept. The DMNDiagram is an ordered collection of mixed DMNShape(s) and DMNEdge(s). The order of the DMNShape(s) and DMNEdge(s) inside a DMNDiagram determines their Z-order (i.e., what is in front of what). DMNShape(s) and DMNEdge(s) that are meant to be depicted “on top” of other DMNShape(s) and DMNEdge(s) MUST appear after them in the DMNDiagram. Thus, the exporting tool MUST order all DMNShape(s) and DMNEdge(s) such that the desired depiction can be rendered. Measurement UnitAs per OMG DD, all coordinates and lengths defined by DMN DI are assumed to be in user units, except when specified otherwise. A user unit is a value in the user coordinate system, which initially (before any transformation is applied) aligns with the device’s coordinate system (for example, a pixel grid of a display). A user unit, therefore, represents a logical rather than physical measurement unit. Since some applications might specify a physical dimension for a diagram as well (mainly for printing purposes), a mapping from a user unit to a physical unit can be specified as a diagram’s resolution. Inch is chosen in this specification to avoid variability, but tools can easily convert from/to other preferred physical units. Resolution specifies how many user units fit within one physical unit (for example, a resolution of 300 specifies that 300 user units fit within 1 inch on the device).

14.4.2 DMNDI [Class]

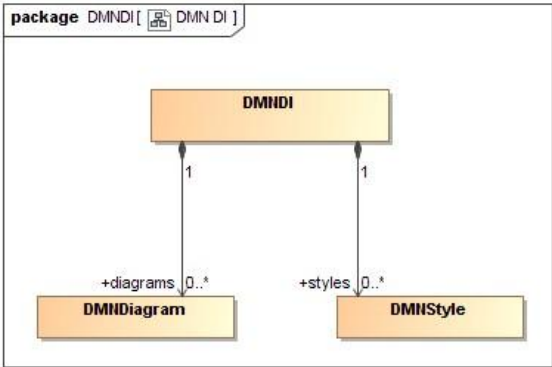


Figure 14-1: DMNDI

The class DMNDI is a container for the shared DMNStyle and all the DMNDiagram defined in a Definitions.

Commented [AF67]: Editorial: corrected figure numbering throughout section 14

Table 95: DMNDI attributes

Attribute	Description
styles: DMNStyle [0..*]	A list of shared DMNStyle that can be referenced by all DMNDiagram and DMNDiagramElement.
diagrams: DMNDiagram [0..*]	A list of DMNDiagram.

14.4.3 DMNDiagram [Class]

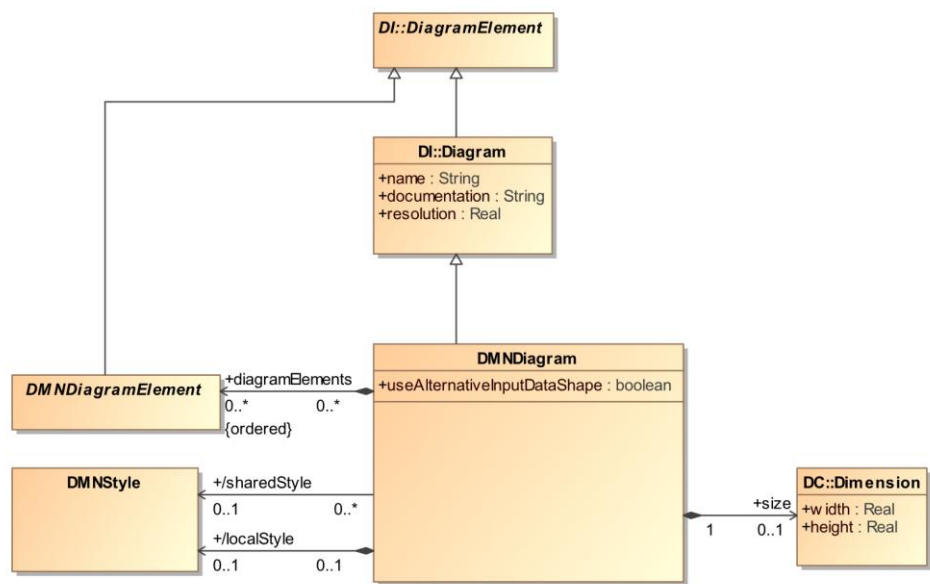


Figure 14-2: DMNDiagram

The class DMNDiagram specializes DI::Diagram. It is a kind of Diagram that represents a depiction of all or part of a DMN model.

DMNDiagram is the container of DMNDiagramElement (DMNShape(s) and DMNEdge(s)). DMNDiagram cannot include other DMNDiagram.

A DMNDiagram can define a DMNStyle locally and/or it can refer to a shared one defined in the DMNDI. Properties defined in the local style overrides the one in the referenced shared style. That combined style (shared and local) is the default style for all the DMNDiagramElement contained in this DMNDiagram.

The DMNDiagram class represents a two-dimensional surface with an origin of (0, 0) at the top left corner. This means that the x and y axes have increasing coordinates to the right and bottom. Only positive coordinates are allowed for diagram elements that are nested in a DMNDiagram.

The DMNDiagram has the following attributes.

Table 96: DMNDiagram attributes

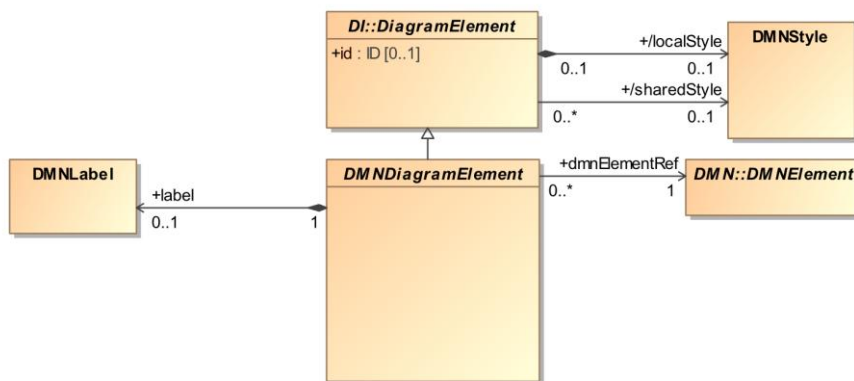
Attribute	Description
name: String	The name of the diagram. Default is empty String.
documentation: String	The documentation of the diagram. Default is empty String.
resolution: Real	The resolution of the diagram expressed in user units per inch. Default is 300

Formatted Table

diagramElements: DMNDiagramElement [0..*]	A list of DMNDiagramElement (DMNShape and DMNEdge) that are depicted in this diagram.
useAlternativeInputDataShape: Boolean [0..1]	If the DMNShape depicts an Input Data element then it is represented either using the paper sheet symbol, harmonized with BPMN and CMMN notations (true) or using the backwards compatible oval symbol (false).
sharedStyle: DMNStyle[0..1]	A reference to a DMNStyle defined in the DMNDI that serves as the default styling of the DMNDiagramElement in this DMNDiagram.
localStyle: DMNStyle [0..1]	A DMNStyle that defines the default styling for this diagram. Properties defined in that style override the ones in the sharedStyle.
size: DC::Dimension [0..1]	The size of this diagram. If not specified, the DMNDiagram is unbounded.

Commented [AF68]: DMN17-66

14.4.4 DMNDiagramElement [Class]



Commented [AF69]: DMN17-177

Figure 14-3: DMNDiagramElement

The DMNDiagramElement class is contained by the DMNDiagram and is the base class for DMNShape and DMNEdge.

DMNDiagramElement inherits its styling from its parent DMNDiagram. In addition, it can refer to one of the shared DMNStyle defined in the DMNDI and/or it can define a local style. See clause 13.4.9 for more details on styling.

DMNDiagramElement MAY also contain a DMNLabel when it has a visible text label. If no DMNLabel is defined, the DMNDiagramElement should be depicted without a label.

DMNDiagramElement has the following attributes:

Table 97: DMNDiagramElement attributes

Attribute	Description
dmnElementRef: DMNElement [1]	A reference to the DMNElement that is being depicted.
sharedStyle: DMNStyle [0..1]	A reference to a DMNStyle defined in the DMNDI.
localStyle: DMNStyle [0..1]	A DMNStyle that defines the styling for this element.
label: DMNLabel [0.. 1]	An optional label when this DMNElement has a visible text label.

14.4.5 DMNShape [Class]

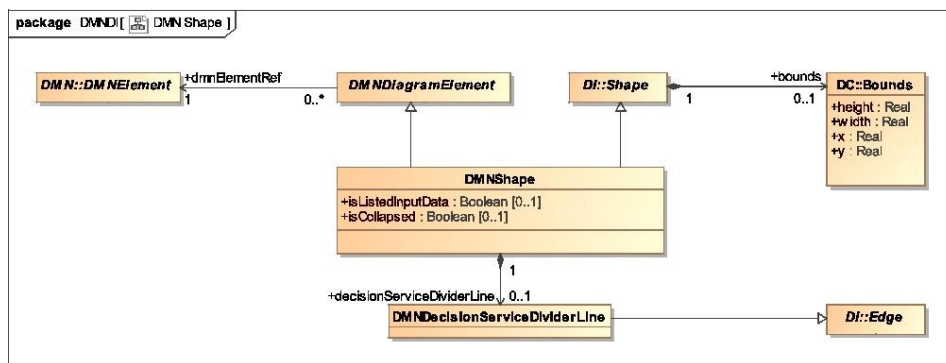


Figure 14-4: DMNShape

The DMNShape class specializes DI::Shape and DMNDiagramElement. It is a kind of Shape that depicts a DMNElement from the DMN model.

DMNShape represents a Decision, a Business Knowledge Model, an Input Data element, a Knowledge Source, a Decision Service or a Text Annotation that is depicted on the diagram.

DMNShape has three additional properties (isListedInputData, isCollapsed and decisionServiceDividerLine) that are used to further specify the appearance of some shapes that cannot be deduced from the DMN model.

DMNShape extends DI::Shape and DMNDiagramElement and has the following attributes:

Table 98: DMNShape attributes

Attribute	Description
bounds: DC::Bounds [1]	The Bounds of the shape relative to the origin of its parent DMNDiagram. The Bounds MUST be specified.
dmnElementRef: DMNElement [1]	A reference to a Decision, a Business Knowledge Model, an Input Data element, a Knowledge Source, a Decision Service, a Group or a Text Annotation MUST be specified.

isListedInputData: Boolean [0..1]	If the DMNShape depicts an Input Data element then this attribute is used to determine if the Input Data is listed on the Decision element (true) or drawn as separate notational elements in the DRD (false).
decisionServiceDividerLine: DMNDecisionServiceDividerLine [0..1]	If the DMNShape depicts a Decision Service, this attribute references a DMNDecisionServiceDividerLine which is a DI::Edge that defines s where the DMNShape is divided into two parts by a straight solid line. This can be the case when a DMNShape depicts a Decision Service, where the set of output decisions is smaller than the set of encapsulated decisions. The start and end waypoints of the decisionServiceDividerLine MUST be on the border of the DMNShape.
isCollapsed Boolean [0..1] = false	If the DMNShape depicts a DecisionService, this attribute indicates if it should be depicted expanded (false) or collapsed (true). Default is false.
useAlternativeInputDataShape: Boolean [0..1]	If the DMNShape depicts an Input Data element then it is represented either using the paper sheet symbol, harmonized with BPMN and CMMN notations (true) or using the backwards compatible oval symbol (false).

Commented [AF70]: DMN17-66

14.4.6 DMNEdge [Class]

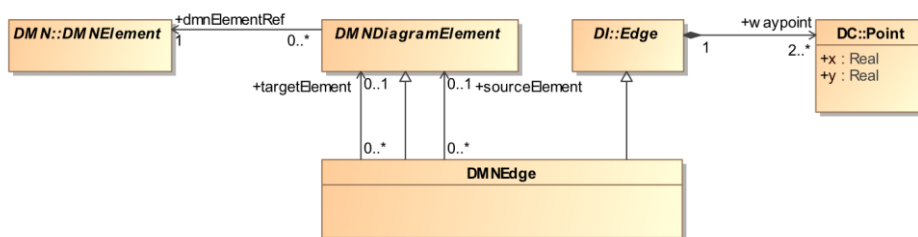


Figure 14-5: DMNEdge

The DMNEdge class specializes DI::Edge and DMNDiagramElement. It is a kind of Edge that can depict a relationship between two DMN model elements.

DMNEdge are used to depict Requirements or Associations in the DMN model. Since DMNDiagramElement might be depicted more than once, sourceElement and targetElement attributes allow to determine to which depiction a DMNEdge is connected. When DMNEdge has a source, its sourceModelElement MUST refer to the DMNDiagramElement it starts from. That DMNDiagramElement MUST resolved to the DMNElement that is the actual source of the Requirement or Association. For Requirement, this is the required DMNElement. When it has a target, its targetModelElement MUST refer to the DMNDiagramElement where it ends. That DMNDiagramElement MUST resolved to the DMNElement that is the actual target of the Requirement or Association. For Requirement, this is the DMNElement holding it.

DMNEdge extends DI : : Edge and has the following properties:

Table 99: DMNEdge attributes

Attribute	Description
wayPoints: DC : : Point [2..*]	A list of points relative to the origin of its parent DMNDiagram that specifies the connected line segments of the edge. At least two (2) waypoints MUST be specified.
dmnElementRef: DMNElement [1]	A reference to an InformationRequirement, KnowledgeRequirement, AuthorityRequirement or Association.
sourceElement: DMNDiagramElement[0.. 1]	The actual DMNDiagramElement this DMNEdge is connecting from. MUST be specified when the DMNEdge has a source.
targetElement: DMNDiagramElement[0.. 1]	The actual DMNDiagramElement this DMNEdge is connecting to. MUST be specified when the DMNEdge has a target.

14.4.7 DMNLabel [Class]

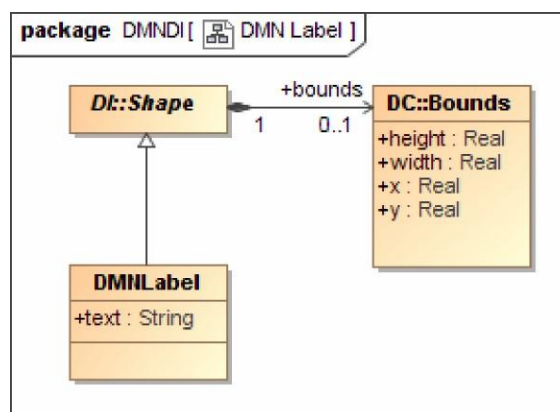


Figure 14-6: DMNLabel

DMNLabel represents the depiction of some textual information about a DMN element.

A DMN label is not a top-level element but is always nested inside either a DMNShape or a DMNEdge. It does not have its own reference to a **DMN** element but rather inherits that reference from its parent DMNShape or DMNEdge. The textual information depicted by the label is derived from the name attribute of the referenced DMNElement.

DMNLabel extends DI : : Shape and has the following properties:

Table 100: DMNLabel attributes

Attribute	Description
bounds: Bounds [0..1]	The bounds of the DMNLabel. When not specified, the label is positioned at its default position as determined in clause 13.5
text: String[0..1]	An optional pretty printed text that MUST be displayed instead of the DMNElement's name if it is present.

14.4.8 DMNStyle [Class]

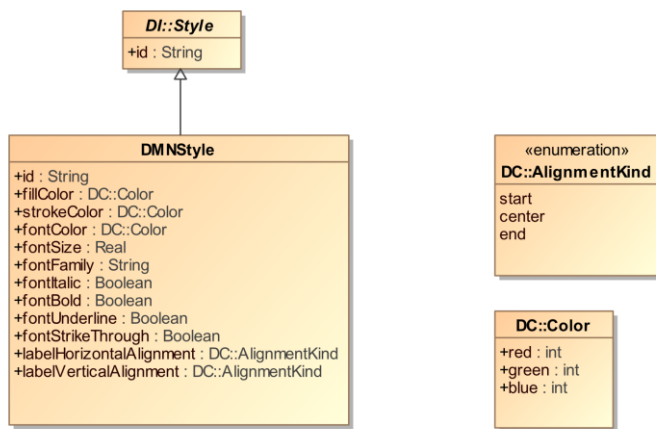


Figure 14-7: DMNStyle

DMNStyle specializes DC::Style. It is a kind of Style that provides appearance options for a DMNDiagramElement.

DMNStyle is used to keep some non-normative visual attributes such as colors and font. DMN doesn't give any semantic to color and font styling, but tools can decide to use them and interchange them.

DMNDiagramElement style is calculated by percolating up DMNStyle attributes defined at a different level of the hierarchy. Each attribute is considered independently (meaning that a DMNStyle attribute can be individually overloaded). The precedence rules are as follow:

- The DMNStyle defined by the localStyle attribute of the DMNDiagramElement
- The DMNStyle referenced by the sharedStyle attribute of the DMNDiagramElement
- The DMNStyle defined by the localStyle attribute of the parent DMNDiagram
- The DMNStyle referenced by the sharedStyle attribute of the parent DMNDiagram

The default attribute value defined in Table 101 (DMNStyle attributes).

For example, let's say we have the following:

- DMNDiagramElement has a local DMNStyle that specifies the fillColor and strokeColor
- Its parent DMNDiagram defines a local DMNStyle that specifies the fillColor and fontColor

Then the resulting `DMNDiagramElement` should use:

- The `fillColor` and `strokeColor` defined at the `DMNDiagramElement` level (as they are defined locally).
- The `fontColor` defined at the `DMNDiagram` level (as the `fillColor` was overloaded locally).
- All other `DMNStyle` attributes would have their default values.

`DMNStyle` extends `DC::Style` and has the following properties:

Table 101: `DMNStyle` attributes

Attribute	Description
id: String [0..1]	A unique id for this style so it can be referenced. Only styles defined in the <code>DMNDI</code> can be referenced by <code>DMNDiagramElement</code> and <code>DMNDiagram</code> .
fillColor: <code>DC::Color</code> [0..1]	The color use to fill the shape. Doesn't apply to <code>DMNEdge</code> . Default is white.
strokeColor: <code>DC::Color</code> [0..1]	The color use to draw the shape borders. Default is black.
fontColor: <code>DC::Color</code> [0..1]	The color use to write the label. Default is black.
fontFamily: String [0..1]	A comma-separated list of Font Name that can be used to display the text. Default is Arial.
fontSize: Real [0..1]	The size in points of the font to use to display the text. Default is 8.
fontItalic: Boolean [0..1]	If the text should be displayed in Italic. Default is false.
fontBold: Boolean [0..1]	If the text should be displayed in Bold. Default is false.
fontUnderline: Boolean [0..1]	If the text should be underlined. Default is false.
fontStrikeThrough: Boolean [0..1]	If the text should be stroke through. Default is false.
labelHorizontalAlignment: <code>AlignmentKind</code> [0..1]	How text should be positioned horizontally within the Label bounds. Default depends of the <code>DMNDiagramElement</code> the label is attached to (see 14.5).
label VerticalAlignment: <code>AlignmentKind</code> [0..1]	How the text should be positioned vertically inside the Label bounds. Default depends of the <code>DMNDiagramElement</code> the label is attached to (see 14.5). Start means "top" and end means "bottom".

14.5 Notation Depiction Library and Abstract Element Resolutions

As a notation, **DMN** specifies the depiction for each of the **DMN** elements.

Serializing a **DMN** diagram for interchange requires the specification of a collection of `DMNShape(s)` (see 14.4.6) and `DMNEdge(s)` (see 14.4.7) in the `DMNDiagram` (see 14.4.4). The `DMNShape(s)` and `DMNEdge(s)` attributes must be populated in such a way as to allow the unambiguous rendering of the **DMN** diagram by the receiving party. More

specifically, the DMNShape(s) and DMNEdge(s) MUST reference **DMN** model elements. If no DMNElement is referenced or if the reference is invalid, it is expected that this shape or edge should not be depicted.

When rendering a **DMN** diagram, the correct depiction of a DMNShape or DMNEdge depends mainly on the referenced **DMN** model element and its particular attributes and/or references. The purpose of this clause is to: provide a library of the **DMN** element depictions, and to provide an unambiguous resolution between the referenced **DMN** model element [DMNElement] and their depiction. Depiction resolution tables are provided below for both DMNShape (see 14.5.2) and DMNEdge (see 14.5.3).

14.5.1 Labels

Both DMNShape and DMNEdge may have labels (its name attribute) placed on the shape/edge, or above or below the shape/edge, in any direction or location, depending on the preference of the modeler or modeling tool vendor.

Labels are optional for DMNShape and DMNEdge. When there is a label, the position of the label is specified by the bounds of the DMNLabel of the DMNShape or DMNEdge. Simply put, label visibility is defined by the presence of the DMNLabel element.

The bounds of the DMNLabel are optional and always relative to the containing DMNDiagram's origin point. The depiction resolution tables provided below exemplify default label positions if no bounds are provided for the DMNLabel (for DMNShape kinds (see 14.5.2) and DMNEdge kinds (see 14.5.3)).

When the DMNLabel is contained in a DMNShape, the text to display is the name of the DMNElement.

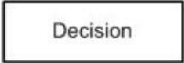
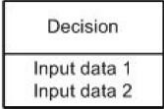
14.5.2 DMNShape Resolution

DMNShape can be used to represent a Decision, a Business Knowledge Model, an Input Data element, a Knowledge Source, a Text Annotation, a Group, and a Decision Service.

14.5.2.1 Decision

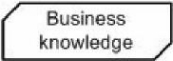
A Decision is represented in a DRD as a rectangle, normally drawn with solid lines. If the Listed Input Data option is exercised, all the Decisions requirements for Input Data shall be listed beneath the Decisions label and separated from it by a horizontal line. The listed Input Data names shall be clearly inside the shape of the DRD element.

Table 102: Depiction Resolution for Decision

DMNElement	DMNShape attributes	Depiction
Decision	None	
Decision and two Input Data	Shapes of Input Data have inListedInputData=true	

14.5.2.2 Business Knowledge Model

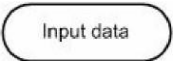
Table 103: Depiction Resolution for Business Knowledge Model

DMNElement	DMNShape attributes	Depiction
Business Knowledge Model	None	

14.5.2.3 Input Data Element

An Input Data element is represented in a DRD as a shape with two parallel straight sides and two semi-circular ends, normally drawn with solid lines.

Table 104: Depiction Resolution for Input Data

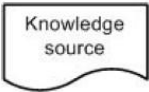
DMNElement	DMNShape-DMN Diagram attributes	Depiction
Input Data	None or useAlternativeInputDataShape = false	
Input Data	useAlternativeInputDataShape = true	

Commented [AF71]: DMN17-66

14.5.2.4 Knowledge Source

A Knowledge Source is represented as a shape with three straight sides and one wavy one, normally drawn with solid lines.

Table 105: Depiction Resolution for Knowledge Source

DMNElement	DMNShape attributes	Depiction
Knowledge Source	None	

14.5.2.5 Artifacts

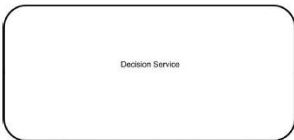
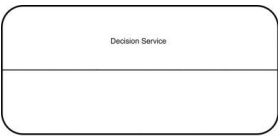
Table 106: Depiction Resolution of Artifacts

DMNElement	DMNShape Attributes	Depiction
TextAnnotation	None	
Group	None	

14.5.2.6 Decision Service

If the set of output decisions is smaller than the set of encapsulated decisions, the Decision Service shall be divided into two parts with a straight solid line.

Table 107: Depiction Resolution of Decision Service

DMNElement	DMNShape attributes	Depiction
Decision Service	None or isCollapsed=false	
Decision Service	DecisionServiceDividerLine isCollapsed=false	
Decision Service	isCollapsed=true	

14.5.3 DMNEdge Resolution

14.5.3.1 Information Requirement

Table 108: Depiction Resolution of Information Requirement

DMNElement	Depiction
Information Requirement	

14.5.3.2 Knowledge Requirement

Table 109: Depiction Resolution of Knowledge Requirement

DMNElement	Depiction
Knowledge Requirement	

14.5.3.3 Authority Requirement

Table 110: Depiction Resolution of Authority Requirement

DMNElement	Depiction
Authority Requirement	

14.5.3.4 Association

When the DMNEdge depicts an Association, its DMNElement MUST be specified.

Table 111: Depiction Resolution of Association

DMNElement	Depiction
Association where associationDirection is none.	
Association where associationDirection is one.	
Association where associationDirection is both.	

This page intentionally left blank.

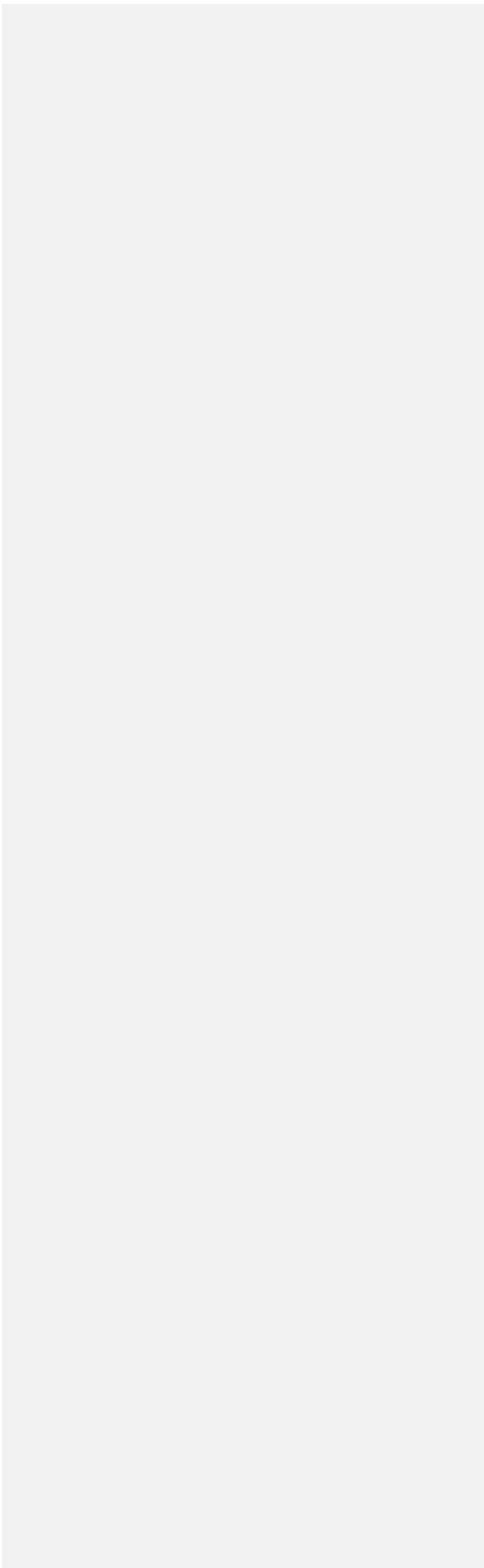
ANNEXES

All the Annexes are informative.

Annex A. discuss issues around the application of **DMN** in combination with **BPMN**. This section is intended to provide some direction to practitioners but is non-normative.

Annex B. provides a non-normative glossary to aid comprehension of the specification.

This page intentionally left blank.



Annex A

Relation to BPMN

(informative)

A.1 Goals of BPMN and DMN

The OMG Business Process Model and Notation Standard provides a standard notation for describing business processes as orchestrations of tasks. The success of **BPMN** has provided a major motivation for **DMN**, and business decisions described using **DMN** are expected to be commonly deployed in business processes described using **BPMN**.

All statements pertaining to **BPMN** below are from the OMG document reference 11-01-03 unless otherwise stated.

BPMN's goals are stated in the specification and provide easy comparisons to **DMN**:

- Goal 1: *“The primary goal of **BPMN** is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the businesspeople who will manage and monitor those processes. Thus, **BPMN** creates a standardized bridge for the gap between the business process design and process implementation.”*. **DMN** users will also be business analysts (designing decisions) and then business users (populating decision models such as decision tables). Technical developers may be responsible for mapping business terms to appropriate data technologies. Therefore, **DMN** can also be said to bridge the decision design by a business analyst, and the decision implementation, typically using some decision execution technology.
- Goal 2: *“... To ensure that XML languages designed for the execution of business processes, such as WSBPEL (Web Services Business Process Execution Language), can be visualized with a businessoriented notation.”* It is not a stated goal of **DMN** to be able to visualize other XML languages (such as W3C RIF or OMG PRR); indeed, it is expected that **DMN** would provide the MDA specification layer for such languages. It does not preclude however the use of **DMN** (such as decision tables) to represent executable forms (such as production rules).
- Goal 3: *“The intent of **BPMN** is to standardize a business process model and notation in the face of many different modeling notations and viewpoints. In doing so, **BPMN** will provide a simple means of communicating process information to other business users, process implementers, customers, and suppliers.”* Similarly, the intent of **DMN** is to standardize the decision model and notation across the many different implementations of broadly semantically similar models. In so doing, **DMN** will also facilitate the communication of decision information across business communities and tools.

A.2 BPMN Tasks and DMN Decisions

Most **BPMN** diagrams contain some tasks which involve decision-making which can be modeled in **DMN**. These tasks take input data acquired or generated earlier in the process and produce decision outputs which are used later in the process. Decision outputs may be used in two principal ways:

- They may be consumed in another process task.
- They may influence the choice of sequence flows out of a gateway.

In the latter case, decisions are used to determine which subprocesses or tasks are to be executed (in the process sense). As such, **DMN** complements **BPMN** as decision modeling complements process modeling (in the sense of defining orchestrations or work tasks).

For example, Figure A.1 shows an example¹ of a **BPMN**-defined process.

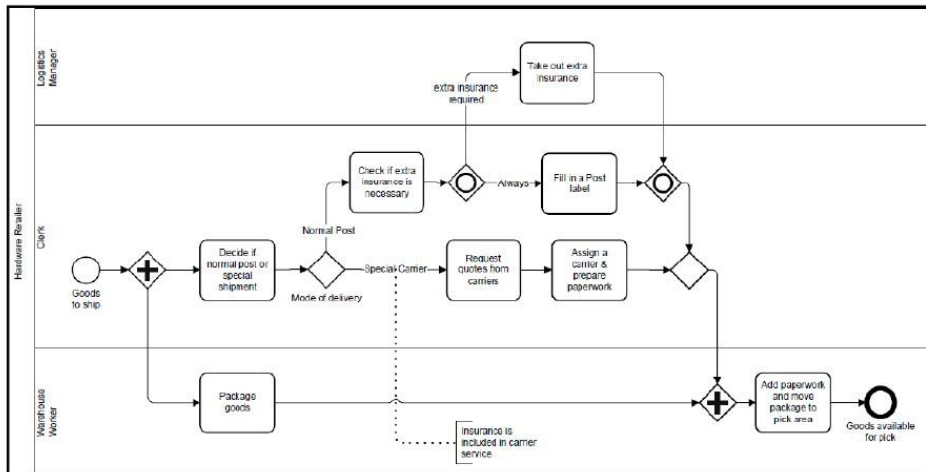


Figure A.1: Decision-making in BPMN

Analyzing this we see:

- A task whose title starts with “Decide...” which makes a decision on (whether to use) normal post or special shipment, and which precedes an exclusive gateway using that decision result.
- A task whose title starts with “Check...” which makes a decision on whether extra insurance is necessary, which precedes an inclusive gateway for which an additional process path may be executed based on the decision result.
- A task whose title starts with “Assign...” which implies a decision to select a carrier based on some selection criteria. The previous task is effectively collecting data for this decision. In an automated system this would probably be a subprocess embedding a decision and some other activities (such as “prepare paperwork”).

From this example we can see that even a simple business process in **BPMN** may have several decision-making tasks.

A.3 Types of BPMN Tasks relevant to DMN

BPMN defines² different types of tasks that can be considered for decision-making roles. The relevant tasks are as shown in Table 111:

1.Shipment Process in a Hardware Retailer example, Ch5.1, BPMN 2.0 By Example, June 2010, OMG reference 10-06-02
2.See ch 10.2.3 in the BPMN Specification.

Table 111: BPMN tasks relevant to DMN

	Task type(s)	Decision role
1	Loop Multi-Instance Compensation 	None explicitly. Although a process for a decision may make iterations or loop (such as production rules executing Run To Completion cycles in a Rete-based rules engine), these are not considered relevant at the business modeling level.
2	Service Task 	Decision tasks will be executed (when automated) by a decision service. However a decision model is not guaranteed to be executed automatically in a business process.
3	User Task 	Decision tasks executed manually as a part of a workflow-oriented business process may be specified as a User Task.
4	Business Rule Task 	The Business Rule Task was defined in BPMN 2 as a placeholder for (business-rule-driven) decisions, and is the natural placeholder for a decision task. <i>Note that business rules (as defined in OMG SBVR) can constrain any type of process activity, not just business decisions.</i>
5	Script Task 	Decision tasks may today be encoded using business process script languages.

A future version of **BPMN** may choose to clarify and extend the definitions of task to better match decision modeling requirements and **DMN** – to wit, to define a **BPMN** Decision Task as some task used to make a decision modeled with **DMN**. In the meantime, the Business Rule Task is the most natural way to express this functionality. However, as noted in clauses 5.2.2 and 6.3.6, a Decision in **DMN** can be associated with any Task, allowing for flexibility in implementation.

A.4 Process gateways and Decisions

Process gateways can be considered of 2 types:

1. A gateway that determines a process route or routes based on existing data
2. A gateway that determines a process route or routes based on the outcome of one or more decisions that are determined by some previous task within the process.

In the latter case, a Decision Task (task used to make a decision using **DMN**) may need an extended notation to clarify the relationship of the decision task to the gateway(s) that use it.

A.5 Linking BPMN and DMN Models

DMN offers two approaches to linking business process models in **BPMN** with decision models: one normative and the other non-normative:

a) Associating Decisions with Tasks and Processes

As described in clause 6.3.6, in **DMN**, the process context for an instance of `Decision` is defined by its association with any number of `usingProcesses`, which are instances of `Process` as defined in **OMG BPMN 2**, and any number of `usingTasks`, which are instances of `Task` as defined in **OMG BPMN 2**. Each decision may therefore be associated with one or more business processes (to indicate that the decision is taken during those processes), and/or with one or more specific tasks (to indicate that the tasks involve making the decision). An implementation **SHALL** allow these associations to be defined for each decision.

An implementation **MAY** perform validation over the two (**BPMN** and **DMN**) models, to check, for example, that:

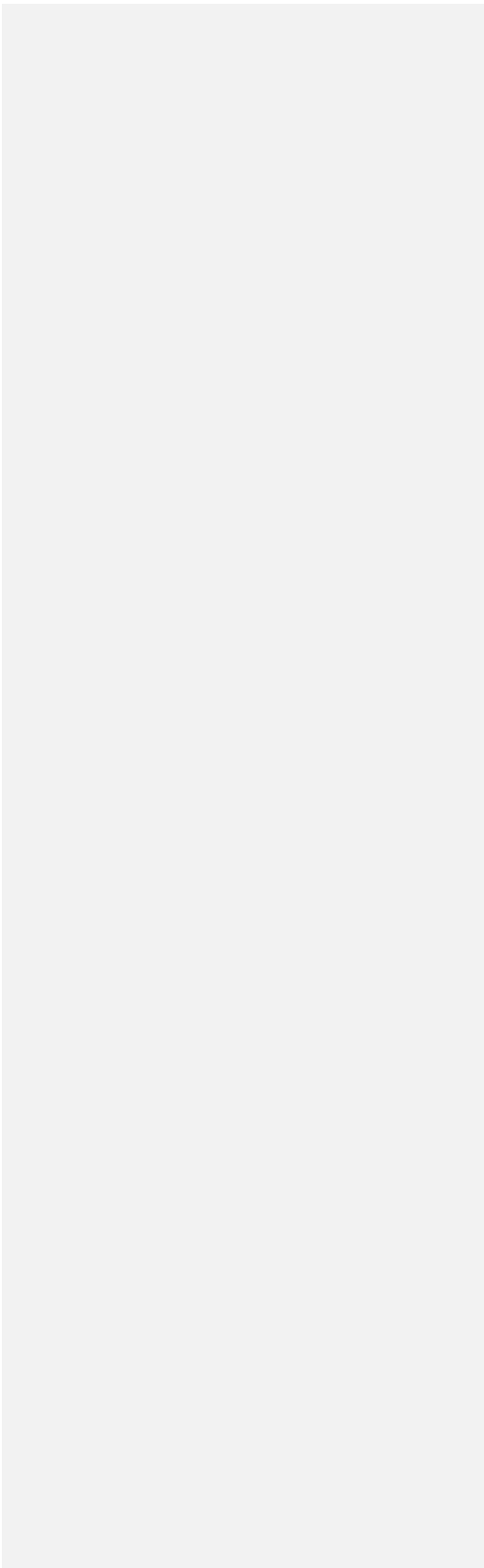
- A Decision is not associated with Tasks that are part of Processes not also associated with the Decision.
- A Decision is not associated with Tasks that are not part of any Process associated with the Decision.

During development it may be appropriate to associate a Decision only with a Process, but inconsistency between Task and Process associations is not allowed.

Note that this approach allows the relationships between business process models and decision models to be defined and validated but does not of itself permit the decisions modeled in **DMN** to be executed automatically by processes modeled in **BPMN**.

b) Decision Services

One approach to decision automation is described non-normatively in Annex A: the encapsulation of **DMN** Decisions in a “decision service” called from a **BPMN** Task (e.g., a Service Task or Business Rule Task, as discussed in Annex A.3 above). The `usingProcesses` and `usingTasks` properties allow definition and validation of associations between **BPMN** and **DMN**; the definition of decision services then provides a detailed specification of the required interface.



Annex B Glossary

(informative)

A

Aggregation

The production of a single result from multiple **hits** on a **decision table**. DMN specifies four aggregation operators on the Collect hit policy, namely: + (sum), < (min), > (max), # (count). If no operator is specified, the results of the Collect hit policy are returned without being aggregated.

Any

A **hit policy** for **single hit decision tables** with overlapping **decision rules**: under this policy any match may be used.

Authority
Requirement

The dependency of one element of a Decision Requirements Graph on another element which provides guidance to it or acts as a source of knowledge for it.

B

Binding

In an **invocation**, the association of the parameters of the invoked expression with the input variables of the invoking expression, using a binding formula.

Boxed Context

A form of **boxed expression** showing a collection of n (name, value) pairs with an optional result value.

Boxed Expression

A notation serving to decompose **decision logic** into small pieces which may be associated graphically with elements of a **DRD**.

Boxed Function	A form of boxed expression showing the kind, parameters, and body of a function.
Boxed Invocation	A form of boxed expression showing the parameter bindings that provide the context for the evaluation of the body of a business knowledge model .
Boxed List	A form of boxed expression showing a list of n items.
Boxed Literal Expression	A form of boxed expression showing a literal expression .
Business Context Element	An element representing the business context of a decision: either an organisational unit or a performance indicator .
Business Knowledge Model	Some decision logic (e.g., a decision table) encapsulated as a reusable function, which may be invoked by decisions or by other business knowledge models .

C

Clause	In a decision table , a clause specifies a subject, which is defined by an input expression or an output domain, and the finite set of the subdomains of the subject's domain that are relevant for the piece of decision logic that is described by the decision table.
--------	--

Collect

A hit policy for multiple hit decision tables with overlapping decision rules: under this policy all matches will be returned as a list in an arbitrary order. An operator can be added to specify a function to be applied to the outputs: see Aggregation.

Context

In **FEEL**, a map of key-value pairs called **context entries**.

Crosstab Table

An **orientation** for **decision tables** in which two **input expressions** form the two dimensions of the table, and the **output entries** form a twodimensional grid.

D

Decision

The act of determining an **output value** from a number of **input values**, using **decision logic** defining how the output is determined from the inputs.

Decision Logic	The logic used to make decisions, defined in DMN as the value expressions of decisions and business knowledge models and represented visually as boxed expressions .
Decision Logic Level	The detailed level of modeling in DMN, consisting of the value expressions associated with decisions and business knowledge models .
Decision Model	A formal model of an area of decision-making, expressed in DMN as decision requirements and decision logic .
Decision Point	A point in a business process at which decisionmaking occurs, modeled in BPMN 2.0 as a business rule task and possibly implemented as a call to a decision service .
Decision Requirements Diagram	A diagram presenting a (possibly filtered) view of a DRG .
Decision Requirements Graph	A graph of DRG elements (decisions , business knowledge models and input data) connected by requirements .

Decision Requirements Level	The more abstract level of modelling in DMN, consisting of a DRG represented in one or more DRDs .
Decision Rule	In a decision table , a decision rule specifies associates a set of conclusions or results (output entries) with a set of conditions (input entries).
Decision Service	A software component encapsulating a decision model and exposing it as a service, which might be consumed (for example) by a task in a BPMN process model.
Decision Table	A tabular representation of a set of related input and output expressions, organized into decision rules indicating which output entry applies to a specific set of input entries .
Definitions	A container for all elements of a DMN decision model . The interchange of DMN files will always be through one or more Definitions.
DMN Element	Any element of a DMN decision model : a DRG Element , Business Context Element , Expression , Definitions , Element Collection , Information Item or Item Definition .

DRD	See Decision Requirements Diagram .
DRG	See Decision Requirements Graph .
DRG Element	Any component of a DRG : a decision , business knowledge model , input data or knowledge source .
E	
Element Collection	Used to define named groups of DRG elements within a Definitions .
Expression	A literal expression , decision table , invocation , list, context , function definition, or relation used to define part of the decision logic for a decision model in DMN . Returns a single value when interpreted.
F	
FEEL	The "Friendly Enough Expression Language" which is the default expression language for DMN .

First	A hit policy for single hit decision tables with overlapping decision rules : under this policy the first match is used, based on the order of the decision rules .
Formal Parameter	A named, typed value used in the invocation of a function to provide an information item for use in the body of the function.
H	
Hit	In a decision table , the successful matching of all input expressions of a decision rule , making the conclusion eligible for inclusion in the results.
Horizontal	An orientation for decision tables in which decision rules are presented as rows, clauses as columns.
I	
Information Item	A DMN element used to model either a variable or a parameter at the decision logic level in DMN decision models .
Information Requirement	The dependency of a decision on an input data element or another decision to provide a variable used in its decision logic .
Input Data	Denotes information used as an input by one or more decisions , whose value is defined outside of the decision model .

Input Entry	An expression defining a condition cell in a decision table (i.e., the intersection of a decision rule and an input clause).
Input Expression	An expression defining the item to be compared with the input entries of an input clause in a decision table .
Input Value	An expression defining a limited range of expected values for an input clause in a decision table .
Invocation	A mechanism that permits the evaluation of one value expression another, using a number of bindings .
Item Definition	Used to model the structure and the range of values of input data and the outcome of decisions , using a type language such as FEEL or XML Schema.

K

Knowledge Requirement	The dependency of a decision or business knowledge model on a business knowledge model which must be invoked in the evaluation of its decision logic .
-----------------------	--

Knowledge Source An authority defined for **decisions** or **business knowledge models**, e.g., domain experts responsible for defining or maintaining them, or source documents from which business knowledge models are derived or sets of test cases with which the decisions must be consistent.

L

Literal Expression Text that represents **decision logic** by describing how an output value is derived from its input values, e.g. in plain English or using the default expression language **FEEL**.

M

Multiple Hit A type of **decision table** which may return **output entries** from multiple **decision rules**.

O

Organisational Unit A **business context element** representing the unit of an organization which makes or owns a **decision**.

Orientation The style of presentation of a **decision table**: horizontal (decision rules as rows; clauses as columns), vertical (rules as columns; clauses as rows), or crosstab (rules composed from two input dimensions).

Output Entry	An expression defining a conclusion cell in a decision table (i.e., the intersection of a decision rule and an output clause).
Output Order	A hit policy for multiple hit decision tables with overlapping decision rules : under this policy all matches will be returned as a list in decreasing priority order. Output priorities are specified in an ordered list of values.
Output Value	An expression defining a limited range of domain values for an output clause in a decision table .

P

Performance Indicator	A business context element representing a measure of business performance impacted by a decision .
-----------------------	--

Priority	A hit policy for single hit decision tables with overlapping decision rules: under this policy the match is used that has the highest output priority. Output priorities are specified in an ordered list of values.
----------	---

R

Relation	A form of boxed expression showing a vertical list of homogeneous horizontal contexts (with no result cells) with the names appearing just once at the top of the list, like a relational table.
----------	--

Requirement	The dependency of one DRG element on another: either an information requirement , knowledge requirement or authority requirement .
Requirement Subgraph	The directed graph resulting from the transitive closure of the requirements of a DRG element ; i.e., the sub-graph of the DRG representing all the decision-making required by a particular element.
Rule Order	A hit policy for multiple hit decision tables with overlapping decision rules : under this policy all matches will be returned as a list in the order of definition of the decision rules .
S	
S-FEEL	A simple subset of FEEL , for decision models that use only simple expressions : in particular, decision models where the decision logic is modeled mostly or only using decision tables .
Single Hit	A type of decision table which may return the output entry of only a single decision rule .
U	
Unique	A hit policy for single hit decision tables in which no overlap is possible and all decision rules are exclusive. Only a single rule can be matched.

V

Variable Represents a value that is input to a **decision**, in the description of its **decision logic**, or a value that is passed as a **parameter** to a function.

Vertical An **orientation** for **decision tables** in which decision rules are presented as columns; clauses as rows.

W

Well-Formed Used of a **DRG element** or **requirement** to indicate that it conforms to constraints on referential integrity, acyclicity etc.

