

Date: January 2014



Decision Model and Notation

Beta1

OMG Document Number:

Standard document URL: <http://www.omg.org/spec/DMN/1.0>

Machine Consumable File(s):

<http://www.omg.org/spec/DMN/20130901/DMN.xmi>

<http://www.omg.org/spec/DMN/20130901/DMN.xsd>

<http://www.omg.org/spec/DMN/20130901/DMNLevel3.xsd>

This OMG document replaces the submission document (bmi/2013-08-01, Alpha). It is an OMG Adopted Beta specification and is currently in the finalization phase. Comments on the content of this document are welcome, and should be directed to issues@omg.org by March 31, 2014.

You may view the pending issues for this specification from the OMG revision issues web page <http://www.omg.org/issues/>.

The FTF Recommendation and Report for this specification will be published on September 26, 2014. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

Copyrights

Copyright © 2013, Decision Management Solutions

Copyright © 2013, Escape Velocity LLC

Copyright © 2013, Fair Isaac Corporation

Copyright © 2013, International Business Machines Corporation

Copyright © 2013, Knowledge Partners International

Copyright © 2013, KU Leuven

Copyright © 2013, Model Systems Limited

Copyright © 2013, Oracle Incorporated

Copyright © 2013, TIBCO Software Inc.

Copyright © 2014, Object Management Group, Inc.

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, 109 Highland Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™, Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™, IMM™, OMG Interface Definition Language (IDL)™, and OMG SysML™ are trademarks of the Object Management Group. 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 <http://www.omg.org>, under Documents, Report a Bug/Issue (http://www.omg.org/report_issue.htm.)

Contents

Contents	6
Preface	11
OMG 11	
OMG Specifications	11
Typographical Conventions	12
1 Scope	13
2 Conformance	14
2.1 Conformance levels	14
2.2 General conformance requirements	14
2.2.1 Visual appearance.....	14
2.2.2 Decision semantics.....	15
2.2.3 Attributes and model associations	15
3 References	16
3.1 Normative	16
3.2 Non-normative	16
4 Additional Information	17
4.1 Acknowledgements	17
4.2 IPR and Patents	17
4.3 Guide to the Specification	17
5 Introduction to DMN	19
5.1 Context	19
5.2 Basic concepts	23
5.2.1 Decision requirements level.....	23
5.2.2 Decision logic level	24
5.3 Scope and uses of DMN	27
5.3.1 Modeling human decision-making.....	27
5.3.2 Modeling requirements for automated decision-making	28
5.3.3 Implementing automated decision-making.....	28
5.3.4 Combining applications of modeling	29
6 Decision Requirements (DRG and DRD)	30
6.1 Introduction	30

6.2	Notation.....	30
6.2.1	DRD Elements.....	31
6.2.2	DRD Requirements.....	33
6.2.3	Connection rules.....	35
6.2.4	Partial views and hidden information.....	35
6.3	Metamodel.....	37
6.3.1	DMN Element metamodel	37
6.3.2	Definitions metamodel	38
6.3.3	Import metamodel.....	40
6.3.4	Element Collection metamodel	40
6.3.5	DRG Element metamodel.....	41
6.3.6	Decision metamodel	42
6.3.7	Business Context Element metamodel.....	45
6.3.8	Business Knowledge Model metamodel.....	47
6.3.9	Input Data metamodel	49
6.3.10	Knowledge Source metamodel.....	49
6.3.11	Information Requirement metamodel.....	50
6.3.12	Knowledge Requirement metamodel.....	51
6.3.13	Authority Requirement metamodel.....	52
6.4	Examples.....	53
7	Relating Decision Logic to Decision Requirements.....	54
7.1	Introduction.....	54
7.2	Notation.....	57
7.2.1	Boxed Expressions	57
7.2.2	Boxed literal expression.....	58
7.2.3	Boxed invocation.....	58
7.3	Metamodel.....	59
7.3.1	Expression metamodel	60
7.3.2	ItemDefinition metamodel.....	62
7.3.3	InformationItem metamodel.....	64
7.3.4	Literal expression metamodel.....	65
7.3.5	Invocation metamodel	67
7.3.6	Binding metamodel.....	68

8	Decision Table	70
8.1	Introduction	70
8.2	Notation	70
8.2.1	Line style and color	71
8.2.2	Table orientation	71
8.2.3	Input expressions	72
8.2.4	Input values	72
8.2.5	Table name and output name	72
8.2.6	Output values	72
8.2.7	Multiple outputs	73
8.2.8	Input entries	74
8.2.9	Merged input entry cells	74
8.2.10	Output entry	75
8.2.11	Hit policy	75
8.2.12	Completeness indicator	77
8.2.13	Aggregation	77
8.3	Metamodel	78
8.3.1	Decision Table metamodel	78
8.3.2	Decision Table Clause metamodel	81
8.3.3	Decision Rule metamodel	82
8.4	Examples	83
9	Simple Expression Language (S-FEEL)	88
9.1	S-FEEL syntax	88
9.2	S-FEEL data types	89
9.3	S-FEEL semantics	90
9.4	Use of S-FEEL expressions	91
9.4.1	Item definitions	92
9.4.2	Invocations	92
9.4.3	Decision tables	92
10	Expression Language (FEEL)	93
10.1	Introduction	93
10.2	Notation	93
10.2.1	Boxed Expressions	93

10.2.2	FEEL	101
10.3	Full FEEL syntax and semantics.....	103
10.3.1	Syntax	103
10.3.2	Semantics	108
10.3.3	XML Data	127
10.3.4	Built-in functions	129
10.4	Relationship of FEEL to DRG and Boxed Expressions.....	135
10.5	Metamodel.....	137
10.5.1	Context metamodel.....	137
10.5.3	FunctionDefinition metamodel.....	138
10.5.4	List metamodel	139
10.5.5	Relation metamodel.....	139
10.6	Examples.....	140
10.6.1	Context	140
10.6.2	Calculation	141
10.6.3	If, In.....	141
10.6.4	Sum entries of a list.....	141
10.6.5	Invocation of user-defined PMT function.....	141
10.6.6	Sum weights of recent credit history.....	141
10.6.7	Determine if credit history contain a bankruptcy event.....	141
11	DMN Example	142
11.1	The business process model.....	142
11.2	The decision requirements level	143
11.3	The decision logic level.....	146
12	Exchange formats.....	157
12.1	Interchanging Incomplete Models.....	157
12.2	Machine Readable Files	157
12.3	XSD	157
12.3.1	Document Structure	157
12.3.2	References within the DMN XSD	157
Annex A.	Relation to BPMN.....	160
1.	Goals of BPMN and DMN.....	160
2.	BPMN Tasks and DMN Decisions.....	160

3.	Types of BPMN Tasks relevant to DMN	162
4.	Process gateways and Decisions	163
5.	Linking BPMN and DMN Models	163
Annex B.	Decision services	165
Annex C.	Glossary	167

Preface

OMG

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 Metamodel); and industry-specific standards for dozens of vertical markets.

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

<http://www.omg.org/spec>

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- **CORBAServices**
- **CORBAFacilities**

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

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
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

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

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

Courier/Courier New - 10 pt. Bold: Programming language elements.

Courier - 12 pt.: Name of modeling element (class or association)

Arial - 12pt.: syntax element.

Arial - 10 pt.: Examples and non-normative remarks

Helvetica/Arial - 10 pt: Exceptions

NOTE: Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

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 business people 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 useable 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 has 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 1.0 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 **MUST** 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. 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 1.0 conformance levels shall comply with all of the requirements set forth in Clause 2.2.

2.2 General conformance requirements

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.

An extension SHALL NOT change the specified shape of a defined graphical element or marker (e.g., changing a dashed line into a plain line, changing a square into a triangle, or changing rounded corners into squared corners).

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

BPMN 2.0

- Business Process Model and Notation, version 2.0, OMG Document Number: dtc/2010-06-05, June 2010.

<http://www.omg.org/spec/BPMN/2.0>

RIF

- RIF Core dialect, H. Boley et al. (Eds.) , W3C Recommendation, 22 June 2010.
- RIF data types and built-ins 1.0, A. Polleres et al. (Eds.) , W3C Recommendation, 22 June 2010.
- RIF production rule dialect, Ch. de Sainte Marie et al. (Eds.) , W3C Recommendation, 22 June 2010.

<http://www.w3.org/TR/rif-core/>

[http://www.w3.org/TR/rif-dtb/XML 1.0 \(Second Edition\)](http://www.w3.org/TR/rif-dtb/XML%201.0%20(Second%20Edition))

<http://www.w3.org/TR/rif-prd/>

RFC-2119

- Key words for use in RFCs to Indicate Requirement Levels, S. Bradner, IETF RFC 2119, March 1997.

<http://www.ietf.org/rfc/rfc2119.txt>

3.2 Non-normative

None

4 Additional Information

4.1 Acknowledgements

The following companies submitted 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.

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 an example of DMN used to model automated decision-making in a simple business process.

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 suggests principles for encapsulating decision models as decision services
- Annex C provides a glossary of terms.

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 1.

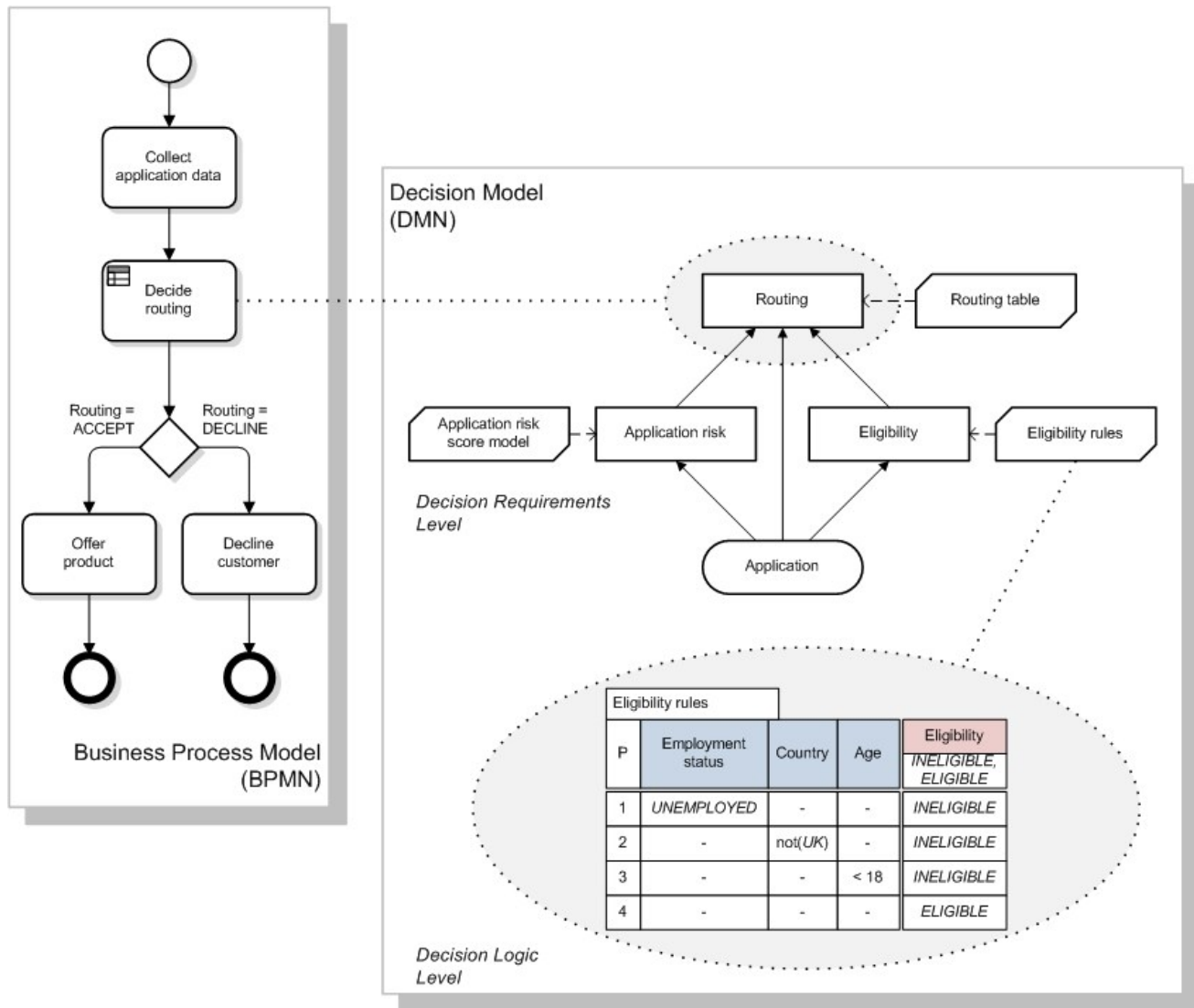


Figure 1: Aspects of modeling

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 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 clause 5.3).

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 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 2.

Again, although Figure 2 depicts these decision modeling constructs as interlinked, it is possible to use them independently or in any combination. For example, it is possible to use DMN only to draw DRDs, or only to define decision tables, or only to write FEEL expressions.

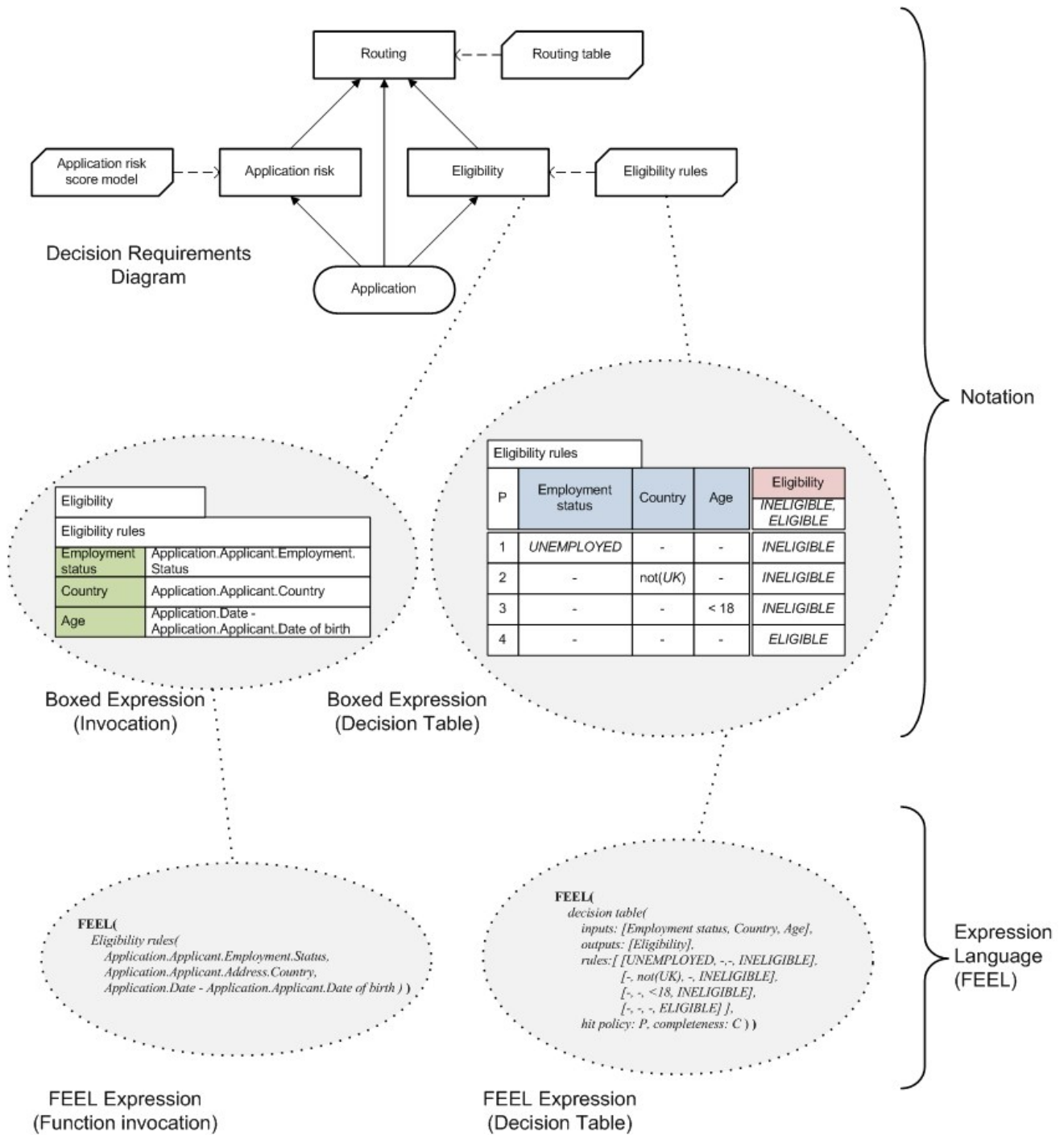


Figure 2: DMN Constructs

5.2 Basic concepts

5.2.1 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 3.

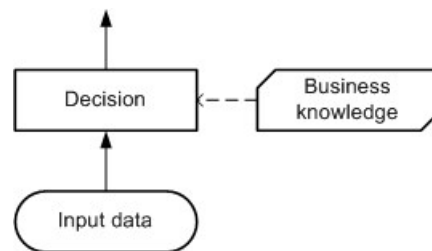


Figure 3: Basic elements of a decision model

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 4).

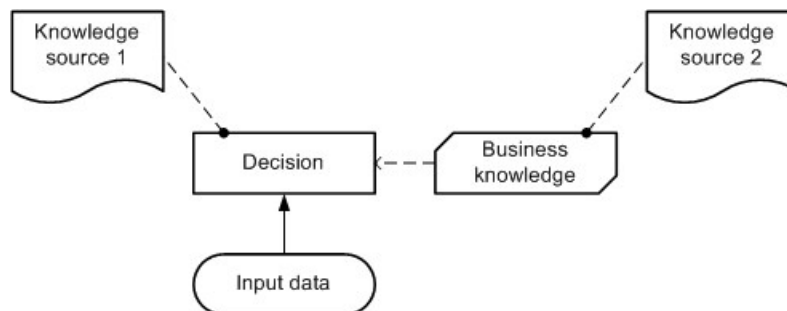


Figure 4: Knowledge sources

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.

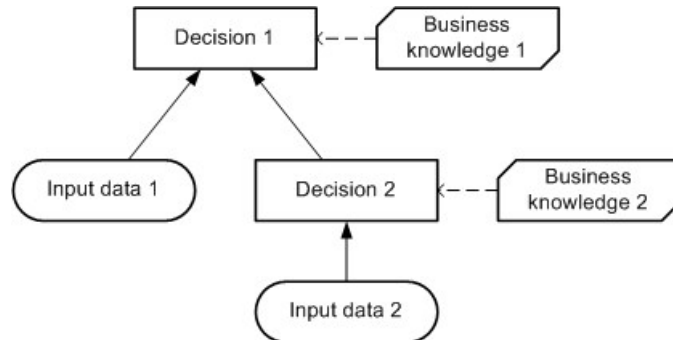


Figure 5: A simple Decision Requirements Diagram (DRD)

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

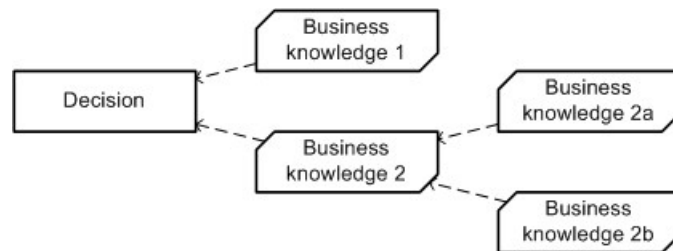


Figure 6: Combining business knowledge models

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

5.2.2 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 1 and Figure 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 7.

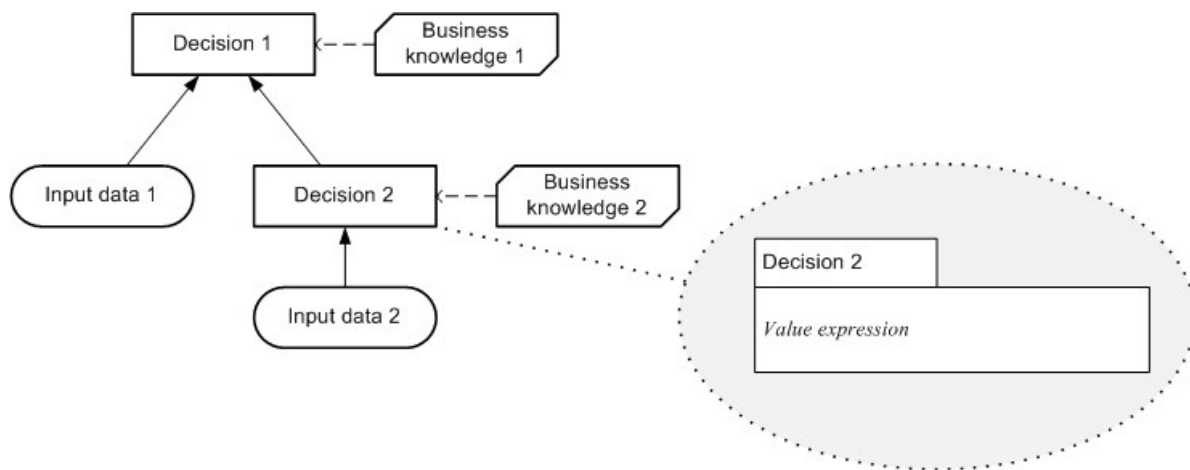


Figure 7: Decision and corresponding value expression

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. Value expressions may be encapsulated as **functions**, which may be invoked from decisions’ value expressions; business knowledge models are examples of such functions (but decision logic may also include functions which do not correspond to business knowledge models). The interpretation of business knowledge models as functions in DMN means that the combination of business knowledge models as in Figure 6 has the clear semantics of functional composition. The value expression of a business knowledge model may be notated using a **boxed function**, as shown in Figure 8.

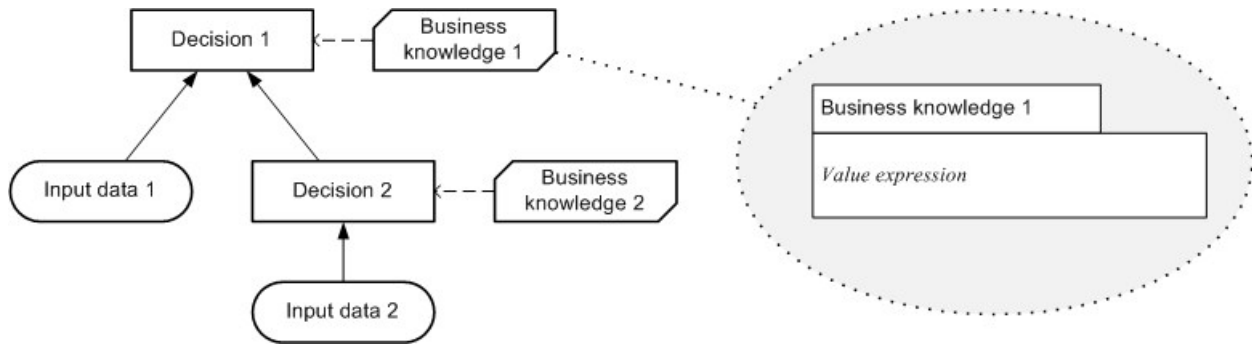


Figure 8: Business knowledge model and corresponding value expression

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 9.

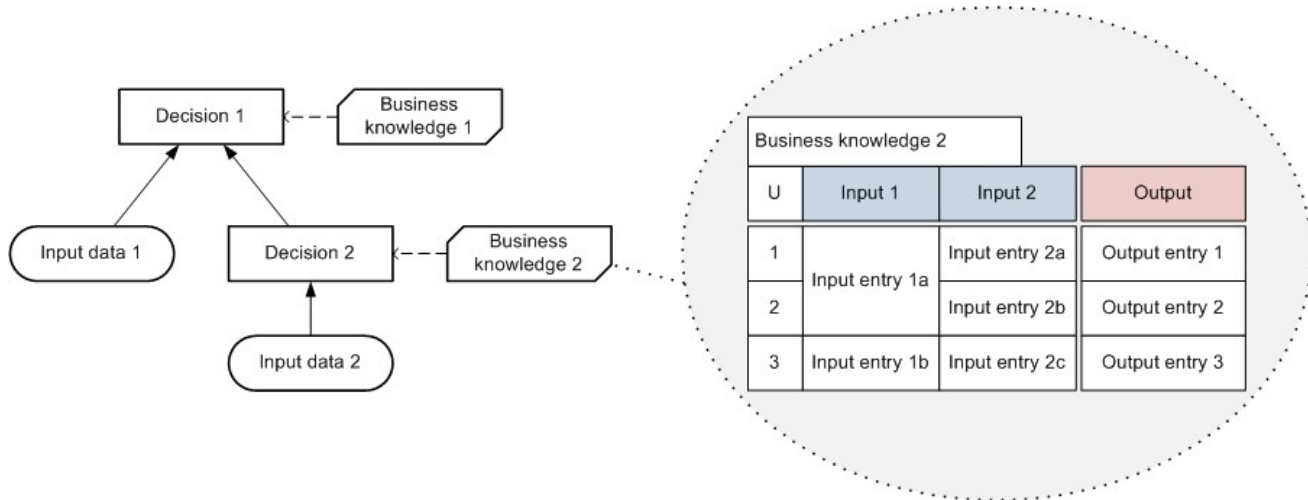


Figure 9: Business knowledge model and corresponding decision table

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 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.

5.3.1 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.

5.3.2 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. In DMN 1.0 no formal mapping of DMN ItemDefinitions 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.

5.3.3 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.

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.

5.3.4 Combining applications of modeling

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.

6 Decision 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, and input data:

- 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.

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 by the decision logic of a Decision.
- An **Authority Requirement** denotes the dependence of a DRG element on a Knowledge Source, or the dependence of a Knowledge Source on a DRG element.

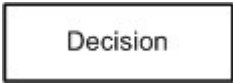

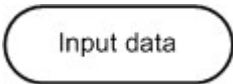
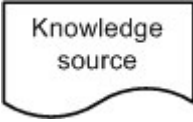

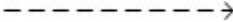

These components are summarized in Table 1 and described in more detail in clause 6.2.

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.	
	Knowledge Source	A knowledge source denotes an authority for a business knowledge model or decision.	
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 a knowledge source, or the dependence of a knowledge source on input data	

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. Implementations **MUST** be able to label each Decision by displaying its Name, and **MAY** be able to label it by displaying other properties such as its Question or Description. If displayed, the label **MUST** be clearly inside the shape of the DRD element.

If the Listed Input Data option is exercised (see 6.2.1.3), all the Decision's requirements for Input Data **MUST** be listed beneath the Decision's label and separated from it by a horizontal line, as shown in Figure 10. The listed Input Data names **MUST** be clearly inside the shape of the DRD element.

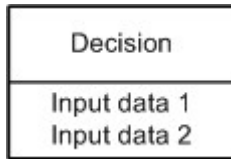


Figure 10: Decision with Listed Input Data option

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. Implementations **MUST** be able to label each Business Knowledge Model by displaying its Name, and **MAY** be able to label it by displaying other properties such as its Description. If displayed, the label **MUST** be clearly inside the shape of the DRD element.

The properties of a Business Knowledge Model are listed and described in 6.3.7.

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. Implementations **MUST** be able to label each Input Data element by displaying its Name, and **MAY** be able to label it by displaying other properties such as its Description. If displayed, the label **MUST** be clearly inside the shape of the DRD element.

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 11 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 **MUST** be listed on all Decisions which require it (unless it is deliberately hidden as discussed in 6.2.4).

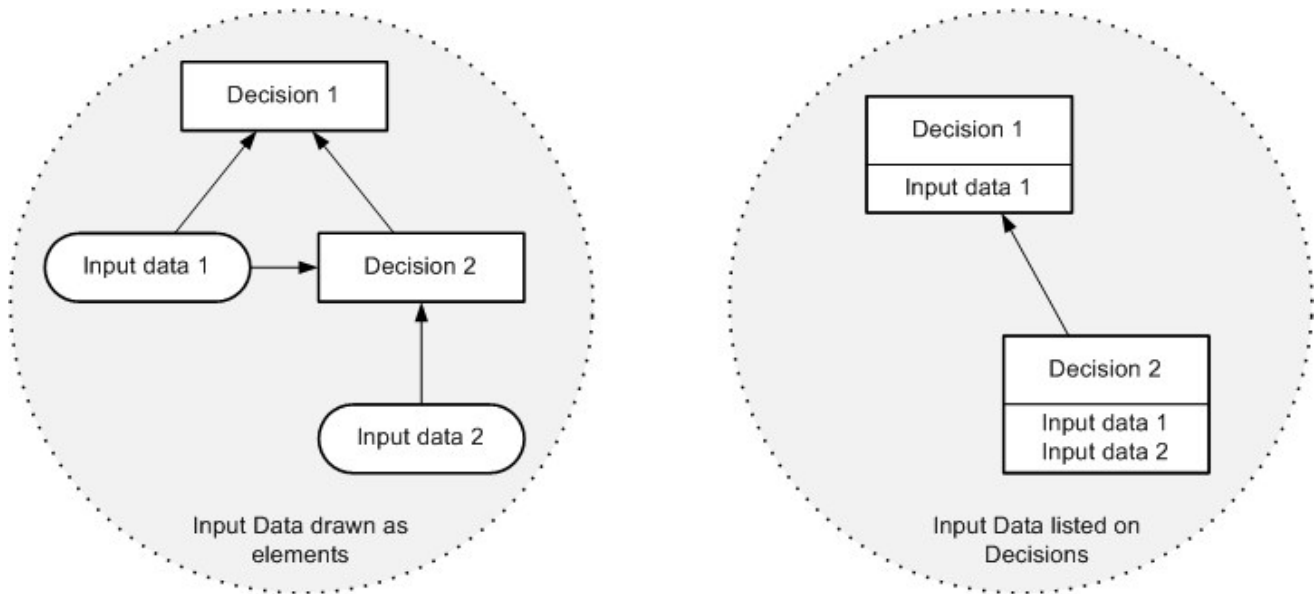


Figure 11: The Listed Input Data option

The properties of an Input Data element are listed and described in 6.3.9.

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. Implementations **MUST** be able to label each Knowledge Source element by displaying its Name, and **MAY** be able to label it by displaying other properties such as its Description. If displayed, the label **MUST** be clearly inside the shape of the DRD element.

The properties of a Knowledge Source element are listed and described in 6.3.10.

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 Business Knowledge Models to Decisions, and from Business Knowledge Models to other Business Knowledge Models. They represent the invocation of business knowledge when making a decision. They may also be interpreted as function calls: a DRD displaying only Decisions, Business Knowledge Models and Knowledge Requirements is equivalent to a function hierarchy showing the function calls involved in evaluating the Decisions. The Knowledge Requirements of a valid DRG form a directed acyclic graph.

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:

- a) 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 would typically be used to record the fact that a set of business rules must be derived from or consistent with a published document, e.g. a piece of legislation or a statement of business policy. An example of this use of Knowledge Sources is shown in Figure 12.

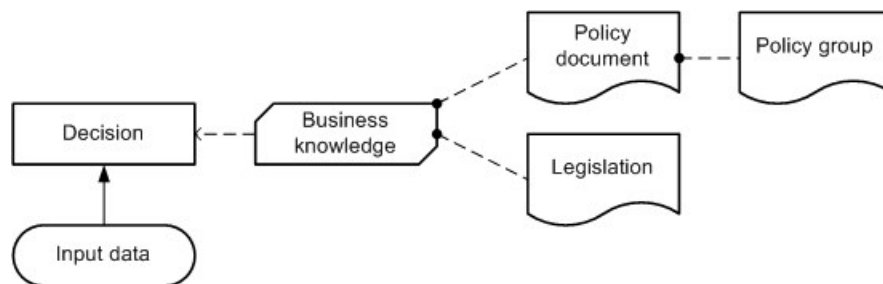


Figure 12: Knowledge Sources representing authorities

- b) 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 13.

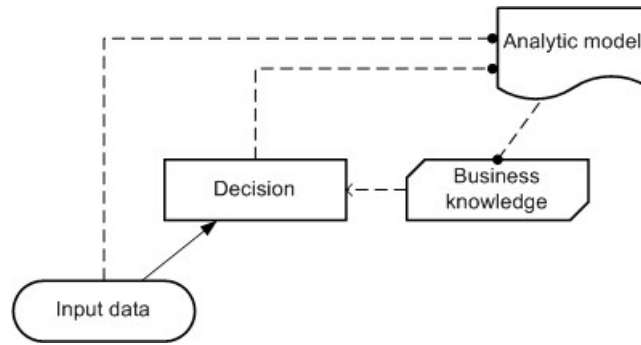


Figure 13: Knowledge source representing predictive analytics

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.

Note that no requirements may be drawn terminating in Input Data.

Table 2: Requirements connection rules

		To			
		Decision	Business Knowledge Model	Knowledge Source	Input Data
From	Decision	Information Requirement		Authority Requirement	
	Business knowledge model	Knowledge Requirement	Knowledge Requirement		
	Knowledge Source	Authority Requirement	Authority Requirement	Authority Requirement	
	Input Data	Information Requirement		Authority Requirement	

6.2.4 Partial views and hidden information

The metamodel (see clause 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 MUST 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. DMN does not specify how such views should be notated, but whenever information is hidden implementations SHOULD provide a clear visual indication that this is the case.

Two examples of DRDs providing partial views of a DRG are shown in Figure 14: DRD 1 shows only the immediate requirements of a single decision; DRD 2 shows only Information Requirements and the elements they connect. In this example, for the purposes of illustration only, the approach taken is to use a fine dashed outline for any element with some hidden requirements.

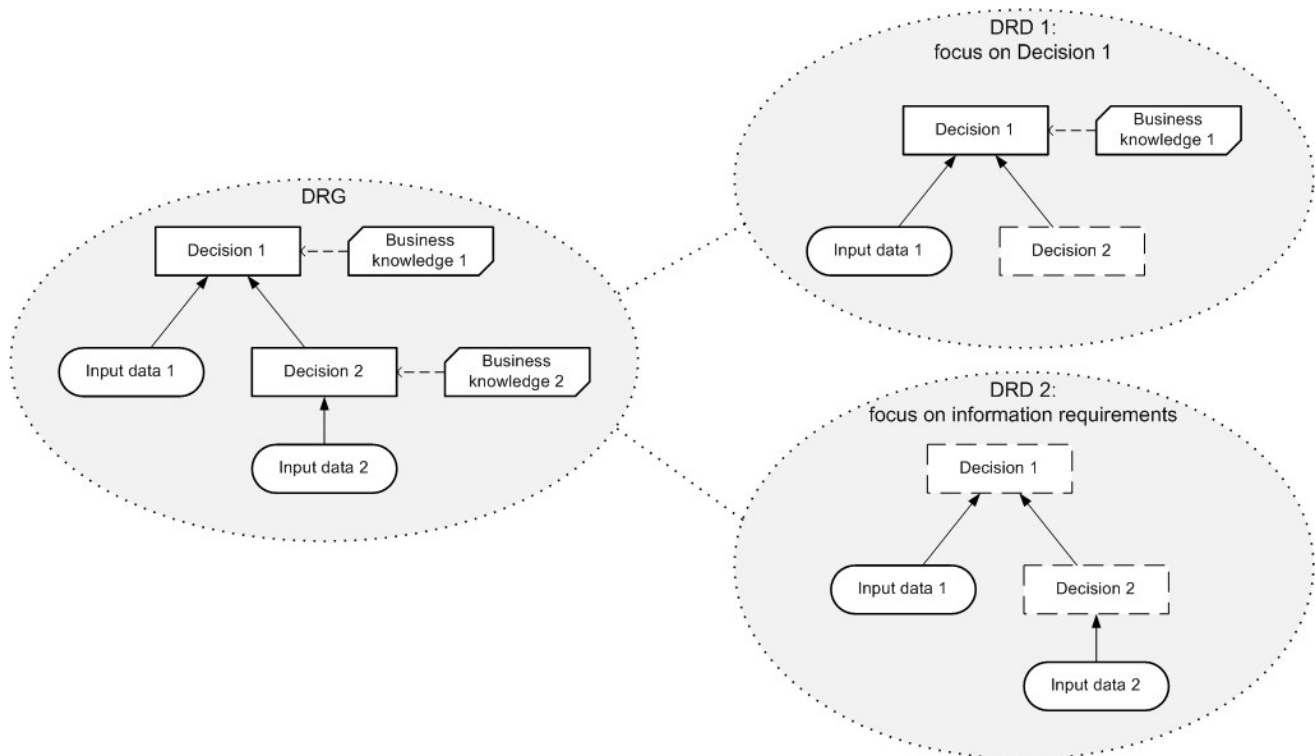


Figure 14: DRDs as partial views of a DRG

In DMN v1.0, DRDs are not represented in the metamodel and may therefore not be interchanged; a set of definitions comprising a DRG may be interchanged, and the recipient may generate any desired DRD from them which is supported by the receiving implementation.

6.3 Metamodel

6.3.1 DMN Element metamodel

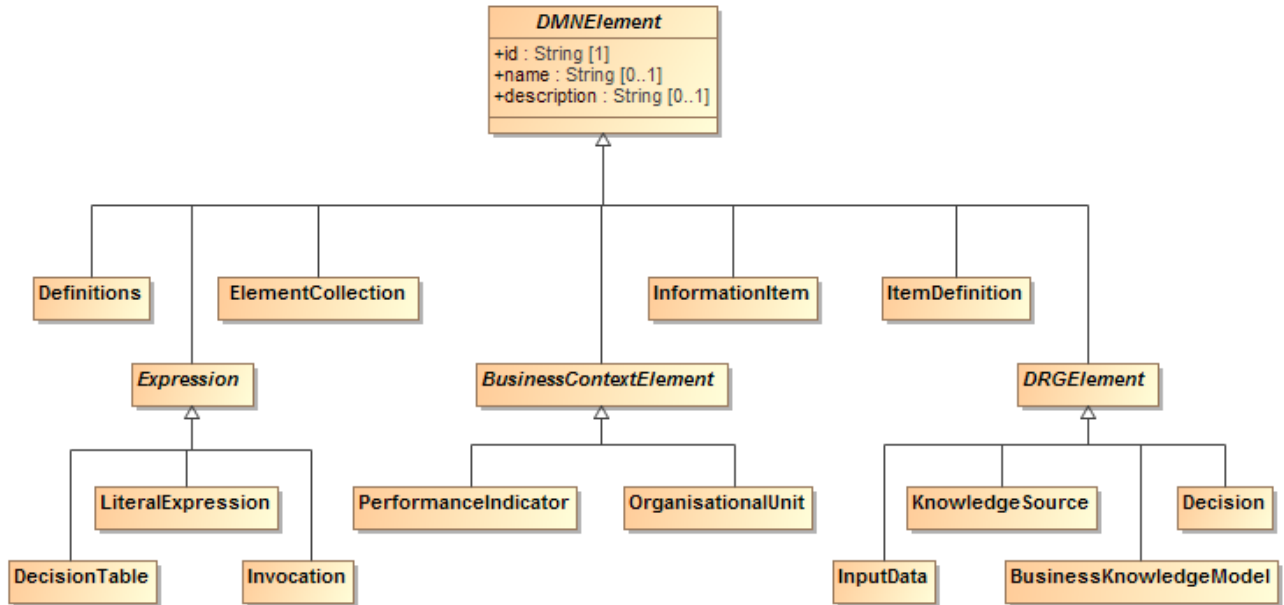


Figure 15: DMNElements Class Diagram

`DMNElement` is the abstract super class for the decision requirement model elements. It provides the mandatory attribute `id` and the optional attributes `name` and `description`, which all are Strings, and which other elements will inherit. The `id` of a `DMNElement` element MUST be unique within the containing element.

`DMNElement` has three abstract specializations: `Expression`, `BusinessContextElement` and `DRGElement` and four concrete specializations: `Definitions`, `ItemDefinition`, `InformationItem` and `ElementCollection`.

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

Table 3: `DMNElement` attributes and model associations

Attribute	Description
name: String [0..1]	The name of this element.
id: String	The string that identifies this <code>DMNElement</code> uniquely within its containing <code>Definitions</code> element.
description: String [0..1]	A description of this element.

6.3.2 Definitions metamodel

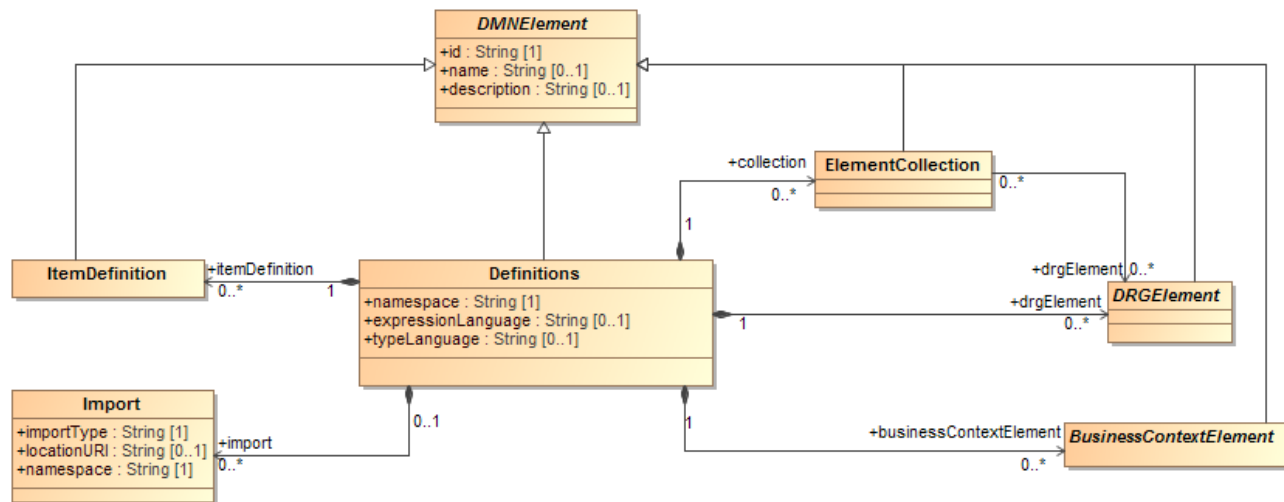


Figure 16: Definitions Class Diagram

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 `DMNElement`, from which an instance of `Definitions` inherits the `id` and optional `name` and `description` 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 `String` 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 **MUST** be specified in a URI format. The Default is FEEL (clause 10).

An instance of `Definitions` may specify a `typeLanguage`, which is a `String` that identifies the default type language used in elements within the scope of this `Definitions`. For example, a `typeLanguage` value of “`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 is FEEL. This value may be overridden on each individual `ItemDefinition`. The `typeLanguage` **MUST** be specified in a URI format.

An instance of `Definitions` is composed of zero or more `drgElements`, which are instances of `DRGElement`, zero or more `collections`, which are instances of `ElementCollection`, zero

or more `itemDefinition`, which are instances of `ItemDefinition` and of zero or more `businessContextElement`, 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 `DMNElement`. Table 4 presents the additional attributes and model associations of the `Definitions` element.

Table 4: Definitions attributes and model associations

Attribute	Description
namespace: String	This attribute identifies the namespace associated with this <code>Definitions</code> and follows the convention established by XML Schema.
expressionLanguage: String [0..1]	This attribute identifies the expression language used in <code>LiteralExpressions</code> within the scope of this <code>Definitions</code> . The Default is FEEL (clause 10). This value MAY be overridden on each individual <code>LiteralExpression</code> . The language MUST be specified in a URI format.
typeLanguage: String [0..1]	This attribute identifies the type language used in <code>LiteralExpressions</code> within the scope of this <code>Definitions</code> . The Default is FEEL (clause 10). This value MAY be overridden on each individual <code>ItemDefinition</code> . The language MUST be specified in a URI format.
itemDefinition: <code>ItemDefinition</code> [*]	This attribute lists the instances of <code>ItemDefinition</code> that are contained in this <code>Definitions</code> .
drgElement: <code>DRGElement</code> [*]	This attribute lists the instances of <code>DRGElement</code> that are contained in this <code>Definitions</code> .
businessContextElement: <code>BusinessContextElement</code> [*]	This attribute lists the instances of <code>BusinessContextElement</code> that are contained in this <code>Definitions</code> .
collection <code>ElementCollection</code> [*]	This attribute lists the instances of <code>ElementCollection</code> that are contained in this <code>Definitions</code> .

import: Import [*]	This attribute is used to import externally defined elements and make them available for use by elements in this Definitions.
---------------------------	---

6.3.3 Import metamodel

The Import class is used when referencing external elements, either DMN DRGElement instances contained in other Definitions elements, or non-DMN elements, such as an XML Schema or a PMML file. Imports must 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. A value of `<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 String that MUST be in URI format.

An instance of Import has a namespace, which is a String that identifies the namespace of the imported element.

Table 5 presents the attributes and model associations of the Import element.

Table 5: Import attributes and model associations

Attribute	Description
importType: String	Specifies the style of import associated with this Import.
locationURI: String [0..1]	Identifies the location of the imported element. MUST be in URI format.
namespace: String	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 DMNElement, from which an instance of ElementCollection inherits the id and optional name and description attributes, which are

Strings. The `id` of a `ElementCollection` element **MUST** 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 `DMNElement`. Table 6 presents the additional attributes and model associations of the `ElementCollection` element.

Table 6: `ElementCollection` attributes and model associations

Attribute	Description
<code>drgElement</code> : <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 super class 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) **MUST** be contained in an instance of `DRGElement`, or in a model element that is contained in an instance of `DRGElement`, recursively.

The concrete specializations of `DRGElement` are `Decision`, `InputData`, `BusinessKnowledgeModel` and `KnowledgeSource`.

`DRGElement` is a specialization of `DMNElement`, from which it inherits the `id` and optional `name` and `description` attributes. The `id` of a `DRGElement` element **MUST** 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 6.3.11, 6.3.12 and 6.3.13). The connection rules are specified in clause 6.2.3).

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

6.3.6 Decision metamodel

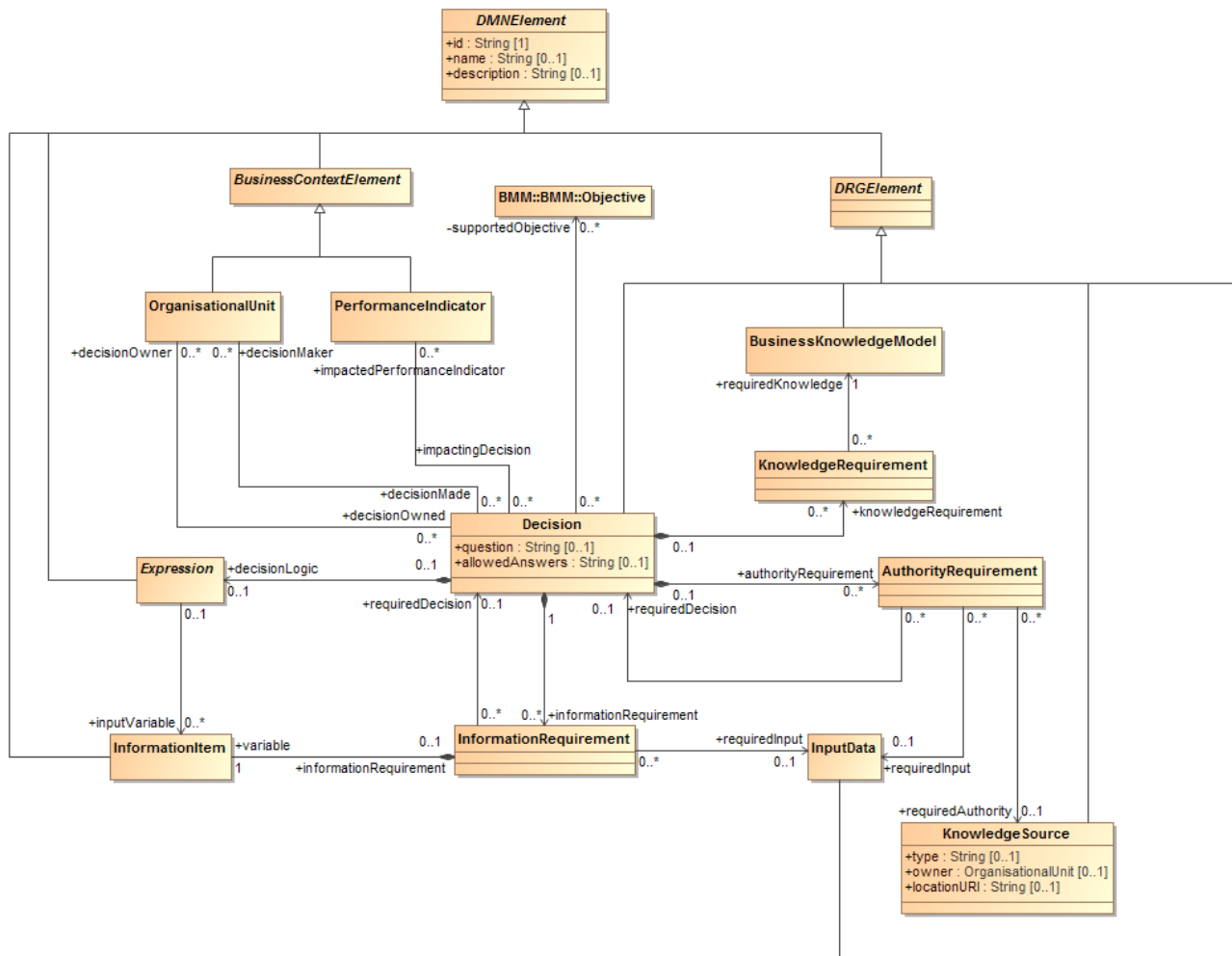


Figure 17: Decision Class Diagram

In DMN 1.0, the class Decision is used to model a decision.

Decision is a concrete specialization of DRGElement and it inherits the mandatory id and optional name and description from DMNElement

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.

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 **MUST** be acyclic, that is, that a `Decision` element **MUST** 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 6.3.7 and Figure 18).

In DMN 1.0, 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 `impactedPerformanceIndicators`, which are instances of `PerformanceIndicator`, any number of `decisionMaker` and any number of `decisionOwner`, which are instances of `OrganisationalUnit`.

`Decision` inherits all the attributes and model associations from `DRGElement`. Table 7 presents the additional attributes and model associations of the `Decision` class.

Table 7: Decision attributes and model associations

Attribute	Description
question: String [0..1]	A natural language question that characterizes the <code>Decision</code> such that the output of the <code>Decision</code> 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.
decisionLogic: Expression [0..1]	The instance of <code>Expression</code> that represents the decision logic for this <code>Decision</code> .

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.

6.3.7 Business Context Element metamodel

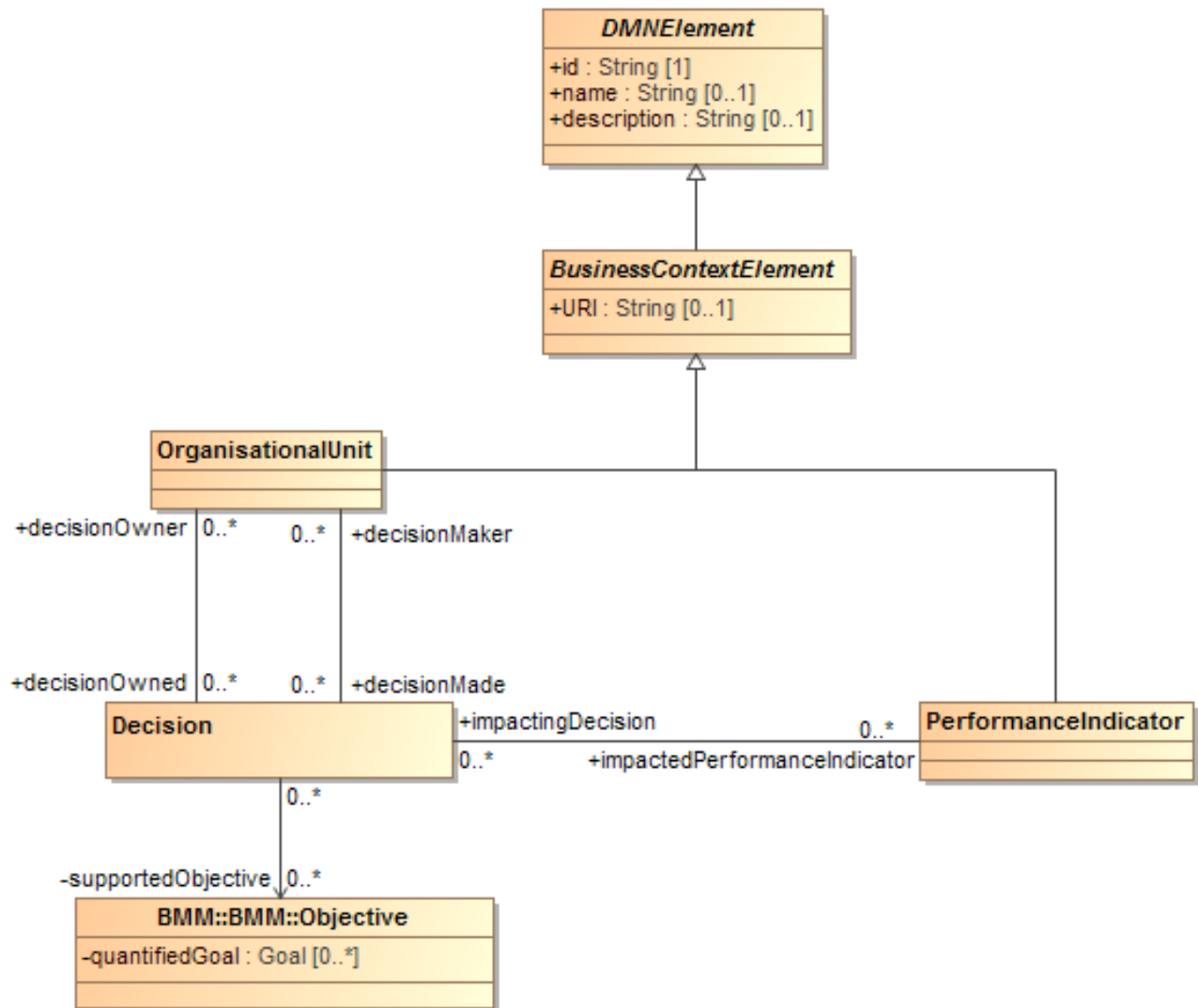


Figure 18: BusinessContextElement class diagram

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.

In `DMN 1.0`, `BusinessContextElement` is a specialization of `DMNElement`, from which it inherits the `id` and optional `name`, a `description` attributes.

In addition, instances of `BusinessContextElements` may have an `URI`, which is a `Strings` that must be in `URI` format, 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 `DMNElement`. Table 8 presents the additional attributes and model associations of the `BusinessContextElement` class.

Table 8: `BusinessContextElement` attributes and model associations

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

`PerformanceIndicator` inherits all the attributes and model associations from `BusinessContextElement`. Table 9 presents the additional attributes and model associations of the `PerformanceIndicator` class.

Table 9: `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 10 presents the additional attributes and model associations of the `OrganisationalUnit` class.

Table 10: `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> .

6.3.8 Business Knowledge Model metamodel

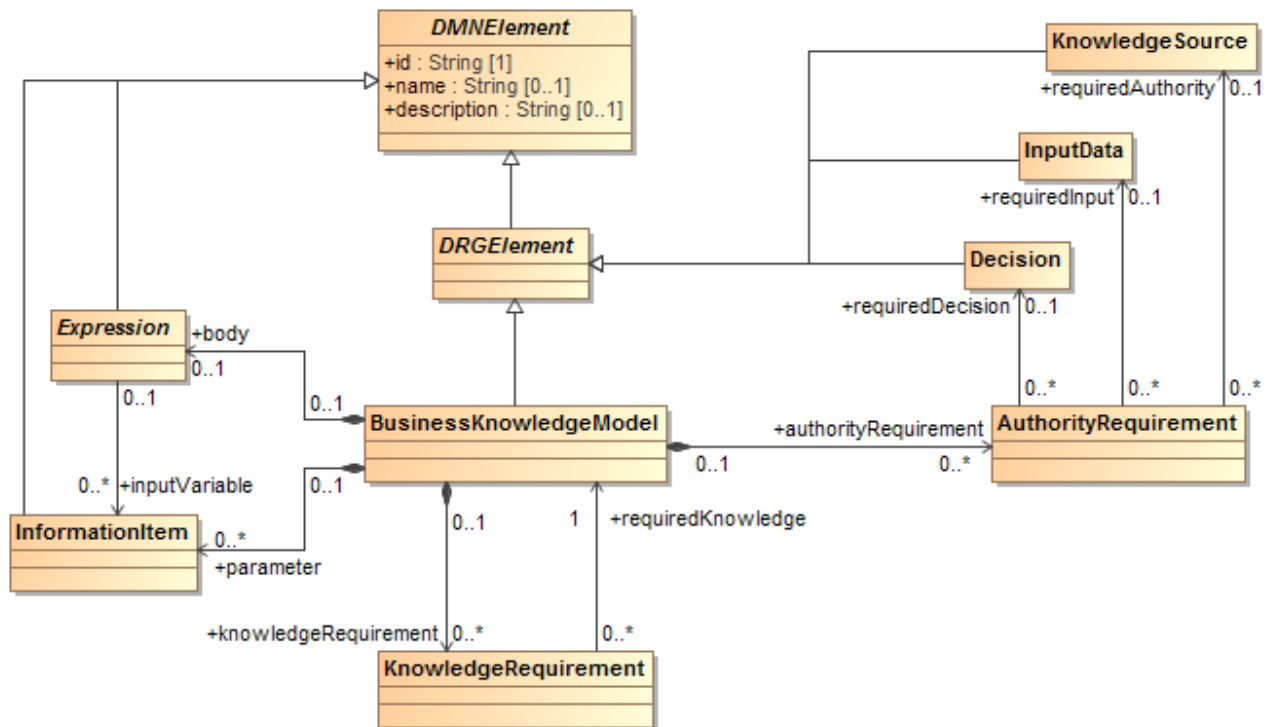


Figure 19: BusinessKnowledgeModel class diagram

The business knowledge models that are associated with a decision are reusable modular expressions of all or part of their decision logic.

In DMN 1.0, the class BusinessKnowledgeModel is used to model a business knowledge model.

BusinessKnowledgeModel is a concrete specialization of DRGElement and it inherits the mandatory id and optional name and description attributes from DMNElement.

In a DRD, an instance of BusinessKnowledgeModel is represented by a **business knowledge model** diagram element.

A BusinessKnowledgeModel element may have zero or more knowledgeRequirement, which are instance of KnowledgeRequirement, and zero or more authorityRequirement, which are instances of AuthorityRequirement.

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 MUST be acyclic, that is, that a BusinessKnowledgeModel element MUST not require itself, directly or indirectly.

At the decision logic level, a BusinessKnowledgeModel element defines a function. It may be composed of an associated body, which is an instance of Expression and of zero or more parameter, which are instances of InformationItem. The body that is associated with a BusinessKnowledgeModel element is the reusable module of decision logic that is represented by this BusinessKnowledgeModel element. The parameters in a BusinessKnowledgeModel element are the inputVariables that are referenced by its body.

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

Table 11: BusinessKnowledgeModel attributes and model associations

Attribute	Description
body: Expression [0..1]	The instance of Expression that describes the logic represented by this BusinessKnowledgeModel, that is, the body of the function that it defines.
parameter: InformationItem [*]	This attribute lists the instances of InformationItem that model the parameters of the function that this BusinessKnowledgeModel defines.
knowledgeRequirement: KnowledgeRequirement [*]	This attribute lists the instances of KnowledgeRequirement that compose this BusinessKnowledgeModel.
authorityRequirement: AuthorityRequirement [*]	This attribute lists the instances of AuthorityRequirement that compose this BusinessKnowledgeModel.

6.3.9 Input Data metamodel

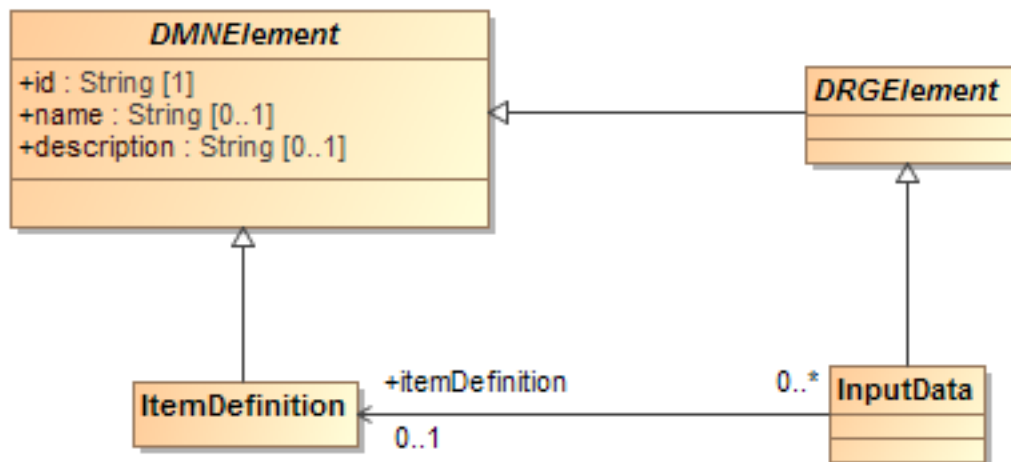


Figure 20: InputData class diagram

DMN 1.0 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 mandatory `id` and optional `name` and `description` from `DMNElement`.

Instances of `InputData` may reference an `itemDefinition`, which is an `ItemDefinition` element that specifies the type of data that is this `InputData` represents.

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 12 presents the additional attributes and model associations of the `InputData` class.

Table 12: InputData attributes and model associations

Attribute	Description
itemDefinition: <code>ItemDefinition</code> [0..1]	The instance of <code>ItemDefinition</code> that describes the data type expected for this <code>InputData</code> .

6.3.10 Knowledge Source metamodel

In DMN 1.0, 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 DMNElement, from which it inherits the mandatory id and optional name and description from DMNElement and the mandatory Id from from DRGElement. In addition, a KnowledgeSource has a locationURI, which is a String that MUST be specified in a URI format. It has a type, which is a String, and an owner, which is <an instance of OrganisationalUnit?>.

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

KnowledgeSource inherits all the attributes and model associations from DRGElement. Table 13 presents the attributes and model associations of the KnowledgeSource class.

Table 13: KnowledgeSource attributes and model associations

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

6.3.11 Information Requirement metamodel

The class InformationRequirement is used to model an **information requirement**, as represented by a plain arrow in a DRD.

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 is composed of a variable, which is an instance of InformationItem, and that 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,
- and the `Decision` element that contains the instance of `InformationRequirement` is not in the requirement subgraph of the referenced `requiredDecision` element, if this `InformationRequirement` element references one.

Table 14 presents the attributes and model associations of the `InformationRequirement` element.

Table 14: `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.
variable: <code>InformationItem</code>	The instance of <code>InformationItem</code> that represents this <code>InformationRequirement</code> in the logic of the requiring <code>Decision</code> .

6.3.12 Knowledge Requirement metamodel

The class `KnowledgeRequirement` is used to model a **knowledge requirement**, as represented by a dashed arrow in a DRD.

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 `BusinessKnowledgeModel`.

Notice that a `KnowledgeRequirement` element must reference the instance of `BusinessKnowledgeModel` 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,
- and, if the `InformationRequirement` element is contained in an instance of `BusinessKnowledgeModel`, that `BusinessKnowledgeModel` element is not in the requirement subgraph of the referenced `requiredKnowledge` element.

Table 15 presents the attributes and model associations of the `KnowledgeRequirement` element.

Table 15: KnowledgeRequirement attributes and model associations

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

6.3.13 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.

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 16 presents the attributes and model associations of the `AuthorityRequirement` element.

Table 16: 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 its containing <code>KnowledgeSource</code> , <code>Decision</code> or <code>BusinessKnowledgeModel</code> element.

requiredDecision: Decision [0..1]	The instance of Decision that this AuthorityRequirement associates with its containing KnowledgeSource element.
requiredInput: InputData [0..1]	The instance of InputData that this AuthorityRequirement associates with its containing KnowledgeSource element.

6.4 Examples

Examples of DRDs are provided in clause 11.2.

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 in one or more value expressions. In DMN 1.0, the elements of decision logic modeled as value expressions are literal expressions, decision tables and invocations:

- 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 theory, a Java computer program and a PMML document. DMN 1.0 specifies an expression language: **FEEL** (see clause 10), and a basic subset of FEEL (see clause 9) 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** may be used as an alternative to FEEL, to model how a decision invokes decision logic that is represented by a Business Knowledge Model.

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 in order 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 represent input values in value expressions: input values are assigned to variables, and value expressions reference 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 includes 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 each of the requiring decision. 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 information requirement at the DRG level is associated with a (variable, expression) pair at the decision logic level. Each input variable that is referenced by a decision's expression must be the variable in one of the decision's information requirements, and each variable in a decision's information requirement must be an input variable of the decision's expression. The expression that is associated with a variable in an information requirement specifies the value to be assigned to the variable when evaluating the decision's expression:

- If a decision requires another decision, the expression in the pair that is associated with the information requirement is the required decision's expression, thus assigning the value of the required decision to an input variable of 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 expression in the pair that is associated with the information requirement is outside of the decision model and is, therefore, not represented as an explicit expression: 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. Notice that, for required input data, FEEL allows test data to be included in the place where the external value expression cannot be explicitly represented.

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 9.3.2.8). 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 variable 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 8.3), 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 must have an explicit value expression that describes how the required business knowledge models are invoked and their results combined or otherwise elaborated.

At the decision logic level, a decision invokes a required business knowledge model by evaluating the business knowledge model's value expression with the parameters bound to its own input value. How this may be achieved depends on how the decision logic is partitioned between the decision and business knowledge models:

- If a decision element requires more than one business knowledge element, its value expression must be a literal expression that specifies how the business knowledge model elements are invoked and how their results are combined into the decision's outcome.
- If a decision does not require any business knowledge models, its value expression must be a literal expression or decision table that specifies the entire decision logic for deriving the output from the inputs.
- Similarly, if a decision element requires only one business knowledge model element, but the logic of the decision elaborates on the logic of its required business knowledge model, the decision element must have a literal expression that specifies how the business knowledge model's value expression is invoked, and how its result is elaborated to provide the decision's outcome.
- In all other cases (i.e. when a decision requires exactly one business knowledge model and does not elaborate the logic), the value expression of a decision element may be a value expression of type invocation. In a value expression of type invocation, only the bindings of the business knowledge model parameters to the decisions input data need be specified: the outcome of the decision is the result returned by the business knowledge model's value expression for the values passed to its parameters.

The binding of a business knowledge model's parameter is a value expression that specifies how the value passed to that parameter is derived from the values of the input variables of the invoking decision.

Using a value expression of type invocation is never required, even when possible: FEEL specifies its own invocation mechanism for more complex usages, and a FEEL literal expression can always be used instead of a value expression of type invocation.

7.2 Notation

7.2.1 Boxed 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 notion 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 **MUST** 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 21:

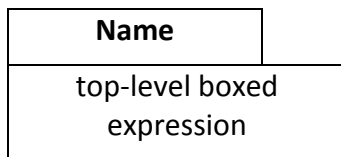


Figure 21: Boxed expression

Alternatively, the name box and expression box can be separated by white space and connected on the left side with a dotted line, as shown in Figure 22:

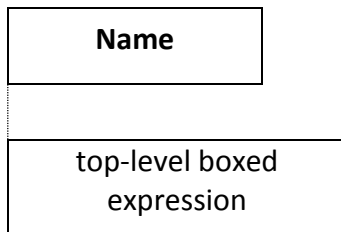


Figure 22: Boxed expression with separated name and expression boxes

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 23 is equivalent to Figure 24:

Credit contingency factor table		
UC	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 23: Decision table with italicized literals

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

Figure 24: Decision table with string literals

To avoid having to discerning whether *HIGH, DECLINE* is "HIGH", "DECLINE" or "HIGH, DECLINE", typographical string literals should be free of "," characters. FEEL typographical string literals must conform to grammar rule 27 (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 must obey the syntax specified in clauses 10.3.2.2.4, 10.3.2.2.5 and 10.3.2.2.6.

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 25).

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 25: 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, in DMN, is that they may contain an expression that describes the logic by which a modeled decision shall be made, or pieces of that logic.

In DMN 1.0, the class `Expression` is the abstract super class 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).

DMN 1.0 defines three concrete kinds of `Expression`; `LiteralExpression`, `DecisionTable` (see clause 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. In DMN 1.0, 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. In DMN 1.0, the class `ItemDefinition` is used to model data structures and ranges.

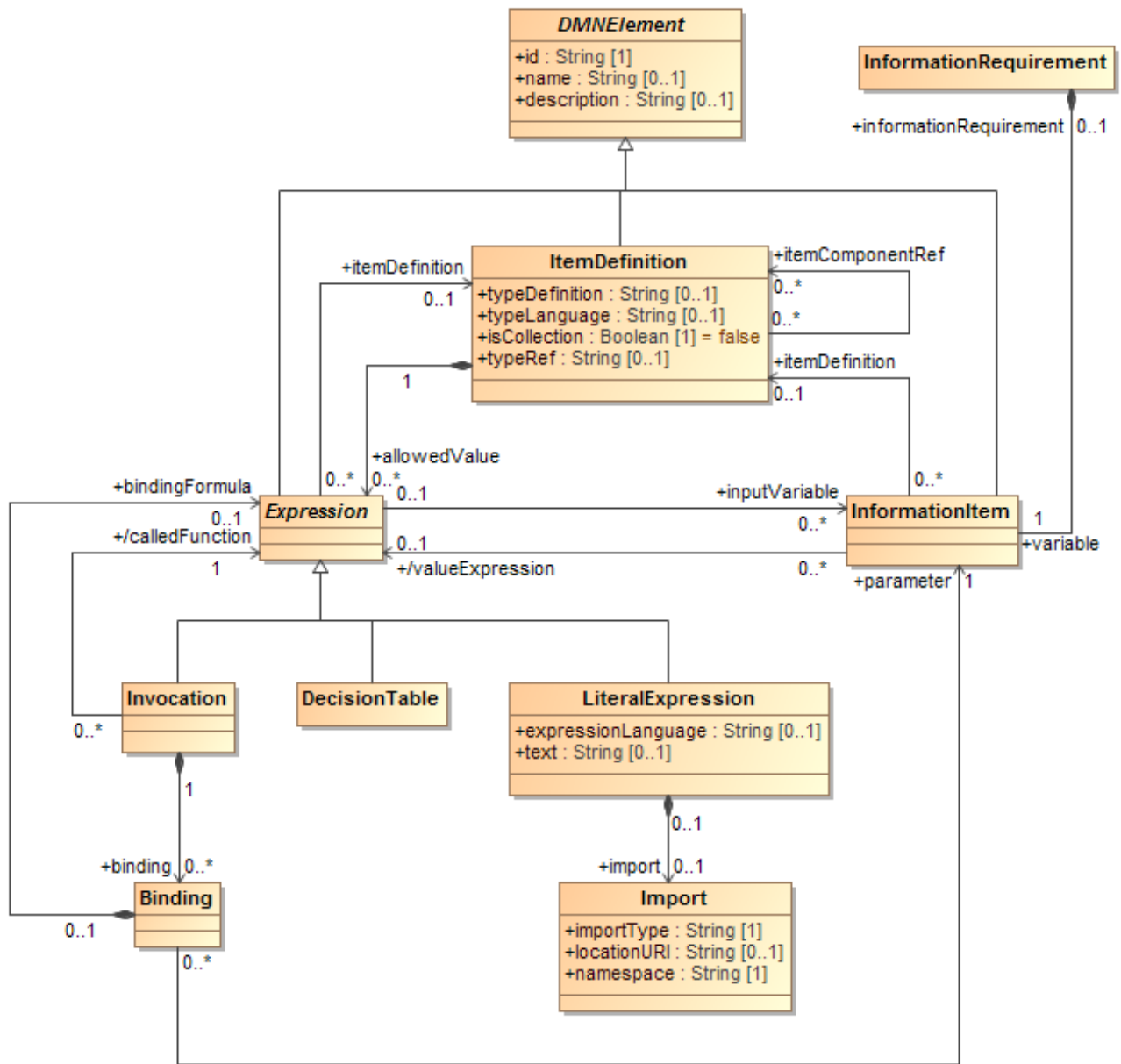


Figure 26: Expression class diagram

7.3.1 Expression metamodel

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

In DMN 1.0, the class `Expression` is the abstract super class 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.

Expression is an abstract specialization of DMNElement, from which it inherits the id, and optional name and description 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. The id of an Expression element MUST be unique within the containing instance of Decision, BusinessKnowledgeModel or ItemDefinition.

An instance of Expression references zero or more inputVariables, which are instances of InformationItem. The inputVariables are lexically scoped, in instances of Expression, and the scope is defined by the instance of Decision that contains them as part of an informationRequirement element, or by the instance of BusinessKnowledgeModel that contains them as parameters. An Expression element that is contained in an instance of ItemDefinition MUST NOT reference any inputVariable.

An instance of Expression references an optional itemDefinition, which is an instance of ItemDefinition that specifies its range of possible values.

An instance of Expression can be interpreted to derive a single value from the values assigned to its inputVariables. How the value of an Expression element is derived from the values assigned to its inputVariables depends on the concrete kind of the Expression.

Expression inherits from the attributes and model associations of DMNElement. Table 17 presents the additional attributes and model associations of the Expression element.

Table 17: Expression attributes and model associations

Attribute	Description
inputVariable: InformationItem [*]	This attributes lists the instances of InformationItem that are free in this Expression.
itemDefinition: ItemDefinition [0..1]	The instance of ItemDefinition to which the value of this Expression must conform.
inputClause: Clause [0..1]	The containing instance of Clause, if this Expression is an inputEntry element in an instance of DecisionTable.
outputClause: Clause [0..1]	The containing instance of Clause, if this Expression is an outputEntry element in an instance of DecisionTable.

7.3.2 ItemDefinition metamodel

In DMN, the inputs and output of decisions are data items whose value, at the decision logic level, is assigned to variables or represented by value 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.

In DMN 1.0, 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 `DMNElement`, an instance of `ItemDefinition` has an `id` and an optional `name` and `description`. The `id` of an `ItemDefinition` element **MUST** be unique within the containing instance of `Definitions`.

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 using 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 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 `typeDefinition`, which is a `String` that defines the data structure using the `typeLanguage`, or a `typeRef`, which is a `String` that references a builtin data type in the associated `typeLanguage` or a type or data structure defined at the top level in an external document using the `typeLanguage`: in the latter case, the external document **MUST** be imported in the `Definitions` element that contains the instance of `ItemDefinition`, using an `Import` element. 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.

By default, the name of an `ItemDefinition` is the name of the type that is defined in its `typeDefinition` or referenced in its `typeRef`. An `ItemDefinition` element **MUST NOT** have both a `typeDefinition` and a `typeRef`.

If the type language is FEEL the builtin types are the FEEL built-in data types: *number*, *string*, *boolean*, *duration*, *time* and *date and time*.

An `ItemDefinition` element may restrict the values that are allowed from the `typeDefinition` or `typeRef`, using the `allowedValue` attribute: each `allowedValue` is an instance of `Expression` that specifies a single allowed value or a range of allowed values from the `typeDefinition` or `typeRef`. The `itemDefinition` of the `allowedValues` MUST be the containing `ItemDefinition` element itself and MAY be omitted. If an `ItemDefinition` element contains one or more `allowedValues`, the list of the `allowedValues` specifies the complete range of values that this `ItemDefinition` represents. If an `ItemDefinition` element does not contain an `allowedValue`, its range of allowed values is the full range of the referenced `typeRef` or defined `typeDefinition`.

In cases where the values that an `ItemDefinition` element represents are collections of values in the allowed range, the multiplicity can be projected into the attribute `isCollection`. The default value for this attribute is *false*.

An alternative way to define an instance of `ItemDefinition` is as a composition of other `ItemDefinition` elements. An instance of `ItemDefinition` may reference zero or more `itemComponentRef`, which are `ItemDefinition` elements: each value in the range of an `ItemDefinition` element that references at least one `itemComponentRef` is made of one value in the range each of the referenced `itemComponentRef` elements.

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

- inline definition of a data type or structure using a `typeDefinition`, possibly restricted with `allowedValues`;
- reference to a built-in or imported `typeRef`, possibly restricted with `allowedValues`;
- composition of other `ItemDefinition` elements, referencing `itemComponentRef`.

That is, an `ItemDefinition` element that references an `itemComponentRef` element MUST NOT have a `typeDefinition`, a `typeRef` or `allowedValues`. Reciprocally, an `ItemDefinition` element that has a `typeDefinition` or a `typeRef` attribute MUST NOT reference any `itemComponentRef`. As already mentioned above, an `ItemDefinition` element MUST NOT have both a `typeDefinition` and a `typeRef`.

The `ItemDefinition` element specializes `DMNElement` and it inherits its attributes and model associations. Table 18 presents the additional attributes and model associations of the `ItemDefinition` element.

Table 18: `ItemDefinition` attributes and model associations

Attribute	Description
typeDefinition: String [0..1]	This attribute is used to define in line the base data structure for this <code>ItemDefinition</code>

typeRef: String [0..1]	This attribute is used to identifies 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 MUST be specified in a URI format.
allowedValue: Expression [*]	This attribute lists the <code>Expression</code> elements that define the values or range of values in the base type that are allowed in this <code>ItemDefinition</code>
itemComponentRef: <code>ItemDefinition</code> [*]	This attribute lists the <code>ItemDefinition</code> elements that compose this <code>ItemDefinition</code>
IsCollection: Boolean	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> .

7.3.3 InformationItem metamodel

In DMN 1.0, the class `InformationItem` is used to model variables at the decision logic level in decision models.

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

In DMN, variables represent the values that are input to a decision, in the description of the decision's logic, or the values that 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 case, a variable is the realization, at the decision logic level, of one of the information requirements (at the decision requirements level) of a decision; in the second 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.

As a consequence, an `InformationItem` element **MUST** be either a `variable` in an instance of `InformationRequirement` or a `parameter` in an instance of `BusinessKnowledgeModel`; it **MUST NOT** be both. The scope of an `InformationItem` element is the `Decision` that contains the containing `InformationRequirement` element, or the containing `BusinessKnowledgeModel` element.

A variable in an instance of `InformationRequirement` MUST be an `inputVariable` in the `decisionLogic` in the `Decision` element that contains the `InformationRequirement` element. A parameter in an instance of `BusinessKnowledgeModel` MUST be an `inputVariable` in the `valueExpression` in that `BusinessKnowledgeModel` element.

As a concrete specialization of `Expression`, an `InformationItem` element can be interpreted and assigned a value. Specifically:

- An `InformationItem` element is assigned the value of the `requiredDecision` that is referenced by its containing instance of `InformationRequirement`, if it references one.
- An `InformationItem` element that is a parameter in a `BusinessKnowledgeModel` element can only be assigned a value using a `Binding` element as part of an instance of `Invocation`.
- Otherwise, an `InformationItem` element is assigned a value by the external data source that is attached at runtime to the `requiredInput` element that its containing instance of `InformationRequirement` references. How a data source is attached to an instance of `InputData` at run time, and how it assigns a value to an `InformationItem` element is out of the scope of DMN 1.0.

In any case, the `valueDefinition` element that is associated with an instance of `InformationItem` must be compatible with the `valueDefinition` that is associated with the DMN model element from which it takes its value.

`InformationItem` inherits of all the attributes and model associations of `DMNElement`. Table 19 presents the additional attributes and model associations of the `InformationItem` element.

Table 19: `InformationItem` attributes and model associations

Attribute	Description
/valueExpression: Expression [0..1]	The Expression whose value is assigned to this <code>InformationItem</code> . This is a derived attribute
informationRequirement: InformationRequirement [0..1]	The instance of <code>InformationRequirement</code> in which this <code>InformationItem</code> is a part, if any.
itemDefinition: ItemDefinition [0..1]	The instance of <code>ItemDefinition</code> to which the value of this <code>InformationItem</code> must conform.

7.3.4 Literal expression metamodel

In DMN 1.0, 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`, `inputVariable` and `valueDefinition` 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` MUST 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 1.0 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 an `import`, which is an instance of `Import` that identifies where the `text` of the `LiteralExpression` is located. An instance of `LiteralExpression` MUST NOT have both a `text` and an `import`. The `importType` of the `import` MUST be the same as the `expressionLanguage` of the `LiteralExpression` element.

`LiteralExpression` inherits of all the attributes and model associations of `Expression`. Table 20 presents the additional attributes and model associations of the `LiteralExpression` element.

Table 20: `LiteralExpression` attributes and model associations

Attribute	Description
text: <code>String</code> [0..1]	The <code>text</code> of this <code>LiteralExpression</code> . It MUST be a valid expression in the <code>expressionLanguage</code> .
expressionLanguage: <code>String</code> [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 MUST be specified in a URI format.
import: <code>Import</code> [0..1]	The instance of <code>Import</code> that specifies where the <code>text</code> of this <code>LiteralExpression</code> is located.

7.3.5 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.

In DMN, the class `Invocation` is used to model invocations as a kind of `Expression`: `Invocation` is a concrete specialization of `Expression`, from which it inherits the `id`, `inputVariable` and `valueDefinition` attributes.

An instance of `Invocation` is made of zero or more `binding`, which are instances of `Binding`, and model how the parameters of the invoked expression are bound to the `inputVariables` of the invoking instance of `Expression`.

An instance of `Invocation` references a `calledFunction`, which is the instance of `Expression` to be invoked.

The value of an instance of `Invocation` is the value of the associated `calledFunction`, with its `inputVariables` 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 body of an instance of `BusinessKnowledgeModel` as the logic for any instance of `Decision` that requires that `BusinessKnowledgeModel`, where each requiring `Decision` element specifies its own bindings for the `BusinessKnowledgeModel` element's parameters.

The `calledFunction` that is associated with the `Invocation` element MUST BE the body of the `BusinessKnowledgeModel` element that is required by the `Decision` element that contains the `Invocation`; that is, the `calledFunction` is a derived attribute. The `Invocation` element MUST have exactly one binding for each parameter in the `BusinessKnowledgeModel` element.

`Invocation` inherits of all the attributes and model associations of `Expression`. Table 21 presents the additional attributes and model associations of the `Invocation` element.

Table 21: Invocation attributes and model associations

Attribute	Description
/calledFunction: Expression	The Expression that is invoked by this Invocation. It MUST BE the body of the BusinessKnowledgeModel element that is required by the instance of Decision that contains this Invocation. This is a derived attribute
binding: Binding [*]	This attribute lists the instances of Binding used to bind the inputVariables of the calledFunction in this Invocation.

7.3.6 Binding metamodel

In DMN 1.0, the class Binding is used to model, in an Invocation element, the binding of the free variables – or *parameters* – in the invoked expression to the input variables of the invoking expression.

An instance of Binding is made of one bindingFormula, which is an instance of Expression, and of one reference to a parameter, which is an instance of InformationItem.

The inputVariables of the bindingFormula in a Binding element **MUST** be a subset of the inputVariables in the owning instance of Invocation.

The parameter referenced by a Binding element **MUST** be one of the parameters of the BusinessKnowledgeModel element that contains the calledFunction element that is invoked by the containing instance of Invocation.

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 22 presents the attributes and model associations of the Binding element.

Table 22: Binding attributes and model associations

Attribute	Description
parameter: InformationItem	The InformationItem on which the calledFunction of the owning instance of Invocation depends that is bound by this Binding.

bindingFormula: Expression [0..1]	The instance of Expression to which the parameter in this Binding is bound when the owning instance of Invocation is evaluated.
--	---

8 Decision Table

8.1 Introduction

One of the ways to express the decision logic corresponding to the DRD decision artifact is 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:

- a name
- a set of inputs, each input optionally associated with a type and list of input values
- a set of outputs, each output optionally associated with a type and list of output values
- a list of rules in rows or columns of the table (depending on orientation), where each rule is composed of the specific input entries and output entries of the table row (or column)

The decision table shows the rules in a shorthand notation by arranging the entries in table cells:

input expression 1	input expression 2	result
x	y	z

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

IF input expression 1 matches x AND input expression 2 matches y THEN a result (a "hit") is z.

This shorthand notation shows all inputs in the same order in every rule and therefore has a number of readability and verification advantages.

The list of rules expresses the logic of the decision. If rules are allowed to contain overlapping input combinations, the table hit policy indicates how overlapping rules have to be interpreted, in order to avoid inconsistency.

The list of rules may contain all possible combinations of input values, in which case the table is called "complete".

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) list of values in standard locations on a grid of cells
- 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
- the aggregation, indicating how multiple hits are aggregated
- placement of table name, hit policy (H), completeness indicator (C), aggregation (A) and rule numbers

8.2.1 Line style and color

Line style is normative. There is a double line between the inputs section and the outputs section, and there is a double line between inputs/outputs and the rule entry cells. 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 inputs, outputs and rule values.

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).

The table must be arranged in one of the following ways (see Figure 27, Figure 28, Figure 29). Cells labeled in *italics* are optional.

table name			
HC	input expression 1	input expression 2	Output name
	<i>value 1a, value 1b</i>	<i>value 2a, value 2b</i>	<i>value 1a, value 1b</i>
1	input entry 1a	input entry 2a	output entry 1a
2		input entry 2b	output entry 1b
3	input entry 1b	-	output entry 1a

Figure 27: Rules as rows

table name				
input expression 1	<i>value 1a, value 1b</i>	input entry 1a		input entry 1b
input expression 2	<i>value 2a, value 2b</i>	input entry 2a	input entry 2b	-
Output name	<i>value 1a, value 1b</i>	output entry 1a	output entry 1b	output entry 1a
HC		1	2	3

Figure 28: Rules as columns

table name			
Output name		input expression 1	
		input entry 1a	input entry 1b
input expression 2	input entry 2a	output entry 1a	output entry 1a
	input entry 2b	output entry 1b	output entry 1a

Figure 29: Rules as crosstab (optional input and output values not shown)

Crosstab tables with more than two inputs are possible (but not shown here).

8.2.3 Input expressions

Input expressions are usually simple, for example, a name (e.g. Customer Status) or a test (e.g. Age<25). The expression can be any text (e.g., natural language text) but SHOULD NOT conflict with FEEL syntax.

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 may be considered complete if its rules cover all combinations of expected input values.

Regardless of how the expected input values are modeled, input values should be exclusive and complete. Exclusive means that input values do not overlap. Complete means that all relevant input values from the domain are covered.

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 restricting the corresponding inputs to values that test *true*. The list can be any text (e.g., natural language text) but SHOULD NOT conflict with FEEL syntax.

8.2.5 Table name and output name

The table name or the output name MUST be specified.

If the table name is specified, the output name MUST be the same as the table name, or left unspecified (empty box for single-output tables, or omitted box for multiple-output tables). If the table name is not specified, the output name MUST be specified.

8.2.6 Output values

The output entries of a decision table are often drawn from a list of output values. The ordering of the list of output values can be used to specify the priority when multiple rules match but only one hit should be returned. The ordering is also used when the hit policy is output order.

The list of output values is optional. If provided, it is a list restricting the value of the output entries to the given list of values. The list can be any text (e.g., natural language text) but SHOULD NOT conflict with FEEL syntax.

8.2.7 Multiple outputs

The decision table can show a compound output (see Figure 30, Figure 31, Figure 32).

table name				
HC	input expression 1	input expression 2	output name	
			output 1	output 2
	<i>input value 1a, input value 1b</i>	<i>input value 2a, input value 2b</i>	<i>output value 1a, output value 1b</i>	<i>output value 2a, output value 2b</i>
1	input entry 1a	input entry 2a	output entry 1a	output entry 2a
2		input entry 2b	output entry 1b	output entry 2a
3	input entry 1b	-	output entry 1a	output entry 2b

Figure 30: Horizontal table with compound output

table name					
input expression 1		<i>input value 1a, input value 1b</i>	input entry 1a		input entry 1b
input expression 2		<i>input value 2a, input value 2b</i>	input entry 2a	input entry 2b	-
output name	output 1	<i>output value 1a, output value 1b</i>	output entry 1a	output entry 1b	output entry 1a
	output 2	<i>output value 2a, output value 2b</i>	output entry 2a	output entry 2a	output entry 2b
HC			1	2	3

Figure 31: Vertical table with compound output

table name			
output name		input expression 1	
output 1, output 2		input entry 1a	input entry 1b
input expression 2	input entry 2a	output entry 1a, output entry 2a	output entry 1a, output entry 2a
	input entry 2b	output entry 1b, output entry 2b	output entry 1a, output entry 2a

Figure 32: Crosstab with compound output

8.2.8 Input entries

Rule input entries are expressions. The expression can be any text (*e.g.*, natural language text) but SHOULD NOT conflict with FEEL syntax. A dash symbol ('-') can be used to mean any input value, *i.e.*, the input is irrelevant for the containing rule.

The values 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 a '-' 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 expressions in a decision table does not produce side-effects. The order of input entries is not related to any execution order in implementation.

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 33 and Figure 34. Rule output cells cannot be merged (except in crosstabs).

table name			
HC	input expression 1	input expression 2	Output name
	<i>input value 1a,</i> <i>input value 1b</i>	<i>input value 2a,</i> <i>input value 2b</i>	<i>output value 1a,</i> <i>output value 1b</i>
1	input entry 1a	input entry 2a	output entry 1a
2		input entry 2b	output entry 1b
3	input entry 1b	-	output entry 1a

Figure 33: Merged rule input cells allowed

table name			
HC	input expression 1	input expression 2	Output name
	<i>input value 1a,</i> <i>input value 1b</i>	<i>input value 2a,</i> <i>input value 2b</i>	<i>output value 1a,</i> <i>output value 1b</i>
1	input entry 1a	input entry 2a	output entry 1a
2		input entry 2b	output entry 1b
3	input entry 1b	input entry 2b	output entry 1b
4		input entry 2a	output entry 1a

Figure 34: Merged rule input cells not allowed

8.2.10 Output entry

A rule output entry is an expression. The expression can be any text (*e.g.*, natural language text) but SHOULD NOT conflict with FEEL syntax.

In vertical (rules as columns) tables with a single output name which is identical to the table name, a shorthand notation may be used to indicate: output applies (X) or does not apply (-), as is common practice in decision tables. The table in Figure 35 is shorthand notation for the table in Figure 36.

table name	input entry 1a		input entry 1b	input entry 1c
input expression 1	input entry 1a		input entry 1b	input entry 1c
input expression 2	input entry 2a	input entry 2b	-	-
output entry a	X	-	-	-
output entry b	-	X	-	X
output entry c	-	-	X	-
HC	1	2	3	4

Figure 35: Shorthand notation for vertical tables (rules as columns)

table name	value 1a, value 1b, value 1c	input entry 1a		input entry 1b	input entry 1c
input expression 1	value 1a, value 1b, value 1c	input entry 1a		input entry 1b	input entry 1c
input expression 2	value 2a, value 2b	input entry 2a	input entry 2b	-	-
Output name	value 1a, value 1b, value 1c	output entry 1a	output entry 1b	output entry 1c	output entry 1b
HC		1	2	3	4

Figure 36: Full notation for vertical tables (rules as columns)

8.2.11 Hit policy

A decision table may have several rules, and in general more than one rule may be matched for a given set of inputs. The hit policy specifies what the result of the decision table is in such cases, and also contains additional information that can be used to check correctness at design-time. For clarity, the hit policy is summarized using a single character in a particular decision table cell. Tools may support only a subset of hit policies, but the table type must be clear and therefore the hit policy indication is mandatory.

The hit policy MUST default to Unique. Decision tables with the Unique hit policy do not contain rules with overlapping input entries.

If rules are allowed to contain overlapping input entries, the hit policy indicates how these overlapping rules have to be interpreted. A single hit table returns 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).

A single hit table returns the output of one rule only. It may or may not contain overlapping rules. In case of overlapping rules, the hit policy has to indicate which of the matching rules to select.

Single hit policies for single output decision tables are:

1. **Unique:** no overlap is possible and all rules are exclusive. Only a single rule can be matched. This is the default.
2. **Any:** there may be overlap, but all of the matching rules show the same output, so any match can be used.
3. **Priority:** multiple rules can match, with different outputs. This policy returns the matching rule with the highest output priority. Output priorities are specified in an ordered list of values, for example, the list of expected output values.
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 a common usage, because it resolves inconsistencies by forcing the first hit. 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 function of the outputs.

Multiple hit policies for single output decision tables can be:

5. **No order:** returns all hits in a unique list in arbitrary order.
6. **Output order:** returns all hits in decreasing priority order. Output priorities are specified in an ordered list of values.
7. **Rule order:** returns all hits in rule order. Note: the meaning may depend on the order of the rules.

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

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

To reduce complexity, decision tables with compound outputs support only the following hit policies: Unique, Any, First, No order, and Rule order.

Note 1

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

Note 2

The order 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 Completeness indicator

Table completeness is an optional attribute. By default, tables are complete, producing a result for every possible case. If not the indicator should read I(ncomplete). Incomplete tables may specify a default output.

8.2.13 Aggregation

Multiple hits must be aggregated into a single result. DMN 1.0 specifies six aggregation indicators, namely: *collect*, *sum*, *min*, *max*, *count*, *average*. Optionally, the aggregation indicator may be included in the table. The default is *collect*.

Aggregation indicators have no incidence on decision tables with single hit policies.

In decision tables with multiple hit policies, the semantics of the aggregation indicators are:

1. *collect*. The result of the decision table is the list of all the outputs, ordered or unordered per the hit policy.
2. *sum*. The result of the decision table is the sum of all the outputs.
3. *min*. The results of the decision table is the smallest value of all the outputs.
4. *max*. The results of the decision table is the largest value of all the outputs.
5. *count*. The results of the decision table is the number of outputs.
6. *average*. The results of the decision table is the average value of all the outputs, defined as the sum divided by the count, where the semantics of *sum* and *count* are as specified above.

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

8.3 Metamodel

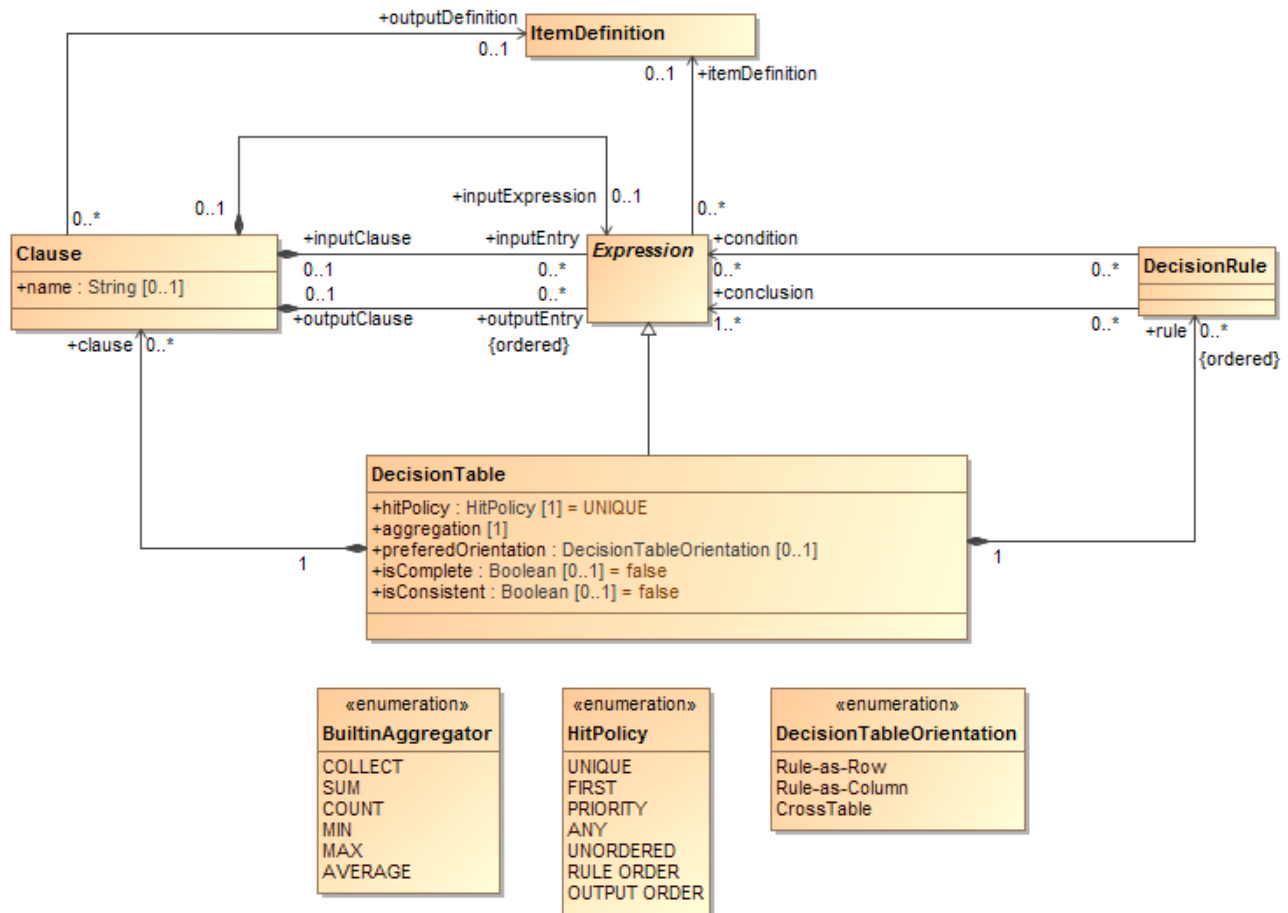


Figure 37: DecisionTable class diagram

8.3.1 Decision Table metamodel

In DMN 1.0, the class `DecisionTable` is used to model a decision table.

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

An instance of `DecisionTable` contains a set of rules, which are instances of `DecisionRule`, and a set of clauses, which are instances of `Clause`.

It has a `preferredOrientation`, which must 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 must be one of the enumerated `HitPolicy`: `UNIQUE`, `FIRST`, `PRIORITY`, `ANY`, `UNORDERED`, `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` **MUST** 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 `outputEntry` of one of the clauses in the `DecisionTable` **MUST** be ordered, and these `outputEntries` **MUST** be associated as conclusions to the rules in the `DecisionTable`. In the diagrammatic representation of the `DecisionTable`, the ordering is represented as specified in clause 8.2.11.

An instance of `DecisionTable` has an associated aggregation, which is one of the enumerated `BuiltinAggregator` (see clause 8.2.13). The default value for aggregation is: `COLLECT`.

As a kind of `Expression`, an instance of `DecisionTable` has a value, which depends on the conclusions 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 following specification:

- if the associated `hitPolicy` is `UNIQUE`, the value of an instance of `DecisionTable` is the value of the conclusion of the only applicable rule (see clause 8.3.3, `Decision Rule`, for the definition of rule applicability);
- if the associated `hitPolicy` is `FIRST`, the value of an instance of `DecisionTable` is the value of the conclusion of the first applicable rule, according to the rule ordering;
- if the associated `hitPolicy` is `PRIORITY`, the value of an instance of `DecisionTable` is the value of the conclusion of the of the first applicable rule, according to the ordering of the `outputEntry` in the clause of its conclusion (see clause 8.3.2, `Decision Table Clause`);
- if the associated `hitPolicy` is `ANY`, the value of an instance of `DecisionTable` is the value of any of the applicable rules;
- if the associated `hitPolicy` is `UNORDERED`, the value of an instance of `DecisionTable` is the result of applying the aggregation function specified by the aggregation attribute of the `DecisionTable` to the unordered set of the values of the conclusions of all the applicable rules;
- if the associated `hitPolicy` is `RULE ORDER`, the value of an instance of `DecisionTable` is the result of applying the aggregation function specified by the aggregation attribute of

the `DecisionTable` to the set of the values of the conclusions of all the applicable rules, ordered according to the rule ordering;

- if the associated `hitPolicy` is `OUTPUT ORDER`, the value of an instance of `DecisionTable` is the result of applying the aggregation function specified by the aggregation attribute of the `DecisionTable` to the set of the values of the conclusions of all the applicable rules, ordered according to the ordering of the `outputEntry` in the clause of its conclusion (see clause 8.3.2, `Decision Table Clause`).

`DecisionTable` has an optional Boolean attribute `isComplete`. If an instance of `DecisionTable` has an `isComplete` attribute, the value of the attribute MUST be *false* if the `DecisionTable` is not *complete*. An instance of `DecisionTable` is said to be *complete* if and only if, for any valid binding of the `DecisionTable` `inputVariables`, at least one of the `DecisionTable` rules is applicable (see also clause 8.2.12).

`DecisionTable` has an optional Boolean attribute `isConsistent`. If an instance of `DecisionTable` has an `isConsistent` attribute, the value of the attribute MUST be *false* unless the `DecisionTable` is *consistent*. An instance of `DecisionTable` is said to be *consistent* if and only if, for any valid binding of the `DecisionTable` `inputVariables`, all the applicable rules have the same value.

`DecisionTable` inherits all the attributes and model associations from `Expression`. Table 23 presents the additional attributes and model associations of the `DecisionTable` element.

Table 23: `DecisionTable` attributes and model associations

Attribute	Description
clause: <code>Clause</code> [*]	This attributes lists the instances of <code>Clause</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>	The aggregation function to be applied to the values of the applicable rules when there are more than one, to determine the value of this <code>DecisionTable</code> . Default is: <code>COLLECT</code> .
isComplete: Boolean [0..1]	If present, this attribute MUST be <i>false</i> unless this <code>DecisionTable</code> is <i>complete</i> . Default is: <i>false</i> .

isConsistent: Boolean [0..1]	If present, this attribute MUST be <i>false</i> unless this DecisionTable is <i>consistent</i> . Default is: <i>false</i> .
preferredOrientation: DecisionTableOrientation [0..1]	The preferred orientation for the diagrammatic representation of this DecisionTable. This DecisionTable SHOULD BE represented as specified by this attribute.

8.3.2 Decision Table Clause metamodel

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 sub-domains of the subject's domain that are relevant for the piece of decision logic that is described by the decision table.

In DMN 1.0, the class `Clause` is used to model a decision table clause.

An instance of `Clause` is made of an optional `inputExpression` and of a set of `inputEntry`, or of an optional name and a set of `outputEntry`, which are instances of `Expression`. A `Clause` element MUST have a set of `inputEntry` if it has an `inputExpression`, it MUST have a set of `outputEntry` if it does not have an `inputExpression`. A `Clause` element MUST NOT have both `inputEntry` and `outputEntry`.

An instance of `Clause` may have a name, which is a `String`, and it may reference an `outputDefinition`, which is an `ItemDefinition` element. An instance of `Clause` that does not have an `inputExpression` MUST reference an `outputDefinition`. An instance of `Clause` that contains an `inputExpression` MUST NOT reference an `outputDefinition`. If a `Clause` element that references an `outputDefinition` does not have a name, its default name is the name of the referenced `ItemDefinition` element.

The `valueDefinition` of an `inputEntry` element MUST be `Boolean` and it MAY be omitted. The `inputEntry` elements MUST test the value of its containing clause's `inputExpression`, possibly implicitly. The `valueDefinition` of an `outputEntry` MUST be the `outputDefinition` or a specialization of the `outputDefinition` of the containing clause, and it MAY be omitted: if the `valueDefinition` of an `outputEntry` is omitted, it defaults to the `outputDefinition` of the containing clause.

In a tabular representation of the containing instance of `DecisionTable`, the representation of an instance of `Clause` depends on the orientation of the decision table. For instance, if the decision table is represented horizontally (rules as row, see clause 8.2.2), instances of `Clause` are represented as columns, with the `inputExpression` or the name of the `Clause` element represented in the top cell, its domain of value optionally listed in the cell below, and each of the cells below representing one of the `inputEntry` or `outputEntry` in the `Clause`. All the instances of `Clause` made of a set of `inputEntry`, MUST be represented on the right of any instance of `Clause` made of a set of `outputEntry`.

Table 24 presents the attributes and model associations of the Clause element.

Table 24: Clause attributes and model associations

Attribute	Description
inputExpression: Expression [0.1]	The subject of this input Clause.
outputDefinition: ItemDefinition [0.1]	The range of this output Clause.
name: String [0.1]	The name of this output Clause.
inputEntry: Expression [*]	This attribute lists the instances of Expression that compose this Clause.
outputEntry: Expression [*]	This attribute lists the instances of Expression that compose this Clause.

8.3.3 Decision Rule metamodel

In DMN 1.0, the class `DecisionRule` is used to model the rules in a decision table (see clause 8.2).

An instance of `DecisionRule` has a set of conditions and a non-empty set of conclusions, which are all instances of `Expression`.

An instance of `Expression` that is referenced by an instance of `DecisionRule` as a condition MUST be associated with a containing clause in which it is an `inputEntry`. In the same way, an instance of `Expression` that is referenced by an instance of `DecisionRule` as a conclusion MUST be associated with a containing clause in which it is an `outputEntry`.

A `DecisionRule` element MUST not have more than one conclusion contained in the same clause.

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 clause 8.2.2), instances of `DecisionRule` are represented as rows, with all the conditions represented on the left of all the conclusions.

By definition, a `DecisionRule` element that has no condition is always applicable. Otherwise, given a set of values for the `inputExpressions` of the clauses of its condition, an instance of `DecisionRule` is said to be *applicable* if and only if, for each Clause element that contains at least one of the rule's condition, at least one the rule's conditions that is contained in the Clause element is *true*. Equivalently, in logical terms, a `DecisionRule` element is said to be *applicable* if the conjunction is true where there is a conjunct per Clause element that has at least one

inputEntry referenced as a condition by the DecisionRule element, and each conjunct is a disjunction of all the rule's conditions that are contained in the same Clause element.

Table 25 presents the attributes and model associations of the DecisionRule element.

Table 25: DecisionRule attributes and model associations

Attribute	Description
condition: Expression [*]	This attribute lists the instances of Expression that compose the condition of this DecisionRule.
conclusion: Expression [1..*]	This attribute lists the instances of Expression that compose the conclusion of this DecisionRule.

8.4 Examples

Table 26 provides examples for the various types of decision table discussed in this section. Further examples may be found in 11.3, in the context of a complete example of a DMN decision model.

Table 26: Examples of decision tables

Single Hit Unique	<table border="1"> <thead> <tr> <th colspan="4">Applicant Risk Rating</th> </tr> <tr> <th>U</th> <th>Applicant Age</th> <th>Medical History</th> <th>Applicant Risk Rating</th> </tr> </thead> <tbody> <tr> <td>1</td> <td rowspan="2">> 60</td> <td>good</td> <td>Medium</td> </tr> <tr> <td>2</td> <td>bad</td> <td>High</td> </tr> <tr> <td>3</td> <td>[25..60]</td> <td>-</td> <td>Medium</td> </tr> <tr> <td>4</td> <td rowspan="2">< 25</td> <td>good</td> <td>Low</td> </tr> <tr> <td>5</td> <td>bad</td> <td>Medium</td> </tr> </tbody> </table>	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																																									
	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																																							
<table border="1"> <thead> <tr> <th colspan="6">Applicant Risk Rating</th> </tr> <tr> <th>Applicant Age</th> <th colspan="2">< 25</th> <th>[25..60]</th> <th colspan="2">> 60</th> </tr> <tr> <th>Medical History</th> <th>good</th> <th>bad</th> <th>-</th> <th>good</th> <th>bad</th> </tr> </thead> <tbody> <tr> <th>Applicant Risk Rating</th> <td>Low</td> <td>Medium</td> <td>Medium</td> <td>Medium</td> <td>High</td> </tr> <tr> <th>U</th> <td>1</td> <td>2</td> <td>3</td> <td>4</td> <td>5</td> </tr> </tbody> </table>	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																																					
Applicant Risk Rating	Low	Medium	Medium	Medium	High																																					
U	1	2	3	4	5																																					
<table border="1"> <thead> <tr> <th colspan="6">Applicant Risk Rating</th> </tr> <tr> <th>Applicant Age</th> <th colspan="2">< 25</th> <th>[25..60]</th> <th colspan="2">> 60</th> </tr> <tr> <th>Medical History</th> <th>good</th> <th>bad</th> <th>-</th> <th>good</th> <th>bad</th> </tr> </thead> <tbody> <tr> <th>Low</th> <td>X</td> <td>-</td> <td>-</td> <td>-</td> <td>-</td> </tr> <tr> <th>Medium</th> <td></td> <td>X</td> <td>X</td> <td>X</td> <td></td> </tr> <tr> <th>High</th> <td></td> <td></td> <td></td> <td></td> <td>X</td> </tr> <tr> <th>U</th> <td>1</td> <td>2</td> <td>3</td> <td>4</td> <td>5</td> </tr> </tbody> </table>	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
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	<table border="1"> <thead> <tr> <th colspan="5">Person Loan Compliance</th> </tr> <tr> <th>A</th> <th>Persons Credit Rating from Bureau</th> <th>Person Credit Card Balance</th> <th>Person Education Loan Balance</th> <th>Person Loan Compliance</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>A</td> <td>< \$10K</td> <td>< \$50K</td> <td>Compliant</td> </tr> <tr> <td>2</td> <td>Not A</td> <td>-</td> <td>-</td> <td>Not Compliant</td> </tr> <tr> <td>3</td> <td>-</td> <td>>= \$10K</td> <td>-</td> <td>Not Compliant</td> </tr> <tr> <td>4</td> <td>-</td> <td>-</td> <td>>= \$50K</td> <td>Not Compliant</td> </tr> </tbody> </table> <p><i>Example case: not A, >= \$10K, >= 50K -> Not Compliant (rules 2,3,4)</i></p>	Person Loan Compliance					A	Persons Credit Rating from Bureau	Person Credit Card Balance	Person Education Loan Balance	Person Loan Compliance	1	A	< \$10K	< \$50K	Compliant	2	Not A	-	-	Not Compliant	3	-	>= \$10K	-	Not Compliant	4	-	-	>= \$50K	Not Compliant											
Person Loan Compliance																																										
A	Persons Credit Rating from Bureau	Person Credit Card Balance	Person Education Loan Balance	Person Loan Compliance																																						
1	A	< \$10K	< \$50K	Compliant																																						
2	Not A	-	-	Not Compliant																																						
3	-	>= \$10K	-	Not Compliant																																						
4	-	-	>= \$50K	Not Compliant																																						

Single Hit Priority	Applicant Risk Rating				
	P	Applicant Age	Medical History	Applicant Risk Rating	
				<i>High, Medium, Low</i>	
	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
	Special Discount				
	Type of Order	Web		-	
	Customer Location	US		-	
	Type of Customer	Wholesaler	Retailer	-	
Special Discount %	10	5	0		
F	1	2	3		
<i>Example case: Web, non-US, Retailer -> 0 (rule 3)</i>					

Multiple Hit No order	<table border="1"> <thead> <tr> <th colspan="8">Holidays</th> </tr> </thead> <tbody> <tr> <td>Age</td> <td>-</td> <td><18</td> <td>>=60</td> <td>-</td> <td>[18..60]</td> <td>>=60</td> <td>-</td> </tr> <tr> <td>Years of Service</td> <td>-</td> <td>-</td> <td>-</td> <td>>=30</td> <td>[15..30]</td> <td>-</td> <td>>=30</td> </tr> <tr> <td>Holidays</td> <td>22</td> <td>5</td> <td>5</td> <td>5</td> <td>2</td> <td>3</td> <td>3</td> </tr> <tr> <td>N</td> <td>1</td> <td>2</td> <td>3</td> <td>4</td> <td>5</td> <td>6</td> <td>7</td> </tr> <tr> <td colspan="8">Aggregation=sum</td> </tr> </tbody> </table> <p><i>Example case: Age=58, Service=31 -> Result=sum(22, 5, 3)=30</i></p>	Holidays								Age	-	<18	>=60	-	[18..60]	>=60	-	Years of Service	-	-	-	>=30	[15..30]	-	>=30	Holidays	22	5	5	5	2	3	3	N	1	2	3	4	5	6	7	Aggregation=sum							
	Holidays																																																
Age	-	<18	>=60	-	[18..60]	>=60	-																																										
Years of Service	-	-	-	>=30	[15..30]	-	>=30																																										
Holidays	22	5	5	5	2	3	3																																										
N	1	2	3	4	5	6	7																																										
Aggregation=sum																																																	
Multiple Hit Output order	<table border="1"> <thead> <tr> <th colspan="4">Holidays</th> </tr> <tr> <th>O</th> <th>Age</th> <th>Years of Service</th> <th>Holidays</th> </tr> </thead> <tbody> <tr> <td></td> <td></td> <td></td> <td>22, 5, 3, 2</td> </tr> <tr> <td>1</td> <td>-</td> <td>-</td> <td>22</td> </tr> <tr> <td>2</td> <td>>= 60</td> <td>-</td> <td>3</td> </tr> <tr> <td>3</td> <td>-</td> <td>>= 30</td> <td>3</td> </tr> <tr> <td>4</td> <td>< 18</td> <td>-</td> <td>5</td> </tr> <tr> <td>5</td> <td>>= 60</td> <td>-</td> <td>5</td> </tr> <tr> <td>6</td> <td>-</td> <td>>= 30</td> <td>5</td> </tr> <tr> <td>7</td> <td>[18..60]</td> <td>[15..30]</td> <td>2</td> </tr> <tr> <td>8</td> <td>[45..60]</td> <td>< 30</td> <td>2</td> </tr> </tbody> </table> <p><i>Example case: Age=58, Service=31 -> Result=(22, 5, 3)</i></p>	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				
	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	<table border="1"> <thead> <tr> <th colspan="5">Student Financial Package Eligibility</th> </tr> <tr> <th>R</th> <th>Student GPA</th> <th>Student Extra-Curricular Activities Count</th> <th>Student National Honor Society Membership</th> <th>Student Financial Package Eligibility List</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>> 3.5</td> <td>>= 4</td> <td>Yes</td> <td>20% Scholarship</td> </tr> <tr> <td>2</td> <td>> 3.0</td> <td>-</td> <td>Yes</td> <td>30% Loan</td> </tr> <tr> <td>3</td> <td>> 3.0</td> <td>>= 2</td> <td>No</td> <td>20% Work-On-Campus</td> </tr> <tr> <td>4</td> <td><= 3.0</td> <td>-</td> <td>-</td> <td>5% Work-On-Campus</td> </tr> </tbody> </table> <p><i>Example case: For GPA=3.6, EC Activities=4, NHS Membership -> result = (20% scholarship, 30% loan)</i></p>	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																		
	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																																													

9 Simple Expression Language (S-FEEL)

DMN 1.0 defines the friendly enough expression language (FEEL) for the purpose of giving standard executable semantics to many kinds of expressions in decision model (see clause 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

9.1 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 and the production numbering is from the FEEL EBNF, clause 10.3.1.2.

Grammar rules:

- 4 arithmetic expression =
 - 4.a addition | subtraction |
 - 4.b multiplication | division |
 - 4.c exponentiation |
 - 4.d arithmetic negation ;
- 5 simple expression = arithmetic expression | simple value ;
- 6 simple expressions = simple expression , { "," , simple expression } ;
- 7 simple positive unary test =
 - 7.a ["<" | "<=" | ">" | ">="] , endpoint |
 - 7.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 simple positive unary tests = simple positive unary test , { "," , simple positive unary test } ;
- 14 simple unary tests =
 - 14.a simple positive unary tests |
 - 14.b “not”, "(", simple positive unary tests, ")" |
 - 14.c “”-“;
- 18 endpoint = simple value ;

- 19 simple value = qualified name | simple literal ;
- 20 qualified name = name , { "." , name } ;
- 21 addition = expression , "+" , expression ;
- 22 subtraction = expression , "-" , expression ;
- 23 multiplication = expression , "*" , expression ;
- 24 division = expression , "/" , expression ;
- 25 exponentiation = expression , "**" , expression ;
- 26 arithmetic negation = "-" , expression ;
- 27 name = name start , { name part | additional name symbols } ;
- 28 name start = name start char , { name part char } ;
- 29 name part = name part char , { name part char } ;
- 30 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] ;
- 31 name part char = name start char | digit | \uB7 | [\u0300-\u036F] | [\u203F-\u2040] ;
- 32 additional name symbols = "." | "/" | "-" | "'" | "+" | "*" ;
- 32 simple literal = numeric literal | string literal | Boolean literal ;
- 33 string literal = "" , { character – ("" | vertical space) } , "" ;
- 34 Boolean literal = "true" | "false" ;
- 35 numeric literal = digits , ["." , digits] | "." , digits ;
- 36 digit = [0-9] ;
- 37 digits = digit , {digit} ;

9.2 S-FEEL data types

S-FEEL supports all FEEL data types: *number*, *string*, *boolean*, *day time duration*, *year month 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*” [XSD]. 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 *day time duration* and *year month duration* datatypes. FEEL *day time duration* and *year month duration* have the same literal and value spaces as the [XPath Data Model](#) `dayTimeDuration` and `yearMonthDuration` datatypes, respectively. That is, FEEL *day 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 *year month 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 details in clause 10.3.2.2.

9.3 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.

For the sake of simplicity, the semantics of S-FEEL expressions are defined, in this section, with respect to the semantics of the XML Schema datatypes and of the corresponding XQuery and XPath functions and operators. A complete stand-alone 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 4a) 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 year month durations;
- `op:add-dayTimeDuration` and `subtract:dayTimeDurations`, when the two operands are day time durations;
- `op:add-yearMonthDuration-to-date` and `op:subtract-yearMonthDuration-from-date`, when the first operand is a year month 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 day 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 day 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.

Arithmetic addition and subtraction are not defined in other cases.

Arithmetic multiplication and division (grammar rule 4b) 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 4c) is defined as the multiplication of the first operand by itself as many times as indicated by the second operand. It is defined only when the first operand is a number and the second operand is an integer (that is, a number with no decimal or, equivalently whose decimals are all zeros).

Arithmetic negation (grammar rule 4d) 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 7.a) between numbers are defined according to the specification of `op:numeric-equal`, `op:numeric-less-than` and `op:numeric-greater-than`; comparison between dates are defined according to the specification of `op:date-equal`, `op:date-less-than` and `op:date-greater-than`; comparison between times are defined according to the specification of `op:time-equal`, `op:time-less-than` and `op:time-greater-than`; comparison between year month durations are defined according to the specification of `op:duration-equal`, `op:yearMonthDuration-less-than` and `op:year-MonthDuration-greater-than`; comparison between day 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 year month duration and day 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 endpoints `e1` and `e2`:

- is in the interval `(e1..e2)`, also noted `]e1..e2[`, if and only if `o > e1` and `o < e2`
- is in the interval `(e1..e2] also noted]e1..e2], if and only if o > e1 and o ≤ e2`
- is in the interval `[e1..e2)` also noted `[e1..e2[`, if and only if `o ≥ e1` and `o < e2`
- is in the interval `[e1..e2] also noted [e1..e2], if and only if o ≥ e1 and o ≤ e2.`

An expression to be tested satisfies an instance of simple unitary tests (grammar rule 9) 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 “-“.

9.4 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.

9.4.1 Item definitions

The expression that defines an **allowed value** must be a *simple unary test* (grammar rule 7), where only the values in the defined or referenced type that satisfy the test are allowed values.

9.4.2 Invocations

In the bindings of an invocation, the **binding formula** must be a *simple expression* (grammar rule 5).

9.4.3 Decision tables

Each **input expression** must be a *simple expression* (grammar rule 5).

Each **input value** must be a *simple unary test* (grammar rule 7), where the value that is tested is the value of the input expression of the containing clause. A list of input values must be an instance of *simple unary tests* (grammar rule 9).

Each list of **output values** must be an instance of *simple expressions* (grammar rule 6).

Each **input entry** must be a *simple unary test* (grammar rule 7), where the value that is tested is the value of the input expression of the containing clause. If a rule has multiple disjunctive conditions applying to the same input expression, the expression in the corresponding decision table cell must be an instance of *simple unary tests* (grammar rule 9).

Each **output entry** must be a *simple expression* (grammar rule 5).

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**) based on SQL and PMML

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**, or
- a **boxed function**.

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 **MUST** 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 38:

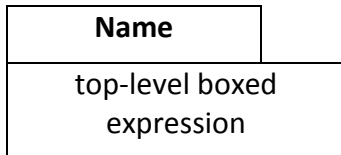


Figure 38: Boxed expression

Alternatively, the name box and expression box can be separated by white space and connected on the left side with a dotted line, as shown in Figure 39:

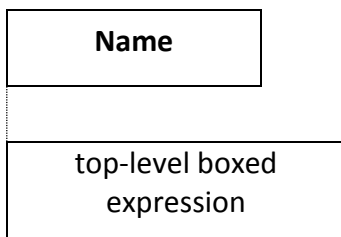


Figure 39: Boxed expression with separated name and expression boxes

Name is the only link defined between DRG elements and decision logic elements. 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 10.3.2.7.

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 40:



Figure 40: Boxed FEEL expression

The meaning of a boxed expression containing e is $\text{FEEL}(e, \mathbf{s})$, where \mathbf{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 builtin 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 41:

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

Figure 41: Boxed invocation

The boxed syntax maps to the textual syntax defined by grammar rules 40, 41, 42, 43. 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.

function-valued expression()

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. Each pair is called a context entry. Context entries may be separated by whitespace and connected with a dotted 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 must be arranged in one of the following ways (see Figure 42, Figure 43):

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

Figure 42: Vertical context

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

Figure 43: 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 44).

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

Figure 44: Use of context entries

Contexts with a final result box are useful for representing calculations (see Figure 45).

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 45: 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.

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

Figure 46: Vertical context with decision table entry

Color is suggested.

The names must 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 47):

Post-Bureau Risk Category

Existing Customer	Applicant. ExistingCustomer
Credit Score	Report. CreditScore
Application Risk Score	Affordability Model(Applicant, Product). Application Risk Score

UC	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 47: 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 }.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.

10.2.1.5 Boxed List

A **boxed list** is a list of n items. Cells must be arranged in one of the following ways (see Figure 48, Figure 49):

Item 1
Item 2
Item n

Figure 48: Vertical list

Item 1, Item 2, Item n

Figure 49: 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 50:

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 50: Relation

10.2.1.7 Boxed Function

The boxed expression associated with a Business Knowledge Model must be a boxed function.

A boxed function has 3 cells:

1. Kind, containing the initial letter of one of the following:
 - FEEL
 - 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 must be arranged as shown in Figure 51:

K	(Parameter1, Parameter2, ...)
Body	

Figure 51: Boxed function

For FEEL functions, the Body is an expression that references the parameters. For externally defined functions, the Body is a context as described in 10.3.2.10.2.

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

10.2.2 FEEL

A subset of FEEL, defined in the next section, serves as the notation "in the boxes" of boxed expressions. FEEL extends JSON objects. A JSON object is a number, a string, a context (JSON calls them maps) or a list of JSON objects. FEEL adds date, time, and duration objects, functions, friendlier syntax for literal values, and does not require the 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

Note that ranges and lists of ranges appear in decision table rule test cells. In the examples in Table 27, this portion of the syntax is shown in ***bold***.

Table 27: FEEL range comparisons

FEEL Expression	Value
<i>5 in <=5</i>	true

$1.0*10^{**3}$	1000.0
----------------	--------

10.3 Full FEEL syntax and semantics

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, \dots . Symbols may contain spaces. The following table summarizes the EBNF notation.

Example	Meaning
$S = S_1 ;$	Symbol S is defined in terms of symbol S_1
$S_1 S_2$	Either S_1 or S_2
S_1, S_2	S_1 followed by S_2
$[S_1]$	S_1 occurring 0 or 1 time
$\{S_1\}$	S_1 repeated 0 or more times
$k * S_1$	S_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" , hexadecimal digit , 4 * [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-f] | [A-F] ;
```

Note that the character range that includes all Unicode characters is `[\u0-\u10FFF]`.

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 55) is used to give execution semantics to boxed expressions.

1. expression = textual expression | boxed expression ;
2. textual expression =
 - a. function definition | for expression | if expression | quantified expression |
 - b. disjunction |
 - c. conjunction |
 - d. comparison |
 - e. arithmetic expression |
 - f. instance of |
 - g. path expression |
 - h. filter expression | function invocation |
 - i. literal | unary test | name | "(" , textual 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. simple positive unary tests = simple positive unary test , { " , " , simple positive unary test } ;
 14. simple unary tests =
 - a. simple positive unary tests |
 - b. “not”, "(, simple positive unary tests,)" |
 - c. “-“;
 15. positive unary test = simple positive unary test | "null" ;
 16. positive unary tests = positive unary test , { " , " , positive unary test } ;
 17. unary tests =
 - a. positive unary tests |
 - b. “not”, “(, positive unary tests,)” |
 - c. “-“
 18. endpoint = simple value ;
 19. simple value = qualified name | simple literal ;
 20. qualified name = name , { "." , name } ;
 21. addition = expression , "+" , expression ;
 22. subtraction = expression , "-" , expression ;
 23. multiplication = expression , "*" , expression ;
 24. division = expression , "/" , expression ;

25. exponentiation = expression, "**", expression ;
26. arithmetic negation = "-", expression ;
27. name = name start , { name part | additional name symbols } ;
28. name start = name start char, { name part char } ;
29. name part = name part char , { name part char } ;
30. 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] ;
31. name part char = name start char | digit | \uB7 | [\u0300-\u036F] | [\u203F-\u2040] ;
32. additional name symbols = "." | "/" | "-" | "'" | "+" | "*" ;
33. literal = simple literal | "null" ;
34. simple literal = numeric literal | string literal | Boolean literal ;
35. string literal = "" , { character – ("" | vertical space) } , "" ;
36. Boolean literal = "true" | "false" ;
37. numeric literal = digits , [".", digits] | ".", digits ;
38. digit = [0-9] ;
39. digits = digit , {digit} ;
40. function invocation = expression , parameters ;
41. parameters = "(" , (named parameters | positional parameters) , ")" ;
42. named parameters = parameter name , ":" , expression , { ",", parameter name , ":" , expression } ;
43. parameter name = name ;
44. positional parameters = [expression , { ",", expression }] ;
45. path expression = expression , ".", name ;
46. for expression = "for" , name , "in" , expression { ",", name , "in" , expression } , "return" , expression ;
47. if expression = "if" , expression , "then" , expression , "else" expression ;
48. quantified expression = ("some" | "every") , name , "in" , expression , { name , "in" , expression } , "satisfies" , expression ;
49. disjunction = expression , "or" , expression ;
50. conjunction = expression , "and" , expression ;
51. comparison =
 - a. expression , ("=" | "!=" | "<" | "<=" | ">" | ">=") , expression |

- b. expression , "between" , expression , "and" , expression |
- c. expression , "in" , (positive unary test | positive unary tests) ;
- 52. filter expression = expression , "[" , expression , "]" ;
- 53. instance of = expression , "instance" , "of" , type ;
- 54. type = qualified name ;
- 55. boxed expression = list | function definition | context ;
- 56. list = "[" [expression , { "," , expression }] , "]" ;
- 57. function definition = "function" , "(" , [formal parameter { "," , formal parameter }] , ")" , ["external"] , expression ;
- 58. formal parameter = parameter name ;
- 59. context = "{" , [context entry , { "," , context entry }] , "}" ;
- 60. context entry = key , ":" , expression ;
- 61. key = name | string literal ;

Additional syntax rules:

- Operator precedence is given by the order of the alternatives in grammar rules 2 and 4. *E.g.*, 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.
- A name may contain spaces but may not contain a sequence of 2 or more spaces.
- Java-style comments can be used, *i.e.* `'/'` to end of line and `/* ... */`.

10.3.1.3 Literals, data types, built-in functions

FEEL supports literal syntax for numbers, strings, booleans, 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
- duration
 - days and time duration
 - years and months duration
- time
- date and time

Duration and date/time datatypes have no literal syntax. They must be constructed from a string representation using a built-in function (10.3.4.1).

10.3.1.4 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 59). 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 56).

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_1 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.*,

$$\begin{aligned} & \{a: [\{b: 1\}, \{b: [2.1, 2.2]\}]\}, \{a: [\{b: 3\}, \{b: 4\}, \{b: 5\}]\}.a.b = \\ & [[\{b: 1\}, \{b: [2.1, 2.2]\}], [\{b: 3\}, \{b: 4\}, \{b: 5\}]].b = \\ & [[1, [2.1, 2.1]], [3, 4, 5]] \end{aligned}$$

Nested lists can be flattened using the *flatten()* built-in function (10.3.4).

10.3.1.5 Ambiguity

Names of context entries and function parameters can contain commonly used characters such as space and apostrophe (but cannot contain a colon or comma (':' or ',')). This naming freedom makes FEEL's syntax ambiguous. Ambiguity is resolved using the scope. Names 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)

10.3.2 Semantics

FEEL semantics is specified by mapping syntax fragments to values in the FEEL semantic domain. Literals (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. 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 $e = \text{FEEL}(e, s)$. Two FEEL expressions e_1 and e_2 are equivalent in scope s if and only if $\text{FEEL}(e_1, s) = \text{FEEL}(e_2, s)$. When s is understood from context (or not important), we may abbreviate the equivalence as $e_1 = e_2$.

10.3.2.1 Semantic Domain

The FEEL semantic domain \mathbf{D} consists of an infinite number of values of the following *kinds*: functions, lists, contexts, ranges, datatypes, and the distinguished value **null**. Each kind is disjoint (*e.g.* a value cannot be both a number and a list).

A function is a lambda expression with lexical closure or is externally defined by Java 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,

$$\text{FEEL}([1+1, 2+2]) = [2, 4]$$

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 Semantics of literals and datatypes

FEEL datatypes are listed in 10.3.2.2. 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.

10.3.2.2.1 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. Numbers are a restriction of the XML Schema type [precisionDecimal](#), and are equivalent to Java [BigDecimal](#) with [MathContext](#) DECIMAL128.

Grammar rule 37 defines literal numbers. Literals consist of base 10 digits and an optional decimal point. -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.* $\text{FEEL}(\text{number}("1.000.000,01", ".", ",")) = \mathbf{1000000.01}$.

FEEL does not support a literal scientific notation. *E.g.*, 1.2e3 is not valid FEEL syntax. Use $1.2*10**3$ instead.

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 [-6111..6176]. Each such pair represents the number $\mathbf{p}/10^s$. To indicate the numeric value, we write **value(p,s)**. *E.g.* $\mathbf{value(100,2)=1}$. 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, $\mathbf{value(1,0) = value(10,1) = value(100,2)}$.

There is no value for notANumber, positiveInfinity, or negativeInfinity. Use **null** instead.

10.3.2.2.2 string

Grammar rule 35 defines literal strings as a double-quoted sequence of characters, *e.g.* "abc".

The literal string "abc" is mapped to the semantic domain as a sequence of three Unicode characters **a**, **b**, and **c**, written "**abc**".

10.3.2.2.3 boolean

The Boolean literals are given by grammar rule 36. The values in the semantic domain are **true** and **false**.

10.3.2.2.4 time

FEEL does not have time literals, although we use boldface time literals to represent values in the semantic domain. Times can be expressed using a string literal and the *time()* built-in function. The literal format of the characters within the quotes of the string literal is defined by the lexical space of the [XML Schema](#) time datatype. A time in the semantic domain is a sequence of numbers for the hour, minute, second, and optional timezone offset.

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 and so it has an inverse function **value_t⁻¹**. *E.g.*, **value_t⁻¹(3661)=01:01:01**.

10.3.2.2.5 date and date-time

FEEL does not have date-time literals, although we use boldface date-time literals to represent values in the semantic domain. Dates can be expressed using a string literal and the *date()* built-in function; date-times can be expressed using a string literal and the *date and time()* built-in function. The literal format of the characters within the quotes of the string literal is defined by the lexical space of the [XML Schema](#) date and dateTime datatypes. A date and time in the semantic domain is a sequence of numbers for the year, month, day, and optional hour, minute, second, and timezone offset. *I.e.*, the value for a date and time consists of a sequence of 3 numbers for the date, an optional 3 additional numbers for the time, and an optional timezone offset. The year must be in the range [-999,999,999..999,999,999].

A date and time **d** 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}⁻¹(value_{dt}(d)) = d**. **value_{dt}⁻¹** returns **null** rather than a date with a year outside the legal range.

10.3.2.2.6 days and time duration

FEEL does not have duration literals although we use boldface duration literals to represent values in the semantic domain. Durations can be expressed using a string literal and the *duration()* built-in function. 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 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**.

10.3.2.2.7 years and months duration

FEEL does not have duration literals, although we use boldface duration literals to represent values in the semantic domain. Durations can be expressed using a string literal and the *duration()* built-in function. 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, $\text{FEEL}(\text{duration}("P0Y13M")) = \mathbf{P1Y1M}$.

The value of a years and months duration can be expressed as a number of months. *E.g.*, $\text{value}_{\text{ymd}}(\mathbf{P1Y1M}) = 13$. The $\text{value}_{\text{ymd}}$ function is one-to-one and so it has an inverse function $\text{value}_{\text{ymd}}^{-1}$. *E.g.*, $\text{value}_{\text{ymd}}^{-1}(13) = \mathbf{P1Y1M}$.

10.3.2.3 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.

10.3.2.4 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 $L[-1]$.

If $\text{FEEL}(L) = \mathbf{L}$ is a list in the FEEL semantic domain, the first element is $\text{FEEL}(L[1]) = \mathbf{L}[1]$. If \mathbf{L} is empty then $\mathbf{L}[1] = \mathbf{null}$.

\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:

$$\begin{aligned} [1, 2, 3, 4][\text{item} > 2] &= [3, 4] \\ [\{x:1, y:2\}, \{x:2, y:3\}][x=1] &= [\{x:1, y:2\}] \\ [[\{x:1, y:2\}, \{x:2, y:3\}]] [x=1] &= [] \end{aligned}$$

The filter expression is evaluated for each item in list, and a list containing only items where the filter expression is **true** is returned.

For convenience, any function or operator that expects a list as input but instead receives a non-list semantic domain element e behaves as if it had received $[e]$ as input.

Also for convenience, a selection using the "." operator with a list of contexts on its left hand side returns a list of selections, *i.e.* $\text{FEEL}(e.f, \mathbf{c}) = [\text{FEEL}(f, \mathbf{c}'), \text{FEEL}(f, \mathbf{c}''), \dots]$ where $\text{FEEL}(e) = [\mathbf{e}', \mathbf{e}'', \dots]$ 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.

10.3.2.5 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 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 *key_l* from context-valued expression *m* is *m.key_l*.

If *key_l* 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, "key_l")*. Selecting a value by key from context **m** in the semantic domain is denoted as **m.key_l** or **get value(m, "key_l")**

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 other context entries in the same context. These references must be acyclic and form a partial order. The expressions in a context must be evaluated consistent with this partial order.

10.3.2.6 Ranges

FEEL supports a compact syntax for a range of values, useful in decision table test cells and elsewhere. A range maps to the semantic domain as a single comparable value (number, date/time/duration, or string) or a pair of comparable endpoint values and an endpoint inclusivity code that indicates whether one or both endpoints are included in the range.

The range syntax supports literal and symbolic endpoints, but not arbitrary expressions. Because date/time/duration values have no literal syntax, symbolic endpoints must be used for ranges of these types. E.g., the following context defines a range of duration named *soon* from one to two minutes, inclusive.

```
{
    one min: duration("PT1M"),
    two min: duration("PT2M"),
    soon: [one min..two min]
}
```

10.3.2.7 Decision Table

A decision table as described in clause 7 has 10 kinds of cells:

1. Hit policy and Completeness Code (HC)
2. Rule number (1,2,3,...)
3. Input (I1, I2, ...)
4. Optional input values (IV1a, IV1b, IV2a, IV2b, ...)
5. Output O
6. Optional multiple outputs (O1, O2, ...)
7. Output values for each output (OV1a, OV1b)
8. Rule tests (RT1a, R21b, RT2a, RT2b, etc.)
9. Rule output (RO1a, RO1b, RO2a, RO2b ...)

10. Aggregation (A)

E.g., the 'rules as rows' layout looks like:

HC	I1	I2	O	
	IV1a, IV1b	IV2a, IV2b	O1	O2
1	RT1a	RT2a	RO1a	RO2a
2		RT2b	RO1b	RO2b
3	RT1b	ANY	RO1a	RO2a
A				

The execution semantics of a decision table as described schematically above is:

```

FEEL(
  decision table(
    inputs: [I1, I2, ...],
    input values: [[IV1a, IV1b], [IV2a, IV2b], ...],
    outputs: [O1, O2, ...],
    output values: [[OV1a, OV1b],[OV2a, OV2b]],
    rules:[
      [RT1a, RT2a, ..., RO1a, RO2a],
      [RT1a, RT2b, ..., RO1b, RO2b],
      [RT1b, ANY, ..., RO1a, RO2a]],
    hit policy: H, completeness: C, aggregation: A ) )

```

That is, the meaning is defined by the meaning of the FEEL built-in *decision table* function (see 10.3.4.5).

10.3.2.7.1 Name

Note that we do not include the Name cell here because only the top-level boxed expression has a Name cell, and the decision table may not be the top-level BE. When a decision table is the top-level boxed expression, it must have a name and the output cell must be blank or the same as the name. When a decision table is an entry in a boxed context, the name cell of the context entry may be omitted. In this case, the output cell must contain the context entry name and cannot be blank unless it is the last entry. When a decision table is not top-level and not a context entry, the decision table has no name and the output cell is uninterpreted.

10.3.2.7.2 Hit policy

The hit policy is described in the specification of the *decision table* built-in (10.3.4.5). The default hit policy is "U", for Unique. Crosstab decision tables always have unique hits.

Customized hit policies such as returning the maximum value, can be added easily using a boxed expression to post-process the output list.

10.3.2.7.3 Complete

Completeness is described in the specification of the *decision table* built-in (10.3.4.5). The default value is "C", for Complete.

10.3.2.7.4 Input

Each input cell is a boxed expression. The expression must be a *textual expression* (grammar rule 2).

10.3.2.7.5 Input values

Input values are optional. If provided, it is a comma-separated list of FEEL expressions. . Each input value must be a *unary test* (grammar rule 10). A list of input values must be an instance of *unary tests* (grammar rule 11).

The list restricts the corresponding input to the given list of values, mapping others to null. The rule test cell can test for null as a proxy "otherwise", that is, to test for values not in the list of input values.

10.3.2.7.6 Output

When a decision table is an entry in a boxed context, the output cell may be used to contain the name of the entry, instead of a separate entry name cell, thus saving space.

10.3.2.7.7 Output Values

If provided, the list of output values must be an instance of *textual expressions* (grammar rule 3).

10.3.2.7.8 Rule tests

Rule test cells, also called input entries, contain FEEL expressions restricted to contain only a (list of) *unary test* (grammar rule 10) or a list of *unary tests* (grammar rule 11).

The meaning of *rule matching* is composed from these rule test expressions. The rule tests are combined with corresponding inputs to form *in* expressions (FEEL grammar rule 51.c) that are *anded* (FEEL grammar rule 50) to determine whether a rule is matched.

10.3.2.7.9 Rule output

A rule output cell is a boxed expression. The expression must be a *textual expression* (grammar rule 2).

10.3.2.7.10 Aggregation

The aggregation is described in the specification of the *decision table* built-in (10.3.4.5). The default value is "collect".

10.3.2.8 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 \mathbf{s} is the *global* context. The first context in \mathbf{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 \mathbf{s} appended with $.N$, where N is the name of entry in the first context of \mathbf{s} containing e . QNs in e are resolved by looking through the contexts in \mathbf{s} from first to last.

10.3.2.8.1 Local context

If e denotes the value of a context entry of context \mathbf{m} , then \mathbf{m} is the local context for e , and \mathbf{m} is the first element of \mathbf{s} . Otherwise, e has no local context and the first element of \mathbf{s} is the global context, or in some cases explained later, the first element of \mathbf{s} is a special context.

All of the entries of \mathbf{m} are in-scope for e , but the *depends on* graph must be acyclic. This provides a simple solution to the problem of the confusing definition above: if \mathbf{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 .

10.3.2.8.2 Global context

The global context is a context provided for convenience and 'pre-compilation'. Any number of expressions can be named and represented in a FEEL context \mathbf{m} . The syntactic description m of this context can be evaluated once, that is, mapped to the FEEL domain as \mathbf{m} , and then re-used to evaluate many expressions.

10.3.2.8.3 Built-in context

The built-in context contains all the built-in functions.

10.3.2.8.4 Special context

Some FEEL expressions are interpreted in a *special* context that is pushed on the front of \mathbf{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 \mathbf{s} . The meaning of a FEEL expression e in scope \mathbf{s} is denoted as $\mathbf{FEEL}(e, \mathbf{s})$. We can also say that e evaluates to \mathbf{e} in scope \mathbf{s} , or $\mathbf{e} = \mathbf{FEEL}(e, \mathbf{s})$. Note that \mathbf{e} and \mathbf{s} are elements of the FEEL domain. \mathbf{s} is a list of contexts.

10.3.2.9 Mapping between FEEL and other domains

A FEEL expression e denotes a value \mathbf{e} in the semantic domain. Some kinds of values can be passed between FEEL and external Java methods, between FEEL and external PMML models, and between FEEL and XML, as summarized in the following table:

<i>FEEL value</i>	<i>Java</i>	<i>XML</i>	<i>PMML</i>
number	java.math.BigDecimal	decimal	decimal, PROB-NUMBER, PERCENTAGE-NUMBER
		integer	integer , INT-NUMBER

		double	double, REAL-NUMBER
string	java.lang.String	string	string, FIELD-NAME
date and time, 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	X
boolean	java.lang.Boolean	boolean	boolean
range	<i>TBD</i>	<i>TBD</i>	<i>TBD</i>
function	X	X	X
list	java.util.List	contain multiple child elements	array (homogeneous)
context	java.util.Map	contain attributes and child elements	X

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 $\mathbf{FEEL}(e, c)$.

10.3.2.10 Function Semantics

FEEL functions can be

- built-in, *e.g.*,
decision table (see clause 10.3.4.5), or
- user-defined, *e.g.*,
function(age) age < 21, or
- externally defined, *e.g.*,
function(angle) external {
 java: {
 class: "java.lang.Math",
 method signature: "cos(double)"
 }}

FEEL built-ins are specified in 10.3.4.

10.3.2.10.1 User-defined functions

User-defined functions have the form

$$\mathit{function}(X_1, \dots, X_n) e$$

The terms X_1, \dots, X_n are parameter names. The function body is e . The meaning of $\mathbf{FEEL}(\mathit{function}(X_1, \dots, X_n) e, s)$ is an element in the FEEL semantic domain that we denote as $\mathbf{function}(\mathbf{argument\ list}: [X_1, \dots$

$X_n]$, **body:** e , **scope:** 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 s .

The invocation of a FEEL user-defined function **f** is denoted as $\mathbf{f}(Y_1, \dots Y_n)$. The meaning $\text{FEEL}(\mathbf{f}(Y_1, \dots Y_n), \mathbf{S})$, where **f** has already been interpreted, is computed as follows:

1. the parameter names $X_1, \dots X_n$ and corresponding values $\mathbf{Y}_1, \dots \mathbf{Y}_n$ are paired in a context $\mathbf{c} = \{X_1 : \mathbf{Y}_1, \dots X_n : \mathbf{Y}_n\}$. $\mathbf{Y}_i = \text{FEEL}(Y_i, \mathbf{S})$.
2. e is interpreted in \mathbf{s}' , where $\mathbf{s}' = \text{insert before}(\mathbf{s}, \mathbf{1}, \mathbf{c})$ (see 10.3.4.4)

10.3.2.10.2 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 must have one of the following forms:

```
{  
  java: {class: class-name, method signature: method-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)**. In the mapping information, *class-name* is a string name of a Java class on the classpath. Classpath configuration is implementation-defined. *Method-signature* is a string consisting of the name of a public static method in the named class, and an argument list containing only Java argument type names. The argument type information is used to resolve overloaded methods and may be used to detect out-of-domain errors before runtime. *IRI* is the resource identifier for a PMML document, and *model-name* is an optional name of a model in the document. If no model-name is specified, the first model in the document is used.

When an externally-defined function is invoked, actual argument values and result value are converted when possible using the type mapping table for Java or PMML [ref to this table in Types and Inference]. 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**.

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.

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.

10.3.2.10.3 Function name

To name a function, define it as a context entry. *E.g.*

```
{
  isPositive : function(x) x > 0,
  isNotNegative : function(x) isPositive(x+1),
  result: isNotNegative(0)
}
```

10.3.2.10.4 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 must be supplied. If named, unsupplied parameters are bound to **null**.

10.3.2.11 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.

Grammar Rule	FEEL Syntax	Mapped to Domain
57	<i>function(n₁, ... n_N) e</i>	function(argument list: [n₁, ... n_N], body: e, scope: s)
57	<i>function(n₁, ... n_N) external e</i>	function(argument list: [n₁, ... n_N], external: e)

See 10.3.2.6.

Grammar Rule	FEEL Syntax	Mapped to Domain
46	<i>for n₁ in e₁, n₂ in e₂, ... return e</i>	[FEEL(e, s'), FEEL(e, s''), ...]
47	<i>if e₁ then e₂ else e₃</i>	if FEEL(e₁) then FEEL(e₂) else FEEL(e₃)
48	<i>some n₁ in e₁, n₂ in e₂, ... satisfies e</i>	FEEL(e, s') or FEEL(e, s'') or ...
48	<i>every n₁ in e₁, n₂ in e₂, ... satisfies e</i>	FEEL(e, s') and FEEL(e, s'') and ...
49	<i>e₁ or e₂ or ...</i>	FEEL(e₁) or FEEL(e₂) or ...

50	e_1 and e_2 and ...	FEEL(e_1) and FEEL(e_2) and ...
51.a	$e = null$	FEEL(e) = null
51.a	$e \neq null$	FEEL(e) != 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	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

a	not(a)
true	false
false	true
otherwise	null

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 $FEEL(e_1) \times FEEL(e_2) \times \dots$, **s''** is **s** with a special first context containing keys bound to the second element of the Cartesian product, etc.

Grammar Rule	FEEL Syntax	Input Domain	Result

51.a	$e_1 < e_2$	e_1 and e_2 must both be of the same kind/datatype – both numbers, both strings, <i>etc.</i>	<i>See below</i>
------	-------------	--	------------------

Equality and inequality map to several kind- and datatype-specific tests:

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 endpoints and the same endpoint inclusivity code.
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 10.3.2.2.1. Precision is not considered.
string	e_1 is the same sequence of characters as e_2
date and time	all 7 components (10.3.2.2.5), treating unspecified optional components as null, must be equal
time	all 4 components (10.3.2.2.4), treating unspecified optional components as null, must be equal
days and time duration	value(e_1) = value(e_2) . Value is defined in 10.3.2.2.6.
years and months duration	value(e_1) = value(e_2) . Value is defined in 10.3.2.2.7.
boolean	e_1 and e_2 must both be true or both be false

By definition, $FEEL(e_1 \neq e_2) = FEEL(not(e_1=e_2))$.

The other comparison operators are defined only for the datatypes as shown in the following table. ‘>’ is similar to ‘<’ and is omitted for brevity. $e_1 \leq e_2$ is defined as $e_1 < e_2$ or $e_1 = e_2$.

datatype	$e_1 < e_2$
number	value(e_1) < value(e_2) . value is defined in 10.3.2.2.1. 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 $\backslash u0$

	characters, stripped of common prefix characters, and then the first character in each sequence is compared.
date and time	$\text{value}_{dt}(e_1) < \text{value}_{dt}(e_2)$. value_{dt} is defined in 10.3.2.2.5. If one input has a null timezone offset, that input uses the timezone offset of the other input.
time	$\text{value}_t(e_1) < \text{value}_t(e_2)$. value_t is defined in 10.3.2.2.4. If one input has a null timezone offset, that input uses the timezone offset of the other input.
days and time duration	$\text{value}_{dtd}(e_1) < \text{value}_{dtd}(e_2)$. value_{dtd} is defined in 10.3.2.2.6.
years and months duration	$\text{value}_{ymd}(e_1) = \text{value}_{ymd}(e_2)$. value_{ymd} is defined in 10.3.2.2.7.

FEEL supports additional syntactic sugar for comparison. Note that Grammar Rules (clause 10.3.1.2) are used in decision table condition cells.

Grammar Rule	FEEL Syntax	Equivalent FEEL Syntax	applicability
51.b	e_1 between e_2 and e_3	$e_1 \geq e_2$ and $e_1 \leq e_3$	
51.c	e_1 in [e_2, e_3, \dots]	$e_1 = e_2$ or $e_1 = e_3$ or...	e_2 and e_3 are endpoints
51.c	e_1 in [e_2, e_3, \dots]	e_1 in e_2 or e_1 in e_3 or...	e_2 and e_3 are ranges
51.c	e_1 in $\leq e_2$	$e_1 \leq e_2$	
51.c	e_1 in $< e_2$	$e_1 < e_2$	
51.c	e_1 in $\geq e_2$	$e_1 \geq e_2$	
51.c	e_1 in $< e_2$	$e_1 < e_2$	
51.c	e_1 in ($e_2..e_3$)	$e_1 > e_2$ and $e_1 < e_3$	
51.c	e_1 in ($e_2..e_3$]	$e_1 > e_2$ and $e_1 \leq e_3$	
51.c	e_1 in [$e_2..e_3$)	$e_1 \geq e_2$ and $e_1 < e_3$	
51.c	e_1 in [$e_2..e_3$]	$e_1 \geq e_2$ and $e_1 \leq e_3$	

In Grammar Rule **51c**, the qualified name must evaluate to a comparable constant value at modeling time, *i.e.* the endpoint must be a literal or a named constant.

Grammar Rule	FEEL	Input Domain and Result
21	$e_1 + e_2$	See below

22	$e_1 - e_2$	See below
----	-------------	-----------

Addition and subtraction are defined in the following table. Note that if input values are not of the listed types, the result is **null**.

type(e₁)	type(e₂)	$e_1 + e_2, e_1 - e_2$	result type
number	number	Let $e_1=(p_1,s_1)$ and $e_2=(p_2,s_2)$ as defined in 10.3.2.2.1. If $\text{value}(p_1,s_1) \pm \text{value}(p_2,s_2)$ requires a scale outside the range of valid scales, the result is null . Else the result is (p,s) such that <ul style="list-style-type: none"> • $\text{value}(p,s) = \text{value}(p_1,s_1) \pm \text{value}(p_2,s_2) + \varepsilon$ • $s \leq \max(s_1,s_2)$ • 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 $\text{value}_{\text{dtd}}^{-1}(\text{value}_{\text{dt}}(e_1) - \text{value}_{\text{dt}}(e_2))$, where value_{dt} is defined in 10.3.2.2.5 and $\text{value}_{\text{dtd}}^{-1}$ is defined in 10.3.2.2.6.	days and time duration
time	time	Addition is undefined. Subtraction is defined as $\text{value}_{\text{dtd}}^{-1}(\text{value}_{\text{t}}(e_1) - \text{value}_{\text{t}}(e_2))$ where value_{t} is defined in 10.3.2.2.4 and $\text{value}_{\text{dtd}}^{-1}$ is defined in 10.3.2.2.6.	days and time duration
years and months duration	years and months duration	$\text{value}_{\text{ymd}}^{-1}(\text{value}_{\text{ymd}}(e_1) \pm \text{value}_{\text{ymd}}(e_2))$ where $\text{value}_{\text{ymd}}$ and $\text{value}_{\text{ymd}}^{-1}$ is defined in 10.3.2.2.7.	years and months duration
days and time duration	days and time duration	$\text{value}_{\text{dtd}}^{-1}(\text{value}_{\text{dtd}}(e_1) \pm \text{value}_{\text{dtd}}(e_2))$ where $\text{value}_{\text{dtd}}$ and $\text{value}_{\text{dtd}}^{-1}$ is defined in 10.3.2.2.6	days and time duration
date and time	years and months duration	$\text{value}_{\text{dt}}^{-1}(\text{value}_{\text{dt}}(e_1) \pm \text{value}_{\text{ymd}}(e_2))$ where value_{dt} and $\text{value}_{\text{dt}}^{-1}$ is defined in 10.3.2.2.5 and $\text{value}_{\text{ymd}}$ is defined in 10.3.2.2.7.	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	$\text{value}_{\text{dt}}^{-1}(\text{value}_{\text{dt}}(e_1) \pm \text{value}_{\text{dtd}}(e_2))$ where value_{dt} and $\text{value}_{\text{dt}}^{-1}$ is defined in 10.3.2.2.5 and $\text{value}_{\text{dtd}}$ is defined in 10.3.2.2.6.	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	$\text{value}_t^{-1}(\text{value}_t(e_1) \pm \text{value}_{\text{dtd}}(e_2))$ where value_t and value_t^{-1} is defined in 10.3.2.2.4 and $\text{value}_{\text{dtd}}$ is defined in 10.3.2.2.6.	time
days and time duration	time	Subtraction is undefined. Addition is commutative and is defined by the previous rule.	time

Grammar Rule	FEEL	Input Domain and Result
23	$e_1 * e_2$	See below
24	e_1 / e_2	See below

Multiplication and division are defined in the following table. Note that if input values are not of the listed types, the result is **null**.

type(e_1)	type(e_2)	$e_1 * e_2$	e_1 / e_2	result type
number $e_1=(p_1,s_1)$	number $e_2=(p_2,s_2)$	<p>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</p> <ul style="list-style-type: none"> $\text{value}(p,s) = \text{value}(p_1,s_1) * \text{value}(p_2,s_2) + \epsilon$ $s \leq s_1+s_2$ s is maximized subject to the limitation that p has 34 digits or less ϵ is a possible rounding error 	<p>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</p> <ul style="list-style-type: none"> $\text{value}(p,s) = \text{value}(p_1,s_1) / \text{value}(p_2,s_2) + \epsilon$ $s \leq s_1-s_2$ s is maximized subject to the limitation that p has 34 digits or less ϵ is a possible rounding error 	number
years and months duration	number	$\text{value}_{\text{ymd}}^{-1}(\text{value}_{\text{ymd}}(e_1) * \text{value}(e_2))$ where $\text{value}_{\text{ymd}}$ and $\text{value}_{\text{ymd}}^{-1}$ are defined in 10.3.2.2.7.	If $\text{value}(e_2)=0$, the result is null . Else the result is $\text{value}_{\text{ymd}}^{-1}(\text{value}_{\text{ymd}}(e_1) / \text{value}(e_2))$ where $\text{value}_{\text{ymd}}$ and $\text{value}_{\text{ymd}}^{-1}$ are defined	years and months duration

			in 10.3.2.2.7.	
number	years and months duration	<i>see above, reversing e_1 and e_2</i>		
days and time duration	number	$\text{value}_{\text{dtd}}^{-1}(\text{value}_{\text{dtd}}(e_1) * \text{value}(e_2))$ where $\text{value}_{\text{dtd}}$ and $\text{value}_{\text{dtd}}^{-1}$ are defined in 10.3.2.2.6.	If $\text{value}(e_2)=0$, the result is null . Else the result is $\text{value}_{\text{dtd}}^{-1}(\text{value}_{\text{dtd}}(e_1) * \text{value}(e_2))$ where $\text{value}_{\text{dtd}}$ and $\text{value}_{\text{dtd}}^{-1}$ are defined in 10.3.2.2.6.	days and time duration
number	days and time duration	<i>see above, reversing e_1 and e_2</i>		

Grammar Rule	FEEL Syntax	Input Domain	Result
25	$e_1 ** e_2$	$\text{type}(e_1)$ is number. $\text{value}(e_2)$ is an integer in the range [-999,999,999..999,999,999].	If $\text{value}(e_1)^{\text{value}(e_2)}$ requires a scale that is out of range, the result is null . Else the result is (p,s) such that <ul style="list-style-type: none"> $\text{value}(p,s) = \text{value}(e_1)^{\text{value}(e_2)} + \epsilon$ p is limited to 34 digits ϵ is rounding error

Grammar Rule	FEEL Syntax	Mapped to Domain
53	e instance of type	true iff $\text{type}(e)$ is type

Note that *type* is not mapped to the domain, and *null* is not the name of a type, and **null** is not an instance of any type.

Grammar Rule	FEEL Syntax	Equivalent FEEL Syntax
26	$-e$	$0-e$

Grammar Rule	FEEL	Mapped to Domain	Applicability
40, 41, 44	$e(e_1, \dots)$	$e(e_1, \dots)$	e is a function with matching arity
40, 41, 42, 43	$e(n_1:e_1, \dots)$	$e(n_1:e_1, \dots)$	e is a function with matching parameter names

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.

Grammar Rule	FEEL	Mapped to Domain	Applicability
20	$e.name$	$e."name"$	$type(e)$ is a context
20	$e.name$	<i>see below</i>	$type(e)$ is a date/time/duration

If $type(e)$ is date and time, time, or a duration, and **name** is a property name as shown in the following table, then the meaning is given by the following table:

$type(e)$	$e.name$	name =
date and time	result is the named component of the date and time object e . Valid names are shown to the right. hour, minute, second, and timezone values may be null .	year, month, day, hour, minute, second, timezone
time	result is the named component of the time object e . Valid names are shown to the right. timezone values may be null .	hour, minute, second, timezone
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
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

For example, $FEEL(date\ and\ time("03-07-2012Z").year) = 2012$.

Grammar	FEEL	Mapped to Domain (scope s)	Applicability
---------	------	----------------------------	---------------

Rule	Syntax		
56	$e_1[e_2]$	$e_1[e_2]$	e_1 is a list and e_2 is an integer (0 scale number)
56	$e_1[e_2]$	e_1	e_1 is not a list and not null and value (e_2) = 1
56	$e_1[e_2]$	list of items e such that i is in e iff i is in e_1 and FEEL (e_2, s') = 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 (FEEL (e_2, s')) is boolean
56	$e_1[e_2]$	[e_1] if FEEL (e_2, s') = 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 (FEEL (e_2, s')) is boolean

Grammar Rule	FEEL Syntax	Mapped to Domain (scope s)
59	$\{n_1 : e_1, n_2 : e_2, \dots\}$	{ " n1 ": FEEL (e_1, s_1), " n2 ": 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_j , then c_j does not contain the entry for n_i .
	$\{ "n_1" : e_1, "n_2" : e_2, \dots \}$	
56	$[e_1, e_2, \dots]$	[FEEL (e_1), FEEL (e_2), ...]

10.3.2.12 Error Handling

When a built-in function encounters input that is outside its defined domain, the function **should** report or log diagnostic information if appropriate, and **must** return **null**.

10.3.3 XML Data

FEEL supports XML Data in the FEEL context by mapping XML Data into the FEEL Semantic Domain. Let $XE(e, \mathbf{p})$ be a function mapping an XML element e and a parent FEEL context \mathbf{p} to a FEEL context, as defined in the following tables. XE makes use of another mapping function, $XV(v)$, 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 $xmlns:p1="http://example.org/foobar"$ and $xmlns:p2="http://example.org/foobar"$, the tags $p1:myElement$ and $p2:myElement$ 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)

Herein, 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 \mathbf{p} is initially empty.

XML	context entry in \mathbf{p}	Remark
$\langle e \rangle$	"e" : null	empty element \rightarrow null-valued entry in \mathbf{p}
$\langle q:e \rangle$	"q\$e" : null	namespaces are ignored. Colonized names are changed to legal identifiers.
$\langle e \rangle v \langle /e \rangle$	"e": $XV(v)$	unrepeated element without attributes
$\langle e \rangle v_1 \langle /e \rangle \langle e \rangle v_2 \langle /e \rangle$	"e": [$XV(v_1), XV(v_2)$]	repeating element without attributes
$\langle e \ a="v" \rangle$ $\langle c_1 \rangle v_1 \langle /c_1 \rangle$ $\langle c_n \rangle v_2 \langle /c_n \rangle \langle c_n \rangle v_3 \langle /c_n \rangle$ $\langle /e \rangle$	"e": { "@a": $XV(v)$, "c ₁ ": $XV(v_1)$, "cn": [$XV(v_2), XV(v_3)$] }	attribute names are prefixed with @. An element containing attributes or child elements \rightarrow context
$\langle e \ a="v_1" \rangle v_2 \langle /e \rangle$	"e": { "@a": $XV(v_1)$, "\$content": $XV(v_2)$ }	v_2 is contained in a generated \$content entry

An entry in the FEEL Domain mapping column such as "e" : null indicates a context entry with string key "e" and value null. The context entries are contained by context \mathbf{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. The following table defines how a typed XML value v is mapped to FEEL.

Type of v	FEEL Semantic Domain
number	FEEL(v)
string	FEEL("v")
date	"@a": FEEL(<i>date</i> ("v"))
dateTime	"@a": FEEL(<i>date and time</i> ("v"))
time	"@a": FEEL(<i>time</i> ("v"))
duration	"@a": FEEL(<i>duration</i> ("v"))
list, e.g. "v ₁ v ₂ "	[XV(v ₁), XV(v ₂)]
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>
```



```

    </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 context

```

{
  Context: {
    tns$Employee: { tns$salary: 13,000 },
    Customer: [
      { loyalty_level: "gold", credit_limit: 10000 },
      { loyalty_level: "gold", credit_limit: 20000 },
      { loyalty_level: "silver", credit_limit: 5000 } ]
  }
}

```

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.

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 the following table. 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**.

A superscript means additional constraints on the domain are described later. Whenever a parameter is outside its domain, the result of the built-in is **null**.

Name(parameters)	Parameter Domain	Description	Example
<code>date(<i>from</i>)</code>	string ¹	convert <i>from</i> to a date	<code>date("2012-12-25") - date("2012-12-24") = duration("P1D")</code>
<code>date(<i>from</i>)</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 and time(<i>from</i>)</code>	string ²	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(<i>from</i>)</code>	string ³	convert <i>from</i> to time	<code>time("23:59:00") + duration("PT2M") = time("00:01:00")</code>
<code>time(<i>from</i>)</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>decimal(<i>from</i>, <i>scale</i>)</code>	number, number ⁴	set the scale of <i>from</i>	<code>number(1/3, 2) = .33</code>
<code>number(<i>from</i>, <i>grouping separator</i>, <i>decimal separator</i>)</code>	all 3 are string ⁵	convert <i>from</i> to a number	<code>number("1 000,0", " ", ";") = number("1,000.0", ",", ".")</code>
<code>string(<i>from</i>)</code>	non-null	convert <i>from</i> to a string	<code>string(1.1) = "1.1"</code> <code>string(null) = null</code>
<code>duration(<i>from</i>)</code>	string ⁶	convert <i>from</i> to a days and time or years and months duration	<code>date and time("2012-12-24T23:59:00") - date and time("2012-12-22T03:45:00") = duration("P2DT20H14M")</code>

			<code>duration("P2Y2M") = duration("P26M")</code>
years and months <code>duration(<i>from</i>, <i>to</i>)</code>	both are date and time	return duration between <i>from</i> and <i>to</i>	<code>years and months duration(date("2011-12-22"), date("2013-08-24")) = duration("P1Y8M")</code>

Additional constraints on input domains:

1. *from* must be in the lexical space of *xs:date* [ref xml schema]
2. *from* must be in the lexical space of *xs:dateTime* [ref xml schema]
3. *from* must be in the lexical space of *xs:time* [ref xml schema]
4. scale is in the range [-6111..6176]
5. grouping separator is space (' '), comma (','), period ('.'), or null. Decimal separator is period, comma, or null, but not the same as the grouping separator (unless null). *from* must have syntax specified by grammar rule 37, after removing grouping separators (if not null) and after changing the decimal separator (if not null) to a period.
6. *from* must be in the lexical space of the *xs:dayTimeDuration* or *xs:yearMonthDuration* [ref xpath]

Issues:

1. Extended time zone, e.g. 23:59:00@America/Los_Angeles

10.3.4.2 Boolean function

<code>not(<i>negand</i>)</code>	boolean	logical negation	<code>not(true) = false</code> <code>not(null) = null</code>
---------------------------------	---------	------------------	---

10.3.4.3 String functions

<code>substring(<i>string</i>, <i>start position</i>, <i>length?</i>)</code>	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	<code>substring("foobar",3) = "obar"</code> <code>substring("foobar",3,3) = "oba"</code> <code>substring("foobar", -2, 1) = "a"</code>
<code>string length(<i>string</i>)</code>	string	return length of <i>string</i>	<code>string length("foo") = 3</code>
<code>upper case(<i>string</i>)</code>	string	return uppercased <i>string</i>	<code>upper case("aBc4") = "ABC4"</code>
<code>lower case(<i>string</i>)</code>	string	return lowercased <i>string</i>	<code>lower case("aBc4") = "abc4"</code>
<code>substring before (<i>string</i>, <i>match</i>)</code>	string, string	return substring of <i>string</i> before the <i>match</i> in <i>string</i>	<code>substring before("foobar", "bar") = "foo"</code> <code>substring before("foobar", "xyz") = ""</code>
<code>substring after (<i>string</i>, <i>match</i>)</code>	string, string	return substring of <i>string</i> after the <i>match</i> in <i>string</i>	<code>substring after("foobar", "ob") = "ar"</code> <code>substring after("", "a") = ""</code>

<code>replace(input, pattern, replacement, flags?)</code>	string ²	regular expression pattern matching and replacement	<code>replace("abcd", "(ab) (a)", "[1=\$1][2=\$2]") = "[1=ab][2=]cd"</code>
<code>contains(string, match)</code>	string	does the <i>string</i> contain the <i>match</i> ?	<code>contains("foobar", "of") = false</code>
<code>starts with(string, match)</code>	string	does the <i>string</i> start with the <i>match</i> ?	<code>starts with("foobar", "fo") = true</code>
<code>ends with(string, match)</code>	string	does the <i>string</i> end with the <i>match</i> ?	<code>ends with("foobar", "r") = true</code>
<code>matches(input, pattern, flags?)</code>	string ²	does the <i>input</i> match the regexp <i>pattern</i> ?	<code>matches("foobar", "^fo*b") = true</code>

Additional constraints on input domains:

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* if *start position* is positive, and –*start position* otherwise.
2. *flags* must be ...

10.3.4.4 List functions

<code>list contains(list, element)</code>	list, any element of the semantic domain including null	does the <i>list</i> contain the <i>element</i> ?	<code>list contains([1,2,3], 2) = true</code>
<code>count(list)</code>	list	return size of <i>list</i>	<code>count([1,2,3]) = 3</code>
<code>min(list)</code> <code>max(list)</code>	list of comparable items	return minimum(maximum) item	<code>min([1,2,3]) = 1</code> <code>max([1,2,3]) = 3</code>
<code>sum(list)</code>	list of numbers	return sum of <i>list</i> items	<code>sum([1,2,3]) = 6</code>
<code>mean(list)</code>	list of numbers	return arithmetic mean (average) of <i>list</i> items	<code>mean([1,2,3]) = 2</code>
<code>and(list)</code>	list	return conjunction of <i>list</i> items	<code>and([true,null,true]) = false</code> <code>and([]) = true</code> <code>and(0) = false</code>
<code>or(list)</code>	list	return disjunction of <i>list</i> items	<code>or([false,null,true]) = true</code> <code>or([]) = true</code> <code>or(0) = false</code>
<code>sublist(list, start position, length?)</code>	list, number ¹ , number ²	return list of <i>length</i> (or all) elements of <i>list</i> , starting with	<code>sublist([1,2,3], 1, 2) = [2]</code>

		<i>list</i> [<i>start position</i>]. 1 st position is 1, last position is -1	
<i>append(list, 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]
<i>concatenate(list...)</i>	list	return new <i>list</i> that is a concatenation of the arguments	<i>concatenate</i> ([1,2],[3]) = [1,2,3]
<i>insert before(list, position, 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) = [1,2,3]
<i>remove(list, position)</i>	list, number ¹	<i>list</i> with item at <i>position</i> removed	<i>remove</i> ([1,2,3], 2) = [1,3]
<i>reverse(list)</i>	list	reverse the <i>list</i>	<i>reverse</i> ([1,2,3]) = [3,2,1]
<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</i> ([1,2,3,2],2) = [2,4]
<i>union(list...)</i>	list	concatenate with duplicate removal	<i>union</i> ([1,2],[2,3]) = [1,2,3]
<i>distinct values(list)</i>	list	duplicate removal	<i>distinct values</i> ([1,2,3,2,1] = [1,2,3]
<i>flatten(list)</i>	list	flatten nested lists	<i>flatten</i> ([[1,2],[[3]], 4]) = [1,2,3,4]

Additional constraints on input domains:

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* if *start position* is positive, and –*start position* otherwise.

10.3.4.5 Decision Table

The normative notation for decision tables is specified in clause 8. A textual representation as a builtin function is provided here in order to link the semantics with the syntax in the same way as is done for the rest of FEEL. Note that the parameters correspond to classes and associations in the metamodel in clause 8.3.

Parameters (* means optional):

Name	Domain
input expressions	a list of N>0 expressions, each legal as the LHS of an <i>in</i> expression
input values*	list of N unary test (15) lists. A unary test restricts the associated input to a particular value or range of values.
outputs*	a name (27) or list of M>0 names
output values*	if outputs is a list, a list of lists of values, one list per output; else a list of values for the one output
rules	a list of 0 or more rules. A rule is a list of N input entries followed by M

	output entries. An input entry is a list or a unary test (15). The list contains 0 or more unary test.
hit policy*	"Unique", "Any", "Priority", "First", "NoOrder", "RuleOrder", "OutputOrder" (default is "Unique")
completeness*	"C", "I" (default is "C")
aggregation*	"collect", "sum", "min", "max", "mean", or "count"

An input entry containing the empty list, corresponding to a “-“ in the notation, matches any input.

The decision table function evaluates a decision table that is specified using N input expressions, M outputs, and R rules. Each rule has N+M entries.

The decision table 'input expressions' parameter is a list of N expressions. The decision table 'rules' parameter is a list of R rules.

Each rule is a list containing N input entries, one per input, and M output entries.

A rule is *matched* if all its *FEEL equivalent tests* are true. *Input entry i* in rule *j* is equivalent to one of the following FEEL expressions:

1. If rules[j][i] is positive unary tests (grammar rule 16), the FEEL equivalent test is inputs[i] in rules[j][i].
2. If rules[j][i] is unary tests of the form not(x), where x is positive unary tests, the FEEL equivalent test is not(inputs[i] in x).
3. If rules[j][i] is unary tests of the form '-', the FEEL equivalent test is true.

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 result in an output list.

The hit policy is specified as one of the following:

- **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,
- **No Order** –return a list of outputs 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

To reduce complexity, decision tables with compound outputs support only the following hit policies: Unique, Any, First, No order, and Rule order.

Completeness is specified using the initial letter of one of the following:

- **Complete** – user asserts all input combinations (rule test values) are accounted for. Tools can challenge this assertion. If the assertion is wrong at runtime, the decision table function will return **null**.

- **Incomplete** – table is incomplete. Return first element of *output values* list (or null if *output values* is not supplied) if no rules match.

Completeness can be checked as follows. The *input entries* must cover all combinations of the input values. If an *input expression* has no corresponding *input values*, then the input values are inferred from the *input entries* as follows: equality tests add the test value to the input values. Inequality tests require that the input entries must cover all non-null values of that input, by using range expressions such as <0 , $[0..100]$, etc.

A decision table may have no rule hit for a set of input values. In this case, the result is **null**.

The value of the Completeness flag is ignored for execution, but should be used for design-time validation.

The semantics of a decision table are as follows:

1. Rules are tested in arbitrary order, except for first-hit tables, where rules are tested in rule order.
2. If no rules are matched, *null* is returned
3. Let *m* be the set of rules matched
4. If No Order, return the list of outputs, one for each *m*
5. if Unique or Any return the output for *m*[1]
6. if Rule Order, order *m* by rule index and return the list of outputs, one for each rule in *m*
7. else if Output Order, order *m* by *output values* and return the list of outputs, one for each rule in *m*.
If multiple outputs, order each from left to right or top to bottom.

10.3.4.6 Sort

Sort a list using an ordering function.

Parameters (* means optional):

Name	Domain
list	list of any element, be careful with nulls
precedes	boolean function of 2 arguments defined on every pair of list elements

For example,

sort(list: [3,1,4,5,2], precedes: function(x,y) $x < y$) = [1,2,3,4,5]

10.4 Relationship of FEEL to DRG and Boxed Expressions

FEEL is used to give execution semantics to Decision Models, that is, to a DRG and its associated Boxed Expressions. Many DRG Elements may have an associated expression, in the form of a boxed expression as described in clause 10.2.1. Usage of expressions by DRG Elements is summarized in the following table.

DRG Element	Associated Boxed Expression
-------------	-----------------------------

Business Knowledge Model	function (parameterized decision logic)
Decision	invocation or any other decision logic
Input Data	sample data

Boxed expressions contain other boxed expressions or FEEL expressions described in 10.2.2. FEEL expressions are given meaning by the FEEL Semantics also described in 10.2.2. Boxed expressions are given meaning by mapping them to FEEL expressions. DRGs can also be given meaning (execution semantics) by mapping them to FEEL expressions.

Let D be a DRG with elements d_1, d_2, \dots . Each element has a name n and a decision logic expression e . The FEEL expression for D is the context $\{n_1:e_1, n_2:e_2, \dots\}$, such that:

- the context entries are partially ordered by requirements (*e.g.* the entry for an Input Data element comes before a Decision that uses it)
- the context entries are partially ordered by name reference (referent first)

In other words, the boxed expressions of some DRG elements refer to other DRG elements by name, and each of those relationships must be reflected as a requirement.

10.5 Metamodel

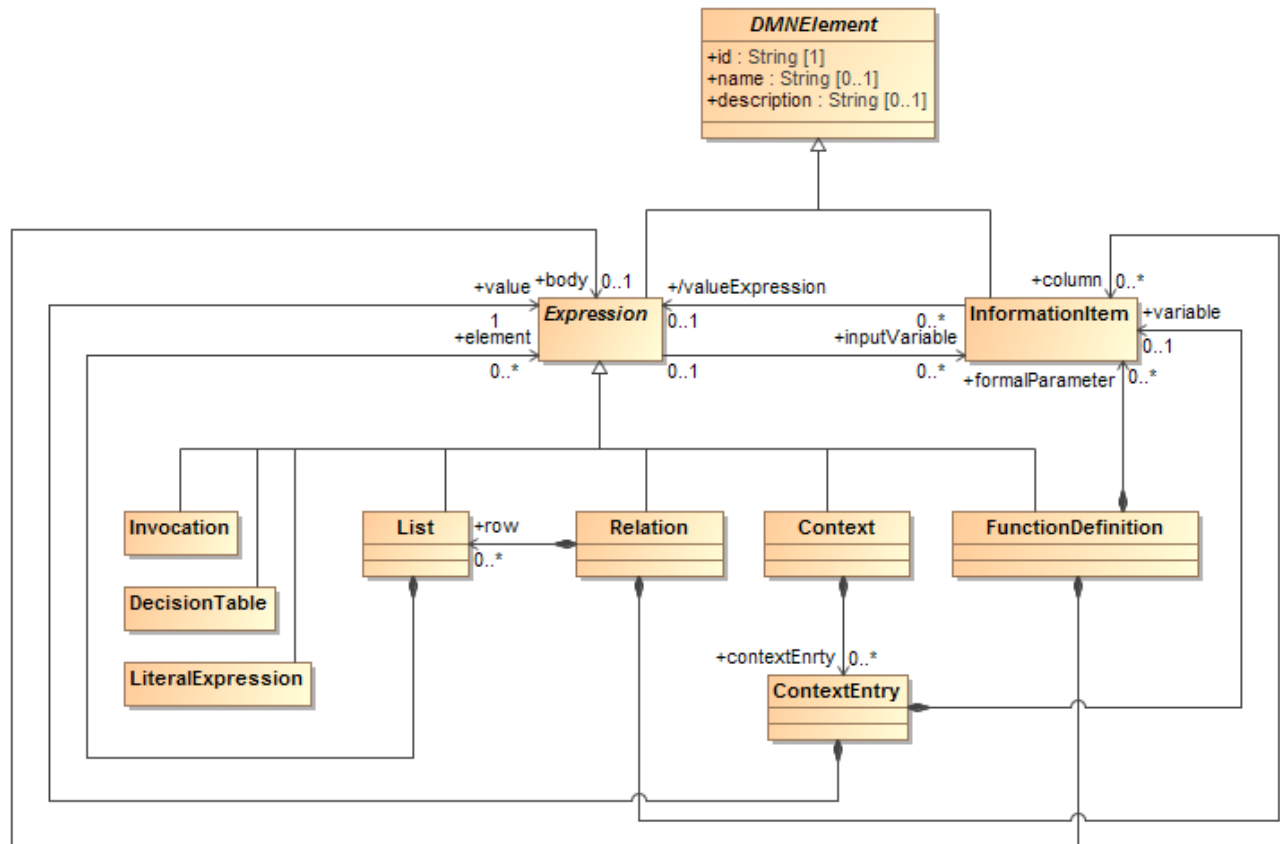


Figure 52: Expression class diagram

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 in DMN (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 59 and clause 10.3.2.5) SHOULD be modeled as a `Context` element whenever possible.

Context inherits all the attributes and model associations from Expression. Table 29 presents the additional attributes and model associations of the Context element.

Table 29: Context attributes and model association

Attribute	Description
contextEntry: ContextEntry [*]	This attributes lists the instances of ContextEntry that compose this Context.

10.5.2 ContextEntry metamodel

The class ContextEntry is used to model FEEL *context entries* when a *context* is modeled as a Context element.

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 30 presents the attributes and model associations of the ContextEntry element.

Table 30: ContextEntry attributes and model associations

Attribute	Description
variable: InformationItem [0..1]	The instance of InformationItem that is contained in this ContextEntry, and whose name is the <i>key</i> in the modeled <i>context entry</i>
value: Expression	The instance of Expression that is the <i>expression</i> in this ContextEntry

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 0. A FEEL *function definition* (grammar rule 57 and clause 10.3.2.11) SHOULD be modeled as a FunctionDefinition element whenever possible.

FunctionDefinition inherits all the attributes and model associations from Expression. Table 31 presents the additional attributes and model associations of the FunctionDefinition element.

Table 31: 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

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 56 and clause 10.3.2.11) SHOULD be modeled as a List element whenever possible.

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

Table 32: 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 ContextEntrys, 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 33 presents the additional attributes and model associations of the Relation element.

Table 33: 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.

10.6 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 53 shows the boxed context used as an instance of input data for the examples.

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)		
	$(\text{amount} * \text{rate} / 12) / (1 - (1 + \text{rate} / 12)^{-\text{term}})$		

Figure 53: 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
```

10.6.3 If, In

```
if applicant.maritalStatus in ["M","S"] then "valid" else "not valid"
```

```
"valid"
```

10.6.4 Sum entries of a list

```
sum(monthly outgoings)
```

```
5500
```

10.6.5 Invocation of user-defined PMT function

```
PMT(requested product . rate,  
     requested product . term,  
     requested product . amount)
```

```
3975.982590125562
```

10.6.6 Sum weights of recent credit history

```
sum(credit history[record date > date("2011-01-01")].weight)
```

```
150
```

10.6.7 Determine if credit history contain a bankruptcy event

```
some ch in credit history satisfies ch.event = "bankruptcy"
```

```
false
```

11 DMN Example

In this section we present an example of the use of DMN to model the decision-making to be automated in decision services called from a business process management system modeled in BPMN.

11.1 The business process model

Figure 54 shows a simple process for loan originations, modeled in BPMN 2.0. The process handles applications for a loan, obtaining data from a credit bureau only if required for the case, and deciding whether the application should be accepted, declined, or referred for human review. 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 **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 Application 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 Application routing.
- The **Application 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 **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 decision 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 the two decision points (calls to decision services) are represented in BPMN 2.0 as business rule tasks.

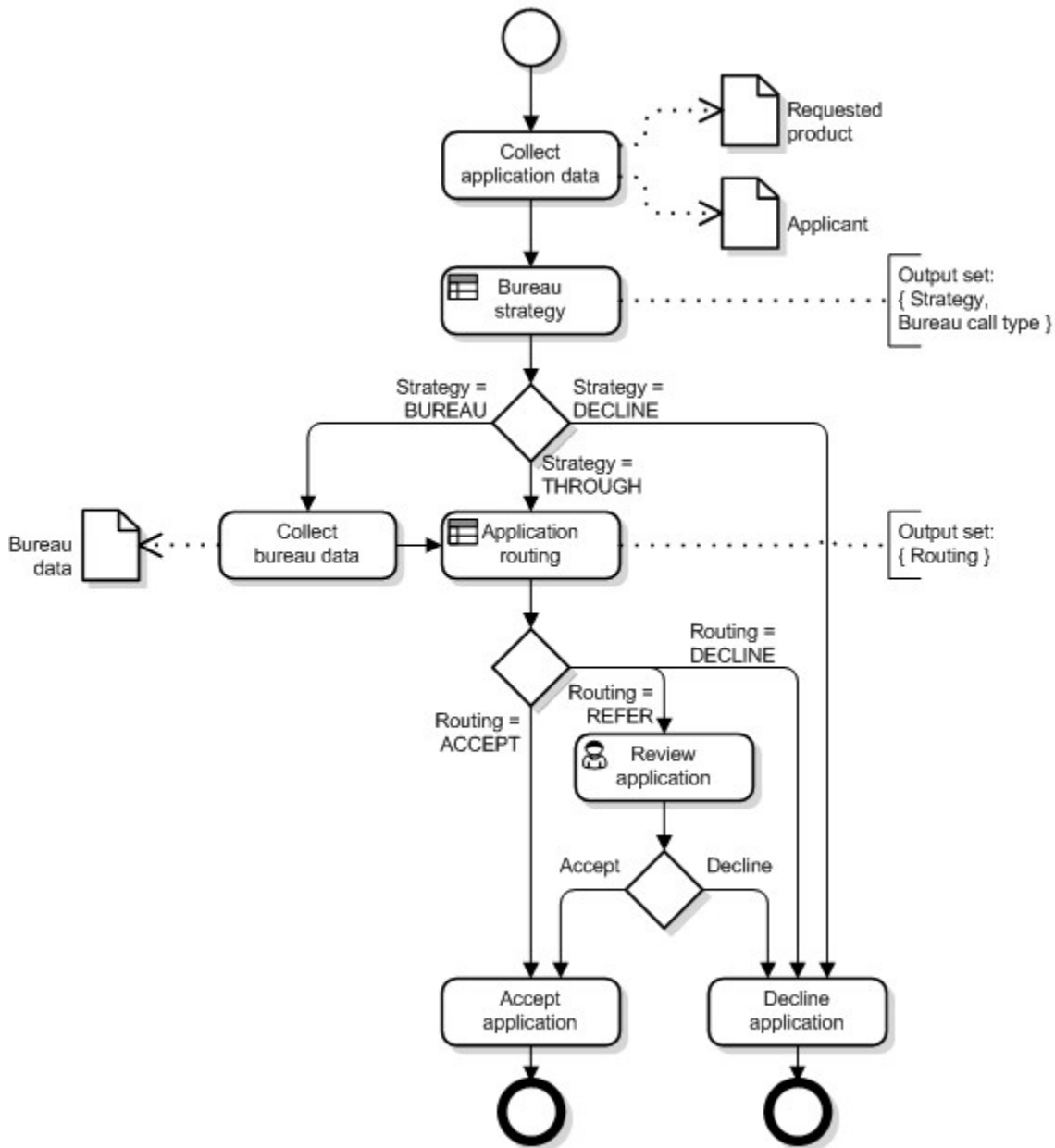


Figure 54: Example business process

11.2 The decision requirements level

Figure 55 shows a DRD of all the decisions to be automated in this business process. There are three data inputs to be provided to the decision services (Requested product, Applicant data and Bureau data), and three decisions to be returned by the decision services (Strategy, Bureau call type and Routing). Between the two are intermediate decisions: evaluations of risk, affordability and eligibility. Note that the two services share some common elements: e.g. the Required Monthly Installment decision and the Affordability calculation business knowledge model are required for both Strategy and Routing. Other notable features of this DRD include:

- 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.

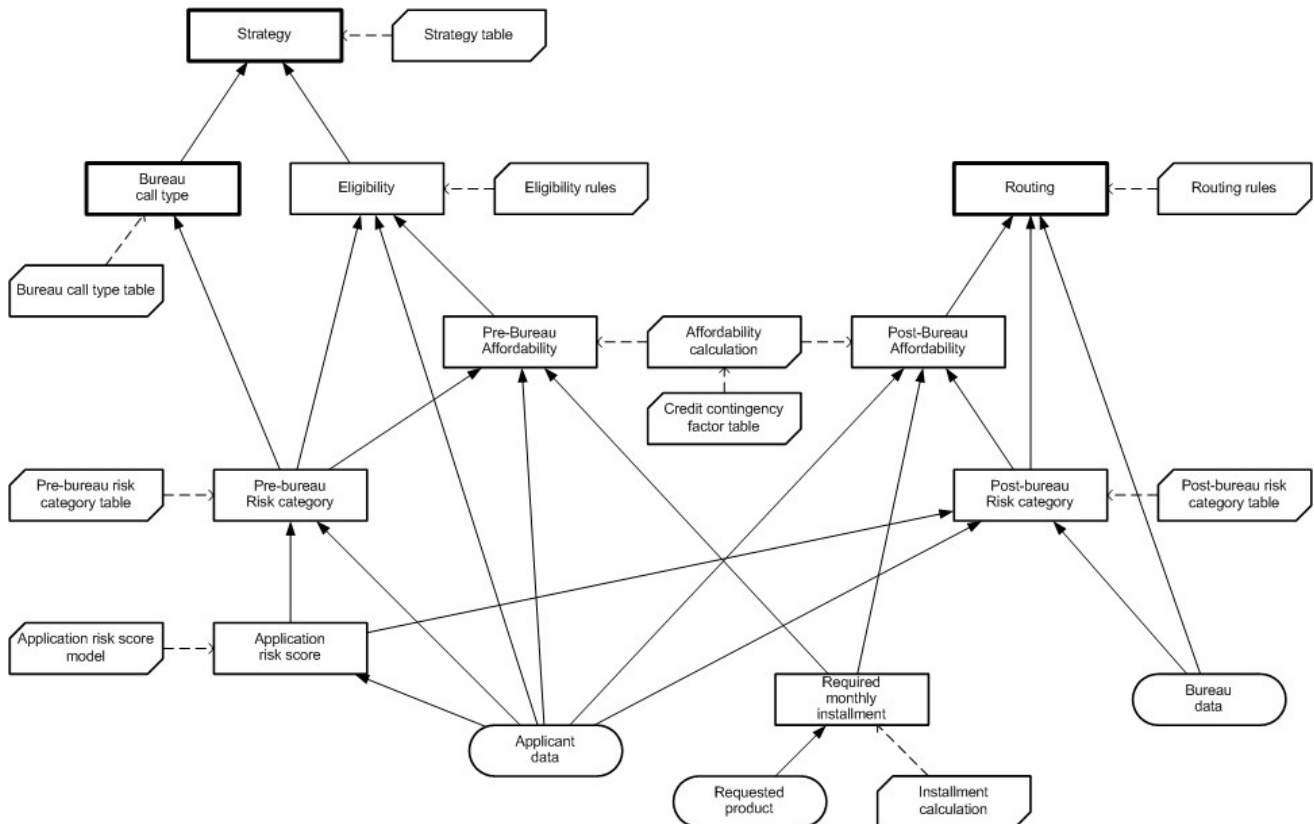


Figure 55: DRD of all automated decision-making

It might be considered more convenient to draw separate (but overlapping) DRDs for the two decision points. Figure 56 shows the DRD of the decisions required for the Bureau Strategy decision point (i.e. the requirements subgraph of the Strategy and Bureau Call Type decisions), and Figure 57 shows the DRD for the Application Routing decision point (i.e. the requirements subgraph of the Routing decision). All three DRDs – Figure 55, Figure 56 and Figure 57 – are views of the same DRG.

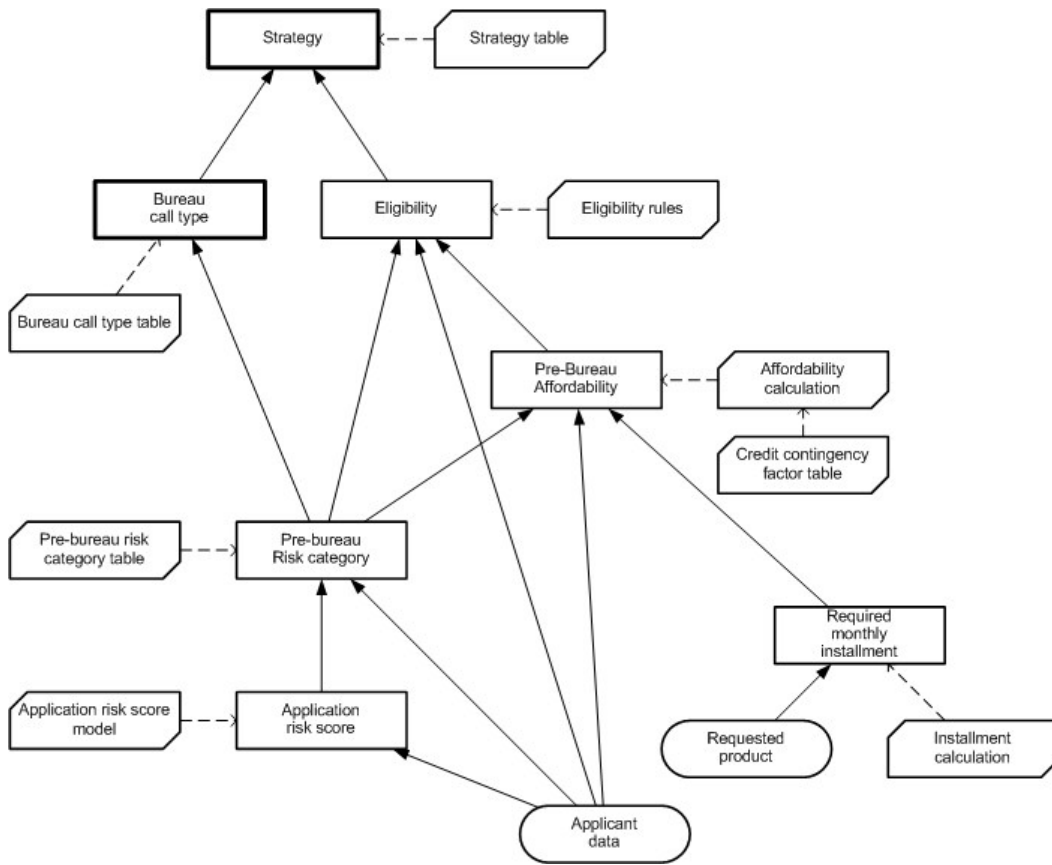


Figure 56: DRD for Bureau Strategy decision point

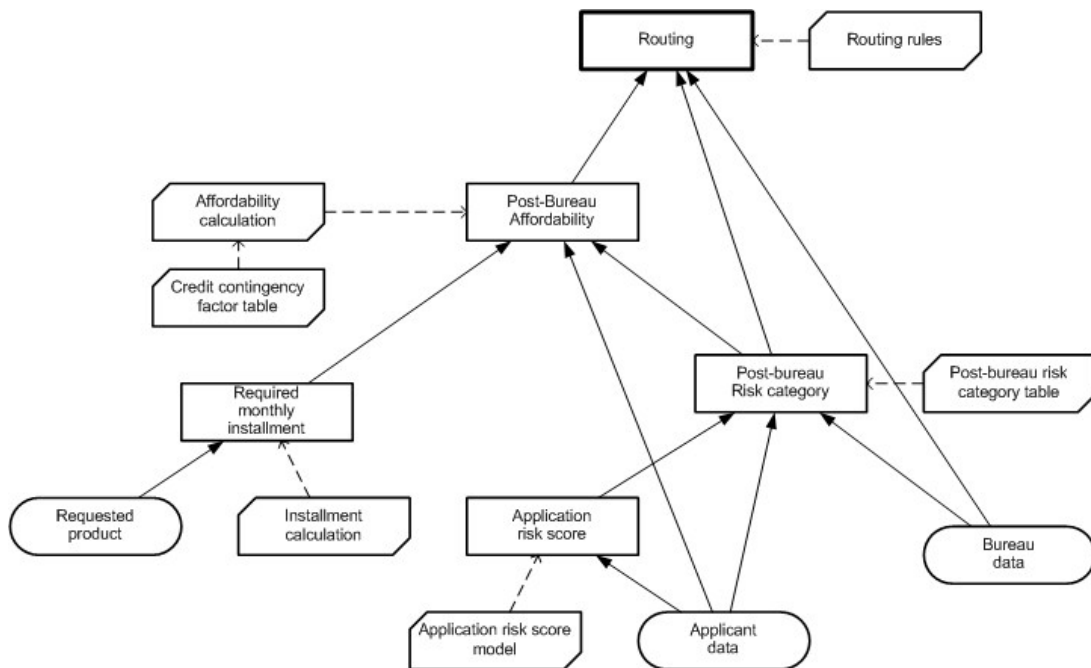


Figure 57: DRD for Application Routing decision point

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 59
- The **Bureau call type** decision, requiring the Pre-bureau risk category decision, invokes the Bureau call type table shown in Figure 61
- The **Eligibility** decision, requiring Applicant data and the Pre-bureau risk category and Pre-bureau affordability decisions, invokes the Eligibility rules shown in Figure 63
- 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 74, which in turn invokes the Credit contingency factor table shown in Figure 75
- 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 65
- The **Application risk score** decision, requiring Applicant data, invokes the score model shown in Figure 67
- The **Routing** decision, requiring Bureau data and the Post-bureau affordability and Post-bureau risk category decisions, invokes the Routing rules shown in Figure 69
- 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 74, which in turn invokes the Credit contingency factor table shown in Figure 75
- 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 71.
- The **Required monthly installment** decision, requiring Requested product data, invokes the Installment calculation boxed expression shown in Figure 77.

11.3 The decision logic level

The DRG in Figure 55 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 58) invokes the Strategy table business knowledge model, passing the output of the Bureau call type decision as the Bureau Call Type parameter, and the output of the Eligibility decision as the Eligibility parameter.
- The **Strategy Table** decision logic (Figure 59) defines a complete, unique-hit decision table deriving Strategy from Eligibility and Bureau Call Type.
- The **Bureau Call Type** decision logic (Figure 60) 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 61) defines a complete, unique-hit decision table deriving Bureau Call Type from Pre-Bureau Risk Category.
- The **Eligibility** decision logic (Figure 62) 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 63) 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 (Figure 64) 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 65) 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 (Figure 66) 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.
- The **Application Risk Score Model** decision logic (Figure 67) 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 (Figure 68) 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 69) 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 (Figure 70) 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 71) 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 (Figure 72) 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 (Figure 73) 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 74) defines a boxed context 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 75) defines a complete, unique-hit decision table deriving Credit contingency factor from Risk Category.
- The **Required monthly installment** decision logic (Figure 76) 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 77) defines a boxed context deriving monthly installment from Product Type, Rate, Term and Amount. One step in this calculation invokes the external function PMT.

Strategy	
Strategy table	
Bureau Call Type	Bureau Call Type
Eligibility	Eligibility

Figure 58: Strategy decision logic

Strategy table			
UC	Eligibility	Bureau Call Type	Strategy
1	<i>INELIGIBLE</i>	-	<i>DECLINE</i>
2	<i>ELIGIBLE</i>	<i>FULL, MINI</i>	<i>BUREAU</i>
3		<i>NONE</i>	<i>THROUGH</i>

Figure 59: Strategy table decision logic

Bureau call type	
Bureau call type table	
Pre-Bureau Risk Category	Pre-Bureau Risk Category

Figure 60: Bureau Call Type decision logic

Bureau call type table		
UC	Pre-Bureau Risk Category	Bureau Call Type
1	<i>HIGH, MEDIUM</i>	<i>FULL</i>
2	<i>LOW</i>	<i>MINI</i>
3	<i>VERY LOW, DECLINE</i>	<i>NONE</i>

Figure 61: 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 62: Eligibility decision logic

Eligibility rules				
PC	Pre-Bureau Risk Category	Pre-Bureau Affordability	Age	Eligibility
				<i>INELIGIBLE, ELIGIBLE</i>
1	<i>DECLINE</i>	-	-	<i>INELIGIBLE</i>
2	-	false	-	<i>INELIGIBLE</i>
3	-	-	< 18	<i>INELIGIBLE</i>
4	-	-	-	<i>ELIGIBLE</i>

Figure 63: 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 64: Pre-Bureau Risk Category decision logic

Pre-bureau risk category table			
UC	Existing Customer	Application Risk Score	Pre-Bureau Risk Category
1	true	< 100	<i>HIGH</i>
2		[100..120[<i>MEDIUM</i>
3		[120..130]	<i>LOW</i>
4		> 130	<i>VERY LOW</i>
5	false	< 80	<i>DECLINE</i>
6		[80..90]	<i>HIGH</i>
7		[90..110]	<i>MEDIUM</i>
8		> 110	<i>LOW</i>

Figure 65: 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 66: Application Risk Score decision logic

Application risk score model				
NC	Age	Marital Status	Employment Status	Partial score
	[18..120]	<i>S, M</i>	<i>UNEMPLOYED, EMPLOYED, SELF-EMPLOYED, STUDENT</i>	
1	[18..21]	-	-	32
2	[22..25]	-	-	35
3	[26..35]	-	-	40
4	[36..49]	-	-	43
5	>=50	-	-	48
6	-	<i>S</i>	-	25
7	-	<i>M</i>	-	45
8	-	-	<i>UNEMPLOYED</i>	15
9	-	-	<i>STUDENT</i>	18
10	-	-	<i>EMPLOYED</i>	45
11	-	-	<i>SELF-EMPLOYED</i>	36
Aggregation = sum				

Figure 67: 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 68: Routing decision logic

Routing rules					
PC	Post-Bureau Risk Category	Post-Bureau Affordability	Bankrupt	Credit Score	Routing
					<i>DECLINE, REFER, ACCEPT</i>
1	-	false	-	-	<i>DECLINE</i>
2	-	-	true	-	<i>DECLINE</i>
3	<i>HIGH</i>	-	-	-	<i>REFER</i>
4	-	-	-	< 580	<i>REFER</i>
5	-	-	-	-	<i>ACCEPT</i>

Figure 69: 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 70: Post-Bureau Risk Category decision logic

Post-bureau risk category table				
UC	Existing Customer	Application Risk Score	Credit Score	Post-Bureau Risk Category
1	true	< 120	< 590	<i>HIGH</i>
2			[590..610]	<i>MEDIUM</i>
3			> 610	<i>LOW</i>
4		[120..130]	< 600	<i>HIGH</i>
5			[600..625]	<i>MEDIUM</i>
6			> 625	<i>LOW</i>
7			> 130	-
8	false	<= 100	< 580	<i>HIGH</i>
9			[580..600]	<i>MEDIUM</i>
10			> 600	<i>LOW</i>
11		< 590	<i>HIGH</i>	
12		[590..615]	<i>MEDIUM</i>	
13		> 100	> 615	<i>LOW</i>

Figure 71: 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 72: 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 73: Post-Bureau Affordability decision logic

Affordability calculation	
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 74: Affordability calculation decision logic

Credit contingency factor table		
UC	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 75: 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 76: Required Monthly Installment decision logic

Installment calculation	
Monthly Fee	if Product Type = "STANDARD LOAN" then 20.00 else if Product Type = "SPECIAL LOAN" then 25.00 else null
Monthly Repayment	PMT(Rate, Term, Amount)
Monthly Repayment + Monthly Fee	

Figure 77: Installment calculation decision logic

Finally, the input data elements may be modeled with some concrete sample data. This allows the decision model to be thoroughly validated and even executed. Figure 78, Figure 79 and Figure 80 show boxed contexts defining instance values for Applicant data, Requested product and Bureau data.

Applicant data		
Age	51	
MaritalStatus	<i>M</i>	
EmploymentStatus	<i>EMPLOYED</i>	
ExistingCustomer	false	
Monthly	Income	10,000.00
	Repayments	2,500.00
	Expenses	3,000.00

Figure 78: Applicant Data input data sample

Requested product	
ProductType	<i>STANDARD LOAN</i>
Rate	0.08
Term	36
Amount	100,000.00

Figure 79: Requested Product input data sample

Bureau data	
Bankrupt	false
CreditScore	600

Figure 80: Bureau Data input data sample

12 Exchange formats

12.1 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.

12.2 Machine Readable Files

All DMN 1.0 machine-readable files, including XSD and XMI files, can be found in OMG Document bmi/2013-08-05, which is a flat zip file.

- For the DMN XMI Model, the main file is DMN10.xmi.
- For the DMN XSD Interchange Part 1 (the interchange definition for Conformance Levels 1 and 2), the main file is DMN10.xsd.
- For the DMN XSD Interchange Part 2 (the interchange definition for Conformance Level 3), the main file is DMN10Level3.xsd.

12.3 XSD

12.3.1 Document Structure

A domain-specific set of model elements is interchanged in one or more DMN files. The root element of each file **MUST** be `<dmn:Definitions>`. The set of files **MUST** be self-contained, i.e. all definitions that are used in a file **MUST** be imported directly or indirectly using the `<dmn:Import>` element.

Each file **MUST** declare a "namespace" 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.

12.3.2 References within the DMN XSD

All the DMN elements that may need 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. The DMN XSD supports referencing by ID, across files, by utilizing QName.

A QName consists of two parts: an optional namespace prefix and a local part. When used to reference a DMN element, the local part is expected to be the ID of the element, and the namespace is expected to be the namespace of the containing Definitions element.

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 from another file, the reference would take the following form:

```
<requiredDecision>decision_ns:prebureauriskDec01</requiredDecision>
```

where “decision_ns” is the namespace prefix associated upon import with the namespace in which the “Pre-Bureau Risk Category” Decision is defined, and “prebureauriskDec01” is the value of the `id` attribute for the Decision.

Notice that the reference can also take the following form:

```
<requiredDecision> prebureauriskDec01</requiredDecision>
```

where the prefix is omitted, if the required decision (in this case Pre-Bureau Risk Category) is defined in the default namespace (e.g. because it is defined in the same instance of Definitions – that is, in the same file – as the requiring decision).

The DMN XSD utilizes IDREFs wherever possible and resorts to QName only when references can span files. In both situations however, the reference is still based on IDs.

ANNEXES

All the Annexes are informative.

Annex A and Annex B discuss issues around the application of DMN in combination with BPMN, including the use of DMN to define the functionality of decision services to be called from tasks defined in BPMN. These sections are intended to provide some direction to practitioners but are non-normative.

Error! Reference source not found. provides cross-reference tables relating the requirements as stated in the RFP to the corresponding responses in this submission.

Annex C provides a non-normative glossary to aid comprehension of the specification.

Annex A. Relation to BPMN

(Informative)

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 business people 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 business-oriented 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.

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 81 shows an example¹ of a BPMN-defined process.

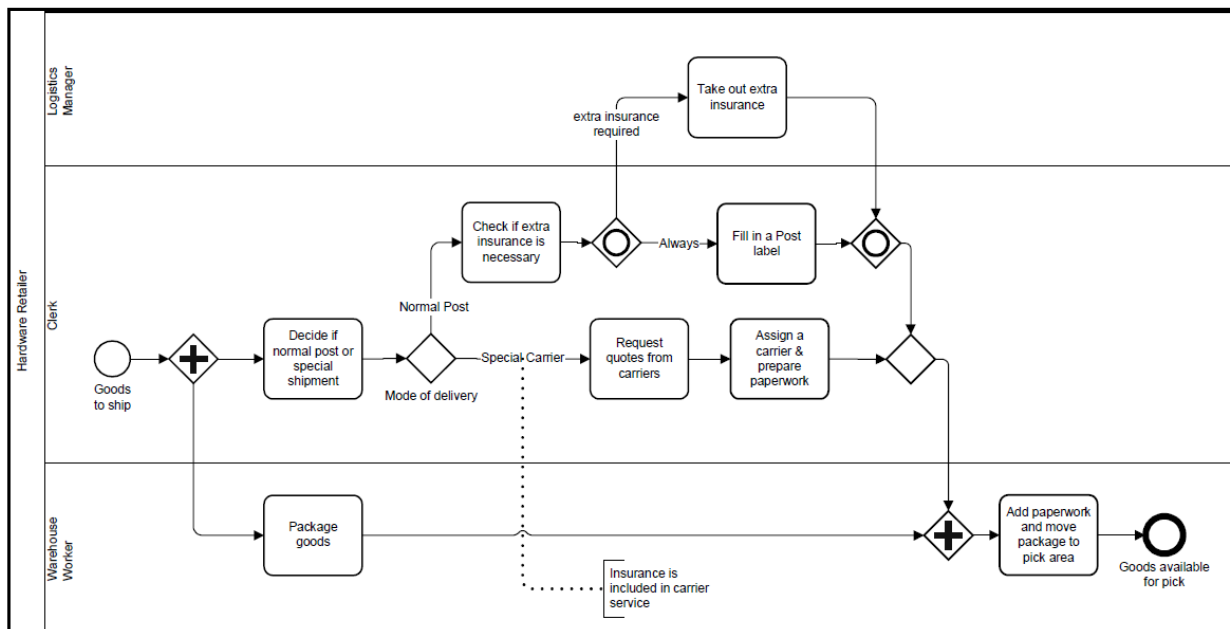


Figure 81: Decision-making in BPMN

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.

¹ Shipment Process in a Hardware Retailer example, Ch5.1, BPMN 2.0 By Example, June 2010, OMG reference 10-06-02

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 34:

Table 34: BPMN tasks relevant to DMN

	Task type(s)	Decision role
1	<p>Loop Multi-Instance Compensation</p> 	<p>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.</p>
2	<p>Service Task </p>	<p>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.</p>
3	<p>User Task </p>	<p>Decision tasks executed manually as a part of a workflow-oriented business process may be specified as a User Task.</p>
4	<p>Business Rule Task </p>	<p>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></p>
5	<p>Script Task </p>	<p>Decision tasks may today be encoded using business process script languages.</p>

² See ch 10.2.3 in the BPMN Specification.

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 5.3.2 and 6.3.6, a Decision in DMN can be associated with any Task, allowing for flexibility in implementation.

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.

5. Linking BPMN and DMN Models

DMN offers two approaches to linking business process models in BPMN with decision models in DMN; one normative and the other non-normative:

a) Associating Decisions with Tasks and Processes

As described in clause 6.3.6, in DMN 1.0, 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 BPMN2, and any number of `usingTasks`, which are instances of `Task` as defined in OMG BPMN2. 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 **MUST** 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 B: 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. Decision services

(Informative)

One important use of DMN will be to define decision-making logic to be automated using “decision services”. A decision service encapsulates a number of decisions and exposes them as a service, which might be consumed (for example) by a task in a BPMN process model. When the service is called, with the necessary input data, it returns the outputs of the encapsulated decisions. Any decision service encapsulating a DMN decision model will be stateless and have no side effects. It might be implemented, for example, as a web service. DMN does not specify how such services should be implemented; this section is to give an indication of the principles which might be followed.

We start with the assumption 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 “minimal output set”). This requires that the service encapsulate not just the minimal output set but also any decisions in the DRG directly or indirectly required by the minimal output set (the “encapsulation set”). The encapsulation set is the transitive closure of the required decision relation on the minimal output set.

The interface to the decision service will consist of:

- Input: a list of contexts, providing instances of all the Input Data required by the encapsulated decisions
- Output: a context, providing (at least) the results of evaluating all the decisions in the minimal output set, using the provided instance data.

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

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

For computational efficiency various improvements to this basic interpretation can be imagined, e.g.

- An optional input parameter specifying a list of “requested decisions” (a subset of the encapsulation 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 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 which are not already known.

Annex C. Glossary

(Informative)

A

Aggregation	The production of a single result from multiple hits on a decision table . DMN specifies six aggregation indicators, namely: <i>collect</i> , <i>sum</i> , <i>min</i> , <i>max</i> , <i>count</i> , <i>average</i> . The default is <i>collect</i> .
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 a decision or business knowledge model on a knowledge source which provides a source of authority for the decision logic.

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 sub-domains of the subject's domain that are
--------	--

	relevant for the piece of decision logic that is described by the decision table.
Completeness Indicator	Indicates whether a decision table produces a result for every possible case. If so (the default) the indicator “C” is optional. If not, the indicator should read “I”.
Context	In FEEL , a map of key-value pairs called context entries .
Context Entry	One key-value pair in a context .
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 two-dimensional 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 decision-making 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 modeling 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 or invocation 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.
Hit Policy	Indicates how overlapping decision rules have to be interpreted. A single hit table returns the output of one rule only; a multiple hit table may return the output of multiple rules or an aggregation of the outputs.
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 .
--------------	--

N

No 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 an arbitrary order.
----------	--

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.