Fact-Based Modelling

Metamodel (version WD08)

Exchanging Fact-Based Conceptual Data Models
VERSION
This document specifies Working Draft 8 (WD08) of a metamodel for fact-based modelling under development by the Fact-Based Modelling Working Group (FBM WG). Members of the FBM WG are technical experts and/or practitioners in fact-based modelling. Any reference to the content of this document should be properly cited.

REVIEW
This document is an official Draft for review by the FBM WG.

DISCLAIMER
As a draft, this document is subject to change without notice.

DISTRIBUTION
This document is primarily intended for use by the FBM WG. Distribution of the document outside this group requires permission from the FBM WG convenor.

CONTACT DETAILS
For further information, please contact:
  serge.valera@esa.int (FBM WG convenor)
  inge.lemmens@pna-group.nl (FBM WG secretariat)

COPYRIGHT
This document is copyrighted to the FBM Working Group, with the exception of Annex A, which is copyrighted to T. Halpin and reused here by permission.
## Contents

1 Introduction 1
   1.1 Document Purpose 1
   1.2 Overview of Fact-Based Modelling 1
   1.3 An Illustrative Example 2
   1.4 Conventions and Abbreviations 7
   1.5 Exchange-specific Aspects 7
   1.6 Document Structure 8

2 Metamodel: Core 10
   2.1 Core Elements 10
   2.2 ObjectType and ObjectTypeName 13
   2.3 Role Names 18
   2.4 FactType Arity 19
   2.5 FactType Predicates 21
   2.6 Predicate Readings 23
   2.7 FactType Readings 26
   2.8 World Assumption 27
   2.9 Role Classification 28
   2.10 Subtyping 30
   2.11 Domain Object Type: Identification 32
   2.12 Is Independent and Is Personal 34
   2.13 Objectification 36
   2.14 Derivation 37
   2.15 Data Type and Data Value 41
   2.16 Constraints 44
   2.17 Alethic Constraints 47
   2.18 Mandatory Role Constraints 48
   2.19 Exclusive-Or Constraints 50
   2.20 Uniqueness Constraints 51
   2.21 Frequency Constraints 53
   2.22 Ring Constraints 56
   2.23 Value-Comparison Constraints 59
   2.24 Set-Comparison Constraints 61
   2.25 Subset Constraints 64
   2.26 Equality Constraints 66
   2.27 Exclusion Constraints 67
   2.28 Cardinality Constraints 68
   2.29 Value Constraints 70
   2.30 General Constraints 73
   2.31 Join Lists 75
   2.32 Sample Populations 80
3 Metamodel: Fine Details 83

3.1 Role Names (Constrained) 83
3.2 Fact Type Names 85
3.3 Fact Type Predicates (Constrained) 86
3.4 Predicate Readings (Constrained) 87
3.5 Fact Type Readings (Derivations) 88
3.6 Predicate Reading Order 89
3.7 Subtyping: Type Consistency 91
3.8 Subtyping: Identity Fact Type 93
3.9 Subtyping: Compatibility 95
3.10 Independence and Implied Mandatory Role Constraints 97
3.11 Objectification: Link Fact Types 99
3.12 Facets 101
3.13 Facets (Constrained) 103
3.14 Scalar and Domain-specific Fact Types 104
3.15 Comparison Operations 105
3.16 Value Comparability 107
3.17 Ring Constraint Combinations 109
3.18 Join Path Specification 111
3.19 Join Lists (Advanced) 113
3.20 Join List Path Specification 114
3.21 Sample Populations (Constrained) 117
3.22 Sample Populations (Instance Compatibility) 119

4 Terms and Definitions 121

Annex A ORM Graphical Notation 131
Annex B References and Websites 139
Index 141
1 Introduction

1.1 Document Purpose

This document aims to provide a standard metamodel for Fact-Based Modelling (FBM). It is hoped that this metamodel will both assist in the development of software tools for storing data models that conform to the FBM approach, and also facilitate the exchange of FBM data models between different software tools. For readers unfamiliar with FBM approaches, the document also includes background on fundamental aspects of FBM as well as detailed explanations of the metamodel itself.

1.2 Overview of Fact-Based Modelling

Fact-Based Modelling is a conceptual approach for modelling, transforming and querying information where all facts of interest are represented in terms of attribute-free structures known as fact types. For example, declarations that the Etna volcano is active and is located in Italy are treated as instances of the fact types Volcano is active, and Volcano is located in Country. Here “is active” denotes a unary predicate, and “is located in” a binary predicate. In Entity-Relationship (ER) and Unified Modeling Language (UML) approaches, the unary fact type would instead typically be modelled as the attribute assignment Volcano.isActive = true. Besides facilitating more natural expression, and ease of population with multiple instances, eschewing attributes in favour of relationships promotes semantic stability (e.g. there is no need to remodel what has already been modelled if we later decide to record facts about an attribute).

Fact types of higher arity (ternaries, quaternaries etc.) are also allowed. For flexibility, and to cater for foreign languages, predicates may be represented in mixfix form, where the terms for the objects being predicated over are inserted in relevant placeholder positions to form the fact sentence. For example, the fact type “occasionally Person plays Sport for Country” involves the ternary predicate reading “occasionally … plays … for …”.

All facts, constraints and derivation rules are expressed in controlled natural language sentences that are intelligible to users in the business domain being modelled. For example, populations of the fact type Person was born on Date would normally be restricted by the uniqueness constraint: Each Person was born on at most one Date. As an example of a derivation rule, instances of the fact type Person is dead might be derived from instances of the fact type Person died on Date by using the rule: For each Person, that Person is dead if that Person died on some Date.

In addition to textual verbalization of data models, FBM includes graphical notations for depicting data models with a rich variety of constraints that goes far beyond the expressivity of the graphical data modelling constructs currently supported in UML class models or industrial ER. Because of their expressive power and intelligibility, FBM languages are especially suited for performing requirements analysis of data-intensive business domains, whether or not they are to be emulated in computerised information systems.

Along with its textual and graphical languages, FBM includes procedures for how to use those languages to design conceptual data models, and transform them to lower level structures for implementation. While different FBM dialects allow for some variations in language syntax, they all adopt
a conceptual schema design procedure that exploits the use of concrete examples to seed the data model, as well as using concrete examples and natural verbalization of constraints and rules to validate models with the relevant domain experts.

The FBM approach originated in Europe in the 1970s, and has since evolved into a family of closely related dialects including Object-Role Modelling (ORM), Cognition enhanced Natural Language Information Analysis Method (CogNIAM), Fully Communication oriented Information Modelling (FCO-IM) and the Developing Ontology-Grounded Methods and Applications (DOGMA) method. The metamodel presented in this document is intended to facilitate interoperability between such different versions of the FBM approach.

1.3 An Illustrative Example

To clarify the information requirements for a given business domain, the FBM modelling procedure employs data use cases (concrete examples of relevant data). These may take the form of input forms, output reports or sample queries. For example, Figure 1-1 shows extracts from two reports about employees, one in tabular form and the other in graphical form. For simplicity, person names are treated here as single character strings.

<table>
<thead>
<tr>
<th>EmployeeNr</th>
<th>Name</th>
<th>Title</th>
<th>Gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Ann Jones</td>
<td>Dr</td>
<td>F</td>
</tr>
<tr>
<td>102</td>
<td>Sue Wong</td>
<td>Mrs</td>
<td>F</td>
</tr>
<tr>
<td>103</td>
<td>John Smith</td>
<td>Dr</td>
<td>M</td>
</tr>
<tr>
<td>104</td>
<td>Beth Morgan</td>
<td>Ms</td>
<td>F</td>
</tr>
<tr>
<td>105</td>
<td>Tom Jones</td>
<td>Mr</td>
<td>M</td>
</tr>
<tr>
<td>106</td>
<td>Ann Bloggs</td>
<td>Ms</td>
<td>F</td>
</tr>
<tr>
<td>107</td>
<td>John Smith</td>
<td>Mr</td>
<td>M</td>
</tr>
</tbody>
</table>

Now suppose that for this business domain, the following example query is of interest. Here, a subordinate of an employee either reports to that employee (and hence is a direct report of that employee) or reports to some other subordinate of that employee. For the sample data, this query returns the result shown:

List the employee number and name of the subordinates of employee 102. → 104 Beth Morgan
105 Tom Jones
106 Ann Bloggs
107 John Smith

Our task is to design a data model to support these information requirements. We begin by verbalizing (typically with the help of a domain expert) the sample information as fact instances expressed in atomic sentences, and documenting the meaning of any terms used in the domain that might possibly be misunderstood by users. For example, using "Nr" as shorthand for "Number", the first row of data in the tabular report may be verbalized as shown below. Since more than one employee may have the same person name (e.g., 'John Smith') we identify employees by their employee numbers.

The Employee with EmployeeNr 101 has the PersonName 'Ann Jones'.
The Employee with EmployeeNr 101 has the PersonTitle 'Dr'.
The Employee with EmployeeNr 101 has the Gender with GenderCode 'F'.
By removing the specific object terms in these fact instance verbalizations, we generalize these instances to the following underlying fact types (kinds of fact). For the purposes of discussion, we use one of the popular FBM notations, in which the names of object types (kinds of objects) start with a capital letter, and their reference schemes (if any) are appended as shorthand reference modes in parentheses. Entities such as employees and genders could in principle be identified in many different ways, so a preferred reference scheme must be provided for them. Values such as person names and person titles are essentially self-identifying, so their reference schemes are empty. The other facts in rows 2 through 7 of the tabular report are instances of one of these three fact types.

Employee(.Nr) has PersonName().
Employee(.Nr) has PersonTitle().
Employee(.Nr) has Gender(.Code).

Similarly, the new information in the organization chart may be verbalized as shown below. We do not bother to restate the person name facts, since these were included in the verbalization of the tabular report. For clarity, we expand “CEO” to “chief executive officer”.

The Employee with EmployeeNr 101 is chief executive officer.
The Employee with EmployeeNr 102 reports to the Employee with EmployeeNr 101.
The Employee with EmployeeNr 103 reports to the Employee with EmployeeNr 101.

Each fact conveyed by the organization chart is an instance of one of the following fact types. There is only one instance of the first fact type because at most one employee is the CEO. Each of the six links in the organization chart depicts an instance of the second fact type.

Employee(.Nr) is chief executive officer.
Employee(.Nr) reports to Employee(.Nr)

Using one FBM graphical notation, the fact types may be displayed and populated as shown in Figure 1-2. Here, entity types are displayed as named, solid, rounded rectangles, with their reference modes underneath in parentheses. Value types are displayed as named, dashed, rounded rectangles. A fact role is a part played by an object in a fact. A role is displayed as a box connected by a line to the object type that hosts it.

![Diagram of fact types and constraints](image-url)
An ordered set of all the roles in a fact type is called a *predicate*. Predicate readings are displayed alongside, and are read from left to right or top to bottom, unless the reading direction is reversed by an arrow-head. Here four *binary fact types* (each with two roles), and one *unary fact type* (with one role) are displayed, along with their sample populations shown in accompanying fact tables. In this example, all object types are simply identified, so each role corresponds to a column in the associated fact table.

If desired, the same fact may be expressed in more than one way by using different predicate readings. For example, the fact that Employee 102 reports to Employee 101 may also be expressed by saying that Employee 101 manages Employee 102. In Figure 1-2 this is indicated by including both the forward predicate reading “reports to” and the inverse predicate reading “manages” separated by a slash.

The next phase of the design procedure involves declaring *constraints* that apply to the data model. If for each state of the fact base, each entry a role’s fact column may appear in that column only once, then the role has a simple *uniqueness constraint*. For a binary or longer fact type, this is depicted as a bar beside the role box. Since FBM fact types are always populated by *sets* of facts (not multisets), the role in a unary fact type implicitly has a uniqueness constraint, so there is no need to display a uniqueness constraint bar for it. The lack of a uniqueness constraint on a role indicates that duplicates may appear in its fact column. See Figure 1-2 for several examples.

If for each state of the fact base, each instance in the population of an object type must play a given role, then the role is said to be mandatory for that object type. A *mandatory role constraint* is displayed by placing a large dot at one end of the line connecting the role to its object type shape. The dot may be displayed at the object type end (as shown in Figure 1-2) or at the role end.

Constraints are validated with the domain expert by verbalizing them in a controlled natural language, and by checking whether they conform to sample data populations. For example, the uniqueness and mandatory role constraints depicted in Figure 1-2 may be automatically verbalized as follows. The phrase “exactly one” combines both uniqueness (at most one) and mandatory (at least one). The lack of a uniqueness constraint on a role is verbalized by indicating that the same object may appear more than once in any given population of that role.

- **Each** Employee has **exactly one** PersonName.
- **It is possible that more than one** Employee has **the same** PersonName.
- **Each** Employee has **exactly one** Gender.
- **It is possible that more than one** Employee has **the same** Gender.
- **Each** Employee has **exactly one** PersonTitle.
- **It is possible that more than one** Employee has **the same** PersonTitle.
- **Each** Employee reports to **at most one** Employee.
- **It is possible that some** Employee manages **more than one** Employee.

Constraints may also be verbalized in a negative way to indicate what counts as a violation of the constraint, and a *counterexample* to the constraint may be used to illustrate such a violation. Asking the domain expert whether such counterexamples are allowed is a good way of checking constraints that are doubtful. For example, the uniqueness constraint on the reporting fact type may be verbalized in negative form as "**It is impossible that some** Employee reports to **more than one** Employee", and checked using the counterexample shown in Figure 1-3 which provides a concrete case of an employee reporting to more than one employee.

![Figure 1-3](image_url)  
*Figure 1-3  Checking a uniqueness constraint with a counterexample*
The FBM design procedure includes a set of instructions for detecting other constraints that may apply. For example: Are there any other relationships of interest, especially functional (many-to-one, or one-to-one) relationships? In this case, the data suggests that some person titles are restricted to a single gender (e.g. 'Mr' is restricted to males, while 'Mrs' and 'Ms' are restricted to females). When this restriction is confirmed by the domain expert, we add the functional fact type PersonTitle is restricted to Gender, as shown in Figure 1-4. The sample data suggests that there are only two possible gender codes ('M' for male, and 'F' for female). When this is confirmed by the domain expert, we add the Gender value constraint displayed as {'M', 'F'} in Figure 1-4.

Other steps in the design procedure prompt us to add the further constraints shown in Figure 1-4. The join subset constraint depicted as a circled subset operator symbol connected to relevant roles may be automatically verbalized as follows. For example, this constraint prevents one from assigning the title 'Mrs' to an employee of male gender.

If some Employee has some PersonTitle that is restricted to some Gender
then that Employee is of that Gender.

The “# ≤ 1” role cardinality constraint on the unary fact type ensures that at any point in time there is at most one chief executive officer. The circled and crossed mandatory dot connected to the CEO and reporting roles depicts an exclusive-or constraint over these roles, indicating that each recorded employee must play exactly one of these roles. These constraints may be verbalized as follows:

For each population of “Employee is chief executive officer”,
the number of Employee instances is at most 1.

For each Employee, exactly one of the following holds:
that Employee is chief executive officer
that Employee reports to some Employee.

The circle with three dots and a stroke attached to the reporting fact type depicts an acyclic ring constraint indicates that the reporting relationship is acyclic. This may be verbalized as follows:

No Employee may cycle back to itself via one or more traversals through Employee reports to Employee.

If desired, a fact role may be given a role name, for use in rules that reference the role, or to control the names of columns generated when mapping the FBM schema to a relational database schema. Diagrammatically, role names appear in square brackets next to the role that they name. Figure 1-4 includes three examples: directReport, manager, and subordinate. The derived fact type Employee is subordinate to Employee is introduced to support the query example discussed earlier. The asterisk appended to its predicate reading indicates that instances of the fact type are derived from other facts rather than simply being asserted. The derivation rule used to infer instances of this fact type is shown below the diagram, and prepended by an asterisk.
This derivation rule is recursive, since the "is subordinate to" predicate is used in both the head and the body of the rule. The acyclicity constraint on the reporting fact type also involves recursion, and may be enforced by checking that the transitive closure of the reporting fact type is irreflexive. Since the extension of the derived fact type Employee is subordinate to Employee is the transitive closure of the Employee reports to Employee fact type, the acyclicity constraint may be checked most efficiently by materializing (storing the population of) the derived fact type, and implementing the constraint incrementally. In other words, when the reporting fact type is updated, only the updates to the population of the derived fact type need be checked for irreflexivity, rather than recomputing the full transitive closure and testing that for irreflexivity.

Hence, for efficiency reasons, the fact type Employee is subordinate to Employee is declared to be derived and stored (shown graphically by appending a double asterisk to its predicate reading) and irreflexive (shown graphically by attaching a constraint circle with a dot and stroke). See Figure 1-5. The irreflexive constraint verbalizes as follows:

No Employee is subordinate to the same Employee.

By default, constraints on derived fact types are implied, and are displayed in the usual colour (violet). In Figure 1-5, the irreflexive constraint shape is coloured red, indicating that the constraint on the derived fact type is asserted. The acyclicity constraint on the asserted fact type Employee reports to Employee is now implied, and is coloured green.

![Diagram](image)

Figure 1-5  Deriving the acyclicity constraint on the reporting fact type from irreflexivity of its transitive closure

Supporting documentation is typically added to clarify the meaning of any aspects of the conceptual schema terminology that might possibly be misunderstood. Entries for role names are shown below. Similar explanations are provided for object type names and fact type readings.

<table>
<thead>
<tr>
<th>RoleName</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>directReport</td>
<td>If employee e1 reports to employee e2, then e1 is a direct report of e2, and e2 is the manager of e1.</td>
</tr>
<tr>
<td>manager</td>
<td>An employee who manages at least one other employee.</td>
</tr>
<tr>
<td>subordinate</td>
<td>If employee e1 reports either to employee e2 or to some employee e3 who ultimately (via a chain of one or more intermediate managers) reports to e2 then e1 is a subordinate of e2.</td>
</tr>
</tbody>
</table>

The conceptual schema for the example business domain is now complete, and may be mapped to various targets systems (e.g. a relational database or a deductive database) for implementation.

Since the conceptual model (both schema and sample population) may be verbalized unambiguously in a controlled natural language that is readily intelligible to domain experts, the conceptual model can be validated with domain experts without their needing to view and understand the schema diagrams. For data modelers, however, the expressive graphical notation is very useful for capturing and visualizing the business data requirements in a compact form, and seeing how different aspects are connected.
1.4 Conventions and Abbreviations

The primary specification of the metamodel is rendered in a controlled natural language (CNL) that has an underlying logical formalization. In this document, the CNL is based on English, but software tools may provide support for languages other than English. Various style options are allowed for the CNL used in this specification. For example, whether an object type name is rendered in PascalCase (e.g. `FactType`) or not (e.g. `fact type`) is a user choice.

The bulk of the metamodel is also rendered in the graphical notation of Object-Role Modelling (ORM), including notes with derivation rules in CNL. The metamodel itself was entered using the Natural ORM Architect (NORMA) tool, and is available as a file with the extension `.orm`. A glossary of the ORM graphical notation is included in Annex A. When available, a rendering in the graphical notation of CogNIAM may also be provided.

The metamodel is spread over many different pages. The same metamodel element may appear on many pages. If a page is the first page on which a shape for a given object type or predicate appears, then that shape is coloured with light blue fill on that page. If a shape for a given object type or predicate appears more than once in the metamodel, it may be displayed with a shadow to indicate this duplication. In the NORMA file for the metamodel, display of shadows may be toggled on or off by selecting the desired choice from the relevant configuration menu (Tools > Options > ORM Designer > Display Shadows). Many pages of the metamodel include model notes in rectangular boxes that are used to provide further explanation or to visually display the verbalization of derivation rules relevant to that page.

The following acronyms or abbreviations may be used within this document.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CogNIAM</td>
<td>Cognition enhanced Natural Language Information Analysis Method</td>
</tr>
<tr>
<td>CNL</td>
<td>Controlled natural Language</td>
</tr>
<tr>
<td>DOGMA</td>
<td>Developing Ontology-Grounded Methods and Applications</td>
</tr>
<tr>
<td>ER</td>
<td>Entity-Relationship modelling</td>
</tr>
<tr>
<td>FBM</td>
<td>Fact-Based Modelling</td>
</tr>
<tr>
<td>FCO-IM</td>
<td>Fully Communication Oriented Information Modelling</td>
</tr>
<tr>
<td>GUID</td>
<td>Globally Unique Identifier</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>NORMA</td>
<td>Natural ORM Architect</td>
</tr>
<tr>
<td>ORM</td>
<td>Object-Role Modelling</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>SBVR</td>
<td>Semantics of Business Vocabulary and Business Rules</td>
</tr>
<tr>
<td>SID</td>
<td>Surrogate Identifier</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>UoD</td>
<td>Universe of Discourse</td>
</tr>
<tr>
<td>UUID</td>
<td>Universally Unique Identifier</td>
</tr>
</tbody>
</table>

1.5 Exchange-specific Aspects

While the bulk of the metamodel is concerned only with purely semantic aspects of FBM, some features are specifically included to facilitate implementation within FBM software tools or exchange of FBM models between different software tools. The two main aspects in this regard concern the ordering of model elements, and the use of surrogate identifiers (SIDs).

As an example of the ordering aspect, the order in which roles are included in a uniqueness constraint might have no significance to users in the business domain being modelled. So from a purely semantic perspective, one might argue for a metamodel in which a uniqueness constraint applies simply to a set of roles. For implementation purposes, however, the order can have an impact. For example, when mapped to a relational database schema, the order chosen for elements of an index can impact the index's
performance. Moreover, in practice a set of elements is always physically stored as an ordered set, so if two FBM tools choose different ordering criteria, and are unaware of these different choices, this complicates detection of local changes when models are exchanged back and forth between the tools. For such reasons, the exchange version of the uniqueness constraint includes an order over the roles.

Where relevant, some parts of the metamodel are presented in two versions, one purely semantic and one exchange-oriented (e.g. predicate reading preferences). For other relevant cases (e.g. mandatory role constraints), it is assumed that an ordering criterion is explicitly agreed to for exchange purposes.

Several elements in the metamodel use surrogate identification schemes. Surrogate identifiers add no conceptual understanding to the model and are usually automatically generated (see UUID below). Beyond a basic expectation that identifiers satisfy XML identifier requirements, the Standard is neutral with respect to the structure of the surrogate identifiers in exchanged documents. While valid surrogate identifiers can often be generated from the contents of the model (known as extensional identifiers), we do not recommend this practice because the generated identifiers are not stable over time, resulting in excessive document changes for a small change in the model. There are also an arbitrary number of algorithms that could be applied to generate extensional identifiers. For example, valid extensional identifiers for the fact type ObjectType plays Role include ObjectTypePlaysRole, ObjectType_plays_Role, and object_type_plays_role. The Standard expects vendors to display extensional identification schemes to the user, but the algorithms for these presentation schemes do not affect exchange and these algorithms are intentionally omitted from this Standard.

Surrogate identifiers reduce the complexity of referencing elements with multi-part extensional identification schemes, preserve identity over time as the (extensional) model contents change, and facilitate merging of multiple models with potentially overlapping contents. Although the ability to exchange structurally incomplete models is not supported in this version of the Standard, many FBM tools support partial model states that have insufficient data to generate unique extensional identifiers. This scenario requires the use of surrogate identifiers to serialize and de-serialize model state.

Taking into account the distributed approach in developing conceptual data models promoted by this Standard, the uniqueness of names assigned by a given conceptual data model supplier cannot be guaranteed at customer level (e.g. when merging supplied conceptual data models). For this reason, to uniquely identify data to be exchanged, the ORM metamodel file provided with this Standard uses universally unique identifiers (UUID, in compliance with ITU-T Rec. X.667 - ISO/IEC 9834-8) as its surrogate identification form. The string representation of any UUID beginning with a number—5/8 of the range of such values—does not constitute a valid XML identifier, so we standardize on upper-cased identifiers with an underscore (_) prepended. For example, 9bbf94d9-cfe7-4140-9ad8-c01f0e94cf35 would appear in identifier form as _9BBF94D9-CFE7-4140-9AD8-C01F0E94CF35.

Nevertheless, to enable different tool vendors to adopt different strategies for generating surrogate identifiers, this metamodel specification does not itself prescribe how such SIDs are created. For this reason, the popular reference mode “(SID)” is provided for types with surrogate identifiers in the metamodel.

As a simple example of pure semantics versus exchange, if an FBM tool vendor has no interest in supporting model exchange or multi-model reuse, or in facilitating renaming of model elements, object types could simply be identified by their name rather than use a SID.

### 1.6 Document Structure

Chapter 2 specifies and explains the core aspects of the metamodel, with each section devoted to one specific part of the metamodel. Although the CNL specification of the metamodel is primary, the ORM diagrams provide a much more compact rendering of the metamodel as well as visually displaying connections between the various components. Hence, for each part of the metamodel, the ORM diagrams are displayed first, so that users familiar with the graphical notation can quickly grasp the intent.
Following the diagram, the verbalization in CNL of newly introduced aspects is provided in the following order: Object Types, Reference Schemes and Subtyping (if any); Asserted Fact Types; Derived Fact Types (if any) and their derivation rules; Constraints. An explanation is then given to clarify the meaning of the metamodel fragment. This may include clarification of the meaning of object type names and fact type readings and related concepts, explanation of graphical notation, and concrete examples to illustrate sample populations of the metamodel fragment for simple cases.

For readers interested in finer details about the metamodel, Chapter 3 provides additional derivation rules and constraints that place further conditions that must be satisfied in order for an FBM model to be well formed. Here, derived fact types are often specified purely to enable complex constraints to be depicted graphically by asserting constraints directly on the derived fact types. In UML, constraints like these would instead typically be specified textually as OCL code.

Chapter 4 provides a comprehensive list of terms used in the FBM approach, along with their definitions, and some illustrative examples.

Annex A provides a concise summary of the graphical notation used in ORM. Annex B provides a list of references and useful websites for further reading. Finally, an Index is provided to facilitate quick access to specific items and topics.
2 Metamodel: Core

2.1 Core Elements

ORM Diagram

![ORM Diagram](image)

**Figure 2-1** Key concepts

Note: Entity types are displayed as named, rounded rectangles with solid line borders. In this document, shapes for object types and fact types are displayed with light blue fill when first introduced. When displayed in a later figure, white fill is used instead. This use of fill-colour is purely to highlight new aspects and avoid repeated verbalization of aspects introduced earlier, and is not a standard aspect of the ORM graphical notation. In this document, shapes for object types and predicates that are displayed multiple times (in the same figure or in separate figures) are highlighted with a shadow.

Verbalization

*Object Types and Reference Schemes*

**ObjectType** is an entity type.
Reference Scheme: ObjectType has ObjectTypeSID.
Reference Mode: .SID.

**Role** is an entity type.
Reference Scheme: Role has RoleSID.
Reference Mode: .SID.

**FactType** is an entity type.
Reference Scheme: FactType has FactTypeSID.
Reference Mode: .SID.

**Asserted Fact Types**

**ObjectType hosts Role.** / Role is hosted by ObjectType.

**FactType contains Role.** / Role belongs to FactType.

**Constraints**

Each Role is hosted by exactly one ObjectType.
It is possible that some ObjectType hosts more than one Role.

Each FactType contains some Role.
Each Role belongs to exactly one FactType.
It is possible that some FactType contains more than one Role.
Explanation

An object is any individual thing of interest (e.g. a specific person or language). An object type (e.g. Person or Language) is a concept used to classify objects into different kinds; it includes all possible instances of that kind. For example, Albert Einstein and Julius Caesar are two instances of Person.

Verbalizations are to be understood within a metalevel context. For example, the verbalization “ObjectType is an entity type” means that ObjectType is an instance of a meta entity type. It does not mean that every object type in a given business domain model is an entity type in that domain model. Similarly, the verbalizations “Role is an entity type” and “FactType is an entity type” mean that Role and FactType are instances of a meta entity type, not a domain entity type.

A fact is a proposition taken to be true by the relevant business community, where the proposition simply declares that (a) some individual exhibits a property (e.g. Pat smokes, Serge is tall), or that (b) one or more individuals participate in a relationship (e.g. Terry speaks Latin, Terry knows Terry, Terry introduced Norma to Matt), or that an individual exists (e.g. Terry exists). A fact type (e.g. Person smokes, Person speaks Language, Person introduced Person to Person) includes all possible instances of a given kind of fact.

A role (i.e. a fact role) is a part that can be played by individual objects within a fact. For example, in the fact “Terry speaks Latin”, Terry plays the speaker role, and Latin plays the role of being spoken. Each role is hosted by an object type, meaning that objects that play that role belong to that object type.

An entity is an object that is referenced by a definite description that relates it to other objects (e.g. the Country that has the CountryCode ‘AU’). An entity can typically change over time while maintaining its identity, unlike a domain value which is always a semantically typed constant (e.g. the CountryCode ‘AU’). An entity type is a kind of entity (e.g. Country, Person, Phone). A value type is a kind of value (e.g. CountryCode, PersonName, PhoneNumber). Note that an entity can change its reference scheme over time, but a value cannot. For example, suppose countries are primarily referenced by their names. When the entity formerly named “Burma” changed its name to “Myanmar” it remained the same entity. However the name “Burma” and the name “Myanmar” can never change. If a later version of the metamodel supports the notion of a structured value (e.g. a person name composed of a family name and given name) a structured value cannot change over time.

In the ORM graphical notation, an entity type is depicted as a named, soft rectangle with a solid line. A value type is distinguished by a dashed line. If the reference scheme for an entity type is simply a fact type that provides a one-to-one mapping from each instance of the type’s population to a single value (e.g. Country has CountryCode) then the manner in which each value refers to its entity is called a reference mode, and the reference fact type may be displayed in abbreviated form by listing its reference mode in parenthesis below the object type name. A popular reference mode is displayed with a leading dot, indicating that the value type name is formed by appending the reference mode name to the object type name. For example, Country,(Code) abbreviates the Country has CountryCode fact type together with its associated constraints (explained later).

As shown in the verbalization, the (SID) reference schemes map ObjectType to ObjectTypeSID, Role to RoleSID, and FactType to FactTypeSID. These three SID types are value types that are implicitly disjoint (mutually exclusive) because they are “top level types” (not subtypes). If a tool vendor wishes to introduce a single global SID type (e.g. UUID), this may be done as an extension (e.g. by declaring the specific SID types as subtypes of the global SID type and explicitly declaring their disjointness).

In the ORM graphic notation, each fact role is displayed as a small role box connected to the object type that hosts that role. The object type provides a pool of object instances, some or all of which may be declared to play that role. Only instances of the associated object type may play that role.

A logical predicate is depicted as an ordered set of role boxes with at least one predicate reading (e.g. “hosts”). By default, predicates are read left-to-right or top-to-bottom, unless the reading direction is reversed by including an arrow-tip pointing the other way. If a predicate is binary (2 roles), readings for both forward and inverse predicates may be displayed, separated by a slash (e.g. “hosts / is hosted by”). A fact type reading is determined by placing its object type names in the appropriate placeholders in a predicate reading (e.g. ObjectType hosts Role).
The bars over the roles hosted by Role depict uniqueness constraints, ensuring that each Role is hosted by at most one ObjectType and belongs to at most one FactType. The large dots connected to three of the roles in the metamodel fragment depict mandatory role constraints, ensuring that each Role is hosted by some ObjectType and belongs to some FactType, and that each FactType contains some Role. The constraint verbalization shown uses “exactly one” to combine “at most one” with “some” (at least one). As a display option, users may choose to place the mandatory role dot either at the object type end or at the role end of the role connector.

To understand metamodel fragments, it often helps to populate them with a concrete example. Consider the sample ORM schema shown in Figure 2-2(a). For simplicity, reference schemes for Person and Country are omitted. Figure 2-2(b) shows a modified version of the metaschema fragment where object types are identified simply by their primary name (e.g. “Person”, “Country”), and fact types are identified by their preferred reading text (e.g. “Person smokes”, “Person was born in Country”). As there is no simple and natural way to identify roles in a global schema (other than to give them each a local name and then combine that with their primary fact type reading) we introduce surrogate identifiers (r1, r2 and r3) for the roles.

![Figure 2-2 Populating metamodel (b) with the model (a), using natural identifiers for ObjectType and FactType](image)

Figure 2-2(b) shows the result of populating the metaschema fragment in (b) with the sample schema in Figure 2-2(a). Each of the two metafact types is populated by a fact table that includes facts conveyed by the sample schema. Each column in a fact table corresponds to the fact type role shown above it. For example, the first fact table tells us that the Person object type hosts roles r1 and r2, and that the Country object type hosts role r3. The other fact table tells us that role r1 belongs to the fact type Person smokes, and that roles r2 and r3 belong to the fact type Person was born in Country. This conveys the basic idea.

Now consider Figure 2-3, where we now use surrogate identifiers throughout. The Person and Country types retain their type names but are now primarily identified by surrogate the identifiers OT1 and OT2. Similarly, the fact type in Figure 2-3(a) retain their readings but are now referenced by the surrogate identifiers FT1 and FT2. In practice, one might generate lengthy UUIDs instead of the short surrogates shown here, but the principle is the same. Use of surrogates allows us to ensure rigid identifiers for the object types, fact types and roles. These identifiers remain unchanged even if we decide later to rename some of the elements (e.g. rename “Person” to “Human”), while facilitating exchange between different tools (and interoperation between different conceptual models) by maintaining stable identifiers that do not clash with identifiers in other models.

![Figure 2-3 Populating metamodel (b) with the model (a), using surrogate identifiers throughout](image)
2.2 Object Type and ObjectTypeName

ORM Diagrams

![ORM Diagram](image)

**Figure 2-4** Partitioning and Naming of Object Types

**Note:** To save space, this figure combines two pages of the metamodel. In this document, the preferred identification scheme for each entity type in the metamodel is displayed only in the figure in which the type is first introduced. Hence the reference mode (.SID) for ObjectType is omitted here. An entity subtype displayed with a solid subtyping arrow connection to a supertype inherits its identification scheme from that supertype, so the four subtypes of ObjectType shown here all have their identification scheme based on ObjectTypeSID.

**Verbalization** (of newly introduced aspects)

*Object Types, Reference Schemes and Subtyping*

**DomainObjectType** is an entity type.
- Each DomainObjectType is an instance of ObjectType.
- Reference Scheme: ObjectType has ObjectTypeSID.

**EntityType** is an entity type.
- Each EntityType is an instance of DomainObjectType.
- Reference Scheme: ObjectType has ObjectTypeSID.
ValueType is an entity type.
   Each ValueType is an instance of DomainObjectType.
   Reference Scheme: ObjectType has ObjectTypeSID.

DataType is an entity type.
   Each DataType is an instance of ObjectType.
   Reference Scheme: ObjectType has ObjectTypeSID.

ObjectName is a value type.

Asserted Fact Types

ValueType maps to DataType.
   DomainObjectType has primary ObjectTypeName. / ObjectTypeName is the primary name of DomainObjectType.
   DomainObjectType has alternate ObjectTypeName. / ObjectTypeName is an alternate name of DomainObjectType.
   DataType has primary ObjectTypeName. / ObjectTypeName is the primary name of DataType.
   DataType has alternate ObjectTypeName. / ObjectTypeName is an alternate name of DataType.

Constraints

   Each ValueType maps to exactly one DataType.
   It is possible that more than one ValueType maps to the same DataType.

   Each DomainObjectType has exactly one primary ObjectTypeName.
   Each ObjectTypeName is the primary name of at most one DomainObjectType.

   Each ObjectTypeName is an alternate name of at most one DomainObjectType.
   It is possible that some DomainObjectType has more than one alternate ObjectTypeName.

   Each DataType has exactly one primary ObjectTypeName.
   Each ObjectTypeName is the primary name of at most one DataType.

   Each ObjectTypeName is an alternate name of at most one DataType.
   It is possible that some DataType has more than one alternate ObjectTypeName.

   For each ObjectTypeName, at most one of the following holds:
      that ObjectTypeName is the primary name of some DomainObjectType;
      that ObjectTypeName is an alternate name of some DomainObjectType.

   For each ObjectTypeName, at most one of the following holds:
      that ObjectTypeName is the primary name of some DataType;
      that ObjectTypeName is an alternate name of some DataType.

   If some DomainObjectType is some EntityType
   then some ObjectType that is that DomainObjectType hosts some Role.
   (i.e. Each EntityType is an ObjectType that hosts some Role).

   For each ObjectType, exactly one of the following holds:
      that ObjectType is some DomainObjectType;
      that ObjectType is some DataType.

   For each DomainObjectType, exactly one of the following holds:
      that DomainObjectType is some EntityType;
      that DomainObjectType is some ValueType.
Explanation

A *partition* of a type is a set of two or more subtypes that are mutually exclusive (they have no instance in common) and collectively exhaustive (their union covers all instances in the overall type). ObjectType is partitioned into DomainObjectType and dataType, and DomainObjectType is partitioned into EntityObject and ValueType.

The populations of *domain object types* come from the business domain being modelled, and are always finite. In contrast, most *data types* (e.g. Integer or String) are theoretically infinite even if implementations typically restrict their size. Instances of value types (e.g. CountryCode, EmployeeNr) are typed constants, where the type names have implicit semantics in the business domain that goes beyond the simple semantics of data types. For example, it is true that the CountryCode ‘CH’ is based on Latin, but this is not true of the character string ‘CH’, so country codes have semantics beyond that of strings. The metatfact type ValueType maps to DataType is used to assign a specific data type for storing instances of a specific value type (e.g. country codes might be represented in storage by character strings of length 2).

Each object type has exactly one primary name (e.g. "Person") used for normal human communication in the business domain, but may be also be given one or more alternate names (e.g. "Human") if desired. In the ORM diagram, the inclusion of *hyphens* in the fact type readings “DomainObject has primary- ObjectTypeName” and “DomainObject has alternate-ObjectTypeName” ensures that the hyphenated qualifiers (in this case adjectives) are bound to their object type term when relevant constraints are automatically verbalized.

For example, “Each DomainObject has exactly one primary ObjectTypeName” reads better than “Each DomainObjectType has primary exactly one ObjectTypeName”. This is known as *hyphen-binding*. The circled “X” in the ORM diagram depicts an *exclusion constraint* to ensure that no object type name of a domain object type can be both primary and alternate, and similarly for data types. Using separate naming fact types for domain object types and data types allows the same name to be used for both types. For example, a model may include an entity type named “Date” as well as a data type named “Date”.

A solid arrow directed from one object type to another indicates that the source object type is a proper *subtype* of the target object type (a *supertype*). This means that each instance in the population of the subtype must also be an instance in the population of its supertype(s), but that it is possible to have a population of the supertype that includes instances not found in that subtype. For example, Woman is a proper subtype of Person.

A circled dot depicts an *inclusive-or constraint* over the roles to it is attached, indicating that each instance in the population of the attached object type must play at least one of the constrained roles. Superimposing an “X” (for exclusion) over this symbol strengthens it to an *exclusive-or constraint*, ensuring that each instance in the population of the attached object type must play exactly one (at least one and at most one) of the constrained roles.

In Figure 2-4, the exclusive-or constraints are displayed with connections to the subtyping relationships. However, this is understood to mean that the constraints actually apply to the roles hosted by the supertypes in the *instance-level, identity fact types* that are implied by the subtyping connections. For example, the exclusive-or constraint between the subtyping connections from DomainObject to Supertype in Figure 2-4 actually apply to the roles depicted in gold fill in Figure 2-5(a).

The subtyping arrows tell us that each DomainObject and each DataType is an ObjectType, which implies that each instance of DomainObject is an instance of Objecttype and that each instance of Data type is an instance of Objecttype. So implicitly we have the fact types DomainObject is / is Objecttype and Data type is / is Objecttype, whose roles (as usual for fact types) are played by instances of the types not the types themselves. These implied fact types are displayed with dashed lines in Figure 2-5(a). The exclusive-or constraint applies to the implicit roles hosted by the supertype, so each instance of Objecttype is an instance of DomainObject or Data type, but not both.
The circled “⊂” with dashed line connections in Figure 2-4 depicts a *subset constraint*, indicating that the population at the source end (with no arrow-tip) must be a subset of the role population at the target end (with the arrow-tip). Although for convenience this is displayed with a subtyping connection at the subset (source) end, the actual source is understood to be the role hosted by the supertype in the implicit instance-level fact type underlying that subtyping connection. Figure 2-5(b) shows this explicitly, with the subset constraint applying between the roles in gold fill.

The subset constraint is used here to ensure that each entity type hosts at least one role. This is required in FBM, since each entity should be well identified, so must play at least one role in a reference scheme. In contrast, value types (which have no relationship-based reference scheme) are not required to host a role, even though they typically do. For example, in an FBM model one may include the value type `CountryCode` without ever using it in a fact type (e.g. to refer to countries). This allows us to simply store a list of country codes without explicitly modelling countries as well.

While implicit displays as in Figure 2-5 are useful for understanding the meaning of constraints displayed on subtyping connections, they are not considered a standard part of the ORM graphical notation.

Figure 2-6 shows a simple example of populating this fragment of the metamodel. For simplicity, datatype mapping is ignored, and no alternate names are used for object types.
Figure 2-6  Populating the object type metamodel with a sample model
2.3 Role Names

ORM Diagram

![ORM Diagram](image)

**Figure 2-7** Role names

**Verbalization** (of newly introduced aspects)

**Object Types and Reference Schemes**

- **RoleName** is a value type.

**Asserted Fact Types**

- Role has RoleName. / RoleName is of Role.

**Constraints**

- Each Role has at most one RoleName.
- It is possible that some RoleName is of more than one Role.
- For each FactType and RoleName, at most one Role belongs to that FactType and has that RoleName.

**Explanation**

Unlike UML class diagrams, where association names are optional but association role names are required, FBM requires each fact type to have a reading while leaving it optional for roles to be named. This difference reflects the FBM practice of requiring that facts can always be declared naturally using sentences, while freeing the modeller from the burden of having to invent names for roles for which no obvious choice is available. For example, in the fact type Person is a brother of Person, the grammatical “subject” role is naturally named “brother”, but a name choice for the grammatical “object” role is likely to be awkward (e.g. “personWithBrother”).

On an ORM diagram, each role name is displayed in square brackets next to its role, in blue colour. For example, in Figure 2-8, six of the eight roles are named. The circled bar in Figure 2-7 denotes an external uniqueness constraint, ensuring that each combination of fact type and role name applies to at most one role (i.e. role names are unique within the context of their fact type). For example, Figure 2-8 include two roles named “owner”, so these must belong to different fact types. For fine details about additional constraints that apply to role names, see Section 3.1.

**Figure 2-8** Role names are optional, but if used they are identifying within the scope of their fact type
2.4 Fact Type Type Arity

ORM Diagram

![ORM Diagram]

Verbalization (of newly introduced aspects)

Object Types, Reference Schemes and Subtyping

Arity is a value type.

UnaryFactType is an entity type.
* Each UnaryFactType is by definition some FactType that has Arity = 1.
  Reference Scheme: FactType has FactTypeSID.

BinaryFactType is an entity type.
* Each BinaryFactType is by definition some FactType that has Arity = 2.
  Reference Scheme: FactType has FactTypeSID.

NaryFactType is an entity type.
* Each NaryFactType is by definition some FactType that has Arity at least 3.
  Reference Scheme: FactType has FactTypeSID.

Derived Fact Types

* FactType has Arity if and only if
  Arity = \text{count(each Role that belongs to that FactType)}.

Constraints

Each FactType has exactly one Arity.

It is possible that some Arity is of more than one FactType.
Explanation

The arity of a fact type is its number of roles. A unary fact type has one role (e.g. Person smokes), a binary fact type has two roles (e.g. Person drives Car), and an $n$-ary fact type has three or more roles (e.g. Person played Sport for Country is a ternary fact type (3 roles) and Product in Region in Year sold in Quantity is a quaternary fact type (4 roles).
2.5 Fact Type Predicates

ORM Diagram

![Diagram of Fact Type Predicates]

Figure 2-10 Fact Type Predicate

Verbalization (of newly introduced aspects)

Object Types and Reference Schemes

- Predicate is an entity type.
  - Reference Scheme: Predicate has PredicateSID.
  - Reference Mode: .SID.

- Position is an entity type.
  - Reference Scheme: Position has PositionNr.
  - Reference Mode: .Nr.

Asserted Fact Types

- FactType has Predicate. / Predicate orders roles of FactType.
  - Predicate orders Role at Position.

Constraints

- Each FactType has some Predicate.
- Each Predicate orders roles of exactly one FactType.
- It is possible that some FactType has more than one Predicate.
- For each Predicate and Role, that Predicate orders that Role at at most one Position.
- For each Predicate and Position, that Predicate orders at most one Role at that Position.
- Each Predicate orders some Role at some Position.
  - Each Role occupies some Position in some Predicate.
  - Each value of PositionNr in "Position has PositionNr" is at least 1.

Explanation

Elementary facts are propositions that apply a logical predicate to one or more objects. These facts are expressed by sentences that place terms for the relevant objects at appropriate slots (placeholders) within a predicate reading. In an ORM diagram, object slots in a predicate reading may be explicitly displayed as an ellipsis "...". Unary predicate readings are assumed to be in postfix position (after the object term) if no ellipsis is displayed. For example, in "Arnold smokes" the unary predicate reading "smokes" is understood as shorthand for "... smokes". Binary predicate readings in infix position
(between two object terms) are implicitly understood to have their object slots at either end. For example, in “Romeo loves Juliet” the binary predicate reading “loves” is understood as shorthand for “... loves ...”. In all other cases, the object slots in predicate readings must be explicitly displayed. For example, the fact type readings “on Date Person was born” and “Person plays Sport for Country” use the predicate readings “on ... was born” and “… plays ... for ...”. This notation which allows object terms to be mixed in anywhere in the sentence is known as mixfix notation.

FBM allows multiple fact type readings to be supplied for a fact type. For example, the fact type readings “Person works for Company” and “Company employs Person” might be given for the same fact type. In this case, two different predicates, with readings “works for” and “employs”, are used, with one predicate being the inverse of the other. Each predicate thus corresponds to one way of navigating through the roles in the fact type. Hence a predicate corresponds to an ordered set of roles for its fact type. With an ordered set of elements, the order of the elements is important, but each element occurs only once in the set (no duplicates).

In Figure 2‐10, the ordered set structure is captured by the two compound uniqueness constraints. For each state of the conceptual database, a uniqueness bar spanning two or more roles requires each instance for that role combination to appear only once. If the constrained roles are not contiguous, a dotted line between role bars indicates the role that is excluded from the constraint. The uniqueness constraint bar over the Predicate and Role roles indicates that each combination of Predicate and Role occurs in only one Position (so roles cannot be duplicated within a predicate). The uniqueness constraint bar over the Predicate and Position roles indicates that each combination of Predicate and Position applies to only one Role (so each object slot corresponds to just one role).

Though rarely used in practice, FBM also allows multiple readings to be supplied for a predicate. For example, the fact type readings “Person works for Company” and “Person is employed by Company” might be given for the same fact type. In this case, both predicate readings “works for” and “is employed by” are used for the same predicate.

The “{1..}” in Figure 2‐10 depicts a value constraint that restricts the possible values of PositionNr used in this fact type to the natural numbers (i.e. integers 1, 2, 3 etc.). Though not shown on the diagram, the value type PositionNr is mapped to the datatype UnsignedSmallInteger, which restricts the largest number allowed. Details of such datatype mappings are largely ignored in this document, but may be accessed in the metamodel file itself.

Figure 2‐11 illustrates how the metamodel fragment may be populated by a simple model. The model in Figure 2‐11(a) includes just one fact type identified by the surrogate FT1. This fact type has the two fact type readings “Person drives Car” and “Car is driven by Person”. The predicate P1 has the predicate reading “... drives ...”, and navigates the roles of this fact type in the order r1, r2. The predicate P2 has the predicate reading “... is driven by ...”, and navigates the roles of this fact type in the order r2, r1.

![Figure 2-11 Populating the Fact Type Predicate metamodel with a sample model](image)

For fine details concerning further constraints on fact type predicates, see Section 3.3.
2.6 Predicate Readings

ORM Diagram

```
 Predicate
      /
     has is of

 PredicateReading (.SID)

 "ReadingRoleText"
     has front-

 Role
     attaches text to

 ReadingText
     has pre-- bound
     has post-- bound
     has following-
```

Figure 2-12 Predicate Readings

Verbalization (of newly introduced aspects)

Object Types and Reference Schemes

PredicateReading is an entity type.
Reference Scheme: PredicateReading has PredicateReadingSID.
Reference Mode: .SID.

ReadingText is a value type.

ReadingRoleText is an entity type.
ReadingRoleText objectifies "PredicateReading attaches text to Role".

Asserted Fact Types

PredicateReading is of Predicate. / Predicate has PredicateReading.
PredicateReading has front ReadingText.
PredicateReading attaches text to Role.
ReadingRoleText has pre-bound ReadingText.
ReadingRoleText has post-bound ReadingText.
ReadingRoleText has following ReadingText.

Constraints

Each Predicate has some PredicateReading.
Each PredicateReading is of exactly one Predicate.
It is possible that some Predicate has more than one PredicateReading.
Each PredicateReading has at most one front ReadingText.
It is possible that more than one PredicateReading has the same front ReadingText.
It is possible that some PredicateReading attaches text to more than one Role and that for some Role, more than one PredicateReading attaches text to that Role.
In each population of PredicateReading attaches text to Role, each PredicateReading, Role combination occurs at most once.

Each ReadingRoleText has at most one pre-bound ReadingText. It is possible that more than one ReadingRoleText has the same pre-bound ReadingText.

Each ReadingRoleText has at most one post-bound ReadingText. It is possible that more than one ReadingRoleText has the same post-bound ReadingText.

Each ReadingRoleText has at most one following ReadingText. It is possible that more than one ReadingRoleText has the same following ReadingText.

Each PredicateReading attaches text to some Role or has some front ReadingText.

Each ReadingRoleText has some pre-bound ReadingText or has some post-bound ReadingText or has some following ReadingText.

Explanation

A *predicate reading* is an occurrence of the use of text to describe a single predicate. Although the same predicate text (e.g. "has") may be used for multiple predicates, these occurrences correspond to different predicate readings. The fact type PredicateReading attaches text to Role is *objectified* as the object type ReadingRoleText. So an instance of ReadingRoleText corresponds to the attachment of text to a specific role within a specific predicate reading.

Diagrammatically, an object type resulting from objectification is displayed as a named, soft rectangle enclosing the predicate shape from which it was formed. In FBM, *objectification* is situational rather than propositional, so instances of the object type resulting from the objectification are in 1:1 correspondence to the relationship instances in the original fact type but are not identical to them.

For each PredicateReading, its *front text* (if any) is non-hyphen-bound text before its first object slot. For each ReadingRoleText, its *pre-bound text* (if any) is hyphen-bound text immediately preceding its object slot, its *post-bound text* (if any) is hyphen-bound text immediately after its slot, and its *following text* (if any) is non-hyphen-bound text immediately after its slot. For constraint verbalization purposes, a hyphen appended to a word binds that word and any other predicate text before the next object slot to the object term immediately after it, and a hyphen prepended to a word binds that word and any other predicate text after the previous object slot to the object term immediately before it. To bind a hyphenated word (e.g. "pre-bound"), an extra hyphen and space are added to the original hyphen (e.g. "pre--bound").

The sample model in Figure 2-13(a) has just one predicate P1, which orders its four roles in the order r1, r2, r3, r4 and has the predicate reading PR1 whose predicate text is as shown. Each of the roles r1 through r4 has one slot (depicted as an ellipsis “…”) in the predicate reading. Read from left-to-right, r1 corresponds to the first slot, r2 to the second slot, r3 to the third slot, and r4 to the fourth slot.

Figure 2-13(b) populates the metamodel with this sample model. In this document, fact tables are either displayed with their columns immediately next to the roles they are populated, or are shown connected by a dotted line to their fact type. Here, each instance of ReadingRoleText is displayed as an ordered (PredicateReading, Role) pair based on the fact entry that it objectifies.

To simplify construction of the overall predicate reading text, entries for ReadingText include leading or trailing spaces where relevant. In this example, the predicate reading PR1 includes front text “in “, and attaches text to three of the roles (r2, r3 and r4). So role r1 has no attached text. In this predicate reading, role r2 has the post-bound text " of note" and the following text " played “, role r3 has the pre-bound text “team “ and the following text “ for”, and role r4 has the following text “ often".
Figure 2-13  Populating the Predicate Readings metamodel with a sample model

For fine details concerning further constraints on predicate readings, see Section 3.4. For fine details on predicate reading order, see section 3.6.
2.7 Fact Type Readings

ORM Diagram

![Diagram showing FactType has * FactTypeReading](image)

<table>
<thead>
<tr>
<th>Informal derivation rule:</th>
</tr>
</thead>
<tbody>
<tr>
<td>* FactType has FactTypeReading if and only if FactTypeReading = a PredicateReading for that FactType with ObjectType terms inserted in order in the placeholders for that PredicateReading e.g. &quot;Person drives Car&quot;.</td>
</tr>
</tbody>
</table>

Uniqueness of fact type readings within the model is not automatic, so the internal uniqueness constraint is not implied.

Formal derivations are supplied in the FactTypeReadings (Derivations) topic.

Figure 2-14 Fact Type Readings

Verbalization (of newly introduced aspects)

Object Types and Reference Schemes

FactTypeReading is a value type.

Derived Fact Types

FactType has FactTypeReading.

Constraints

**Each** FactType has some FactTypeReading.

For each FactTypeReading, **at most one** FactType has that FactTypeReading.

It is possible that some FactType has more than one FactTypeReading.

Explanation

Informally, a fact type reading (e.g. "Person drives Car") may be derived from a predicate reading (e.g. "... drives ...") for that fact type by inserting the object type names (e.g. “Person”, “Car”) in the relevant object slots of that predicate reading. The formal specification of the derivation rule is complex, and is provided in section 3.5.

As indicated by the red colour, the uniqueness constraint on the derived fact type is asserted (not implied), so explicit checks need to performed to ensure that the constraint is satisfied (i.e. that each fact type reading applies to only one fact type).
2.8 World Assumption

ORM Diagram

![ORM Diagram]

```
Derivation Rule:
*UnaryFactType has WorldAssumption1 if and only if
that UnaryFactType has that explicit WorldAssumption1
or that UnaryFactType has no explicit WorldAssumption
and that WorldAssumption1 = 'ClosedWorld'.
```

Figure 2-15  World Assumption for unary fact types

Verbalization (of newly introduced aspects)

Object Types and Reference Schemes

WorldAssumption is an entity type.
Reference Scheme: WorldAssumption has WorldAssumptionName.
Reference Mode: .Name.

Asserted Fact Types

UnaryFactType has explicit WorldAssumption.

Derived Fact Types

*UnaryFactType has WorldAssumption, if and only if
that UnaryFactType has that explicit WorldAssumption,
or that UnaryFactType has no explicit WorldAssumption and that WorldAssumption, = 'ClosedWorld'.

Constraints

Each UnaryFactType has exactly one WorldAssumption.
It is possible that more than one UnaryFactType has the same WorldAssumption.

Each UnaryFactType has at most one explicit WorldAssumption.
It is possible that more than one UnaryFactType has the same explicit WorldAssumption.

The possible values of WorldAssumption are 'ClosedWorld', 'OpenWorld', 'OpenWorldWithNegation'.

Explanation

This metamodel fragment allows the modeller to indicate, for each unary fact type, how complete its fact population is with respect to the business domain being modelled. By default, unary fact types are assumed to conform to the Closed World Assumption (CWA), meaning that all facts of that type are known (i.e. appear in the model, either as asserted facts or derived facts). If an instance of the fact type is neither asserted nor derived then it is assumed to be false. The Open World Assumption (OWA) allows that our knowledge may be incomplete, so the absence of a fact does not imply that it is false. The Open World with Negation Assumption (OWN) is like OWA in allowing that some facts of that kind are unknown, but it also allows negations of fact instances to be explicitly recorded.
For example, given three people Ann, Fred and Colin, suppose that the unary fact type Person smokes is populated with exactly one fact: Fred smokes. Under CWA, we may now infer that neither Ann nor Colin smoke. Under OWA, it is unknown whether Ann or Colin smoke. Under OWN, we can also assert negations for that fact type. For example, if we now assert that Ann does not smoke, then it is only Colin whose smoking status is unknown. When mapped to attribute-based structures, such as relational tables, CWA unaries may appear as mandatory Booleans, OWA unaries as optional Booleans restricted to True, and OWN unaries as optional Booleans. For further discussion, see Halpin & Morgan (2008, secs. 10.6, 11.4).

In the metamodel fragment shown in Figure 2-15, the fact type UnaryFactType has explicit WorldAssumption allows modellers to override the CWA default for any specific unary fact type by specifying OWA or OWN for that fact type. This fact type may also be used to record that the CWA option is explicitly chosen by the modeller rather than merely accepted by default.

### 2.9 Role Classification

**ORM Diagram**

- **Role**
  - is mandatory *
  - is functional *

**Derivation Rules:**

- *Role is mandatory if and only if some AlethicMandatoryConstraint restricts that Role and has Arty=1.*
- *Role is functional if and only if some AlethicUniquenessConstraint restricts that Role and has Arty=1.*

*Figure 2-16  Role Classification*

**Derived Fact Types and Derivation Rules**

See Figure 2-16.

**Constraints**

- **In each population of Role is mandatory, each Role occurs at most once.**
- **In each population of Role is functional, each Role occurs at most once.**

**Explanation**

A role is *mandatory* if and only if it is restricted by a simple mandatory role constraint that is alethic (and hence must be satisfied). This means that for each state of the model, each instance in the population of its object type must play that role.

A role is *functional* if and only if it is restricted by a simple uniqueness constraint that is alethic. This means that for each state of the model, each instance in the role’s population appears there just once.

In FBM, the population of each fact type is always a set, not a multiset, and hence cannot contain duplicate facts. This implies that each fact type has an implied, spanning uniqueness constraint (covering all its roles in combination). If a fact type is unary, a uniqueness constraint on its role is thus implied, so there is no need to display it explicitly on a diagram by adding a uniqueness bar to the role. Although these uniqueness constraints are not displayed in Figure 2-16, they do exist, and hence are included in the constraint verbalization shown above.

In the example shown in Figure 2-17, role r1 is functional because it has an implied alethic uniqueness constraint (all FBM fact types have sets as their populations, not multisets). Roles r6 and r7 are neither functional nor mandatory because the uniqueness and mandatory constraints on them also span another role (and hence are not simple).
Figure 2-17  Populating the Role Classification metamodel with a sample model
2.10 Subtyping

**ORM Diagram**

![ORM Diagram](image)

**Figure 2-18** Subtyping

**Verbalization** (of newly introduced aspects)

**Object Types and Reference Schemes**

Subtyping is an entity type. Subtyping objectifies "ObjectType is a direct subtype of ObjectType".

Subtyping is independent (it may have instances that play no other roles).

**Asserted Fact Types**

ObjectType is a direct subtype of ObjectType. / ObjectType is a direct supertype of ObjectType.

**Constraints**

It is possible that some ObjectType is a direct subtype of more than one ObjectType and that some ObjectType is a direct supertype of more than one ObjectType.

In each population of ObjectType is a direct subtype of ObjectType, each ObjectType, ObjectType combination occurs at most once.

No ObjectType may cycle back to itself via one or more traversals through ObjectType is a direct subtype of ObjectType.

If ObjectType, is a direct subtype of some ObjectType, then it is not true that ObjectType, is indirectly related to ObjectType, by repeatedly applying this fact type.

**Explanation**

The fact type ObjectType is a direct subtype of ObjectType is used to record direct subtyping relationships. In the Figure 2-19 example, AsianWoman is a direct subtype of Woman, and Woman is a direct subtype of Person. AsianWoman is not a direct subtype of Woman (although indirectly it is a subtype of Woman).

![Subtyping example displayed using (a) an ORM notation and (b) an Euler diagram](image)
Direct subtyping relationships are displayed graphically by an arrow directed from the subtype shape to the supertype shape, as in Figure 2-19(a). For explanatory purposes, an equivalent Euler diagram is shown in Figure 2-19(b).

Asserting that Woman is a subtype of Person requires that for each state of the model, each instance in the population of Woman also occurs as an instance in the population of Person. Only proper subtyping is supported (i.e. there is a possible state of the model in which the supertype population includes an instance that is not in the subtype). Hence no object type can be a subtype of itself.

In Figure 2-18, the constraint shape attached to this direct subtyping fact type depicts a compound ring constraint that restricts the fact type to be acyclic (as verbalized in the third constraint above) and strongly intransitive (as verbalized in the fourth constraint above). Enforcement of these ring constraints involves recursion. For further technical discussion of these ring constraints, see Halpin & Curland (2011).

A domain object type is independent if and only if an instance of it may exist independently of playing any roles other than roles used in its preferred reference scheme. Formal coverage of derivation rules and constraints needed to establish independence is provided in section 3.10.

In the ORM graphic notation, domain object types that are independent have an exclamation mark "!" appended to their name. In this metamodel fragment, Subtyping is an independent entity type resulting from the objectification of the fact type ObjectType is a direct subtype of ObjectType.

Fine details on various restrictions applicable to subtyping are covered in sections 3.7, 3.8 and 3.9.
2.11 Domain Object Type: Identification

ORM Diagram

Derivation Rules:

*EntityType is indirectly identified through Subtyping if and only if that EntityType is some ObjectType1 that is a direct subtype of some ObjectType and is involved as subtype in that Subtyping that provides identification path.

*EntityType1 has identifying AlethicUniquenessConstraint if and only if that EntityType1 is directly identified by that AlethicUniquenessConstraint or is indirectly identified through some Subtyping that involves some supertype ObjectType that is some EntityType2 that has that identifying AlethicUniquenessConstraint.

*DomainObjectType1 has a simple identifier based on ValueType1 if and only if that DomainObjectType1 is ValueType1 or that DomainObjectType1 is some EntityType that has some identifying AlethicUniquenessConstraint that has Arity = 1 and restricts some Role that is hosted by some DomainObjectType2 that has a simple identifier based on that ValueType1.

*FactType is existential for EntityType if and only if that EntityType has some identifying AlethicUniquenessConstraint that restricts some Role that belongs to that FactType.

[UNDONE: This rule needs to consider joined preferred identification schemes]

*EntityType has candidate-identifying AlethicUniquenessConstraint

Figure 2-20 Domain Object Type: Identification

Verbalization (of newly introduced aspects)

**Asserted Fact Types**

Subtyping provides identification path.

EntityType is directly identified by AlethicUniquenessConstraint.

**Derived Fact Types**

See Figure 2-20.
Constraints

In each population of Subtyping provides identification path, each Subtyping occurs at most once.

Each Subtyping provides identification path for at most one EntityType.
It is possible that some EntityType is indirectly identified through more than one Subtyping.

Each EntityType is directly identified by at most one AlethicUniquenessConstraint.
Each AlethicUniquenessConstraint directly identifies at most one EntityType.

Each EntityType has exactly one identifying AlethicUniquenessConstraint.
It is possible that some AlethicUniquenessConstraint identifies more than one EntityType.

It is possible that some EntityType has more than one candidate identifying AlethicUniquenessConstraint and that for some AlethicUniquenessConstraint, more than one EntityType has that candidate identifying AlethicUniquenessConstraint.

In each population of EntityType has candidate identifying AlethicUniquenessConstraint, each EntityType, AlethicUniquenessConstraint combination occurs at most once.

It is possible that some FactType is existential for more than one EntityType and that for some EntityType, more than one FactType is existential for that EntityType.

In each population of FactType is existential for EntityType, each FactType, EntityType combination occurs at most once.

Each DomainObjectType has a simple identifier based on at most one ValueType.
It is possible that more than one DomainObjectType has a simple identifier based on the same ValueType.

For each EntityType, exactly one of the following holds:
that EntityType is directly identified by some AlethicUniquenessConstraint;
that EntityType is indirectly identified through some Subtyping.

Explanation

This metamodel fragment aims to ensure that each domain object type is well identified. Value types (e.g. CountryCode) are self-identifying. Each entity type is identified by its preferred reference scheme. This ultimately maps each entity of that type to either a single value or a combination of values, where the roles instantiated by those values are constrained by a uniqueness constraint that ensures this mapping is one-to-one. As indicated by the asserted exclusive-or constraint (coloured red in Figure 2-20) an entity type’s reference scheme either stems directly from the entity type (e.g. Country has CountryCode) or is inherited through a supertype (e.g. Woman may inherit its reference scheme from Person), but not both.

FBM supports context-dependent reference, allowing a subtype to introduce a preferred reference scheme different from that of its supertype(s).
2.12 Is Independent and Is Personal

ORM Diagram

![Diagram showing DomainObjectType and EntityType relationships]

**Note:** This figure combines aspects from two pages of the metamodel.

**Verbalization** (of newly introduced aspects)

** Asserted Fact Type **

+DomainObjectType is independent.

** Semiderived Fact Type (for derivation rule, see Figure 2-21) **

+EntityType is personal.

**Constraints**

- In each population of DomainObjectType is independent, each DomainObjectType occurs at most once.

- In each population of EntityType is personal, each EntityType occurs at most once.

**Explanation**

A domain object type is *independent* if and only if an instance of it may exist independently of playing any roles other than roles used in its preferred reference scheme (if any). On schema diagrams, independent types are displayed with an exclamation mark “!” appended to their name. For example, in a model used to simply list country codes, the value type CountryCode is independent—see Figure 2-22(a). The schema in Figure 2-22(b) includes two independent types: Course and Enrolment. By making Course independent, we can record courses whether or not any students enrol in them. By making the objectified type Enrolment independent, we can record that a student enrolled in a course before knowing what rating (if any) he or she ultimately obtains for that enrolment.

If an entity type hosts a role that is mandatory and not in its preferred reference scheme, it cannot be independent. For example, if we extend the Figure 2-22(b) schema by adding the mandatory fact type Course has CourseTitle, then Course is no longer independent. For formal coverage of derivation rules and constraints relating to independence, as well as when a mandatory constraint is implied, see section 3.10.
Figure 2.22 In these models, CountryCode, Enrolment and Course are independent object types.

The fact type EntityType is personal is used to determine the pronoun used in relative clauses for that entity type in constraint or rule verbalizations. For English verbalizations, if the type is personal, then "who" is used (e.g. Each Person who smokes is cancer-prone); otherwise "that" is used for the relative pronoun (e.g. Each Product that is defective is withdrawn from sale.)
2.13 Objectification

ORM Diagram

![Diagram](image)

Figure 2.23 Objectifying a Fact Type as a Domain Object Type

Verbalization (of newly introduced aspects)

**Asserted Fact Types**

FactType is objectified by DomainObjectType. / DomainObjectType objectifies FactType.

**Constraints**

Each FactType is objectified by **at most one** DomainObjectType.

Each DomainObjectType objectifies **at most one** FactType.

For each FactType and ObjectType, **at most one of the following holds:**

- that FactType is objectified by **some** DomainObjectType **that is that** ObjectType;
- that FactType contains **some** Role that is hosted by **that** ObjectType.

**Explanation**

As discussed earlier, in FBM, **objectification** is situational rather than propositional, so instances of the object type resulting from the objectification are in 1:1 correspondence to the relationship instances in the original fact type but are not identical to them. Diagrammatically, an object type resulting from objectification is displayed as a named, soft rectangle enclosing the predicate shape from which it was formed, as in Figure 2-22 (b).

The exclusion constraint ensures that the object type resulting from the objectification cannot be an object type in the fact type that was objectified.

Though often not displayed, implicit link fact types provide relationships to enable navigation between the object type resulting from objectification and the object type(s) involved in the fact type that was objectified. For further discussion of fine details about link fact types, see section 3.11.
2.14 Derivation

ORM Diagram

Figure 2.24  Derivation options for fact types and subtypes

Verbalization (of newly introduced aspects)

Object Types, Reference Schemes and Subtyping

InstantiationMethod is an entity type.
  Reference Scheme: InstantiationMethod has InstantiationMethodName.
  Reference Mode: .Name.

Rule is an entity type.
  Reference Scheme: Rule has RuleSID.
  Reference Mode: .SID.

DerivationRule is an entity type.
  Reference Scheme: Rule has RuleSID.
  Reference Mode: .SID.
DerivationDescription is a value type.

DerivationEvaluation is an entity type.
   Reference Scheme: DerivationEvaluation has DerivationEvaluationName.
   Reference Mode: .Name.

Subtype is an entity type.
   * Each Subtype is by definition some ObjectType, that is a direct subtype of some ObjectType.
   Reference Scheme: ObjectType has ObjectTypeSID.
   Reference Mode: .SID.

DerivedOrSemiderivedSubtype is an entity type.
   * Each DerivedOrSemiderivedSubtype is by definition some Subtype that has some InstantiationMethod where the possible values of that InstantiationMethod are 'Derived', 'Semiderived'.
   Reference Scheme: ObjectType has ObjectTypeSID.
   Reference Mode: .SID.

DerivedOrSemiderivedFactType is an entity type.
   * Each DerivedOrSemiderivedFactType is by definition some FactType that has some InstantiationMethod where the possible values of that InstantiationMethod are 'Derived', 'Semiderived'.
   Reference Scheme: FactType has FactTypeSID.
   Reference Mode: .SID.

FormalLanguage is an entity type.
   Reference Scheme: FormalLanguage has FormalLanguageName.
   Reference Mode: .Name.

RuleName is a value type.

Asserted Fact Types

   FactType has InstantiationMethod.
   Subtype has InstantiationMethod.

   DerivedOrSemiderivedFactType is derived using DerivationRule.  
      /  DerivationRule derives DerivedOrSemiderivedFactType.

   DerivedOrSemiderivedSubtype is derived using DerivationRule.  
      /  DerivationRule derives DerivedOrSemiderivedSubtype.

   DerivationRule has standard DerivationEvaluation.
   DerivationRule has custom DerivationEvaluation.
   DerivationRule has DerivationDescription.

   Rule is expressed in FormalLanguage.
   Rule has RuleName.

Constraints

   Each FactType has exactly one InstantiationMethod,
   It is possible that more than one FactType has the same InstantiationMethod.

   Each Subtype has exactly one InstantiationMethod.
   It is possible that more than one Subtype has the same InstantiationMethod.

   It is possible that some DerivedOrSemiderivedFactType is derived using more than one DerivationRule and that some DerivationRule derives more than one DerivedOrSemiderivedFactType.
In each population of DerivedOrSemiderivedFactType is derived using DerivationRule, each DerivedOrSemiderivedFactType, DerivationRule combination occurs at most once.

Each DerivedOrSemiderivedFactType is derived using some DerivationRule.

It is possible that some DerivedOrSemiderivedFactType is derived using more than one DerivationRule and that some DerivationRule derives more than one DerivedOrSemiderivedFactType.

In each population of DerivedOrSemiderivedFactType is derived using DerivationRule, each DerivedOrSemiderivedFactType, DerivationRule combination occurs at most once.

Each DerivedOrSemiderivedFactType is derived using some DerivationRule.

Each DerivationRule has at most one standard DerivationEvaluation. It is possible that some DerivationRule has the same standard DerivationEvaluation.

Each DerivationRule has at most one custom DerivationEvaluation. It is possible that more than one DerivationRule has the same custom DerivationEvaluation.

Each DerivationRule has at most one DerivationDescription. It is possible that more than one DerivationRule has the same DerivationDescription.

The possible values of InstantiationMethod are 'Asserted', 'Derived', 'Semiderived'.

The possible values of standard DerivationEvaluation in DerivationRule has standard DerivationEvaluation are 'Immediate', 'OnDemandCached', 'OnDemandUncached'.

For each DerivationRule, exactly one of the following holds: that DerivationRule has some standard DerivationEvaluation; that DerivationRule has some custom DerivationEvaluation.

It is possible that some Rule is expressed in more than one FormalLanguage and that for some Rule, more than one Rule is expressed in that FormalLanguage.

In each population of Rule is expressed in FormalLanguage, each Rule, FormalLanguage combination occurs at most once.

Each Rule has at most one RuleName.

Each RuleName is of at most one Rule.

For each Rule, some DerivationRule that is that Rule has some DerivationDescription or that Rule is expressed in some FormalLanguage.

Each Rule is some DerivationRule or has some RuleName.

Explanation

The choice of instantiation method determines how instances of fact types or subtypes are added to a model's population. A fact type or subtype is asserted if and only if all its population instances must be asserted. It is derived if and only if all its population instances must be derived from other information using a derivation rule. It is semiderived if and only if it is possible for some of its population instances to be asserted and some instances of its population to be derived.

On diagrams, derived fact types and derived subtypes are indicated by appending an asterisk "*" to their name. Semiderived fact types and semiderived subtypes are displayed with a plus superscript "+" appended to their name. For example, in Figure 2-25 the parenthood and gender fact types are asserted, and the fact type Person is male is derived from gender information using the first derivation rule shown. In this example, parenthood knowledge may be incomplete, so we might know that somebody is a grandparent of someone without knowing who the intermediate parent is. For cases where we do know
the relevant parenthood facts, we can derive grandparent facts using the second derivation rule shown. Since the fact type Person is a grandparent of Person can have some asserted instances and some derived instances, it is semiderived.

![Diagram of grandparent relationship]

**Figure 2-25 Sample model with asserted, derived, and semiderived fact types**

In Figure 2-26 the Doctor subtype is simply asserted, the MalePerson subtype is derived from gender facts using the first derivation rule shown. The Grandparent subtype is semiderived, allowing that we may simply assert that someone is a grandparent without knowing the relevant parenthood details, and we may infer its derived instances using the second derivation rule shown when the relevant parenthood facts are available.

![Diagram of Doctor, MalePerson, and Grandparent subtypes]

**Figure 2-26 Sample model with asserted, derived, and semiderived subtypes**

The choice of **derivation evaluation** determines when the derivation is performed. Here, “Immediate” means derive-on-update (eager evaluation), so as soon as the relevant facts in the rule body are added the derivation is performed. “OnDemandUncached” means simple derive-on-query (lazy evaluation), so the derivation is performed only when the user requests it.

“OnDemandCached” (lazy cache evaluation) means that derivation is performed only when requested, but the result is stored in cache memory for later derivation requests in the current session; if updates are made to the underlying facts used to derive them, the formerly derived facts are either deleted or marked as “dirty” (outdated). A subsequent request for the derived data is processed as follows: if the derived data is stored (and not dirty) it is simply read; otherwise the derived data is recalculated and stored.

While these three derivation evaluation options are standardly allowed in the metamodel, the fact type DerivationRule has custom DerivationEvaluation allows tool vendors to support further options in this regard.

The Rule entity is meant to encompass a completely general rule definition, including rules that can recalculate a population from the static state of a model (represented by the DerivationRule subtype) more general rules, such as rules that create new entities (including top level object types) and facts in response to model changes.

The structure of a rule is undefined. The expectation is for extensions to define a specific structure corresponding to a named formal language, then attach additional elements to derived subtypes of Rule and Derivation that match the given language. The design allows for multiple formal languages on one rule. For example, a single rule can contain both a textual rule representation (in one formal language) and the parse tree corresponding to that textual language (represented as an abstract syntax tree with a different formal language).
2.15 Data Type and Data Value

ORM Diagrams

Derivation Rules:

* ValueType bases value restrictions on DataType1 if and only if
  that ValueType maps to that DataType1
  or maps to some DataType2 that is a subtype of that DataType1.

* ObjectType1 bases value restrictions on DataType1 if and only if
  that ObjectType1 is some DomainObjectType that has a simple identifier based on some ValueType
  that bases value restrictions on that DataType1
  or that ObjectType1 is some DataType1
  and is a subtype of that DataType1.

* ObjectType1 allows values from DataType1 if and only if
  that ObjectType1 is some DomainObjectType that has a simple identifier based on some ValueType
  that maps to some DataType2 that = DataType1
  or that DataType2 is a supertype of that DataType1
  or that ObjectType1 is DataType1
  or that ObjectType1 is a supertype of that DataType1.

* DataType is toposm if and only if
  it is not true that that DataType is some ObjectType1 that is a direct subtype of some ObjectType.

* ObjectType1 can restrict values if and only if
  that ObjectType1 is some DataType
  and is a direct subtype of some ObjectType
  or that ObjectType1 is some DomainObjectType that has a simple identifier based on some ValueType.

Canonical data values are expected to be stored with the default language (en-US) and
a standardized default lexical value obtained by using the standard string conversion for
the specified data type. Data type parsing and string formatting operations are outside
the scope of this document, but are well-defined by the external standards used for
top-level data types in this standard.

Derivation Rules:

* CanonicalDataValue is expressed in LanguageCulture if and only if
  that CanonicalDataValue is some DataValue that is expressed in that LanguageCulture.

Figure 2-27  Data Type and Data Value

Note: This figure combines two pages of the metamodel
Verbalization (of newly introduced aspects)

Object Types, Reference Schemes and Subtyping

DataValue is an entity type.
Reference Scheme: DataValue is of DataType; DataValue is represented by LexicalValue; DataValue is expressed in LanguageCulture.

CanonicalDataValue is an entity type.
Each CanonicalDataValue is an instance of DataValue.
Reference Scheme: DataValue is of DataType; DataValue is represented by LexicalValue; DataValue is expressed in LanguageCulture.

LexicalValue is a value type.

LanguageCulture is an entity type.
Reference Scheme: LanguageCulture has LanguageCultureCode.

Asserted Fact Types

DataType has standard ObjectType.
DataType is of DataType.
DataType is represented by LexicalValue.
DataType is expressed in LanguageCulture.

Derived Fact Types (see Figure 2-27 for the derivation rules)

*DataType is topmost
*ValueType bases value restrictions on DataType
*ObjectType allows values from
*DataType can restrict values
*CanonicalDataValue is expressed in LanguageCulture

Constraints

It is possible that some ValueType bases value restrictions on more than one DataType and that for some DataType, more than one ValueType bases value restrictions on that DataType.

In each population of ValueType bases value restrictions on DataType, each ValueType, DataType combination occurs at most once.

It is possible that some ObjectType bases value restrictions on more than one DataType and that for some DataType, more than one ObjectType bases value restrictions on that DataType.

In each population of ObjectType bases value restrictions on DataType, each ObjectType, DataType combination occurs at most once.

It is possible that some ObjectType allows values from more than one DataType and that for some DataType, more than one ObjectType allows values from that DataType.

In each population of ObjectType allows values from DataType, each ObjectType, DataType combination occurs at most once.

The possible values of ObjectType.standardDataTypeName are 'int32', 'float', 'double', 'uint32', 'etc'.

Each DataType has at most one standard ObjectType.
For each standard ObjectTypeName, at most one DataType has that standard ObjectTypeName.

If some DataType is topmost then that DataType has some standard ObjectTypeName.

Each DataValue is of exactly one DataType.
It is possible that more than one DataValue is of the same DataType.

Each DataValue is represented by exactly one LexicalValue.
It is possible that more than one DataValue is represented by the same LexicalValue.

Each DataValue is expressed in exactly one LanguageCulture.
It is possible that more than one DataValue is expressed in the same LanguageCulture.
Each CanonicalDataValue is expressed in exactly one LanguageCulture.
It is possible that more than one CanonicalDataValue is expressed in the same LanguageCulture.

The value of LanguageCulture in CanonicalDataValue is expressed in LanguageCulture is 'en-US'.

For each DataType, LexicalValue, and LanguageCulture, at most one DataValue is of that DataType and
is represented by that LexicalValue and
is expressed in that LanguageCulture.

This association with DataType, LexicalValue, LanguageCulture provides the preferred identification scheme for DataValue.

Explanation

An example of a data value is the number 33.7, which is identified as (a) being of datatype Integer, (b) being represented by the lexical value “33.7” (a character string), expressed in US English.

Regardless of the user’s locale, data values are stored in canonical form, using the default language (en-US) and a standardized default lexical value obtained by using the standard string conversion for the specified data type. It is the responsibility of the software tool to display values using the user’s locale (e.g., “33.7” is rendered as “33,7” in France).

Datatype parsing and string formatting operations are outside the scope of this document, but are well-defined by the external standards used for top-level data types in this standard.

Related details concerning datatype facets and comparison operations may be found in sections 3.12 through 3.15.
2.16 Constraints

ORM Diagram

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Reference Scheme</th>
<th>Reference Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint is an entity type.</td>
<td>Constraint has ConstraintSID.</td>
<td>.SID.</td>
</tr>
<tr>
<td>Modality is an entity type.</td>
<td>Modality has ModalityName.</td>
<td>.Name.</td>
</tr>
<tr>
<td>ConstraintName is a value type.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MandatoryConstraint is an entity type.</td>
<td>Each MandatoryConstraint is an instance of Constraint.</td>
<td>.SID.</td>
</tr>
<tr>
<td>UniquenessConstraint is an entity type.</td>
<td>Each UniquenessConstraint is an instance of Constraint.</td>
<td>.SID.</td>
</tr>
<tr>
<td>FrequencyConstraint is an entity type.</td>
<td>Each FrequencyConstraint is an instance of Constraint.</td>
<td>.SID.</td>
</tr>
</tbody>
</table>

Constraints on derived fact types are typically derivable, but determination of this for specific cases is out of scope. We allow the user to manually declare that such a constraint is asserted, but this metamodel does not require that checks be performed on the correctness of such declarations.

Figure 2-28  Constraints

Verbalization (of newly introduced aspects)

Object Types, Reference Schemes and Subtyping
RingConstraint is an entity type.
Each RingConstraint is an instance of Constraint.
Reference Scheme: Constraint has ConstraintSID.
Reference Mode: .SID.

ValueComparisonConstraint is an entity type.
Each ValueComparisonConstraint is an instance of Constraint.
Reference Scheme: Constraint has ConstraintSID.
Reference Mode: .SID.

SetComparisonConstraint is an entity type.
Each SetComparisonConstraint is an instance of Constraint.
Reference Scheme: Constraint has ConstraintSID.
Reference Mode: .SID.

EqualityConstraint is an entity type.
Each EqualityConstraint is an instance of SetComparisonConstraint.
Reference Scheme: Constraint has ConstraintSID.
Reference Mode: .SID.

ExclusionConstraint is an entity type.
Each ExclusionConstraint is an instance of SetComparisonConstraint.
Reference Scheme: Constraint has ConstraintSID.
Reference Mode: .SID.

SubsetConstraint is an entity type.
Each SubsetConstraint is an instance of SetComparisonConstraint.
Reference Scheme: Constraint has ConstraintSID.
Reference Mode: .SID.

GeneralConstraint is an entity type.
Each GeneralConstraint is an instance of Constraint.
Reference Scheme: Constraint has ConstraintSID.
Reference Mode: .SID.

ValueConstraint is an entity type.
Each ValueConstraint is an instance of Constraint.
Reference Scheme: Constraint has ConstraintSID.
Reference Mode: .SID.

CardinalityConstraint is an entity type.
Each CardinalityConstraint is an instance of Constraint.
Reference Scheme: Constraint has ConstraintSID.
Reference Mode: .SID.

Asserted Fact Types

Constraint has Modality.
Constraint has ConstraintName.
Constraint is asserted on a derived fact type.

Constraints

Each Constraint has exactly one Modality.
It is possible that more than one Constraint has the same Modality.
Each Constraint has at most one ConstraintName.
Each ConstraintName is of at most one Constraint.

In each population of Constraint is asserted on a derived fact type, each Constraint occurs at most once.

The possible values of Modality are 'Alethic', 'Deontic'.

For each Constraint, exactly one of the following holds:
  that Constraint is some MandatoryConstraint;
  that Constraint is some UniquenessConstraint;
  that Constraint is some FrequencyConstraint;
  that Constraint is some RingConstraint;
  that Constraint is some ValueComparisonConstraint;
  that Constraint is some SetComparisonConstraint;
  that Constraint is some ValueConstraint;
  that Constraint is some CardinalityConstraint;
  that Constraint is some GeneralConstraint.

For each SetComparisonConstraint, exactly one of the following holds:
  that SetComparisonConstraint is some EqualityConstraint;
  that SetComparisonConstraint is some ExclusionConstraint;
  that SetComparisonConstraint is some SubsetConstraint.

Explanation

If a constraint has alethic modality, each state of the information model must satisfy the constraint (e.g. Each Person as born on at most one Date). So any attempt to update the model population in a way that would violate an alethic constraint must be rejected.

If a constraint has deontic modality, it ought to be obeyed but may be violated (e.g. Each Person is wife of at most one Person). If an update to the model population violates a deontic constraint, the update is accepted so that the violation is recorded. Ideally, an action should then be taken to discourage such violations in the future. Consideration of such actions is outside the scope of this metamodel.

Constraints on asserted fact types are always asserted. In typical models, most constraints on derived fact types are derivable from other constraints. Although automatic determination of such derivability is outside the scope of this metamodel, the ability to assert constraints on derived fact types is extremely convenient, as it often leads to much simpler constraint formulations for complex cases. The fact type Constraint is asserted on a derived fact type is included to allow users to manually and explicitly indicate when a constraint on a derived fact type is asserted. In this specification, the graphical symbols for asserted constraints on derived fact types are coloured red to distinguish them from other constraints.

The various categories of constraints are discussed in later sections.
2.17 Alethic Constraints

ORM Diagram

![ORM Diagram]

**Verbalization** (of newly introduced aspects)

**Object Types, Reference Schemes and Subtyping**

- **AlethicMandatoryConstraint** is an entity type.
  - *Each AlethicMandatoryConstraint is by definition a MandatoryConstraint that has Modality 'Alethic'.
  - **Reference Scheme:** Constraint has ConstraintSID.
  - **Reference Mode:** .SID.

- **AlethicUniquenessConstraint** is an entity type.
  - *Each AlethicUniquenessConstraint is by definition a UniquenessConstraint that has Modality 'Alethic'.
  - **Reference Scheme:** Constraint has ConstraintSID.
  - **Reference Mode:** .SID.

Similar formulations apply to the other constraint subtypes in Figure 2-29.

**Explanation**

Subtypes for the alethic versions of mandatory role and uniqueness constraints are used elsewhere in the metamodel. The other subtypes are listed merely for completeness. Subtypes for deontic versions are not used elsewhere in the metamodel, so are not introduced here.
2.18 Mandatory Role Constraints

ORM Diagram

Figure 2-30  Mandatory Role Constraints

Note: The fact type Role is compatible with Role is defined in section 3.9, so is not elaborated here, other than to note that roles are compatible just in case they are played by compatible object types.

Verbalization (of newly introduced aspects)

Asserted Fact Types

MandatoryConstraint restricts Role.

Derived Fact Types

*MandatoryConstraint has Arity if and only if
Arity = count(each Role that is restricted by that MandatoryConstraint).

*MandatoryConstraint is a simple mandatory constraint if and only if
that MandatoryConstraint has Arity = 1.

*MandatoryConstraint is an inclusive-or constraint if and only if
that MandatoryConstraint has Arity >= 2.

*Role1 is in inclusive-or constraint with Role2 if and only if
some MandatoryConstraint restricts that Role1
and restricts that Role2
where Role1 <> Role2.
Constraints

- It is possible that some MandatoryConstraint restricts more than one Role and that some Role is restricted by more than one MandatoryConstraint.
- In each population of MandatoryConstraint restricts Role, each MandatoryConstraint, Role combination occurs at most once.
- Each MandatoryConstraint has exactly one Arity.
- It is possible that more than one MandatoryConstraint has the same Arity.
- It is possible that for some Role, more than one Role is in inclusive-or constraint with that Role, and that some Role is in inclusive-or constraint with more than one Role.
- In each population of Role is in inclusive-or constraint with Role, each Role, Role combination occurs at most once.
- In each population of MandatoryConstraint is a simple mandatory constraint, each MandatoryConstraint occurs at most once.
- In each population of MandatoryConstraint is an inclusive-or constraint, each MandatoryConstraint occurs at most once.
- For each MandatoryConstraint, exactly one of the following holds:
  - that MandatoryConstraint is a simple mandatory constraint;
  - that MandatoryConstraint is an inclusive-or constraint.
- If some Role, is in inclusive-or constraint with some Role, then that Role, is compatible with that Role.

Explanation

In the metamodel, the term “mandatory constraint” is used as shorthand for “mandatory role constraint”. A simple mandatory role constraint applies to exactly one role, and requires that, for each state of the model, each instance in the population of the object type that hosts that role must play that role. Graphically, this is depicted by a large dot placed at either end of the line connecting the role shape to the object type shape.

An inclusive-or constraint (also called a disjunctive mandatory role constraint). Applies to a set of two or more roles, and requires that each instance in the population of the relevant object type must play at least one of the constrained roles. Typically each role in the inclusive-or constraint is hosted by the same object type, but the constraint may also be applied to roles of different object types only if those object types are compatible. Graphically, an inclusive-or constraint is depicted by a circled dot connected by dashed lines to the shapes for the constrained roles.

The example model in Figure 2-31 includes three mandatory role constraints. One of these is a simple mandatory role constraint (Each Academic holds some Degree). The top inclusive-or constraint applies to roles hosted by the same object type (Each Academic is intelligent or is industrious). The “or” is interpreted as an inclusive-or, allowing that an academic may be both intelligent and industrious. The other inclusive-or constraint applies to roles hosted by compatible, but not identical, object types (For each Academic, that Academic teaches some Course or some Professor who is that Academic chairs some Department).

Figure 2-31  Examples of Mandatory Role Constraints
2.19 Exclusive-Or Constraints

**ORM Diagram**

![ ORM Diagram of Exclusive-Or Constraints ]

**Derivation Rule:**  
*MandatoryConstraint* can form exclusive-or constraint with *ExclusionConstraint* if and only if  
that *MandatoryConstraint* has some *Arity* and has some *Modality*  
and that *ExclusionConstraint* has some *NrArguments* and has *ArgumentLength = 1*  
and compares no *SetComparisonArgument* that includes some *Role* at some *Position*  
where it is not true that that *Role* is restricted by that *MandatoryConstraint*  
and that *ExclusionConstraint* has *Modality*  
where *Arity = NrArguments*.

**Figure 2.32**  Exclusive-Or Constraints

**Verbalization** (of newly introduced aspects)

**Asserted Fact Types**

-MandatoryConstraint forms exclusive-or constraint with ExclusionConstraint.  
/ ExclusionConstraint forms exclusive-or constraint with MandatoryConstraint.

**Derived Fact Types**

*MandatoryConstraint* can form exclusive-or constraint with *ExclusionConstraint* if and only if  
that *MandatoryConstraint* has some *Arity* and has some *Modality*  
and that *ExclusionConstraint* has some *NrArguments* and has some *ArgumentLength = 1*  
and compares no *SetComparisonArgument* that includes some *Role* at some *Position*  
where it is not true that that *Role* is restricted by that *MandatoryConstraint*  
and that *ExclusionConstraint* has *Modality*  
where *Arity = NrArguments*.

**Constraints**

- Each *MandatoryConstraint* forms exclusive-or constraint with at most one *ExclusionConstraint*.  
- Each *ExclusionConstraint* forms exclusive-or constraint with at most one *MandatoryConstraint*.  
- Each *MandatoryConstraint* can form exclusive-or constraint with at most one *ExclusionConstraint*.  
- Each *ExclusionConstraint* can form exclusive-or constraint with at most one *MandatoryConstraint*.  
- If some *MandatoryConstraint* forms exclusive-or constraint with some *ExclusionConstraint*  
then that *MandatoryConstraint* can form exclusive-or constraint with that *ExclusionConstraint*.

**Explanation**

An exclusive-or constraint is the combination of an inclusive-or constraint and an exclusion constraint applied to the same roles. The derivation rule and the asserted subset constraint specify the conditions needed for these two component constraints to be legally combined.
2.20 Uniqueness Constraints

**ORM Diagram**

![ORM Diagram](image)

**Derivation Rules:**

*UniquenessConstraint restricts Role if and only if that UniquenessConstraint restricts that Role at some Position.

*UniquenessConstraint has Arity if and only if Arity = count(each Role that is restricted by that UniquenessConstraint).

*UniquenessConstraint is internal if and only if that UniquenessConstraint restricts some Role1 that belongs to some FactType1 and it is not true that that UniquenessConstraint restricts some Role2 that belongs to some FactType2 where FactType1 <> FactType2.

*FactType has spanning UniquenessConstraint if and only if that UniquenessConstraint is internal and has some Arity that is of that FactType.

**Figure 2-33 Uniqueness Constraints**

**Verbalization** (of newly introduced aspects)

**Asserted Fact Types**

UniquenessConstraint restricts Role at Position.

**Derived Fact Types**

*UniquenessConstraint restricts Role / Role is restricted by UniquenessConstraint if and only if that UniquenessConstraint restricts that Role at some Position.

*UniquenessConstraint has Arity if and only if Arity = count(each Role that is restricted by that UniquenessConstraint).

*UniquenessConstraint is internal if and only if that UniquenessConstraint restricts some Role1 that belongs to some FactType1 and it is not true that that UniquenessConstraint restricts some Role2 that belongs to some FactType2 where FactType1 <> FactType2.

*FactType has spanning UniquenessConstraint if and only if that UniquenessConstraint is internal and has some Arity that is of that FactType.
Constraints

Each UniquenessConstraint restricts some Role at some Position.
For each UniquenessConstraint and Role,
  that UniquenessConstraint restricts that Role at at most one Position.
For each UniquenessConstraint and Position,
  that UniquenessConstraint restricts at most one Role at that Position.

It is possible that some UniquenessConstraint restricts more than one Role
and that some Role is restricted by more than one UniquenessConstraint.

In each population of UniquenessConstraint restricts Role,
  each UniquenessConstraint, Role combination occurs at most once.

Each UniquenessConstraint restricts some Role.
Each UniquenessConstraint has exactly one Arity.

It is possible that more than one UniquenessConstraint has the same Arity.

In each population of UniquenessConstraint is internal,
  each UniquenessConstraint occurs at most once.

Each FactType has at most one spanning UniquenessConstraint.
Each UniquenessConstraint spans at most one FactType.

Explanation

From a purely business semantics perspective, a uniqueness constraint applies to a set of roles. For the exchange metamodel however, we require the constrained roles to be ordered, so each uniqueness constraint applies to an ordered set of roles. This enables the same role order to be maintained as models are moved back and forth between modeling tools, and also addresses performance aspects such as the order in which index components are ordered when mapping to relational database structures.

An internal uniqueness constraint applies only to roles from a single fact type, and is displayed as a bar over the constrained roles. An internal uniqueness constraint that applies to the combination of all the roles of the constrained fact type is said to be a spanning uniqueness constraint (since it spans all the roles).

An external uniqueness constraint applies to roles from two or more fact types, and is displayed as a circled bar connected by dashed lines to the constrained roles.

Figure 2-1 displays four uniqueness constraints, labelled C1..C4 for easy reference. Constraints C1, C2 and C3 are internal uniqueness constraints. Constraint C4 is an external uniqueness constraint, ensuring that each combination of software name (e.g. “Microsoft Word”) and version number (e.g. “2010”) applies to at most one software product. Constraint C4 is a spanning uniqueness constraint, allowing a person to use many software products, and many people to use the same software product, so only the combination of person and software product need be unique when populating the Person uses SoftwareProduct fact type. Constraints C2 and C3 are each simple, applying to just one role of their binary fact types, so each software product has only one name and only one version number.
2.21 Frequency Constraints

ORM Diagram

Verbalization (of newly introduced aspects)

Object Types and Reference Schemes

FrequencyRange is an entity type.
Reference Scheme:
FrequencyConstraint allows FrequencyRange;
FrequencyRange has minimum FrequencyBound;
FrequencyRange has maximum FrequencyBound.

FrequencyBound is a value type.

Asserted Fact Types

FrequencyConstraint allows FrequencyRange. / FrequencyRange is allowed in FrequencyConstraint.
FrequencyConstraint allows discrete FrequencyBound.
FrequencyConstraint restricts Role. / Role is restricted by FrequencyConstraint.
FrequencyRange has minimum FrequencyBound.
FrequencyRange has maximum FrequencyBound.

Derived Fact Types

*FrequencyConstraint has Arity if and only if
Arity = count(each Role that is restricted by that FrequencyConstraint).

Constraints

It is possible that some FrequencyConstraint restricts more than one Role
and that some Role is restricted by more than one FrequencyConstraint.
In each population of FrequencyConstraint restricts Role,
each FrequencyConstraint, Role combination occurs at most once.
Each FrequencyConstraint restricts some Role.
Each FrequencyConstraint has exactly one Arity.

It is possible that more than one FrequencyConstraint has the same Arity.

It is possible that some FrequencyConstraint allows more than one discrete FrequencyBound and that for some FrequencyBound, more than one FrequencyConstraint allows that discrete FrequencyBound.

In each population of FrequencyConstraint allows discrete FrequencyBound, each FrequencyConstraint, FrequencyBound combination occurs at most once.

Each FrequencyRange is allowed in exactly one FrequencyConstraint.

It is possible that some FrequencyConstraint allows more than one FrequencyRange.

Each FrequencyRange has at most one minimum FrequencyBound.

It is possible that more than one FrequencyRange has the same minimum FrequencyBound.

Each FrequencyRange has at most one maximum FrequencyBound.

It is possible that more than one FrequencyRange has the same maximum FrequencyBound.

Each FrequencyConstraint allows some FrequencyRange or allows some discrete FrequencyBound.

Each FrequencyRange has some minimum FrequencyBound or has some maximum FrequencyBound.

For each FrequencyConstraint, minimum FrequencyBound, and maximum FrequencyBound, at most one FrequencyRange is allowed in that FrequencyConstraint and has that minimum FrequencyBound, and has that maximum FrequencyBound.

This association with FrequencyConstraint, FrequencyBound, FrequencyBound provides the preferred identification scheme for FrequencyRange.

The possible values of FrequencyBound are at least 1.

Each value of FrequencyBound in "FrequencyRange has maximum FrequencyBound" is at least 2.

Explanation

A frequency constraint (also called an occurrence frequency constraint) applies to a set of one or more roles, selected from one or more fact types, and is used to restrict, for each state of the fact base, the number of times each entry in the fact population for that role (or role combination) may occur in that population. The number restriction is composed of one or more component number restrictions, each of which is either a discrete integer that is greater than 1 (unlike uniqueness constraints where the number of occurrences is restricted to 1), or a range of numbers with only a minimum (e.g. >= 2), only a maximum (e.g. <= 5), or both a minimum and maximum (e.g. 1..7).

Diagrammatically, a frequency constraint is displayed by attaching its frequency restriction to the constrained role(s). Figure 2-36 shows six simple frequency constraint examples, labelled C1..C6, each of which constrains a single role. The discrete value frequency constraints C1 and C2 ensure that each duet includes at most 2 persons (the mandatory role constraint strengthens this to exactly 2 persons), and that each person who is recorded as the child of someone has both parents recorded (so each person has either 0 or 2 parents recorded). Unlike UML multiplicity constraints, frequency constraints apply to existing role entries, so a frequency constraint of 0 is impossible. For example, in Figure 2-36, frequency constraint C2 on the child role is equivalent to a UML multiplicity constraint of 0..2 on the other (parent) role.
The range frequency constraints $C3$, $C4$, $C5$ and $C6$, ensure that each person is assigned to review at most 5 books and speaks at most 5 languages, each book is reviewed by at least 3 persons and is translated into at most 5 languages. In this metamodel fragment, the term “FrequencyRange” means “frequency range occurrence”. For example, the $C5$ and $C6$ “1..5” frequency constraints involve two different frequency ranges in this sense.

As a partial illustration of populating the metamodel fragment, FrequencyConstraint $C1$ allows discrete FrequencyBound 2, and there exists the FrequencyRange that allows FrequencyConstraint $C6$ and has minimum FrequencyBound 1 and has maximum FrequencyBound 5. The frequency range for constraint $C3$ has maximum FrequencyBound 5, but no minimum bound, and the frequency range for constraint $C4$ has minimum FrequencyBound 3 but no maximum bound.

Though rarely used, frequency constraint combinations are allowed, e.g. “2, 3, 5..7, 9, 12..15”. For examples of multi-role frequency constraints and external frequency constraints see Annex A or Halpin & Morgan (2008).
2.22 Ring Constraints

ORM Diagram

Figure 2-37 Ring Constraints

Verbalization (of newly introduced aspects)

Asserted Fact Types

- RingConstraint is irreflexive.
- RingConstraint is antisymmetric.
- RingConstraint is asymmetric.
- RingConstraint is transitive.
- RingConstraint is strongly intransitive.
- RingConstraint is acyclic.
- RingConstraint is reflexive.
- RingConstraint is symmetric.
- RingConstraint is intransitive.
- RingConstraint restricts first Role.
- RingConstraint restricts second Role.

Constraints

In each population of RingConstraint is irreflexive, each RingConstraint occurs at most once.
In each population of RingConstraint is antisymmetric, each RingConstraint occurs at most once.
In each population of RingConstraint is asymmetric, each RingConstraint occurs at most once.
In each population of RingConstraint is transitive, each RingConstraint occurs at most once.
In each population of RingConstraint is strongly intransitive, each RingConstraint occurs at most once.
In each population of RingConstraint is reflexive, each RingConstraint occurs at most once.
In each population of RingConstraint is symmetric, each RingConstraint occurs at most once.
In each population of RingConstraint is intransitive, each RingConstraint occurs at most once.

Each RingConstraint restricts exactly one first Role.
It is possible that more than one RingConstraint restricts the same first Role.
Each RingConstraint restricts exactly one second Role. 
It is possible that more than one RingConstraint restricts the same second Role.

Each RingConstraint is irreflexive or is antisymmetric or is asymmetric or is transitive or is strongly intransitive or is acyclic or is reflexive or is symmetric or is intransitive.

For each RingConstraint and Role, at most one of the following holds:
that RingConstraint restricts that first Role;
that RingConstraint restricts that second Role.

If for some Role₁, some RingConstraint restricts that first Role₁ and restricts some second Role₂ then that Role₁ is compatible with that Role₂.

Explanation

Ring constraints are formalized here using the symbols shown for logical operators and quantifiers. The definitions are given for a binary ring predicate $R$. In general, ring constraints may apply to a compatible role pair, in which case each binary relation that populates the role pair obeys the definition given.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Reading</th>
<th>Operation Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>not</td>
<td>Negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>and</td>
<td>Conjunction</td>
</tr>
<tr>
<td>∨</td>
<td>inclusive-or</td>
<td>Inclusive disjunction</td>
</tr>
<tr>
<td>→</td>
<td>only if</td>
<td>Material implication</td>
</tr>
<tr>
<td>←</td>
<td>if</td>
<td>Converse of Material Implication</td>
</tr>
<tr>
<td>∀</td>
<td>for each</td>
<td>Universal Quantifier</td>
</tr>
<tr>
<td>∃</td>
<td>there exists some</td>
<td>Existential Quantifier</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>reflexive</td>
<td>A ring predicate $R$ is locally reflexive if and only if $\forall x \forall y (xRy \rightarrow xRx)$</td>
</tr>
<tr>
<td>irreflexive</td>
<td>$\forall x \sim xRx$</td>
</tr>
<tr>
<td>symmetric</td>
<td>$\forall x \forall y (xRy \rightarrow yRx)$</td>
</tr>
<tr>
<td>asymmetric</td>
<td>$\forall x \forall y (xRy \rightarrow \sim yRx)$</td>
</tr>
<tr>
<td>antisymmetric</td>
<td>$\forall x \forall y [(xRy &amp; x \neq y) \rightarrow \sim yRx]$</td>
</tr>
<tr>
<td>transitive</td>
<td>$\forall x \forall y (xRy &amp; yRz) \rightarrow xRz$</td>
</tr>
<tr>
<td>intransitive</td>
<td>$\forall x \forall y (xRy &amp; yRz) \rightarrow \sim xRz$</td>
</tr>
<tr>
<td>strongly intransitive</td>
<td>A ring predicate $R$ is strongly intransitive if and only if $\forall x \forall y \forall z [(xRy &amp; yPz) \rightarrow \sim xRz], where the predicate $P$ is recursively defined to give the transitive closure of $R$, i.e. $\forall x \forall y [xPy \leftarrow (xRy \vee \exists z (xRz &amp; zPy))]$</td>
</tr>
<tr>
<td>acyclic</td>
<td>A ring predicate $R$ is acyclic if and only if its transitive closure is irreflexive, i.e. $\forall x \sim xPx$ where $P$ is defined thus $\forall x \forall y [xPy \leftarrow (xRy \vee \exists z (xRz &amp; zPy))]$</td>
</tr>
</tbody>
</table>
Note that reflexivity is defined here as *local* reflexivity, which is much more useful than either global reflexivity or reflexivity over a set. The graphical notation for the various individual ring constraints is summarized in Annex A. For coverage of which combinations of ring constraints are allowed, as well as derivation rules for some implications between ring constraints, see section 3.16. Detailed explanations and examples of most ring constraints are provided in Halpin & Morgan (2008). For further technical discussion, especially of recently introduced ring constraints such as strong intransitivity, see Halpin & Curland (2011).
2.23 Value-Comparison Constraints

ORM Diagram

![Diagram of Value-Comparison Constraints]

Figure 2.38  Value-Comparison Constraints

Verbalization (of newly introduced aspects)

**Asserted Fact Types**

ValueComparisonConstraint compares first Role.
ValueComparisonConstraint compares second Role.
ValueComparisonConstraint compares values with StandardComparator.

**Constraints**

Each ValueComparisonConstraint compares exactly one first Role.
It is possible that more than one ValueComparisonConstraint compares the same first Role.

Each ValueComparisonConstraint compares exactly one second Role.
It is possible that more than one ValueComparisonConstraint compares the same second Role.

Each ValueComparisonConstraint compares values with exactly one StandardComparator.
It is possible that more than one ValueComparisonConstraint compares values with the same StandardComparator.

For each ValueComparisonConstraint and Role, at most one of the following holds:
- that ValueComparisonConstraint compares that first Role;
- that ValueComparisonConstraint compares that second Role.

If for some Role₁, some ValueComparisonConstraint compares that first Role₁, and that ValueComparisonConstraint compares some second Role₂, then that Role₁ is value comparable with that Role₂.

**Explanation**

A value-comparison constraint compares two values using a standard comparator (standard comparison operator) such as “<” (is less than) or “≤” (is than or equal to), where these values instantiate two compatible roles projected from either a join path (see section 2.31) or the same fact type. On a diagram, these constraints are displayed by a circle with two dots (indicating the values) enclosing the comparator, connected to the relevant roles by dashed lines, with an arrow tip at one end indicating the direction of the comparison.
The value-comparison constraint $C_1$ in Figure 2-39 ensures that the birthdate of each dead person occurs on or before his/her deathdate. This constraint may be captured in the metamodel by the following facts: ValueComparisonConstraint $C_1$ compares first Role $r_1$; ValueComparisonConstraint $C_1$ compares second Role $r_2$; ValueComparisonConstraint $C_1$ compares values with StandardComparator ‘LessThanOrEqual’; Role $r_1$ is value comparable with Role $r_2$. The join path from which the constrained roles were projected involves a conceptual join on the roles hosted by Person in this example.
2.24 Set-Comparison Constraints

**ORM Diagram**

Derivation Rules:

*SetComparisonArgument has Arity if and only if
  Arity = count(each Role where that SetComparisonArgument includes that Role at some Position).

*SetComparisonConstraint has ArgumentLength if and only if
  that SetComparisonConstraint compares some SetComparisonArgument that has some Arity = that ArgumentLength.

*SetComparisonConstraint compares SetComparisonArgument if and only if
  that SetComparisonConstraint is some EqualityConstraint that compares that SetComparisonArgument
  or that SetComparisonConstraint is some ExclusionConstraint that compares that SetComparisonArgument
  or that SetComparisonConstraint is some SubsetConstraint that has that subset SetComparisonArgument
  or has that superset SetComparisonArgument.

*SetComparisonConstraint has NrArguments if and only if
  NrArguments = count(each SetComparisonArgument that is compared by that SetComparisonConstraint).

*Role1 in set comparison argument is compared with Role2 in compared argument if and only if
  that Role1 is included in some SetComparisonArgument1 at some Position
  where that SetComparisonArgument1 is compared by some SetComparisonConstraint
  that compares some SetComparisonArgument2 that includes that Role2 at that Position
  where SetComparisonArgument1 <> SetComparisonArgument2.

**Figure 2-40** Set-Comparison Constraints

**Verbalization** (of newly introduced aspects)

**Object Types, Reference Schemes and Subtyping**

ArgumentLength is a value type.

  Each ArgumentLength is an instance of Arity.
  NrArguments is a value type.

**Asserted Fact Types**

SetComparisonArgument includes Role at Position.

**Derived Fact Types**

*SetComparisonArgument has Arity if and only if
  Arity = count(each Role where that SetComparisonArgument includes that Role at some Position).

*SetComparisonConstraint has ArgumentLength if and only if
  that SetComparisonConstraint compares some SetComparisonArgument
  that has some Arity = that ArgumentLength.

*SetComparisonConstraint compares SetComparisonArgument
  if and only if
  SetComparisonArgument is compared by SetComparisonConstraint.
that SetComparisonConstraint is some EqualityConstraint that compares that SetComparisonArgument or that SetComparisonConstraint is some ExclusionConstraint that compares that SetComparisonArgument
or that SetComparisonConstraint is some SubsetConstraint that has that subset SetComparisonArgument or has that superset SetComparisonArgument.

*SetComparisonConstraint has NrArguments if and only if NrArguments = count(each SetComparisonArgument that is compared by that SetComparisonConstraint).

*Role1 in set comparison argument is compared with Role2 in compared argument if and only if that Role1 is included in some SetComparisonArgument, at some Position where that SetComparisonArgument, is compared by some SetComparisonConstraint that compares some SetComparisonArgument, that includes that Role, at that Position where SetComparisonArgument, <> SetComparisonArgument,

Constraints

For each SetComparisonArgument and Role,
    that SetComparisonArgument includes that Role at at most one Position.
Each SetComparisonArgument includes some Role at some Position.
For each SetComparisonArgument and Position,
    that SetComparisonArgument includes at most one Role at that Position.

Each SetComparisonConstraint compares some SetComparisonArgument.
Each SetComparisonArgument is compared by exactly one SetComparisonConstraint.
It is possible that some SetComparisonConstraint compares more than one SetComparisonArgument.

Each SetComparisonArgument has exactly one Arity.
It is possible that more than one SetComparisonArgument has the same Arity.

Each SetComparisonConstraint has exactly one ArgumentLength.
It is possible that more than one SetComparisonConstraint has the same ArgumentLength.

Each SetComparisonConstraint has exactly one NrArguments.
It is possible that more than one SetComparisonConstraint has the same NrArguments.

It is possible that some Role in set comparison argument is compared with more than one Role in compared argument
and that for some Role, more than one Role in set comparison argument is compared with that Role,
in compared argument.

In each population of Role in set comparison argument is compared with Role in compared argument,
each Role, Role combination occurs at most once.

For each SetComparisonConstraint and Arity,
    that SetComparisonConstraint has some ArgumentLength that is that Arity if and only if that SetComparisonConstraint compares some SetComparisonArgument that has that Arity.

If some Role, in set comparison argument is compared with some Role, in compared argument
then that Role, is compatible with that Role,

Explanation

Each SetComparisonArgument is an ordered set of roles that is selected as an argument of a SetComparisonConstraint. Arguments of the same set comparison constraint must have the same arity (number of roles). Each set comparison constraint is either a subset constraint, an equality constraint, or an exclusion constraint.
NrArguments is the number of arguments compared by the constraint, and ArgumentLength is the number of roles in each argument. A subset constraint compares exactly two arguments, ensuring that the set of tuples populating one argument is a subset of the set of tuples populating the other argument. An equality constraint compares two or more arguments, ensuring that the sets of tuples populating each argument are always equal. An exclusion constraint compares two or more arguments, ensuring that the sets of tuples populating each argument are always mutually exclusive.

Graphically, a subset constraint is depicted by a circled “⊆”, connected by an arrowed, dashed line directed from the subset argument to the superset argument. An equality constraint is depicted by a circled “=”, connected by dashed lines to its arguments. An exclusion constraint is depicted by a circled “×”, connected by dashed lines to its arguments. If a dashed line connects to the junction of two roles, it applies to that pair of roles.

For example, the exclusion constraint in Figure 2-41 applies to three arguments (NrArguments = 3), each of which is a pair of roles (ArgumentLength = 2). This constraint ensures that for each person and team combination, that person can play at most one of the owner, captain and vice-captain roles for that team. This does not prevent a person from owning one team, captaining another team, and being vice-captain of yet another team.

Figure 2-41  Example of an exclusion constraint with three arguments of length 2

The subset and equality constraints in the metamodel fragment involve at least one derived fact type, but are asserted (as indicated by their red colour). For each of these constraints, NrArguments = 2 and ArgumentLength = 2.

These three varieties of set-comparison constraints are discussed further in the next few sections.
2.25 Subset Constraints

ORM Diagram

![Subset Constraint Diagram](image)

**Figure 2-42** Subset constraints

**Verbalization** (of newly introduced aspects)

**Asserted Fact Types**

- SubsetConstraint has subset SetComparisonArgument.  
  / SetComparisonArgument is superset for SubsetConstraint.
- SubsetConstraint has superset SetComparisonArgument.  
  / SetComparisonArgument is subset for SubsetConstraint.

**Constraints**

- **Each** SubsetConstraint has **exactly one** subset SetComparisonArgument.  
- **Each** SetComparisonArgument is superset for **at most one** SubsetConstraint.
- **Each** SubsetConstraint has **exactly one** superset SetComparisonArgument.  
- **Each** SetComparisonArgument is subset for **at most one** SubsetConstraint.
- **For each** SetComparisonArgument, **at most one of the following holds:**
  - that SetComparisonArgument is subset for **some** SubsetConstraint;
  - that SetComparisonArgument is superset for **some** SubsetConstraint.

**Explanation**

A subset constraint compares exactly two arguments, ensuring that the set of tuples populating one argument is a subset of the set of tuples populating the other argument. Graphically, a subset constraint is depicted by a circled “⊆”, connected by an arrowed, dashed line directed from the subset argument to the superset argument. For example, the subset constraint depicted in Figure 2-43 ensures that, for each state of the fact base, the set of (student, course) pairs populating the ordered role pair (r3, r4) is a subset of the set of (student, course) pairs populating the ordered role pair (r1, r2). In other words, if a student passed a course then that student must have enrolled in that course.

![Subset Constraint Example](image)

**Figure 2-43** Example of a subset constraint between role pairs
Figure 2-44 shows how to populate the metamodel with the example subset constraint. Here $A_1$ and $A_2$ denote the surrogate identifiers for the set comparison arguments.
2.26 Equality Constraints

ORM Diagram

![Diagram](image)

Figure 2-45  Equality constraints

Verbalization (of newly introduced aspects)

Asserted Fact Types

EqualityConstraint compares SetComparisonArgument.
/ SetComparisonArgument is compared by EqualityConstraint.

Constraints

Each EqualityConstraint compares some SetComparisonArgument.
Each SetComparisonArgument is compared by at most one EqualityConstraint.
It is possible that some EqualityConstraint compares more than one SetComparisonArgument.
Each EqualityConstraint in the population of “EqualityConstraint compares SetComparisonArgument” occurs there at least 2 times.

Explanation

An equality constraint compares two or more arguments, ensuring that the sets of tuples populating of each argument are always equal. Graphically, an equality constraint is depicted by a circled “=”, connected by dashed lines to its arguments. In Figure 2-46 the equality constraint has three arguments, each comprising a single role. This constraint ensures that each paper either has no reviews or has three reviews.

![Diagram](image)

Figure 2-46  An equality constraint with three 1-role arguments

Although this metamodel fragment applies an equality constraint to a set of arguments, for exchange purposes some order must be chosen for those arguments.
2.27 Exclusion Constraints

ORM Diagram

![Exclusion Constraint Diagram]

*Figure 2-47 Exclusion constraints*

**Verbalization** (of newly introduced aspects)

**Asserted Fact Types**

- ExclusionConstraint compares SetComparisonArgument.
- / SetComparisonArgument is compared by ExclusionConstraint.

**Constraints**

- Each ExclusionConstraint compares some SetComparisonArgument.
- Each SetComparisonArgument is compared by at most one ExclusionConstraint.
- It is possible that some ExclusionConstraint compares more than one SetComparisonArgument.
- Each ExclusionConstraint in the population of “ExclusionConstraint compares SetComparisonArgument” occurs there at least 2 times.

**Explanation**

An exclusion constraint compares two or more arguments, ensuring that the sets of tuples populating each argument are always mutually exclusive. Graphically, an exclusion constraint is depicted by a circled “×”, connected by dashed lines to its arguments. See Figure 2-41 for an example.

Although this metamodel fragment applies an exclusion constraint to a set of arguments, for exchange purposes some order must be chosen for those arguments.
2.28 Cardinality Constraints

ORM Diagram

![Diagram showing CardinalityConstraint, ObjectType, Role, CardinalityRange, and Cardinality.](image)

For each CardinalityConstraint, CardinalityRanges do not overlap.

**Verbalization** (of newly introduced aspects)

**Object Types and Reference Schemes**

CardinalityRange is an entity type.

Reference Scheme:
- CardinalityConstraint has CardinalityRange;
- CardinalityRange specifies minimum Cardinality;
- CardinalityRange specifies maximum Cardinality.

Cardinality is a value type.

**Asserted Fact Types**

- CardinalityConstraint applies to ObjectType.
- CardinalityConstraint applies to Role.
- CardinalityConstraint has CardinalityRange.
- CardinalityRange specifies minimum Cardinality.
- CardinalityRange specifies maximum Cardinality.

**Constraints**

- Each CardinalityConstraint applies to at most one ObjectType.
- Each ObjectType has at most one CardinalityConstraint.
- Each CardinalityConstraint applies to at most one Role.
- Each Role has at most one CardinalityConstraint.
- Each CardinalityConstraint has some CardinalityRange.
- Each CardinalityRange is for exactly one CardinalityConstraint.

It is possible that some CardinalityConstraint has more than one CardinalityRange.
Each value of minimum Cardinality in CardinalityRange specifies minimum Cardinality is at least 1. Each CardinalityRange specifies at most one minimum Cardinality. It is possible that more than one CardinalityRange specifies the same minimum Cardinality.

Each value of maximum Cardinality in CardinalityRange specifies maximum Cardinality is at least 1. Each CardinalityRange specifies at most one maximum Cardinality. It is possible that more than one CardinalityRange specifies the same maximum Cardinality.

Each value of Cardinality is at least 0. Each value of minimum Cardinality in CardinalityRange specifies minimum Cardinality is at least 1. Each value of maximum Cardinality in CardinalityRange specifies maximum Cardinality is at least 1.

For each CardinalityConstraint, exactly one of the following holds:
  that CardinalityConstraint applies to some ObjectType;
  that CardinalityConstraint applies to some Role.

Each CardinalityRange specifies some minimum Cardinality or specifies some maximum Cardinality.

For each CardinalityConstraint, minimum Cardinality, and maximum Cardinality, at most one CardinalityRange is for that CardinalityConstraint and
  specifies that minimum Cardinality, and
  specifies that maximum Cardinality.

This association with CardinalityConstraint, Cardinality, Cardinality provides the preferred identification scheme for CardinalityRange.

For each CardinalityRange, if that CardinalityRange specifies some minimum Cardinality, and
  specifies some maximum Cardinality,
then Cardinality, is less than or equal to Cardinality.

Note: The note in Figure 2-48 specifies an additional, general constraint for which no graphical notation exists.

Explanation

For each state of the fact base, a cardinality constraint limits the number of instances that may occur in the population of the constrained object type or role. Here each “cardinality range” is specific to a given cardinality constraint (so is an occurrence of a range), and must specify a minimum number, a maximum number, or both. Graphically, a cardinality constraint is depicted by placing next to the constrained object type or role the symbol “#” (meaning cardinality) followed by an expression to indicate the range of numbers allowed.

For example, the cardinality constraint depicted in Figure 2-49 applies to the role of the unary fact type, ensuring that, for each state of the fact base, at most one employee populates that role. Over time, different employees may play this role, but at any given time, at most one employee is chief executive officer.

![Figure 2-49](image_url) A role cardinality constraint to ensure that there is at most one chief executive officer at any given time
2.29 Value Constraints

ORM Diagram

Derivation Rules:

*Role allows values from DataType if and only if
that Role is hosted by some ObjectType that allows values from that DataType.

*ValueConstraint allows values from DataType if and only if
that ValueConstraint restricts some Role that allows values from that DataType
or that ValueConstraint restricts some ObjectType that allows values from that DataType.

If a DataValueRange has both minimum and maximum values,
then these values must be compatible, with the minimum value $\leq$ the maximum value.

Join Constraints:

(DataValueRangeMinMatchesDataType subset constraint)
If some ValueConstraint allows some DataValueRange
that has some minimum CanonicalDataValue1 that is some DataValue1 that is of some DataType
then that ValueConstraint allows values from that DataType.

(DataValueRangeMaxMatchesDataType subset constraint)
If some ValueConstraint allows some DataValueRange
that has some maximum CanonicalDataValue1 that is some DataValue1 that is of some DataType
then that ValueConstraint allows values from that DataType.

(ValueConstraintValueMatchesDataType subset constraint)
If some ValueConstraint allows some CanonicalDataValue that is some DataValue that is of some DataType
then that ValueConstraint allows values from that DataType.

Figure 2.50 Value constraints
Verbalization (of newly introduced aspects)

Object Types and Reference Schemes

DataValueRange is an entity type.
Reference Scheme: ValueConstraint allows DataValueRange; DataValueRange has minimum CanonicalDataValue; DataValueRange has maximum CanonicalDataValue.

ValueRangeMinimum is an entity type.
ValueRangeMinimum objectifies "DataValueRange has minimum CanonicalDataValue".

ValueRangeMaximum is an entity type.
ValueRangeMaximum objectifies "DataValueRange has maximum CanonicalDataValue".

Asserted Fact Types

ValueConstraint restricts Role. / Role is restricted by ValueConstraint.
ValueConstraint restricts ObjectType. / ObjectType is restricted by ValueConstraint.
ValueConstraint allows DataValueRange.
DataValueRange has minimum CanonicalDataValue.
DataValueRange has maximum CanonicalDataValue.
ValueRangeMinimum is included.
ValueRangeMaximum is included.

Derived Fact Types

*Role allows values from DataType
  if and only if
    that Role is hosted by some ObjectType that allows values from that DataType.

*ValueConstraint allows values from DataType
  if and only if
    that ValueConstraint restricts some Role that allows values from that DataType
    or that ValueConstraint restricts some ObjectType that allows values from that DataType.

Constraints

Each ValueConstraint restricts at most one Role.
Each Role is restricted by at most one ValueConstraint.

Each ValueConstraint restricts at most one ObjectType.
Each ObjectType is restricted by at most one ValueConstraint.

Each DataValueRange is allowed in exactly one ValueConstraint.
It is possible that some ValueConstraint allows more than one DataValueRange.

Each DataValueRange has at most one minimum CanonicalDataValue.
It is possible that more than one DataValueRange has the same minimum CanonicalDataValue.

Each DataValueRange has at most one maximum CanonicalDataValue.
It is possible that more than one DataValueRange has the same maximum CanonicalDataValue.

In each population of ValueRangeMinimum is included, each ValueRangeMinimum occurs at most once.
In each population of ValueRangeMaximum is included, each ValueRangeMaximum occurs at most once.

It is possible that some ValueConstraint allows values from more than one DataType
and that for some DataType, more than one ValueConstraint allows values from that DataType.
In each population of ValueConstraint allows values from DataType, each ValueConstraint, DataType combination occurs at most once.

Each ValueConstraint allows values from some DataType.

It is possible that some Role allows values from more than one DataType and that for some DataType, more than one Role allows values from that DataType.

In each population of Role allows values from DataType, each Role, DataType combination occurs at most once.

For each ValueConstraint, exactly one of the following holds:

- that ValueConstraint restricts some Role;
- that ValueConstraint restricts some ObjectType.

Each ValueConstraint allows some DataValueRange or allows some CanonicalDataValue.

Each DataValueRange has some minimum CanonicalDataValue or has some maximum CanonicalDataValue.

For each ValueConstraint, minimum CanonicalDataValue, and maximum CanonicalDataValue, at most one DataValueRange is allowed in that ValueConstraint and has that minimum CanonicalDataValue, and has that maximum CanonicalDataValue,

This association with ValueConstraint, CanonicalDataValue, CanonicalDataValue provides the preferred identification scheme for DataValueRange.

If some Role is restricted by some ValueConstraint then that Role allows values from some DataType.

If some ObjectType is restricted by some ValueConstraint then that ObjectType can restrict values.

If some ValueConstraint allows some DataValueRange that has some maximum CanonicalDataValue, that is some DataValue that is some CanonicalDataValue and is of some DataType then that ValueConstraint allows values from that DataType.

If some ValueConstraint allows some CanonicalDataValue that is some DataValue that is of some DataType then that ValueConstraint allows values from that DataType.

If some ValueConstraint allows some DataValueRange that has some minimum CanonicalDataValue that is some DataValue that is of some DataType then that ValueConstraint allows values from that DataType.

Explanation

A value constraint restricts the values of the role or object type that it constrains. The specification of the allowed values may include one or more discrete data values and/or one or more data value ranges. Each data value range is specific to a value constraint (and hence is a range occurrence), and specifies a minimum value or a maximum value or both, with a qualification to indicate which of these extreme values are to be included in the range. Graphically, a value constraint is displayed next to the constrained object type or role.

By default, ranges are assumed to be closed, so the end values are included. For example, \{1..10\} includes 1 and 10 as well as the numbers in between. A left parenthesis "(" before a start value excludes that value, and a right parenthesis ")" after an end value excludes that value. A square bracket may be used to explicitly indicate inclusion (the default). For example, "[0.10]" and "(0..10)" each denote a range of positive (above 0) numbers up to and including 10. Value constraints are to be understood in terms of the relevant data type. For example, for integer datatypes the constraint \{(0..10)\} means the same as \{1..10\}, but for a floating point datatype \{(0..10)\} includes all the positive real numbers up to 10 (including fractional values that are above 0 but below 1). For exchange purposes, data values are stored in canonical format.
2.30 General Constraints

ORM Diagram

FormalLanguage is used the same way here as with Rule. Setting a formal language indicates that additional structural elements will be added to GeneralConstraint by extensions to this model.

If a formal language is specified, then the constrained object types, fact types, and roles are expected to be derivable based on the formal rule. Therefore, the derivation rules for this fact type are also considered external.

Informal rules are considered ad hoc and must explicitly set what they affect and provide an informal constraint description.

Figure 2.51 General constraints

Verbalization (of newly introduced aspects)

Object Types and Reference Schemes

ConstraintDescription is a value type.

Asserted Fact Types

GeneralConstraint has ConstraintDescription.
GeneralConstraint is expressed in FormalLanguage.

Semiderived Fact Types

GeneralConstraint applies to ObjectType. / ObjectType has GeneralConstraint.
GeneralConstraint applies to Role. / Role has GeneralConstraint.
GeneralConstraint applies to FactType. / FactType has GeneralConstraint.

Constraints

It is possible that some GeneralConstraint applies to more than one ObjectType and that some ObjectType has more than one GeneralConstraint.
In each population of GeneralConstraint applies to ObjectType, each GeneralConstraint, ObjectType combination occurs at most once.

It is possible that some GeneralConstraint applies to more than one FactType and that some FactType has more than one GeneralConstraint.
In each population of GeneralConstraint applies to FactType,
each GeneralConstraint, FactType combination occurs at most once

It is possible that some GeneralConstraint applies to more than one Role and that some Role has more than one GeneralConstraint.

In each population of GeneralConstraint applies to Role,

each GeneralConstraint, Role combination occurs at most once.

Each GeneralConstraint has exactly one ConstraintDescription.

It is possible that more than one GeneralConstraint has the same ConstraintDescription.

It is possible that some GeneralConstraint is expressed in more than one FormalLanguage and that for some FormalLanguage, more than one GeneralConstraint is expressed in that FormalLanguage.

In each population of GeneralConstraint is expressed in FormalLanguage, each GeneralConstraint, FormalLanguage combination occurs at most once.

Each GeneralConstraint applies to some ObjectType or applies to some Role or applies to some FactType.

Explanation

A general constraint is a constraint for which no graphical FBM notation yet exists. Such constraints may be specified textually in a constraint description expressed in either a formal language or an informal language. The specification of a textual language for this purpose is outside the scope of this metamodel, but it is anticipated that later versions of the metamodel will provide a formal grammar for at least one such language.

Graphically, general constraints may be displayed in various ways (e.g. in a note or footnote). While this metamodel requires that each general constraint be associated with at least one object type fact type or role, it does not prescribe how or even if such associations to model elements attached are displayed. (e.g. by footnote numbers, or by dashed or dotted line connections). Figure 2-52 provides an example using footnotes, with the general constraint expressed in the FORML language (Halpin & Wijbenga, 2010). This is an example of a restricted uniqueness constraint, which strengthens the uniqueness constraint requirement on a fact type for a subpopulation of one of the object types involved. Although employees in general may be funded to attend more than one conference, junior employees are funded for at most one conference.

![Diagram](image_url)

*Each Employee who has EmployeeRank ‘JuniorEmployee’ is funded for at most one Conference.*

Figure 2-52  Example of an general constraint displayed in a footnote
2.31 Join Lists

ORM Diagram

Derivation Rules:
*Join is first if and only if that Join is at Position = 1.
*Role1 occurs on JoinSide of Join if and only if that Join is first and is from some Role2 that belongs to some FactType1 that contains that Role1 where JoinSide = "From" or that Join is to some Role3 that belongs to some FactType2 that contains that Role1 where Role3 <> Role1 where JoinSide = "To".
*Join1 is directly followed by Join2 if and only if that Join1 is at some Position1 and is in some JoinList where that Join2 is in that JoinList and is at some Position2 where Position2 = Position1 + 1.
*Join1 is preceded by to Role if and only if that Join1 is directly preceded by some Join2 that is to that Role.
*Join1 is from Role1 if and only if that Join1 is explicitly from that Role1 or that Join1 is explicitly from no Role and is directly preceded by some Join2 that is from that Role1.

Note on JoinFactTypeRoleVariable:
In a join list, each join is to a role that indicates a use of the fact type for that role. Using a FactType makes each role in the fact type available for projection and possibly other uses of the path. A reuse of the same fact type via a later join operation indicates a different use, producing different variables for the roles in the FactType.

However, if we look just at the toRoles to determine fact type uses then we miss the initial fact type use, which can be determined by the fromRole of the first join. If the first join does a self-join (to the same fact type), then the {Role,Join} combination is not unique, so JoinSide is required to ensure uniqueness in self-join situations.

JoinFactTypeRoleVariables are created for all roles in the first fact type, but not for toRoles in subsequent fact types. The JoinFactTypeRoleVariable for the from-role is sufficient for the toRoles. For consistency in associating variables with roles, we disallow any explicit fromRole that is the same as the toRole of the previous join.

Figure 2-53  Join Lists
**Verbalization** (of newly introduced aspects)

**Object Types and Reference Schemes**

- **Join** is an entity type.
  - Reference Scheme: Join has JoinSID.
  - Reference Mode: SID.

- **JoinType** is an entity type.
  - Reference Scheme: JoinType has JoinTypeName.
  - Reference Mode: Name.

- **JoinList** is an entity type.
  - Reference Scheme: JoinList has JoinListSID.
  - Reference Mode: SID.

- **JoinSide** is an entity type.
  - Reference Scheme: JoinSide has JoinSideName.
  - Reference Mode: Name.

- **JoinFactTypeRoleVariable** is an entity type.
  - JoinFactTypeRoleVariable objectifies “Role occurs on JoinSide of Join”.
  - JoinFactTypeRoleVariable is independent (it may have instances that play no other roles).

**Asserted Fact Types**

- Join has JoinType.
- Join is explicitly from Role.
- Join is to Role.
- Join is in JoinList.
- Join is at Position.

**Derived Fact Types** (for derivation rules see Figure 2-53)

- *Join is first.
- *Role occurs on JoinSide of Join.
- *Join is directly followed by Join / Join is directly preceded by Join.
- *Join is preceded by to- Role.
- *Join is from Role.
Constraints

Each Join has exactly one JoinType.
It is possible that more than one Join has the same JoinType.

Each Join is explicitly from at most one Role.
It is possible that more than one Join is explicitly from the same Role.

Each Join is to exactly one Role.
It is possible that more than one Join is to the same Role.

Each Join is in exactly one JoinList.
For each JoinList, some Join is in that JoinList.
It is possible that more than one Join is in the same JoinList.

Each Join is at exactly one Position.
It is possible that more than one Join is at the same Position.

In each population of Join is first, each Join occurs at most once.
It is possible that for some Role and JoinSide, that Role occurs on that JoinSide of more than one Join.
and that for some Role and Join, that Role occurs on more than one JoinSide of that Join.
and that for some JoinSide and Join, more than one Role occurs on that JoinSide of that Join.

In each population of Role occurs on JoinSide of Join, each Role, JoinSide, Join combination occurs at most once.

Each Join is preceded by at most one to Role.
It is possible that more than one Join is preceded by the same to Role.

Each Join is from exactly one Role.
It is possible that more than one Join is from the same Role.

For each JoinList and Position, at most one Join is in that JoinList and is at that Position.

The possible values of JoinSide are 'From', 'To'.

The possible values of JoinType are 'Inner', 'Outer'.

If some Join is first then that Join is explicitly from some Role.

For each Join and Role, at most one of the following holds:
that Join is explicitly from that Role;
that Join is preceded by that to Role.

If for some Role, some Join is to that Role,
and that Join is from some Role,
then that Role is compatible with that Role.

Explanation

In navigating across a role path in an FBM model, a conceptual join from one role (the fromRole) to another role (the toRole) requires an object (if any) playing the second role to be identical to the object playing the first role. If the join type is Inner, each object playing the first role also plays the second role. If the join type is Outer, an object playing the first role may, but need not, play the second role. A join list is an ordered set of joins. Each join has a surrogate identifier, but can also be identified by its position in a join list. If the same two roles are joined more than once in the same join list (indicating a cycle in the join path), these are treated as different joins (so joins are actually join occurrences).

If a constraint has at least one argument composed of roles projected from a role path that includes at least one join, the constraint is said to be a join constraint. Moreover, it is fairly common for derivation rules to involve navigation across a role path with at least one join. This metamodel fragment is included to capture simple joins involved in join constraints and basic derivation rules. It is not intended to directly
cover composite joins that could be expressed in a general derivation or query language (e.g. List each person and sport where that person plays and coaches that sport).

Figure 2-54 shows a simple example adapted from Halpin & Morgan (2008, p. 404), of a join subset constraint to ensure that each advisor who serves in a country also speaks a language that is used by that country. In this verbalization of the constraint, the relative pronoun “that” signifies the conceptual join. The superset argument of the subset constraint is projected from a path that involves an inner join from role r1 to role r2. For this join, r1 is the fromRole and r2 is the toRole.

Figure 2-54  A join subset constraint where the superset roles are projected from a path that joins the Language roles

A join is allowed between type-compatible roles, even if they are not directly hosted by the same object type. For example, in Figure 2-55 the external uniqueness constraint applies to roles r1 and r4, which are projected from a role path involving the inner join r2 ⊙ r3 between the roles with gold fill. This single join is equivalent to 3 joins if we include the joins on the Employee roles in the implicit subtyping identity relationships.

Figure 2-55  An external uniqueness constraint over roles r1 and r4 projected a path that joins roles r2 and r3

An explicit fromRole of a join is optional for all but the first join. This naturally caters for cases like the one shown in Figure 2-56, adapted from Halpin & Morgan (2008). Here the join list comprises two left outer joins. The left outer join operator, denoted here by “*•”, is left-associative, so the join list results in (r1 *• r2) *• r3. The second join has no explicit fromRole but its derived fromRole is r1.

Figure 2-56  An external uniqueness constraint over roles projected a path that outer joins roles r1, r2 and r3
In a join list, each join is to a role that indicates a use of the fact type for that role. Using a fact type makes each role in the fact type available for projection and possibly other uses of the path. A reuse of the same fact type via a later join operation indicates a different use, producing different variables for the non-join roles (i.e., the roles that are not joined) in the fact type.

However, if we look just at the toRoles to determine fact type uses then we miss the initial fact type use, which can be determined by the fromRole of the first join. If the first join does a self-join (to the same fact type), then the (Role, Join) combination is not unique, so the JoinSide (From or Left, versus To or Right) is required to ensure uniqueness in self-join situations.

For example, consider the query: List each Person who speaks English, Latin and Japanese. This may be formalized in logic as shown below. In logic, joining one role occurrence to another is indicated by using the same object variable (here, x) for both.

\{x | Person x & x speaks y & y hasLanguageName 'English' & x speaks z & z hasLanguageName 'Latin' & x speaks w & w hasLanguageName 'Japanese'}

Similarly, the derivation rule for the fact type “Person speaks english and latin and Japanese” may be formalized:

\( \forall x [\text{PersonSpeaksEnglishLatinAndJapanese} x = \neg (\text{Person} x & x \text{ speaks} y \& y \text{ hasLanguageName} 'English' \& x \text{ speaks} z \& z \text{ hasLanguageName} 'Latin' \& x \text{ speaks} w \& w \text{ hasLanguageName} 'Japanese')] \)

As shown in Figure 2-57(a), this involves navigating three times through the fact type Person speaks Language. The body of the query or derivation rule may be visualized in terms of joins as in Figure 2-57(b). Here, for explanatory purposes, the fact type Person speaks Language is displayed three times, with the self-joins depicted as arrows directed from the source role (the fromRole) to the target role (the toRole).

![Figure 2-57](image)

**Figure 2-57** Two consecutive self-joins on the person role (r1) of the Person speaks Language fact type

**Join fact type role variables** are created for all roles in the first fact type occurrence. Here, the variable x is used for role r1 on the From-side of Join 1, and the variable y is used for role r2 on the From-side of Join 1. Note that variables are not created for toRoles of joins, but variables are created for all other roles in subsequent fact type occurrences on the From-side of a join. For the Figure 2-57(b) example, the variables are thus determined by the triples shown in the table below, corresponding to the composite reference scheme for JoinFactTypeRoleVariable in the metamodel.

The JoinFactTypeRoleVariable for the initial from-role is implicitly reused as the fromRole for second and later joins. For consistency in associating variables with roles, we disallow any explicit fromRole that is the same as the toRole of the previous join.

Further fine details about join lists and associated constraints and projections are provided in sections 3.17, 3.18 and 3.19.

<table>
<thead>
<tr>
<th>JoinFactTypeRoleVariable</th>
<th>Join</th>
<th>JoinSide</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Join-1</td>
<td>From</td>
<td>r1</td>
</tr>
<tr>
<td>y</td>
<td>Join-1</td>
<td>From</td>
<td>r2</td>
</tr>
<tr>
<td>z</td>
<td>Join-1</td>
<td>To</td>
<td>r2</td>
</tr>
<tr>
<td>w</td>
<td>Join-2</td>
<td>To</td>
<td>r2</td>
</tr>
</tbody>
</table>
2.32 Sample Populations

ORM Diagram

Verbalization (of newly introduced aspects)

Object Types, Reference Schemes and Subtyping

Fact is an entity type.
Reference Scheme: Fact has FactSID.
Reference Mode: .SID. Join

DomainObject is an entity type.
Reference Scheme: DomainObject has DomainObjectSID.
Reference Mode: .SID.

DomainValue is an entity type.
*Each DomainValue is some DomainObject that is an instance of some DomainObjectType that is some ValueType.
Reference Scheme: DomainObject has DomainObjectSID.
Reference Mode: .SID.

TypedObject is an entity type.
TypedObject objectifies "DomainObject is an instance of DomainObjectType".
TypedObject is independent (it may have instances that play no other roles).

Asserted Fact Types

Fact is an instance of FactType.
DomainObject plays Role in Fact.
TypedObject objectifies Fact. / Fact is objectified by TypedObject.
DomainObject is an instance of DomainObjectType.
DomainValue has CanonicalDataValue.
Constraints

Each Fact is an instance of exactly one FactType.
It is possible that more than one Fact is an instance of the same FactType.

For each Fact and Role,
 at most one DomainObject plays that Role in that Fact.

Each.TypedObject objectifies at most one Fact.
Each Fact is objectified by at most one TypedObject.

It is possible that some DomainObject is an instance of more than one DomainObjectType
and that for some DomainObjectType, more than one DomainObject is an instance of that DomainObjectType.

In each population of DomainObject is an instance of DomainObjectType, each DomainObject,
DomainObjectType combination occurs at most once.
This association with DomainObject, DomainObjectType provides the preferred identification scheme for TypedObject.

Each DomainObject is an instance of some DomainObjectType.
Each DomainValue has exactly one CanonicalDataValue.
It is possible that more than one DomainValue has the same CanonicalDataValue.

Explanation

Most fact types may be explicitly populated with facts (i.e. fact instances). A single fact is recorded by
assigning a domain object (i.e. an entity or value) to each role in the fact type. Surrogate identifiers are
used initially, but ultimately each entity is mapped via its reference scheme to a combination of one or
more values, and each value is mapped to a canonical data value, which itself is identified by combining
its lexical value with its data type for the default language culture.

Figure 2-59(a) shows how a simple ORM model might typically be displayed. Persons are referenced
by their name and intelligence quotients (IQs) by numbers. Here the fact table displays two facts in a form
that is easily understood by the user. Figure 2-59(b) shows the same model, with the reference mode
expanded to display the underlying reference fact type (Person has PersonName), and various surrogate
identifiers: Here OT1, OT2 and OT3 denote the object types Person, PersonName and IQ. FT1 and FT2
denote the fact types with readings Person has PersonName and Person has IQ. The roles of these fact types are
numbered r1, r2, r3 and r4 as shown. At the instance level, the two persons are denoted by the surrogates
p1 and p2, and the four facts by F1, F2, F3 and F4.

![Figure 2-59](image)

To fully capture the model, more work is needed, because the PersonName and IQ instances are
domain values, so they also are assigned surrogate identifiers (e.g. n1, n2 for the names, and iq1, iq2 for
the IQs), each of which is then related to a canonical value (e.g. the literal 'Albert Einstein' of datatype
string in language culture en-US).

Ignoring the ultimate mapping to canonical data values, Figure 2-60 shows how the information
conveyed by the surrogate entries in Figure 2-59 (supplemented by surrogates for person names and IQs)
may be captured in the metamodel.
To reduce the amount of memory and work required for storing facts, the metamodel does not support explicit population of instance level, identity fact types underlying subtyping relationships, or link fact types implied by objectification. Instead, identity fact types underlying subtyping are implicitly populated by associating a single object instance with multiple compatible types (e.g., Woman, Person). Moreover, link fact types can be implicitly populated by recording instance level objectifications via the fact type TypedObject objectifies Fact. This allows the model to subtype objectifications of different fact types, and to allow alternate identification schemes (including simple surrogates) for objectifying types.

Objectified facts are related to typed objects instead of just objects, to ensure 1:1 correspondence between them. For example, consider a model that objectifies Student enrolled in Course as Enrolment, and objectifies Student enrolled early in Course as EarlyEnrolment, and then declares EarlyEnrolment as a subtype of Enrolment. While an instance of EarlyEnrolment is also an instance of Enrolment (and hence may have the same SID), to objectify the fact that a specific student enrolled early in a specific course, one should specifically identify this as an instance of EarlyEnrolment, so the type information is also required, not just a surrogate for the enrolment object.

For further discussion of fine details regarding sample populations, see sections 3.20 and 3.21.
3 Metamodel: Fine Details

3.1 Role Names (Constrained)

ORM Diagram

```
Derivation Rules:
*ObjectType has far RoleName from BinaryFactType if and only if
  that ObjectType plays some Role1 that belongs to that BinaryFactType
  that contains some Role2 that has that RoleName
  where Role1 <> Role2.

*ObjectType has dot RoleName from FactType if and only if
  that ObjectType has that far RoleName from that FactType
  or hosts some Role
  that has that RoleName
  and belongs to that FactType that is some UnaryFactType.
```

Note on derived fact types:
These fact types exist to describe the far role and dot role
concepts, and to apply the internal uniqueness constraints,
which are not implied. This model assumes set semantics for
all fact types, including derived fact types, so a spanning uniqueness
constraint on a binary 'ObjectType has far RoleName' fact type
would be implied by the set projection and add no additional
restriction. The ternaries are necessary to form a derived set
with the possibility of duplicate ObjectType/RoleName pairs
in the first two columns so that the applied uniqueness constraints
can be stronger than the implied spanning uniqueness constraint.

![Figure 3-1 Far RoleNames and Dot RoleNames of Object Types](image)

Verbalization (of newly introduced aspects)

Derived Fact Types

*ObjectType has far RoleName from BinaryFactType if and only if
  that ObjectType hosts some Role, that belongs to that BinaryFactType that contains some Role,
  that has that RoleName where Role1 <> Role2.

*ObjectType has dot RoleName from FactType if and only if
  that ObjectType has that far RoleName from that FactType
  or hosts some Role that has that RoleName
  and belongs to that FactType that is some UnaryFactType.
Constraints

For each ObjectType and far RoleName,
  that ObjectType has that far RoleName from at most one BinaryFactType.

For each ObjectType and dot RoleName,
  that ObjectType has that dot RoleName from at most one FactType.

Explanation

In FBM, role names are used mainly to enable constraints and derivation rules to be formulated textually in "attribute-style", treating some roles as if they are "attributes" of a relevant object type. This is an alternative to the usual "relational style" formulation in which navigation across fact types is expressed in terms of fact type readings. Attribute-style is especially useful for rules that involve arithmetic comparisons between roles of binary fact types. For example, assuming the role names "birthdate" and "hiredate" are provided on the Date roles in the fact types Employee was born on Date and Employee was hired on Date, the value-comparison constraint expressed in relational style as "For each Employee, if that Employee was born on Date1 and was hired on Date2 then Date2 > Date1" is rendered more compactly in attribute style as "For each Employee, hiredate > birthdate". In this example, the birthdate and hiredate roles are said to be far roles of Employee, because from the perspective of Employee they are placed at the far end of the birth and hire fact types. They are also called dot roles of Employee, because they can be referenced using dot notation (e.g., Employee.birthdate, Employee.hiredate). If a role of a unary fact type is named, it is also considered a dot role for its object type.

The asserted uniqueness constraints on the derived fact types in the metafragment Figure 3-1 ensure that for each object type, all its dot names names are distinct. This ensures that navigational access via ObjectType.dotRole expressions unambiguously determines the relevant role. For example, in the model shown in Figure 3-2, the named far role names of Person can be accessed as Person.brother, Person.isSmoker, Person.carDriven and Person.carOwned, the far roles of Car can be accessed as Car.driver and Car.owner, and the far role of Dog can be accessed as Dog.owner.

![Figure 3-2 Simple Example of Far RoleNames and Dot Role Names](image-url)
3.2 Fact Type Names

ORM Diagram

![Fact Type Diagram](image)

Figure 3-3 Fact Type Names

Verbalization (of newly introduced aspects)

Object Types and Reference Schemes

FactTypeName is a value type.

Asserted Fact Types

FactType has FactTypeName. / FactTypeName is of FactType.

Constraints

- Each FactType has at most one FactTypeName.
- Each FactTypeName is of at most one FactType.

Explanation

Fact types are primarily identified by their FactTypeSID, and may also be identified by their one or more Fact Type Readings. Optionally, they may also be assigned a FactTypeName. Some FBM methods require fact type names, and some FBM tools use these names to facilitate generation of element names for various mapping targets (e.g. relational table names).

Different FBM tool vendors may decide whether to include support for fact type names as well as how such names are determined. For example, fact type names could be based at least partly on fact type readings (e.g. "PersonDrivesCar"), or be numbered codes (e.g. "FT1"), or be simply entered by the modeller.
3.3 Fact Type Predicates (Constrained)

ORM Diagram

Verbalization (of newly introduced aspects)

Derived Fact Types

*Predicate has Arity if and only if
Arity = count(each Role that orders that Predicate at some Position).

Constraints

Each Predicate has exactly one Arity.
It is possible that more than one Predicate has the same Arity.

If some FactType has some Predicate that orders some Role at some Position then that FactType contains that Role.

For each Predicate and Arity,
that Predicate has that Arity if and only if
that Predicate orders roles of some FactType that has that Arity.

Explanation

The Predicate note on the diagram indicates that predicates may also be identified by their extension (so each ordered set of roles corresponds to at most one predicate).

The role pair in the first argument of the subset constraint is projected from a join path that joins the roles hosted by Predicate in two asserted fact types.

The role pair in the second argument of the equality constraint is projected from a join path that joins the roles hosted by FactType in the two fact types mentioned in the constraint verbalization. This equality constraint is depicted in red colour to indicate that it is asserted (not derivable), even though it includes a derived fact type in its arguments.
3.4 Predicate Readings (Constrained)

ORM Diagram

Figure 3-5  Predicate Readings (Constrained)

Verbalization (of newly introduced aspects)

Constraints

If for some Role, some PredicateReading attaches text to that Role and that PredicateReading is of some Predicate then that Role occupies some Position in that Predicate.

Explanation

This simply adds a join-subset constraint to the relevant metamodel fragment considered in the previous chapter. The argument at the subset end of the constraint is projected from the join path formed by joining the PredicateReading roles in the fact types PredicateReading attaches text to Role and PredicateReading is of Predicate.
3.5 Fact Type Readings (Derivations)

**ORM Diagram**

Derivation Rules:

*Predicate has ObjectTypeName at Position if and only if that Predicate orders some Role at that Position where that Role is played by some ObjectType that has that primary ObjectTypeName or has that alternative ObjectTypeName.*

*ReadingRoleText1 has ExpandedText if and only if that ReadingRoleText1 involves some Role and involves some PredicateReading that is of some Predicate that orders that Role at some Position and has some ObjectTypeName at that Position and for some ReadingText1, that ReadingRoleText1 has that pre-bound ReadingText1 or that ReadingRoleText1 has no pre-bound ReadingText

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Position (Nr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>... has at ... *</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PredicateReading</th>
<th>Position (Nr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>... has ... at ... *</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FactType</th>
</tr>
</thead>
<tbody>
<tr>
<td>has *</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ObjectTypeName</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1..)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ExpandedText</th>
</tr>
</thead>
<tbody>
<tr>
<td>has *</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FactTypeReading</th>
</tr>
</thead>
<tbody>
<tr>
<td>has *</td>
</tr>
</tbody>
</table>

Derivation Rules: (continued)

*PredicateReading has ExpandedText at Position if and only if that PredicateReading is of some Predicate1 that orders some Role1 at that Position and that PredicateReading is involved in some ReadingRoleText that involves that Role1 and has that ExpandedText or that PredicateReading is of some Predicate2 where it is not true that that PredicateReading attaches text to that Role2 and that Predicate2 has some ObjectTypeName that is that ExpandedText at that Position.

*PredicateReading has cumulative ExpandedText1 at Position1 if and only if that PredicateReading has some front ReadingText1 and has some ExpandedText2 at Position1=1 where ExpandedText1 = concat(ReadableText1, ExpandedText2) or that PredicateReading has no front ReadingText

<table>
<thead>
<tr>
<th>PredicateReading</th>
</tr>
</thead>
<tbody>
<tr>
<td>... has cumulative ... at ... *</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FactType</th>
</tr>
</thead>
<tbody>
<tr>
<td>has *</td>
</tr>
</tbody>
</table>

**Explanation**

Informally, a fact type reading (e.g. “Person drives Car”) may be derived from a predicate reading (e.g. “... drives ...”) for that fact type by inserting the object type names (e.g. “Person”, “Car”) in the relevant object slots of that predicate reading. This metamodel fragment is included simply to enable this rule for deriving fact type readings to be formally specified.

The top five derived fact types in Figure 3-6 are introduced as helper fact types to enable the final derivation rule for FactType has FactTypeReading to be simply stated. The textual formulation of the six derivation rules is provided in the Note within Figure 3-6.
3.6 Predicate Reading Order

ORM Diagram (exchange version)

Figure 3-7 Predicate Reading Order (exchange version)

Verbalization (of newly introduced aspects)

Asserted Fact Types

PredicateReading has preference Position for its fact type.

Derived Fact Types

See Figure 3-7.

Constraints

Each PredicateReading has exactly one preference Position for its fact type.
It is possible that more than one PredicateReading has the same preference Position for its fact type.

Each PredicateReading describes exactly one FactType.
For each FactType, some PredicateReading describes that FactType.
It is possible that more than one PredicateReading describes the same FactType.

Each PredicateReading is preferred for at most one Predicate.
For each Predicate, exactly one PredicateReading is preferred for that Predicate.

Each PredicateReading is preferred for at most one FactType.
For each FactType, exactly one PredicateReading is preferred for that FactType.
For each FactType and preference Position, at most one PredicateReading describes that FactType and has that preference Position for its fact type.

Explanation

FBM allows a fact type to have many predicates, each of which determines one ordering of roles within the fact type. Although rare in practice, a predicate may also be given many readings. For exchange purposes, the various predicate readings for any given fact type need to be stored in a specific order. This metamodel fragment directly supports this exchange requirement by recording the preference order for all such readings.

If instead we wish to focus on purely semantic aspects of relevance to the business domain, it is sufficient to restrict the recording of preferences only to the preferred predicate for each fact type and the preferred reading for each predicate, as shown in Figure 3-8.

![Diagram](image)

**Figure 3-8** Predicate Reading Order (business semantics version)
3.7 Subtyping: Type Consistency

ORM Diagram

![ ORM Diagram showing relationships between entity types, value types, and data types. ]

<table>
<thead>
<tr>
<th>Derivation Rules (the first 3 rules assume implicit subtyping connections):</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Subtyping is between entity types if and only if</em> &lt;br&gt;that Subtyping involves some subtype EntityType &lt;br&gt;and that Subtyping involves some supertype EntityType.</td>
</tr>
<tr>
<td><em>Subtyping is between value types if and only if</em> &lt;br&gt;that Subtyping involves some subtype ValueType &lt;br&gt;and that Subtyping involves some supertype ValueType.</td>
</tr>
<tr>
<td><em>Subtyping is between data types if and only if</em> &lt;br&gt;that Subtyping involves some subtype DataType &lt;br&gt;and that Subtyping involves some supertype DataType.</td>
</tr>
</tbody>
</table>

(The exclusion constraint is implied by these derivation rules. <br>The emphasized disjunctive mandatory constraint is not implied.)

*ValueType1 is a direct subtype of ValueType2 if and only if* <br>that ValueType1 is some ObjectType1 that is a direct subtype of some ObjectType2 that is that ValueType2.

*ValueType1 can be a subtype of ValueType2 if and only if* <br>that ValueType1 maps to some DataType1 <br>or that ValueType1 maps to some DataType2 <br>and that ValueType2 maps to some DataType3 that is a supertype of that DataType2.

Figure 3-9 Subtyping: Type Consistency

Verbalization (of newly introduced aspects)

Derived Fact Types

See Figure 3-9.

Constraints

For each Subtyping, at most one of the following holds:

that Subtyping is between entity types;

that Subtyping is between value types;

that Subtyping is between data types.

Each Subtyping is between entity types

or is between value types

or is between data types.

It is possible that some ValueType is a direct subtype of more than one ValueType

and that for some ValueType, more than one ValueType is a direct subtype of that ValueType.

In each population of ValueType is a direct subtype of ValueType,

each ValueType, ValueType combination occurs at most once.
It is possible that some `ValueType` can be a subtype of more than one `ValueType` and that for some `ValueType_i`, more than one `ValueType` can be a subtype of that `ValueType_i`.

In each population of `ValueType` can be a subtype of `ValueType_i`, each `ValueType, ValueType` combination occurs at most once.

If some ` ValueType_i` is a direct subtype of some ` ValueType_j`, then that ` ValueType_i` can be a subtype of that ` ValueType_j`.

**Explanation**

The derived fact types are introduced to enable type restriction consistency conditions to be declared on subtyping relationships (e.g. no entity type may be declared to be a subtype of a value type). The inclusive-or constraint and the subset constraint in Figure 3-9 are coloured red to indicate that they are asserted (i.e. not implied by other constraints or derivation rules).
3.8 Subtyping: Identity Fact Type

ORM Diagram

![ORM Diagram]

**Derivation Rules:**

*IdentityFactType has subtype Role if and only if some Subtyping corresponds to that IdentityFactType and involves some subtype ObjectType and that IdentityFactType is some FactType that contains that Role that is hosted by that ObjectType.*

*IdentityFactType has supertype Role if and only if some Subtyping corresponds to that IdentityFactType and involves some supertype ObjectType and that IdentityFactType is some FactType that contains that Role that is hosted by that ObjectType.*

**Implementation note:**
It is expected that the IdentityFactType structure may be automatically created by a tool when a Subtyping relationship is specified. Although a formal creation rule describing this operation is beyond the scope of this model, the requirements for such a creation rule can be clearly inferred from the subtype and subset constraints on this diagram.

**Figure 3-10** Subtyping: Identity Fact Type

**Verbalization** (of newly introduced aspects)

**Object Types, Reference Schemes and Subtyping**

- **IdentityFactType** is an entity type.
  - Each IdentityFactType is an instance of BinaryFactType.
  - Reference Scheme: FactType has FactTypeSID.
  - Reference Mode: .SID.

**Asserted Fact Types**

Subtyping corresponds to IdentityFactType.

**Derived Fact Types**

See Figure 3-10.

**Constraints**

- Each Subtyping corresponds to at most one IdentityFactType.
- For each IdentityFactType, exactly one Subtyping corresponds to that IdentityFactType.
Each IdentityFactType has **exactly one** subtype Role.
For each subtype Role, **at most one** IdentityFactType has **that** subtype Role.

Each IdentityFactType has **exactly one** supertype Role.
For each supertype Role, **at most one** IdentityFactType has **that** supertype Role.

For each Role,
if some IdentityFactType has **that** subtype Role then **that** Role is mandatory.

For each Role,
if some IdentityFactType has **that** subtype Role then **that** Role is functional.

For each Role,
if some IdentityFactType has **that** supertype Role then **that** Role is functional.

**Explanation**

*Identity fact types* were explained earlier in the discussion accompanying Figure 2-5. For example, declaring Woman as a subtype of Person entails that each woman is a person, and hence each instance in the population of a subtype relates via an instance level identity relationship (given the predicate reading “is”) to an instance of the supertype (e.g. the woman Cleopatra is the person Cleopatra).

The metamodel fragment in Figure 3-10 is used to establish to the correspondence between the subtyping metarelationship (between types) and the instance level identity relationship. The subset constraints in Figure 3-10 are coloured red to indicate that they are asserted (i.e. not implied by other constraints or derivation rules).
3.9 Subtyping: Compatibility

ORM Diagram

![ORM Diagram](image)

**Derivation Rules:**

- ObjectType1 is a subtype of ObjectType2 if and only if ObjectType1 is a direct subtype of ObjectType2 or is a direct subtype of some ObjectType3 that is a subtype of ObjectType2.
- ObjectType1 has ancestor supertype Role if and only if ObjectType1 is involved as subtype in some Subtyping that corresponds to some IdentityFactType that has that supertype Role or that Subtyping involves some supertype ObjectType2 that has that ancestor supertype Role.
- ObjectType1 shares a supertype with ObjectType2 on another branch if and only if some ObjectType3 is a supertype of ObjectType1 and is a supertype of ObjectType2 and it is not true that ObjectType2 is a subtype of ObjectType1 and it is not true that ObjectType2 is a supertype of ObjectType1.
- ObjectType1 is a disjoint relative of ObjectType2 if and only if ObjectType1 shares a supertype with ObjectType2 on another branch and has some ancestor supertype Role1 and that ObjectType2 has some ancestor supertype Role2 and for some ExclusionConstraint, that ExclusionConstraint is some SetComparisonConstraint that has ArgumentLength=1 and that ExclusionConstraint is some Constraint that has Modality='Alethic' and that SetComparisonConstraint compares some SetComparisonArgument1 that includes that Role1 at some Position and that SetComparisonConstraint compares some SetComparisonArgument2 that includes that Role2 at some Position where Role1 <> Role2.
- ObjectType1 is compatible with ObjectType2 if and only if ObjectType1 is a subtype of ObjectType2 or is a supertype of ObjectType2 or is a subtype of some ObjectType3 that is a supertype of ObjectType2 where it is not true that ObjectType2 is a disjoint relative of ObjectType1 or ObjectType1=ObjectType2.
- Role1 is compatible with Role2 if and only if Role1 is played by some ObjectType1 and that Role2 is played by some ObjectType2 that is compatible with that ObjectType1 where Role1 <> Role2.
- ObjectType1 has direct supertype ObjectType2 distinct from other direct supertype ObjectType3 if and only if ObjectType1 is a direct supertype of ObjectType2 and is a direct supertype of some ObjectType3 where ObjectType2 <> ObjectType3.

**Figure 3-11** Subtyping: Compatibility

**Verbalization** (of main, newly introduced aspects)

**Derived Fact Types**

See Figure 3-11.
**Subset Constraint**

If some ObjectType has some direct supertype ObjectType, distinct from some other direct supertype ObjectType, then that ObjectType is compatible with that ObjectType.

**Explanation**

This metamodel fragment formally defines what it means for two object types to be compatible and for two roles to be compatible. The basic intuition is that types (or roles) are compatible if they can share a common instance.

Figure 3-12 provides an example to clarify the meaning of the fact type ObjectType shares a supertype with ObjectType on another branch.

![Diagram](figure3-12.png)

*Figure 3-12*  OT₁ shares a supertype with OT₂ on another branch in case (a), but not in cases (b) or (c)*
3.10 Independence and Implied Mandatory Role Constraints

**OR M Diagram**

![Diagram showing independence and implied mandatory role constraints.]

**Derivation Rules:**

*DomainObjectType can be independent if and only if*
- it is not true that that DomainObjectType is some ObjectType that hosts some Role1 that is restricted by some AlethicMandatoryConstraint and that Role1 belongs to no FactType1 that is existential for some EntityType that is DomainObjectType and that Role1 belongs to no FactType2 that has InstantiationMethod='Derived' and it is not true that some FactType3 is objectified by that DomainObjectType that is directly identified by some AlethicUniquenessConstraint that is internal and restricts some Role2 at Position=1 where that Role2 belongs to that FactType3 where it is not true that that FactType3 has that spanning AlethicUniquenessConstraint.

*(The second half of the 'can be independent' rule (regarding a non-spanning internal uniqueness constraint on an objectified fact type where that uniqueness identifies the objectifying domain object type) is implied by the first half of the rule if link fact types are always created for the objectification. However, link fact types are optional in this model, so this is not implied.)*

*DomainObjectType implies a mandatory constraint on Role if and only if*
- that DomainObjectType can be independent and that Role1 is independent and that DomainObjectType hosts that Role that is restricted by no AlethicMandatoryConstraint and that Role belongs to no FactType1 that is existential for some EntityType that is DomainObjectType and that Role belongs to no FactType2 that has InstantiationMethod='Derived'.

*(The non-emphasized subset and exclusion constraints are implied by this derivation rule.)*

Figure 3-13 Determining when a domain object type can be independent, and implied mandatory role constraints

**Verbalization (of main, newly introduced aspects)**

**Derived Fact Types**

See Figure 3-13.

**Subset and Exclusion Constraints**

If some DomainObjectType is independent then that DomainObjectType can be independent.

If some DomainObjectType implies a mandatory constraint on some Role then that DomainObjectType can be independent.

For each DomainObjectType, at most one of the following holds:
- that DomainObjectType implies a mandatory constraint on some Role;
- that DomainObjectType is independent.
Explanation

This metamodel fragment establishes necessary conditions for a domain object type to be independent. For example, if an entity type hosts a mandatory role in a fact type that is not part of its reference scheme, then it cannot be independent.

The metamodel fragment also determines when a mandatory role constraint (simple or disjunctive) is implied in the global schema. A simple mandatory role constraint applies to exactly one role, while a disjunctive mandatory (also called inclusive-or) constraint applies to a combination of two or more roles. A referential role (also called an existential role) of an entity type is a role it hosts within its reference scheme. A domain value type has no referential roles (since it has no reference scheme). The rule to determine implicit mandatory role constraints may be informally stated as follows:

For each non-independent domain object type with no explicit mandatory role constraint, an implicit mandatory role constraint spans all its non-referential roles.

While a tool vendor might choose to include an option to display implied mandatory constraints on request, this standard does not require it.

Use of implied mandatory role constraints has three main advantages: (a) it facilitates schema changes by avoiding continual change of mandatory constraints as roles are added; (b) it enables schema equivalence and implication theorems to be stated in general form; and (c) it distinguishes the important mandatory role constraints from ones with no implementation consequence.

As a simple example, suppose we always wish to record the current name of employees. Figure 3-14 (a) below reflects this decision, while allowing that at some later time FamilyName might play other roles, so the implied mandatory role constraint on FamilyName is left implicit rather than being made explicit as in Figure 3-14(b). Now suppose that later we wish to record the previous family name (if any) of employees (e.g. a name change on recent marriage or for some other reason). This is best handled by Figure 3-14(c), which simply adds a new fact type and some constraints, with no impact at all on the original schema fragment Figure 3-14(a). There is now an implied disjunctive mandatory constraint between the two roles played by FamilyName, as shown explicitly in Figure 3-14(d).

Note that if we had started with the explicit schema Figure 3-14(b), we would have to modify this schema fragment by removing the simple mandatory role constraint on FamilyName, and then explicitly add the disjunctive mandatory role constraint. The implicit approach leads to greater stability and less work. Moreover, if we map the schemas to a relational database the implicit mandatory constraints come for free, with no implementation overhead.

![Figure 3-14 Implied mandatory role constraints simplify schema evolution](image-url)
3.11 Objectification: Link Fact Types

ORM Diagram

Derivation Rules:
*Role is in objectified fact type if and only if that Role belongs to some FactType that is objectified by some DomainObjectType.

*Role1 has mirror Role2 in link fact type if and only if that Role1 is hosted by some ObjectType and implies some link BinaryFactType that is some FactType that contains that Role2 that is hosted by that ObjectType.

*Role1 has objectifying Role2 in link fact type if and only if that Role1 implies some link BinaryFactType that contains that Role2 that is hosted by some DomainObjectType that objectifies some FactType that contains that Role1.

This model is currently silent on the constraints associated with a mirror role. The associated discussion is placed outside this model.

Similarly, no fact type is provided for implied objectification. Objectification may be implied for all fact types that can be asserted that are either n-ary, or binary with a spanning uniqueness constraint.

Figure 3-15 Objectification: Link Fact Types

Verbalization (of main, newly introduced aspects)

Asserted Fact Types

Role implies link BinaryFactType. / link BinaryFactType is implied by Role.

Derived Fact Types

See Figure 3-15.

Constraints

Each Role implies at most one link BinaryFactType.
For each link BinaryFactType, at most one Role implies that link BinaryFactType.
For each mirror Role, at most one Role has that mirror Role in link fact type.
Each Role has at most one mirror Role in link fact type.
Each Role has at most one objectifying Role in link fact type.
For each objectifying Role, at most one Role has that objectifying Role in link fact type.
If some Role implies some link BinaryFactType then that Role is in objectified fact type.
For each Role, if some Role has that objectifying Role in link fact type then that Role is mandatory.

For each Role, if some Role has that objectifying Role in link fact type then that Role is functional.

For each Role, that Role has some objectifying Role in link fact type if and only if that Role implies some link BinaryFactType.

For each Role, that Role has some mirror Role in link fact type if and only if that Role implies some link BinaryFactType.

For each FactType and Role, at most one of the following holds:
that FactType contains that Role;
that Role implies some link BinaryFactType that is that FactType.

Explanation

The ideas are most easily understood by example. Figure 3-16(a) shows the usual graphical depiction of the objectification of the fact type Student enrolled in Course as the object type Enrolment. Figure 3-16(b) shows how this is actually interpreted, by displaying the implicit link fact types Enrolment is of Student and Enrolment is of Course. Link fact types are displayed with their role boxes bordered by dashed lines, to distinguish them from normal fact types.

The Enrolment object type has a composite, preferred reference scheme based on the external uniqueness constraint shown. For example, given the fact that "Student s1 enrolled in Course c1", the object resulting from the objectification of this fact may be referenced by the definite description "the Enrolment that is of Student s1 and is in Course c1". The join equality constraint shown ensures the 1:1 correspondence between enrolment object and enrolment facts.

Using the surrogate identifiers provided in Figure 3-16(b), object type OT (i.e. Enrolment) objectifies fact type FT (i.e. Student enrolled in Course), role r1 has role r2 as its mirror role and role r4 as its objectifying role. Role r2 has role r5 as its mirror role and role r6 as its objectifying role.

\[\text{Figure 3-16 Link fact types are displayed using dashed lines}\]
3.12 Facets

ORM Diagram

![Facets ORM Diagram](image)

Verbalization (of main, newly introduced aspects)

**Object Types, Reference Schemes and Subtyping**

- **Facet** is an entity type.
  - Reference Scheme: Facet has FacetSID.
  - Reference Mode: .SID.

- **FacetName** is a value type.

- **FacetRestrictionStyle** is an entity type.
  - Reference Scheme: FacetRestrictionStyle has FacetRestrictionStyleName.
  - Reference Mode: .Name.

- **FacetRestriction** is an entity type.
  - FacetRestriction objectifies "ObjectType restricts Facet".

**Asserted Fact Types**

- Facet draws values from DataType.
- DataType defines Facet.
- Facet has FacetName.
- Facet has FacetRestrictionStyle.
- Facet requires restriction.
Facet has default CanonicalDataValue.
FacetRestriction has CanonicalDataValue.
ObjectType restricts Facet.
FacetRestriction is fixed.

Constraints

- Each Facet draws values from exactly one DataType.
- It is possible that more than one Facet draws values from the same DataType.
- For each Facet, exactly one DataType defines that Facet.
- It is possible that some DataType defines more than one Facet.
- Each Facet has exactly one FacetName.
- It is possible that more than one Facet has the same FacetName.
- Each Facet has exactly one FacetRestrictionStyle.
- It is possible that more than one Facet has the same FacetRestrictionStyle.
- In each population of Facet requires restriction, each Facet occurs at most once.
- Each Facet has at most one default CanonicalDataValue.
- It is possible that more than one Facet has the same default CanonicalDataValue.
- Each FacetRestriction has exactly one CanonicalDataValue.
- It is possible that more than one FacetRestriction has the same CanonicalDataValue.
- It is possible that some ObjectType restricts more than one Facet and that for some Facet, more than one ObjectType restricts that Facet.
- In each population of ObjectType restricts Facet, each ObjectType, Facet combination occurs at most once.
- In each population of FacetRestriction is fixed, each FacetRestriction occurs at most once.
- For each DataType and FacetName, there is at most one Facet such that that DataType defines that Facet and that Facet has that FacetName.
- For each Facet, at most one of the following holds:
  - that Facet has some default CanonicalDataValue;
  - that Facet requires restriction.
- The possible values of FacetRestrictionStyleName in FacetRestrictionStyle has FacetRestrictionStyleName are 'FixedValue', 'AscendingRestriction', 'DescendingRestriction'.

Explanation

An example of a facet the datatype String is length (number of characters allowed). Facet restriction styles are discussed in the Note in Figure 3-17. Fine detailed constraints on facets are provided in the next section.
3.13 Facets (Constrained)

ORM Diagram

![ORM Diagram of Facets (Constrained)](image)

Join Constraints:

1. (RestrictedFacetMatchesDataType Subset Constraint)
   If some ObjectType restricts some Facet
   then that ObjectType bases value restrictions on some DataType that defines that Facet.

2. (FacetDefaultValueMatchesFacetDataType Subset Constraint)
   If some Facet draws values from some DataType1 and has some default CanonicalDataValue that is of some DataType2
   then some ObjectType that is that DataType1 allows values from that DataType2.

3. (FacetRestrictionValuesMatchFacetDataType Subset Constraint)
   If some Facet draws values from some DataType1 and is involved in some FacetRestriction that has some CanonicalDataValue that is of some DataType2
   then some ObjectType that is that DataType1 allows values from that DataType2.

Figure 3-18  Asserted subset constraints on Facets

**Verbalization** (of newly introduced aspects)

**Asserted Subset Constraints**

**If some ObjectType restricts some Facet then that ObjectType can restrict values.**

**If some ObjectType restricts some Facet then that ObjectType bases value restrictions on some DataType that defines that Facet.**

**If some Facet draws values from some DataType, and has some default CanonicalDataValue, that is some DataValue that is some CanonicalDataValue and is of some DataType, then some ObjectType that is that DataType1 allows values from that DataType2.**

**If some Facet draws values from some DataType, and is involved in some FacetRestriction that has some CanonicalDataValue that is some DataValue that is of some DataType, then some ObjectType that is that DataType1 allows values from that DataType2.**
3.14 Scalar and Domain-specific Fact Types

ORM Diagram

![Diagram showing the relationship between FactType and scalar or domain-specific attributes.]

**Derivation Rules:**
*FactType is scalar if and only if each Role that belongs to that FactType is hosted by some DataType.

*FactType is domain-specific if and only if each Role that belongs to that FactType is hosted by some DomainObjectType.

(Note: The disjunctive mandatory constraint is asserted (as indicated by red color) The exclusion constraint is implied.

**Figure 3-19** Scalar and Domain-specific Fact Types

**Verbalization (of main, newly introduced aspects)**

**Derived Fact Types**

See Figure 3-19.

**Constraints**

*For each FactType, at most one of the following holds:*

  * that FactType is scalar;
  * that FactType is domain-specific.

*Each FactType is scalar or is domain-specific.*
3.15 Comparison Operations

ORM Diagram

Derivation Rules:

*FactType can be comparison operation if and only if
that FactType is scalar
and has some BinaryFactType
and contains some Role1 that is hosted by some ObjectType
and that FactType contains some Role2 that is hosted by that ObjectType
where Role1 <> Role2.

*ComparisonOperation is defined on DataType if and only if
that ComparisonOperation is some FactType
that contains some Role that is hosted by some ObjectType that is DataType.

*ComparisonOperation compares values for DataType1 if and only if
that ComparisonOperation is defined on that DataType1
or that DataType1 is some ObjectType1 that is a subtype of some ObjectType2
where that ComparisonOperation is defined on some DataType2 that is that ObjectType2.

*DataType is equatable if and only if
some ComparisonOperation compares values for that DataType
and uses some StandardComparator where the possible values of that StandardComparator are ‘Equals’, ‘NotEquals’.

*DataType is strictly ordered if and only if
some ComparisonOperation compares values for that DataType
and uses some StandardComparator where the possible values of that StandardComparator are ‘LessThan’, ‘GreaterThan’.

Figure 3-20 Scalar and Domain-specific Fact Types

Verbalization (of main, newly introduced aspects)

Object Types, Reference Schemes and Subtyping

ComparisonOperation is an entity type.
* Each ComparisonOperation is by definition some FactType that is comparison operation.
Reference Scheme: FactType has FactTypeSID.
Reference Mode: .SID.

StandardComparator is an entity type.
Reference Scheme: StandardComparator has StandardComparatorName.
Reference Mode: .Name.
**Asserted Fact Types**

- FactType is comparison operation.
- ComparisonOperation uses StandardComparator.

**Derived Fact Types**

- See Figure 3-20.

**Constraints**

- In each population of FactType is comparison operation, each FactType occurs at most once.
- In each population of DataType is equatable, each DataType occurs at most once.
- In each population of FactType can be comparison operation, each FactType occurs at most once.
- In each population of DataType is strictly ordered, each DataType occurs at most once.
- Each ComparisonOperation uses at most one StandardComparator.
- It is possible that more than one ComparisonOperation uses the same StandardComparator.
- Each ComparisonOperation is defined on exactly one DataType.
- It is possible that more than one ComparisonOperation is defined on the same DataType.
- It is possible that some ComparisonOperation compares values for more than one DataType and that for some DataType, more than one ComparisonOperation compares values for that DataType.
- In each population of ComparisonOperation compares values for DataType, each ComparisonOperation, DataType combination occurs at most once.
- Each ComparisonOperation compares values for some DataType.
- Each ComparisonOperation uses at most one StandardComparator.
- It is possible that more than one ComparisonOperation uses the same StandardComparator.
- The possible values of StandardComparatorName in StandardComparator has StandardComparatorName are 'Equals', 'NotEquals', 'LessThan', 'LessThanOrEqual', 'GreaterThanOrEqual', 'GreaterThan'.
- If some FactType is comparison operation then that FactType can be comparison operation.
3.16 Value Comparability

ORM Diagram

![ORM Diagram](image)

Derivation Rules:
*ValueType1 is comparable to ValueType2 if and only if
  for some DataType,
  that ValueType1 bases value restrictions on that DataType
  and that ValueType2 bases value restrictions on that DataType
  and that DataType is strictly ordered.

*DomainObjectType1 is value comparable with DomainObjectType2 if and only if
  that DomainObjectType1 has a simple identifier based on some ValueType1
  and that DomainObjectType2 has a simple identifier based on some ValueType2 that is comparable to that ValueType1.

*Role1 is value comparable with Role2 if and only if
  that Role1 is hosted by some DomainObjectType1
  and that Role2 is hosted by some DomainObjectType2 that is value comparable with that DomainObjectType1
  where Role1 <> Role2.

Figure 3-21 Value Comparability

Verbalization (of main, newly introduced aspects)

Derived Fact Types

See Figure 3-21.

Constraints

It is possible that some ValueType is comparable to more than one ValueType
and that for some ValueType, more than one ValueType is comparable to that ValueType.

In each population of ValueType is comparable to ValueType, each ValueType, ValueType combination occurs at most once.

If ValueType1 is comparable to ValueType2,
then ValueType2 is comparable to ValueType1.

It is possible that some DomainObjectType is value comparable with more than one
DomainObjectType and that for some DomainObjectType, more than one DomainObjectType is value comparable with that DomainObjectType.

In each population of DomainObjectType is value comparable with DomainObjectType, each DomainObjectType, DomainObjectType combination occurs at most once.

It is possible that some Role is value comparable with more than one Role
and that for some Role, more than one Role is value comparable with that Role.

In each population of Role is value comparable with Role, each Role, Role combination occurs at most once.
Types are compatible only if they may share instances. Value types are comparable if their data types are compatible and support comparison operations. For example, unless declared as subtypes of a common type (e.g. Name), FamilyName and CompanyName are incompatible; but they are still comparable if they are represented by mapping to compatible datatypes (e.g. variable length character strings) on which comparators are defined. The comparison condition "FamilyName = CompanyName" is then interpreted as a check that the character strings representing the names are equal.

In Figure 3-21, the external shape attached to the fact type ValueType is comparable to ValueType depicts a symmetric ring constraint, so if one value type is comparable to a second value type, the second value type must be comparable to the first.

Entity types whose reference schemes are reference modes may be value-compared if the value types used to reference them are compatible. For example, if the entity types Height and Width are referenced by comparable value types (e.g. cmValue, mmValue) then the comparison condition "Height > Width" is interpreted as a comparison between their values. In the interests of enabling shortcuts for expressing comparisons, this version of the metamodel allows even comparisons such as "Product > Company" if their referencing values types are comparable. This involves automatically expanding such conditions to longer formulations based on comparison of their referencing values (e.g. "ProductName > CompanyName"). It is expected that tools should make such expansions explicit when reacting to such user input. It is anticipated that a future version of the metamodel will better address such issues by providing deeper support for reference schemes, especially unit-based ones.

Similarly, roles that are hosted by comparable value types or by value-comparable entity types are also value-comparable.
3.17 Ring Constraint Combinations

ORM Diagram

Figure 3-22  Ring Constraint Combinations

Verbalization (of main, newly introduced aspects)

Derived Fact Types

See Figure 3-22.

Exclusion Constraints (implicit uniqueness constraints on unary fact types are omitted here)

For each RingConstraint, at most one of the following holds:
that RingConstraint is irreflexive;
that RingConstraint is antisymmetric.

For each RingConstraint, at most one of the following holds:
that RingConstraint is asymmetric;
that RingConstraint is acyclic;
that RingConstraint is symmetric.

For each RingConstraint, at most one of the following holds:
that RingConstraint is intransitive;
that RingConstraint is strongly intransitive;
that RingConstraint is transitive.
For each RingConstraint, at most one of the following holds:

- that RingConstraint is irreflexive;
- that RingConstraint is implicitly irreflexive;
- that RingConstraint is reflexive.

For each RingConstraint, at most one of the following holds:

- that RingConstraint is antisymmetric;
- that RingConstraint is implicitly antisymmetric;
- that RingConstraint is symmetric.

**Explanation**

The exclusion constraints indicate combinations of individual ring constraints that are not allowed. The final two exclusion constraints verbalized above involve a derived fact type, but are asserted (as indicated by the red colour).

The Ring Combination Notes in Figure 3-22 summarize the allowed combinations of ring constraints, as well as some of the main implications that hold between ring constraints. To simplify the task of combining constraints, combinations that include implied ring constraints are not allowed. For example, “acyclic + asymmetric + irreflexive” is not allowed, because acyclicity implies asymmetry, which itself implies irreflexivity.
3.18 Join Path Specification

ORM Diagram

Derivation Rules:

*JoinPathSpecification applies to constrained Role if and only if that JoinPathSpecification applies to some UniquenessConstraint that restricts that Role at some Position or that JoinPathSpecification applies to some FrequencyConstraint that restricts that Role or that JoinPathSpecification applies to some RingConstraint that restricts that first Role or restricts that second Role or that JoinPathSpecification applies to some ValueComparisonConstraint that compares that first Role or compares that second Role or that JoinPathSpecification applies to some SetComparisonArgument that includes that Role at some Position.

*UniquenessConstraint restricts multiple fact types if and only if that UniquenessConstraint restricts some Role1 that belongs to some FactType1 and that UniquenessConstraint restricts some Role2 that belongs to some FactType2 where FactType1 <> FactType2.

(Derivation rules for frequency, ring and value comparison constraint restricts multiple fact types use the same pattern as the above rule.)

*SetComparisonArgument restricts multiple fact types if and only if that SetComparisonArgument includes some Role1 at some Position where that Role1 belongs to some FactType1 and that SetComparisonArgument includes some Role2 at some Position where that Role2 belongs to some FactType2 that <> FactType1.

Figure 3-23  Join Path Specification
Verbalization (of main, newly introduced aspects)

Object Types and Reference Schemes

JoinPathSpecification is an entity type.
Reference Scheme: JoinPathSpecification has JoinPathSpecificationSID.
Reference Mode: SID.

Asserted Fact Types

JoinPathSpecification applies to UniquenessConstraint.
JoinPathSpecification applies to FrequencyConstraint.
JoinPathSpecification applies to RingConstraint.
JoinPathSpecification applies to ValueComparisonConstraint.
JoinPathSpecification applies to SetComparisonArgument.
JoinPathSpecification is expressed in FormalLanguage.

Derived Fact Types

See Figure 3-23.

Exclusive-or Constraint and an Equality Constraint (For other constraints, see Figure 3-23)

For each JoinPathSpecification, exactly one of the following holds:
that JoinPathSpecification applies to some UniquenessConstraint;
that JoinPathSpecification applies to some FrequencyConstraint;
that JoinPathSpecification applies to some RingConstraint;
that JoinPathSpecification applies to some ValueComparisonConstraint;
that JoinPathSpecification applies to some SetComparisonArgument.

For each UniquenessConstraint,
some JoinPathSpecification applies to that UniquenessConstraint
if and only if
that UniquenessConstraint restricts multiple fact types.
3.19 Join Lists (Advanced)

ORM Diagram

**Derivation Rules:**
- **Join** fromRole subsets toRole if and only if that Join is to some Role1 that is mandatory or that Join is from some Role4 that is included in some SetComparisonArgument1 at some Position where that SetComparisonArgument1 is subset for some SubsetConstraint that has some superset SetComparisonArgument2 that includes some Role5 at that Position where that Join is to that Role5 and that SubsetConstraint is some AlethicSubsetConstraint.
- **Join** is outer if and only if that Join has JoinType='Outer'.

**Verbalization** (of main, newly introduced aspects)

**Derived Fact Types**

See Figure 3-24.

**Exclusion Constraint** (for other constraints, see Figure 3-24)

**For each** Join, **at most one of the following holds:**
- that Join is outer;
- that Join fromRole subsets toRole.

**Explanation**

Core aspects of join lists were discussed earlier in section 2.31. The derived fact types and the asserted exclusion constraint that are introduced in this metamodel fragment help ensure that join paths within FBM models are well formed.
3.20 Join List Path Specification

ORM Diagram

Derivation Rules:

*Each JoinListPathSpecification is by definition some JoinPathSpecification that is expressed in some FormalLanguage where FormalLanguage="FBMJoinList".*

*JoinFactTypeRoleVariable1 can implicitly project on Role if and only if some JoinListPathSpecification projects roles from some JoinList that JoinList is involved in / involves and that JoinListPathSpecification is some JoinPathSpecification that applies to that constrained Role and represents that Role and no JoinFactTypeRoleVariable2 is available to that JoinList and represents that Role where JoinFactTypeRoleVariable1 <> JoinFactTypeRoleVariable2.*

*JoinFactTypeRoleVariable is implicitly projected on Role if and only if it is not true that the JoinFactTypeRoleVariable is explicitly projected on that Role and that JoinVariable can implicitly project on that Role.*

*JoinFactTypeRoleVariable is projected on Role if and only if that JoinFactTypeRoleVariable is explicitly projected on that Role or is implicitly projected on that Role.*

Join Constraints:

*(JoinListPathProjectsOnConstrainedRole Subset Constraint)*
If some Join is involved in some JoinFactTypeRoleVariable that is explicitly projected on some Role and that Join is in some JoinList where some JoinListPathSpecification projects roles from that JoinList then some JoinPathSpecification that is that JoinListPathSpecification applies to that constrained Role.

*(JoinListPathProjectionIsComplete Equality Constraint)*
For each JoinPathSpecification and Role, that JoinPathSpecification applies to that constrained Role if and only if some JoinFactTypeRoleVariable is available to some JoinList where that JoinPathSpecification is some JoinListPathSpecification that projects roles from that JoinList and that JoinFactTypeRoleVariable is projected on that Role.

*(JoinListPathProjectionIsCompatible Subset Constraint)*
If for some Role1, some JoinFactTypeRoleVariable is projected on that Role1 and that JoinFactTypeRoleVariable represents some Role2 then that Role1 is compatible with that Role2.

Figure 3-25 Join List Path Specification
Verbalization (of main, newly introduced aspects)

Object Types, Reference Schemes and Subtyping

JoinListPathSpecification is an entity type.

* Each JoinListPathSpecification is by definition some JoinPathSpecification that is expressed in some FormalLanguage where the possible value of that FormalLanguage is 'FBMJoinList'.

Reference Scheme: JoinPathSpecification has JoinPathSpecificationSID.

Reference Mode: .SID.

Asserted Fact Types

JoinListPathSpecification projects roles from JoinList.

JoinFactTypeRoleVariable is explicitly projected on Role.

Derived Fact Types

See Figure 3-25.

External Constraints (For other constraints, see Figure 3-25)

If some Join is involved in some JoinFactTypeRoleVariable that is explicitly projected on some Role and that Join is in some JoinList where some JoinListPathSpecification projects roles from that JoinList then some JoinPathSpecification that is that JoinListPathSpecification applies to that constrained Role.

For each JoinPathSpecification and Role, that JoinPathSpecification applies to that constrained Role if and only if some JoinFactTypeRoleVariable is available to some JoinList where that JoinPathSpecification is some JoinListPathSpecification that projects roles from that JoinList and that JoinFactTypeRoleVariable is projected on that Role.

If for some Role1, some JoinFactTypeRoleVariable is projected on that Role1 and that JoinFactTypeRoleVariable represents some Role2 then that Role1 is compatible with that Role2.

If some JoinFactTypeRoleVariable is implicitly projected on some Role then that JoinFactTypeRoleVariable can implicitly project on that Role.

Explanation

A constraint join list is a join list that specifies an ordered set of one or more joins along a role path from which roles are projected (i.e. selected or chosen) to form an argument of a uniqueness, frequency, ring, value-comparison or set-comparison constraint. The new aspects introduced in this metamodel fragment are designed to help ensure that explicit role projections and associated join variables used within join constraints are properly constructed. Some of the derived fact types introduced are also used to help determine implicit projections.

This metamodel fragment, in combination with that of the previous section, is designed to simplify the task of users adding constraints to a model, where those constraints include an argument whose roles are projected from a join path. It does this by enabling FBM modelling tools to automatically determine, where possible, the joins that are implicitly intended by the user.

For example, the external uniqueness constraint depicted in Figure 3-26 applies to roles r2 and r4 that are projected from a role path involving an inner join between roles r1 and r3. The intended join may be inferred without requiring the user to identify it explicitly, as there is no other join possibility for that constraint. Moreover, the derivation rules enable the join variable for Room within this context to be
determined. The external uniqueness constraint verbalization “For each Building and RoomNr, at most one Room is in that Building and has that RoomNr” verbalizes the join variable as “Room”.

Figure 3-26  Example where the join intended for the external uniqueness constraint may be inferred.
3.21 Sample Populations (Constrained)

ORM Diagram

Derivation Rules:

*Each DomainValue is some DomainObject that is an instance of some ValueType.

Join Constraints:

(ObjectificationRequiresTypedObjectObjectifiesFactEquality Constraint)
For each TypedObject and Fact, that TypedObject objectifies that Fact if and only if that TypedObject is typed as some DomainObjectType that is some EntityObject that objectifies some FactType where that Fact is an instance of that FactType.

(FactTypeInstancePopulatesAllFactRoles Equality Constraint)
For each Fact and Role, some DomainObject plays that Role in that Fact if and only if that Fact is an instance of some FactType that contains that Role.

(PopulatedRoleInstanceMatchesRolePlayerType Subset Constraint)
If some DomainObject plays some Role in some Fact then that DomainObject is an instance of some DomainObject that hosts that Role.

(DomainValueHasConsistentDataType Subset Constraint)
If for some DomainObject, some DomainObject is an instance of that DomainObject then some DomainObject that has some CanonicalDataValue that is of some DataType then some ObjectType that is that DomainObject allows values from that DataType.

Figure 3-27 Sample populations (constrained)

Verbalization (of main, newly introduced aspects)

Asserted Fact Types

TypedObject is typed as DomainObjectType.
Join Constraints

If for some DomainObjectType, some DomainObject is an instance of that DomainObjectType
where that DomainObject is some DomainValue that has some CanonicalDataValue
that is some DataValue that is of some DataType
then some ObjectType that is that DomainObjectType allows values from that DataType.

For each TypedObject and Fact,
    that TypedObject objectifies that Fact
    if and only if
    that TypedObject is typed as some DomainObjectType that objectifies some FactType
    where that Fact is an instance of that FactType.

For each Fact and Role,
    some DomainObject plays that Role in that Fact
    if and only if
    that Fact is an instance of some FactType that contains that Role.

If some DomainObject plays some Role in some Fact
then that DomainObject is an instance of some DomainObjectType that hosts that Role.

Explanation

This metamodel fragment simply extends work provided earlier on sample populations (section 2.32) by
displaying an implicit link fact type and adding the four subset/equality constraints verbalized above.
3.22 Sample Populations (Instance Compatibility)

ORM Diagram

![ORM Diagram]

<table>
<thead>
<tr>
<th>Join Constraints:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SubtypeInstanceMustBeSupertypeInstance Subset Constraint)</td>
</tr>
<tr>
<td>If some DomainObject is an instance of some DomainObjectType1 that is some ObjectType1</td>
</tr>
<tr>
<td>then that DomainObject is an instance of some DomainObjectType2 that is some ObjectType2.</td>
</tr>
</tbody>
</table>

| (SharedInstancesMustBeOfCompatibleTypes Exclusion Constraint) |
| For each DomainObject and ObjectType1, at most one of the following holds: |
| that DomainObject is an instance of some DomainObjectType1 that is some ObjectType1; |
| where it is not true that that DomainObjectType2 is some ObjectType2 that is compatible with that ObjectType1. |

**Verbalization (of main, newly introduced aspects)**

**Subset and Exclusion Constraints**

- If some DomainObject is an instance of some DomainObjectType, that is some ObjectType, that is a direct subtype of some ObjectType, then that DomainObject is an instance of some DomainObjectType, that is that ObjectType.

- For each DomainObject and ObjectType, at most one of the following holds:
  - that DomainObject is an instance of some DomainObjectType, that is that ObjectType;
  - that DomainObject is an instance of some DomainObjectType, where it is not true that that DomainObjectType is some ObjectType, that is compatible with that ObjectType.

**Explanation**

This metamodel fragment simply adds the subset and exclusion constraints verbalized above, to help ensure that instances used in sample populations are properly compatible. The subset constraint ensures that each instance in the population of a subtype also occurs in the population of each of its supertypes. The exclusion constraint ensures that the same object cannot be in the population of two incompatible types.
4 Terms and Definitions

For the purposes of this document, the following definitions of terms apply.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
</table>
| acyclic constraint | Ring constraint that restricts the population of the constrained role pair to be an acyclic relation. A binary relation whose roles are played by instances of either the same type or compatible types is acyclic if and only if no instance may cycle back to itself via one or more applications of that relationship.  
Example:  
The fact type "Person is a parent of Person" is acyclic. |
| alethic constraint | Restriction that cannot be violated by any state or state transition of the fact base. It may be expressed positively as a necessity, or negatively by indicating an impossibility.  
Examples:  
Each Person was born on at most one Date.  
No Person is a parent of itself. |
| alethic modality   | Modality of necessity and possibility. Note: The alethic modal operators used are: it is necessary that; it is possible that; it is impossible that. |
| antisymmetric constraint | Ring constraint that restricts the population of the constrained role pair to be an antisymmetric relation. A binary relation R whose roles are played by instances of either the same type or compatible types is antisymmetric if and only if, given any instances x and y where x is not equal to y, if x bears the relation R to y, then y cannot bear relation R to x.  
Example:  
The fact type "Person is at least as tall as Person" is antisymmetric (but not asymmetric). |
| arity              | Number of roles in a fact type, predicate or constraint argument. Note: A fact type with exactly one role is called a unary fact type. A fact type with exactly two roles is called a binary fact type. A fact type with more than two roles is called an n-ary fact type. |
| asserted fact      | Fact instance that is simply asserted (declared to be the case) rather than being derived from other facts. Synonyms: primitive fact; base fact; extensional fact. |
| asserted fact type | Fact type, each of whose population instances is an asserted fact. Synonyms: primitive fact type; base fact type; extensional fact type. |
asserted subtype

**Subtype** where instances in the **population** of that subtype are simply asserted rather than being derived.

**Example:**
If LaptopComputer is an asserted subtype of Computer, then each fact that declares a specific computer to be a laptop computer is simply asserted, rather than being derived (e.g. from instances of another fact type that classifies computers).

asymmetric constraint

**Ring constraint** that restricts the population of the constrained role pair to be an asymmetric relation. A binary relation \( R \) whose roles are played by instances of either the same type or compatible types is asymmetric if and only if, given any instances \( x \) and \( y \), if \( x \) bears the relation \( R \) to \( y \) then \( y \) cannot bear relation \( R \) to \( x \).

**Examples:**
The fact type “Person is a parent of Person” is asymmetric.
The fact type “Node points to Node” in a directed, ring network with at least three nodes is asymmetric but not acyclic.

atomic fact

**Fact** that cannot be decomposed, without loss of information, into multiple facts involving exactly the same **object types**.

**Note:** An atomic fact is either an elementary fact or an existential fact.

**Synonym:** irreducible fact.

atomic fact type

**Fact type**, each of whose instances is an atomic fact.

cardinality constraint

**Constraint** on an **object type** or **role** that determines the minimum and/or the maximum number of population instances that the object type or the role may contain for each state of the fact base.

**Note:** Use of the more specific terms “**object cardinality constraint**” and “**role cardinality constraint**” helps to distinguish FBM cardinality constraints from cardinality constraints in other approaches (e.g. ER and OWL).

comparable object types

**Value types** or entity types with **reference modes**, where values that refer to the different types may be compared using a standard comparator such as =, \( > \), because they are based on the same datatype.

**Example:**
Instances of the value types CountryCode and AirlineCode may be compared if they are both based on a character string datatype. This is despite the fact that they are incompatible, since their domain type names carry additional semantics (e.g. the country code ‘CH’ is based on Latin, but the airline code ‘CH’ is not).

compatible object types

**Object types** whose **populations** may have some instances in common.

**Note:** Two object types are compatible if and only if one of the following applies:
(1) the object types are the same, (2) one is a subtype of the other; (3) the object types share a common supertype and are not declared mutually exclusive.

compatible roles

**Roles** hosted by compatible object types

compound fact

**Fact** that can be decomposed, without loss of information, into a conjunction multiple facts involving exactly the same **object types**.

**Note:** A compound fact is not an atomic fact.

compound fact type

**Fact type** whose instances are compound facts
conceptual schema  Structure that determines the possible and permitted states and the possible and permitted transitions of the fact base, as well as the meaning of every included term that could be misunderstood by the intended audience or business community.

Note: A conceptual schema declares the fact types, constraints, derivation rules, and concept definitions relevant to the universe of discourse.

conceptualization principle  Principle that requires conceptual models to capture only those details about the universe of discourse that are conceptually relevant, ignoring implementation aspects.

Note: As noted in refer to ISO TC97/SC5/WG3 TR9007 (1987), conceptual data models include only conceptually relevant aspects, both static and dynamic, of the universe of discourse, thus excluding all aspects of (external or internal) data representation, physical data organization and access as well as all aspects of a particular external user representation.

constraint  Restriction on what states or transitions of the fact base are possible or permitted.

deontic constraint  An obligation (a restriction that ought to be obeyed but may be violated).

Note: A deontic constraint can be expressed in a positive way to indicate what is obligatory, or in a negative way as a prohibition to indicate what is not permitted.

deontic modality  Modality of obligations and permissions.

Note: The deontic modal operators used are: it is obligatory that; it is permitted that; it is forbidden that.

derivation rule  Rule that specifies how to derive instances of a derived or semiderived fact type from other facts, or how to derive membership of instances in a derived or semiderived subtype from properties of its supertype(s).

derivation status  Indication that the instances of a fact type fact type or subtype(s) are all asserted, or (b) are all derived, or (c) may include some asserted instances and some derived instances.

Synonym: instantiation method

derived fact  Fact that is deduced from other facts by means of a derivation rule

Note: A fact that is not derived is an asserted fact.

derived fact type  Fact type, each of whose instances is a derived fact.

derived subtype  Subtype, each of whose instances is derived from facts of its supertype(s) by means of a derivation rule.

domain object  Individual thing of interest that is either an entity or a value (other than a data value)

domain object type  Type, each of whose instances is a domain object.

Note: The population of a domain object type is always finite.

dot role of object type  Either a far role of the object type, or the role hosted by the object type in a unary fact type.

Examples: The role named “birthCountry” in the fact type “Person was born in Country
[birthCountry]” is a dot role of Person. The role named “isSmoker” in the fact type “Person smokes [isSmoker]” is a dot role of Person.

**Note:** For any given object type, its dot role names must be distinct.

### elementary fact

**Fact** that declares that an object has a property, or that one or more objects participate in a relationship, where the fact cannot be split into (rendered as a conjunction of) simpler facts with the same objects without information loss.

**Example:**
Each of the following two fact readings expresses the same fact:
The Politician named ‘Julia Gillard’ governs the Country named ‘Australia’.
The Country named ‘Australia’ is governed by the Politician named ‘Julia Gillard’.

**Note:** An object in an elementary fact is an individual, in the classical logic sense meaning an individual thing. An elementary fact is expressed by a fact reading that predicates over one or more individuals (i.e. applies a logical predicate to a non-empty sequence of individuals). Fact readings that mean the same express the same fact. Existence is not treated as a property or predicate. Simply asserting that an object exists is an atomic (and existential) fact, but not an elementary fact.

### elementary fact type

**Fact type**, each of whose instances is an elementary fact.

### entity

**Object** that is referenced by a definite description that relates it to other objects.

**Note:** An entity can typically change its state over time (e.g. by participating in new facts). An entity is not a value, such as a name or numeral.

**Synonym:** Non-lexical object.

### entity type

**Object type**, each of whose instances is an entity.

### equality constraint

**Set-comparison constraint** that specifies that, for each state of the fact base, the populations of the constrained role sequences must be equal.

### exclusion constraint

**Set-comparison constraint** that specifies that, for each state of the fact base, the populations of the constrained role sequences must be mutually exclusive (i.e. do not overlap).

### exclusive-or constraint

Combination of an inclusive-or constraint and an exclusion constraint over the same set of roles.

**Synonym:** xor constraint

### existential fact

**Atomic fact** that imply asserts the existence of an object.

**Examples:**
- There exists a Country named ‘Australia’.
- There exists a CountryCode ‘AU’.

### existential fact type

**Atomic fact type**, each of whose instances is an existential fact.

### fact

**Proposition** that is taken to be true by the relevant community, and is expressed by a sentence that either simply applies a predicate to one or more individuals or simply asserts the existence of an individual.

**Note:** A fact is neither a constraint nor a derivation rule.
**fact base**

Set of **facts** that are of interest to the relevant community and conform to the **conceptual schema**.

*Note*: The set of asserted facts is called the **extensional database**, and the set of derived facts is called the **intensional database**.

**fact reading**

Sentence expressing a fact.

**fact type**

Type, each of whose instances are facts that express the same kind of information.

*Note*: An elementary fact type is a non-empty set of typed predicates (i.e. predicates with specific object types assigned for its object placeholders).

**fact type reading**

Reading of a **fact type** that inserts the relevant object type names in the placeholders within a **predicate reading**.

*Example*:

Person was born in Country.

*Synonym*: sentential form

**far role of object type**

**Role** in a binary **fact type** that has another role hosted by that **object type**.

*Examples*:

The role named “birthCountry” in the fact type “Person was born in Country [birthCountry]” is a far role of Person.

The roles named “parent” and “child” in the fact type “Person [parent] is a parent of Person [child]” are far roles of Person.

*Note*: For any given object type, its far role names must be distinct.

**frequency constraint**

**Constraint** that restricts, for each state of the **fact base**, the number of times any given sequence of **objects** that instantiate the constrained **role** sequence in the relevant context may appear in the population of that role sequence in that context.

*Note*: The frequency can be expressed by a list of one or more discrete numbers or numeric ranges having minimum and/or maximum values. A frequency constraint with a maximum frequency of 1 is not allowed (instead, a uniqueness constraint should be used for this case).

*Synonym*: occurrence frequency constraint

**functional role**

**Role** with an **alethic uniqueness constraint** that applies to that role only.

**implicit identity fact type**

**Fact type** implied by a direct subtyping relationship, where each instance of the implicit identity fact type indicates an identity between an instance within the **subtype** and an instance within the direct **supertype**.

**inclusive-or constraint**

**Constraint** over two or more **roles** hosted by either the same **object type** or by **compatible object types**, which specifies that all instances of the object type(s) must play at least one of the constrained roles.

*Synonym*: Disjunctive mandatory role constraint (mandatory constraint over a logical disjunction of two or more roles).

**independent object type**

A **domain object type** is independent if and only if it may have some instances that participate in no **elementary facts**.

**instance**

Member of a specific type.

**intransitive constraint**

**Ring constraint** that restricts the population of the constrained role pair to be an intransitive relation. A binary relation \( R \) whose roles are played by instances of
either the same type or compatible types is intransitive if and only if, given any instances $x, y$ and $z$, if $x$ bears the relation $R$ to $y$, and $y$ bears the relation $R$ to $z$ then $x$ cannot bear the relation $R$ to $z$.

Example:
The fact type “Event immediately precedes Event” is intransitive.

**irreflexive constraint**

Ring constraint that restricts the population of the constrained role pair to be an irreflexive relation. A binary relation $R$ whose roles are played by instances of either the same type or compatible types is irreflexive if and only if no instance can bear the relation $R$ to itself.

Example:
The fact type “Person is a parent of Person” is irreflexive.

**join path**

Role path that involves at least one conceptual join between roles hosted by compatible object types.

Example:
Given the fact types F1 “Advisor speaks Language” and F2 “Language is used by Country”, the join path “Advisor speaks Language that is used by Country” navigates from the advisor role in F1, then conceptually inner joins the language role in F1 to the language role in F2, and then ends at the country role in F2.

**lazy evaluation**

Instantiation method in which derived instances of a derived or semiderived fact type are evaluated only when requested (derive on query rather than derive on update).

**lazy cached evaluation**

Instantiation method in which: (1) derived instances of a derived or semiderived fact type are evaluated when first requested, and are then stored; (2) if updates are made to the underlying facts used to derive them, the formerly derived facts are either deleted or marked as “dirty” (outdated); (3) a subsequent request for the derived data is processed as follows: if the derived data is stored (and not dirty) it is simply read; otherwise the derived data is recalculated and stored.

**mandatory role**

Role that must be played by all instances in the population of the role’s object type.

*Synonym:* Total role

**mandatory role constraint**

Constraint ensuring that each instance in the population of a given object type must play the constrained role or at least one of the constrained roles.

*Note:* A mandatory role constraint applying to one role only is a simple mandatory role constraint. A mandatory role constraint applying to more than one role is a disjunctive mandatory role constraint (also known as an inclusive-or constraint).

*Synonyms:* mandatory constraint; total role constraint

**modality**

Construct from modal logic that classifies the mode in which a proposition applies (e.g. necessity, obligation, recommendation).

*Note:* Fact-based modelling currently supports only alethic and deontic modalities, and uses them only for classifying constraints.

**object**

An individual thing of interest, about which the community wishes to communicate

*Examples:*
The Country named “Australia”.
The country name “Australia”
The character string “Australia”
Note: An object is either an entity, a domain value, or a data value.

**Synonym:** Individual

**object type**

Concept used to classify individual things into different kinds.

**Examples:** Country; CountryName; String

*Note:* An object type is an entity type, a value type, or a datatype.

**objectification**

Process of making an object from a fact in order to talk about the state of affairs that corresponds to, but is not identical to, the fact

*Note:* In fact-based modelling, objectification is always situational nominalization, not propositional nominalization. Situational nominalization is the use of a noun phrase to refer to the state of affairs denoted by another linguistic expression. For example, in the sentence “That snowing is beautiful” the noun phrase “that snowing” is used to nominalize the state of affairs referenced by the sentence “It’s snowing outside”. Propositional nominalization is the use of a noun phrase to refer to the proposition expressed by a declarative sentence. For example, in the sentence “I know that it’s snowing outside” the noun phrase “that it’s snowing outside” is used to nominalize the proposition expressed by the sentence “It’s snowing outside”.

**Synonyms:** Situational nominalization; nesting.

**objectified fact type**

Object type resulting from objectification of a fact type.

**population**

Set of instances in the current state of the structure (fact base, object type, fact type, or role) under discussion.

*Note:* The population of a structure may change over time.

**predicate**

Result of replacing the objects of interest in a proposition by placeholders for those objects.

*Note:* A predicate is denoted by a declarative sentence with the object terms of interest replaced by placeholders. A predicate is a semantic, not syntactic construct.

**Synonym:** Logical predicate.

**predicate reading**

Result of replacing the object terms of interest in a sentence by placeholders, thereby expressing a predicate.

*Examples:*

“... smokes”; “... employs ...”; “... is employed by ...”; “... played ... for ...”

*Note:* Each predicate must have at least one predicate reading.

**preferred identifier of an entity type**

Reference scheme chosen by the domain expert as the preferred way of identifying instances of the entity type.

*Example:*

The entity type Country may have its instances identified using either one of the injective fact types “Country has CountryCode” and “Country has CountryName”. If the “Country has CountryCode” relationship is chosen as the preferred reference scheme, the “Country has CountryName” relationship may still be used as an alternative way to refer to countries.

**property**

Unary predicate in which an object participates.

*Example:*

In the unary fact “Fred smokes”, the property of smoking is assigned to Fred.
proposition  That which is asserted by a declarative sentence.

*Note:* Each proposition is true or false, but not both.

reference mode  Mode or manner in which a single value refers to a single entity.

reference scheme  Means for identifying instances of the entity type under discussion.

*Note:* A reference scheme may be simple (e.g. a reference mode) or composite.

reflexive constraint  *Ring constraint* that restricts the population of the constrained role pair to be a locally reflexive relation. A binary relation $R$ whose roles are played by instances of either the same type or compatible types is locally reflexive if and only if, given any instances $x$ and $y$, if $x$ bears the relation $R$ to $y$ then $x$ bears the relation $R$ to itself.

*Example:*  The fact type “Person knows Person” is locally reflexive.

ring constraint  Logical *constraint* between two compatible roles that specifies how the population instances of these roles must be related.

*Examples:*  Varieties of ring constraint include irreflexive, asymmetric, intransitive, antisymmetric and acyclic ring constraints, as well as others.

role  Part played by an object in a fact.

role occurrence  Instance of navigating through a role of a fact type.

Example:  The derivation rule “Person$_1$ is a grandparent of Person$_2$ if and only if Person$_1$ is a parent of some Person$_3$ who is a parent of Person$_2$” involves a role path that includes two occurrences of each of the roles in the fact type “Person is a parent of Person”.

semiderived fact type  Fact type that may have a population that includes some for which derived facts and some asserted facts.

semiderived subtype  Subtype that may have a population that includes some derived objects and some asserted objects.

set-comparison constraint  Constraint that specifies a subset, equality or exclusion condition to be satisfied when comparing the populations of compatible role sequences.

*Note:* Each set-comparison constraint is either a subset constraint, an equality constraint, or an exclusion constraint. Compatible role sequences require the role occurrences across matching positions in the sequences to be compatible.

strongly intransitive constraint  *Ring constraint* that restricts the population of the constrained role pair to be a strongly intransitive relation. A binary relation $R$ whose roles are played by instances of either the same type or compatible types is strongly intransitive if and only if, given any instances $x$ and $y$, $x$ cannot be both directly related and indirectly related to $y$ via $R$. Direct relationships use a single application of $R$ while indirect relationships use multiple applications of $R$.

*Example:*  The fact type “Person is a parent of Person” is deontically, strongly intransitive
subset constraint: **Set-comparison constraint** that specifies that, for each state of the fact base, the population instances of a sequence of one or more role occurrences must be a subset of the population instances of another compatible sequence of role occurrences.

subtype: **Object type**, each of whose instances belong to an encompassing type.  
*Example:*  
Woman is a (proper) subtype of Person.

supertype: **Object type** that has at least one subtype, and may have some instances not in any of its subtypes.

symmetric constraint: **Ring constraint** that restricts the population of the constrained role pair to be a symmetric relation. A binary relation $R$ whose roles are played by instances of either the same type or compatible types is symmetric if and only if, given any instances $x$ and $y$, if $x$ bears the relation $R$ to $y$ then $y$ bears relation $R$ to $x$.

*Example:*  
The fact type “Person is a sibling of Person” is symmetric.

uniqueness constraint: **Constraint** over a sequence of one or more roles that ensures that in each state of the fact base, each instantiation of that role sequence occurs only once.

universe of discourse: The aspects of the world that the relevant community wishes to talk about in the conceptual model.

*Synonym:* Business domain

value: Unchangeable object that is identified by combining its type with a constant such as a character string or numeral.

*Examples:*  
The country code “AU”; the IQ 120.

*Synonyms:* domain value; lexical object; label

value comparison constraint: **Constraint** that specifies how the values of population instances of two roles with co-roles played by the same object are related by one of the following comparison operators: $<$, $\leq$, $>$, $\geq$.

value constraint: **Constraint** that specifies the possible (if alethic) or permitted (if deontic) values for a role or a value type.

value type: **Object type**, each of whose instances is a (domain) value.
## Annex A  ORM Graphical Notation

### Construct and Examples

<table>
<thead>
<tr>
<th><strong>Entity type</strong></th>
<th><strong>Description/Notes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Country</td>
<td>Named soft rectangle, named hard rectangle, or named ellipse. The soft rectangle shape is the default.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Value type</strong></th>
<th><strong>Description/Notes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>CountryCode</td>
<td>Named, dashed, soft rectangle (or hard rectangle or ellipse).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Entity type with popular reference mode</strong></th>
<th><strong>Description/Notes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Country (code) Course (code) Company (name) Building (.nr)</td>
<td>Abbreviation for injective reference relationship to value type, e.g.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Entity type with unit-based reference mode</strong></th>
<th><strong>Description/Notes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Height (cm) Mass (kg) Salary (USD) Price (EUR)</td>
<td>Abbreviation for reference fact type, e.g.</td>
</tr>
<tr>
<td>Height (cm; Length) Salary (USD; Money) Price (EUR; Money)</td>
<td>Optionally, the unit type may be displayed (as shown opposite).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Entity type with general reference mode</strong></th>
<th><strong>Description/Notes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Website (URL) Weblink (URL)</td>
<td>Abbreviation for reference fact types, e.g.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Independent object type</strong></th>
<th><strong>Description/Notes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Country! CountryCode!</td>
<td>Instances of the type may exist, without playing any elementary fact roles.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>External object type</strong></th>
<th><strong>Description/Notes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Address^</td>
<td>Object type is defined in another model. This notation is tentative (yet to be finalized)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Predicate</strong> (unary, binary, ternary, etc.)</th>
<th><strong>Description/Notes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>smokes</td>
<td>Ordered set of 1 or more role boxes with at least one predicate reading in mixfix notation. If shown, object placeholders are denoted by “…” If placeholders are not shown, unaries are in prefix and binaries are in infix notation.</td>
</tr>
<tr>
<td>was born in</td>
<td></td>
</tr>
<tr>
<td>... speaks ... very well</td>
<td></td>
</tr>
<tr>
<td>... played ... for ...</td>
<td></td>
</tr>
<tr>
<td>... in ... on ... ate ...</td>
<td></td>
</tr>
</tbody>
</table>
**Duplicate type or predicate shape**

If an object type or predicate shape is displayed more than once (on the same page or different pages) it is shadowed.

**Unary fact type**

Attaching a role box to an object type shape mean that only instances of that object type may play that role (e.g. here, the smokes role may be played by instances of the Person object type). A role name may be added in square brackets.

**Binary fact type**

By default, predicate readings (binary or longer) are read left-to-right or top-to-bottom. An arrow-tip is used to display a different reading direction. Role names may be displayed in square brackets beside their role. Forward and inverse readings for binaries may be shown together, separated by "/".

**Ternary fact type**

Role names may be added in square brackets. Arrow-tips are used to reverse the default left-right or top-down reading order. Reading orders other than forward and reverse are shown using named placeholders.

**Quaternary fact type**

The above notes for the ternary case apply here also. Fact types of higher arity (number of roles) are also permitted.

**Objectification**

The enrolment fact type is objectified as an entity type whose instances can play roles. In this example, the objectification type is independent, so we can know about an enrolment before the grade is obtained.

**Internal uniqueness constraint (UC) on unary**

These examples are equivalent. By default, fact types are assumed to be populated with sets of facts (not bags of facts), so no whole fact may be duplicated.
**Internal UCs on a binary fact type**

The examples show the 4 possible patterns: 1:n (one-to-many); n:1 (many-to-one); m:n (many-to-many); 1:1 (one-to-one)

**Internal UCs on ternaries**

The first example has two, 2-role UCs: the top UC forbids ties; the other UC ensures that each team gets only place per competition (a dotted line excludes its role from the UC). The second example has a spanning UC (many-to-many-to-many). For an n-ary \( (n > 2) \) fact type to be atomic, each UC on it must span at least n-1 roles.

**Simple mandatory role constraint**

The example constraint means that each person was born in some country. The mandatory role dot may be placed at either end of the role connector.

**Inclusive-or constraint**

An inclusive-or constraint is also called a disjunctive mandatory role constraint. The constraint is displayed as a circled dot connected to the constrained roles. The example constraint means that each visitor referenced in the model must have a passport or a driver licence (or both).

**Preferred internal UC**

A double bar on a UC indicates it underlies the preferred reference scheme.

**External UC**

A double-bar indicates that the constrained roles provide the preferred reference for the object type at the other end. Here, each state is primarily identified by combining its country and state code. Each combination of country and state name also applies to only one state.

**Object type value constraint**

The allowed values may be specified as a list of discrete values and/or value ranges. The two examples shown opposite specify an enumerated list of values.
Ranges are inclusive of end values by default. Round brackets are used to exclude an end value. Square brackets may be added to explicitly declare inclusion, e.g. the constraint on PositiveScore may also be specified as \((0..100]\).

Multiple combinations may also be specified.

**Role value constraint**

As for object type value constraints, but connected to the constrained role. Here, an age of a person must be at most 140 years.

**Subset constraint**

The arrow points from the subset end to the superset end (e.g. if a person smokes then that person is cancer prone). The role sequences at both ends must be compatible. A connection to the junction of 2 roles constrains that role pair.

**Join subset constraint**

The constrained role pair at the superset end is projected from a role path that involves a conceptual join on Language. The constraint declares that if an advisor serves in a country then that advisor must speak a language that is often used in that country.

**Exclusion constraint**

These exclusion constraints mean that no person is both married and widowed, and no person reviewed and authored the same book. Exclusion may apply between 2 or more compatible role sequences, possibly involving joins.

**Exclusive-or constraint**

An exclusive-or constraint is simply the conjunction of an inclusive-or constraint and an exclusion constraint. Also known as an xor constraint.
Equality constraint

This equality constraint means that a patient's systolic BP is recorded if and only if his/her diastolic BP is recorded. An equality constraint may apply between 2 or more compatible role sequences, possibly involving joins.

Derived fact type, and derivation rule

A fact type is either asserted, derived, or semiderived. A derived fact type is marked with an asterisk “*”. A derivation rule is supplied. A double asterisk “**” indicates derived and stored (eager evaluation).

Semiderived fact type, and derivation rule

A fact type is semiderived if some of its instances may be derived, and some of its instances may be simply asserted. It is marked by “++” (half an asterisk). “++” indicates semiderived and stored (eager evaluation for derived instances).

Subtyping

All subtypes are proper subtypes. An arrow runs from subtype to supertype. A solid arrow indicates a path to the subtype's preferred identifier (e.g. here, student employees are primarily identified by their employee number). A dashed arrow indicates the supertype has a different preferred identifier.

Subtyping constraints

A circled "X" indicates the subtypes are mutually exclusive. A circled dot indicates the supertype equals the union of the subtypes. The combination (xor constraint) indicates the subtypes partition the supertype (exclusive and exhaustive).
**Subtype derivation status**

A subtype may be asserted, derived (denoted by "*"), or semiderived (denoted by "+"). If the subtype is asserted, it has no mark appended and has no derivation rule. If the subtype derived or semiderived, a derivation rule is supplied.

**Internal frequency constraint**

This constrains the number of times an occurring instance of a role or role sequence may appear in each population. Here: each jury has exactly 12 members; each panel that includes an expert includes at least 4 and at most 7 experts; each expert reviews at most 5 papers; each paper that is reviewed is reviewed by at least 2 experts; and each department and year that has staff numbers recorded in the quaternary appears there twice (once for each gender).

**External frequency constraint**

The example external frequency constraint has the following meaning. In this context, each combination of student and course relates to at most two enrolments (i.e. a student may enroll at most twice in the same course).

**Value-comparison constraint**

The example value-comparison constraint verbalizes as: **For each Project, existing enddate >= startdate.**

**Object cardinality constraint**

The example constraints ensure there is exactly one president and at most 100 senators (at any given time).

**Role cardinality constraint**

The example constraint ensures that at most one politician is the president (at any given time).
Ring constraints

A ring predicate $R$ is locally reflexive if and only if, for all $x$ and $y$, $xRy$ implies $xRx$. E.g. “knows” is locally but not globally reflexive. Reflexive, symmetric and transitive properties may also be enforced using semiderivation rather than by constraining asserted fact types.

The example constrains the subtyping relationship in ORM to be both acyclic (no cycles can be formed by a chain of subtyping connections) and strongly intransitive (no object type $A$ can be both a direct subtype of another type $B$ and an indirect subtype of $B$, where indirect subtyping means there is a chain of two or more subtyping relationships that lead from $A$ to $B$).

Ring constraints may be combined only if they are compatible, and one is not implied by the other. ORM tools ensure that only legal combinations are allowed.

Deontic constraints

Unlike alethic constraints, deontic constraint shapes are colored blue rather than violet. Most include “o” for “obligatory”. Deontic ring constraints instead use dashed lines.

In the parenthesis example, the alethic frequency constraint ensures that each person has at most two parents, the alethic ring constraint ensures that parenthood is acyclic, and the deontic ring constraint makes it obligatory for parenthood to be strongly intransitive.

Textual constraints

Textual constraints are also known as *general* constraints. First-order constraints with no graphic notation may be expressed textually in the FORML 2 language. These examples use footnoting to capture a restricted uniqueness constraint and a restricted mandatory role constraint.
Objectification display options

Internally, link fact types connect objectified associations to their component object types. By default, display of link fact types is suppressed. If displayed, link predicate shapes use dashed lines instead of solid lines. Objectification object types may also be displayed without their defining components, using an object type shape containing a small predicate shape, as shown in this Enrolment example.
Annex B  References and Websites

References


Some Relevant Websites:

www.factbasedmodeling.org
www.ORMFoundation.org
www.orm.net
http://dataconstellation.com/ActiveFacts/index.shtml
http://www.casetalk.com/
http://www.infagon.com/