OFML – Standardized Data Description Format of the Office Furniture Industry
Version 2.0
3rd revised edition

Copyright © 1998 – 2015
Der Verband Büro-, Sitz- und Objektmöbel e.V. (BSO)

November 4, 2015

Copyright © 1998 – 2015
Verband Büro-, Sitz- und Objektmöbel e.V. (BSO)
Bierstadter Strasse 3
D-65189 Wiesbaden
www.buero-forum.de

The scientific support and coordination of the development of the OFML data standard was performed by Dr. Ing. habil. Ekkehard Beier from the Institute of Practical Informatics and Media Informatics of the Faculty of Informatics and Automation at the Technical University Ilmenau.

Ekkehard Beier holds the intellectual copyright for the OFML object model, including the scene architecture, rules, and base interfaces. Referring to this, any scientific, patent-related or in any other way copyright-related exploitation requires the permission of Ekkehard Beier.

The OFML standard (parts I-III) was developed by EasternGraphics GmbH on behalf of industrial association Büro-, Sitz- und Objektmöbel e.V. (BSO).

EasternGraphics GmbH holds the intellectual copyright for the segments Global Planning Types, Product Data Model, and Planning Environment. The same applies to the OFML Database (ODB), the OFML Metafile Format EGM, and the OFML 2D Interface. Referring to this, any scientific, patent-related or in any other way copyright-related exploitation requires the permission of EasternGraphics GmbH.

Basic syntax and semantics of OFML are based on the *Cobra* programming language from EasternGraphics GmbH. Copyright © 1995 – 2015 EasternGraphics GmbH and Jochen Pohl.

The OFML standard was developed with great care. Nevertheless, mistakes and inconsistencies cannot be ruled out. The industrial association Büro-, Sitz- und Objektmöbel e.V. as well as EasternGraphics GmbH refuse to accept any respective liability in this regard.

Contents

References 6

1 Introduction **7**

2 Concepts **10**

2.1 Types 10
2.2 Entities 11
2.3 Property 14
2.4 Methods 14
2.5 Rules 15
2.6 Categories 16
2.7 Initialization 16
2.8 Interactors 17

3 Basic Syntax and Semantics **18**

3.1 Introduction 18
3.2 Lexical Structure 19
3.3 Types 25
3.4 Predefined Reference Types 30
3.5 Statements 43
3.6 Expressions 51
3.7 Packages and Namespaces 64
3.8 Classes 69
3.9 Predefined Functions 73

4 Basic Interfaces **77**

4.1 MObject 77
4.2 Base 79
4.3 Material 87
4.4 Property 89
4.5 Complex 94
4.6 Article 98

5 Predefined Rule Reasons **103**

5.1 Element Rules 103
5.2 Selection Rules 105
5.3 Move Rules 105
5.4 Persistence Rules 107
5.5 Other Rules 108

6 Global functions **109**

6.1 Formatted Output 109
6.2 oiApplPaste() 110
6.3 oiClone() 110
6.4 oiCollision() 111
6.5 oiCopy() 111
6.6 oiCut() 111
6.7 oiDialog() 111
6.8 oiDump2String() 112
6.9 oiExists() 113
6.10 oiGetDistance() 113
6.11 oiGetNearestObject() 113
6.12 oiGetRoots() 113
6.13 oiGetStringResource() 113
6.14 oiLink() 114
6.15 oiOutput() 114
6.16 oiPaste() 114
6.17 oiReplace() 115
6.18 oiSetCheckString() 115
6.19 oiTable() 115

7 Geometric types **117**

7.1 OiGeometry 117
7.2 OiBlock 118
7.3 OiCylinder 119
7.4 OiEllipsoid 120
7.5 OiFrame 121
7.6 OiHole 122
7.7 OiHPolygon 124
7.8 OiImport 125
7.9 OiPolygon 126
7.10 OiRotation 127
7.11 OiSphere 128
7.12 OiSweep 129
7.13 OiSurface 131

8 Global Planning Types **132**

8.1 OiPlanning 132
8.2 OiProgInfo 140
8.3 OiPlElement 142
8.4 OiPart 148
8.5 OiUtility 153
8.6 OiPropertyObj 153
8.7 OiOdbPlElement 154

9 Types for Product Data Management **156**

9.1 OiPDManager 157
9.2 OiProductDB 160

10 Types of the Planning Environment **164**

10.1 The Wall Interface 164
10.2 OiLevel 164
10.3 OiWall 166
10.4 OiWallSide 166

A Product Data Model 167

B The 2D Interface 169

B.1 Introduction . . . 169
B.2 The 2D Object Hierarchy . . . 169
B.3 Coordinates . . . 170
B.4 Methods . . . 170
B.5 Object Types . . . 171
B.6 Attributes . . . 174

C The 2D vector file format 180

C.1 Introduction . . . 180
C.2 data types . . . 180
C.3 File header . . . 186
C.4 General structured data types . . . 187
C.5 Graphic 2D objects . . . 188

D External data formats 200

D.1 Geometries . . . 200
D.2 Materials . . . 201
D.3 Fonts . . . 203
D.4 External Tables . . . 204
D.5 Text Resources . . . 204
D.6 Archives . . . 205

E Format Specifications 206

E.1 Format Specifications for Properties . . . 206
E.2 Definition Format for Properties . . . 207

F Additional Types 209

F.1 Interactor . . . 209
F.2 Light . . . 210
F.3 MLine . . . 211
F.4 MSymbol . . . 213
F.5 MText . . . 214

G Applied Notation **216**

G.1 Class Diagrams based on Rumbaugh 216

H Categories **218**

H.1 Interface Categories 218

H.2 Material Categories 218

H.3 Planning Categories 219

I Terms **220**

Index **223**

References

[GO] EasternGraphics GmbH: *GO – Generic OFML types (OFML part II).*

[OAM] Verband Büro-, Sitz- und Objektmöbel e.V.: *OAM – OFML Article Mappings (OFML Part VI).*

[OAS] Verband Büro-, Sitz- und Objektmöbel e.V.: *OAS – OFML Article Selection (OFML Part V).*

[OCD] Verband Büro-, Sitz- und Objektmöbel e.V.: *OCD – OFML Commercial Data (OFML Part IV).*

[ODB] EasternGraphics GmbH: *ODB – OFML database (OFML part I).*

[OEX] Verband Büro-, Sitz- und Objektmöbel e.V.: *OEX – OFML Business Data Exchange (OFML Part VII).*

[Rumb91] J.Rumbaugh et al: *Object–Oriented Modelling And Design*. Prentice–Hall, New Jersey, 1991

Chapter 1

Introduction

The motivation for the new standard of office furniture (OFML[1]) is the result of a series of requirements that could generally not be met with past and present solutions:

- The new requirements in the area of planning and visualization of (office) furniture cannot be met sufficiently by CAD-based systems. The main problems of CAD-based solutions are their enormous data size, the poor parameterizability and configurability, insufficient coverage of product logics, insufficient display quality in the interactive range, complicated operation, and costly licensing.
- These disadvantages are magnified in the area of marketing-oriented solutions that may, for example, set the framework for using end user-oriented systems on CD-ROM or the Internet.
- A platform-independent and (software) manufacturer-independent data format allows an unlimited number of software manufacturers to offer systems and solutions so that monopolizing conditions can be avoided or eliminated.
- The new data format also allows for the implementation of a series of applications that are compatible with respect to the data in spite of a different orientation. In this way it is possible to achieve a compatibility and, therefore, technological uniformity between manufacturer, trade, and end user systems.

(Traditional) CAD systems continue to have a raison d'tre, especially through their abilities in the design and manufacturing sector. Consequently, the new standard does not lay claim to a complete removal of existing CAD-based solutions. Instead, a coexistence between traditional CAD solutions and the new standard is aimed at. The coexistence should be implemented on the basis of directly compatible data formats or suitable conversion tools.

In particular, the OFML standard offers the following features:

- consistent application of the object-oriented paradigm,

[1] *Office Furniture Modeling Language*

- conversion of concepts of semantic modeling to achieve a match of virtual objects with actual products,
- combination of geometric, visual, interactive, and semantic features of real products in a uniform and holistic data model,
- mapping of real configuration logics and parametrics,
- independence of system or interface platforms, and
- independence of a concrete runtime environment.

The OFML standard consists of the following **parts**, each covering different aspects of OFML data creation or various application processes. The parts are more or less strongly linked together, primarily by cross-reference such as article numbers and type identifiers.

1. OFML database (ODB)

   The OFML database [ODB] defines a table-based interface for description of hierarchical geometries in 2D and 3D.

2. Generic Office library (GO)

   The class library GO [GO] provides basic functionality for the scope of the office furniture industry.

3. Object model

   This part defines a complete programming language, basic interfaces of OFML types, predefined rule reasons, global (type-independent) functions as well as a set of base types. On the basis of this object model arbitrarily complex data can be created and external commercial data can be integrated.

4. OFML Commercial Data (OCD)

   OCD [OCD] defines a set of tables for the creation of (commercial) product data which is needed and exchanged within business processes of the furniture trade. Primarily, OCD is supposed to cover tasks like configuration of complex articles, price determination and creation of offer resp. order forms.

5. OFML Article Selection (OAS)

   OAS [OAS] describes a format for structured representation and selection of articles in digital catalogs.

6. OFML Article Mappings (OAM)

   The tables specified in this part [OAM] are used to define more complex relationships between data that has been created according to the specification of various other OFML parts.

7. OFML Business Data Exchange (OEX)

   OEX [OEX] describes a format for the electronic exchange of business documents, such as purchase orders and invoices.

Parts I-III were developed by EasternGraphics GmbH on behalf of industrial association Büro-, Sitz- und Objektmöbel e.V. (BSO). All other parts are specified by the standardization committee of the BSO.

In the following, in this document only the object model (part III) is described. All other parts are specified in separate documents (see references above).

This document is structured as follows:

Introduction and Overview

- This chapter describes motivation, features and the parts of the OFML standard, and presents an overview of the document.
- Chapter 2 summarizes the relevant concepts and metaphors of the object model.

OFML part III (Object model)

- Chapter 3 describes the basic syntax and semantics of the programming language underlying OFML.
- Chapter 4 presents an overview of the basic interfaces that form the basis for the concrete types of the standard.
- Chapter 5 describes the set of predefined rule reasons.
- Chapter 6 describes the set of type-independent functions predefined for OFML.
- Chapters 7 and 8 describe the complete set of OFML basic types.
- Chapter 9 specifies types that are required for access to external product data.
- Chapter 10 describes the generic types of planning environments.

Appendix

- Appendix A describes a generic format for the external writing of product data.
- Appendix B documents an explicit 2D programming interface available in OFML.
- Appendix C documents a metafile format that is used by OFML to describe 2D vector graphics.
- Appendix D documents the set of external data formats and their application.
- Appendix E describes the formats relevant for using properties.
- Appendix F describes the additional types that may be applied within the framework of OFML.
- Appendix G describes the notational conventions used within the framework of this standard.
- Appendix H describes the categories predefined within the framework of this standard.
- Appendix I defines the most important terms used within the framework of this standard.

Chapter 2

Concepts

This chapter contains a description of the basic OFML concepts. All concepts documented in the subsequent chapters are invariably based on these fundamentals. As such, an understanding of these concepts is a necessary basis for additional dealings with the standard.

2.1 Types

A type[1] is a combination of entities of the same kind. A type defines the following for these entities:

- a set of methods,
- a set of rules,
- a set of instance variables, and
- exactly one initialization function.

Each instance belongs to exactly one immediate type. A type may have only one direct super type; the features are inherited from this super type in a certain way. As such, a type should always be considered in connection with its (direct or indirect) super types.

A type name must be unique within the defining module. A type name must also be unique within the global context. This is accomplished using a uniform name prefix or by integrating it in a name range.

A type is either abstract or concrete. An abstract type cannot be used to form entities.

Example: The concept of a carcass cabinet can be interpreted as type. A certain carcass cabinet (type) that also features a corresponding order number is one example of a concrete type. The generalization of all carcass cabinet types is an example of an abstract type.

[1] The terms type and class are synonymous.

The term interface resembles the term type with respect to its use within OFML. However, the following exception exists:

- An interface is a descriptive tool and does not necessarily correspond to a type.
- An interface is not derived from another interface.
- The name of an interface does not receive a name prefix.

2.2 Entities

An instance[2] is a concrete embodiment of a type. It distinguishes itself from other entities through its identity which is implemented through a hierarchical name. In general, it also distinguishes itself through the assignment of the instance variables of which it always possesses its own copy.

Example: Two carcass cabinets with the same order number are referred to as entities of the same type. They have common features, such as the same order number or the same physical dimensions. They differ from each other, for example, through the or the material design.

In general, entities should be **topologically independent**. This means:

- An instance should not store any object references in its instance variables, that is, name references to objects.

  Example: This would be violated if one instance *remembers* a certain other instance (e.g., on the same topological level).

- An instance cannot assume that its topological ancestors are from a certain type.

  Example: In the course of the temporary generation of entities, any random instance can be an ancestor of an instance.

Under special circumstances, these rules may be violated. The resulting consequences may include:

- loss of ability to save and
- incorrect behavior during copying and inserting.

[2] The terms instance and object are synonymous.

2.2.1 Children

An instance can have a number of children. A child is an instance that exists in the name space of the father object. The father-child-relation is described as follows in OFML:

- Children are generated, modified and deleted during runtime. As such, the number of children is time-dependent.
- The father must be indicated at the generation of an instance and cannot be changed afterwards.
- Deleting an instance always results in the deletion of its children.
- A child inherits the features of its father in a certain way. For example, the complete global space modeling of the child results from interlinking the global space modeling of the father and the local space modeling of the child.
- A child knows its father. This fact may be used for an upward traversing within the scene.
- A father knows his children. This fact may be used for a downward traversing within the scene.

Entities are placed in a scene. Based on the features of the father-child relation described above, the resulting scene topology is a set of trees.

The set of elements is a subset of the set of children. An element is a special child whose generation and removal can be controlled via rules (see below). Thus, every element is a child, but not every child is an element. Elements are normally used for accessible components of a complex instance. Non-elements are normally components of a combined instance that evade access by the user.

Example: The children of a carcass cabinet are stringer, back wall, base, front, and built-in components. The shelves are elements that can be inserted, moved, and deleted separately.

The individual boards of the carcass, on the other hand, are non-elements since there is usually no access to them.

Moving a shelf is controlled by the father of the shelf, that is, the carcass cabinet (rasterization, avoiding collision, sector monitoring). The shelf must, therefore, know its father to transfer control over the desired move to him.

If the carcass cabinet is moved, the children must be moved accordingly. For this reason, the children must be known to the father.

If a carcass cabinet is deleted, all shelves, etc. of this carcass cabinet are automatically deleted.

Syntactically, children are treated like instance variables (Section 3.8.4). Since they are created dynamically, access by name within methods must be carried out using a prefixed *self* in addition to an access operator, e.g., for the child *b5*: *self.b5*.

2.2.2 Instance Identity

The identity of an instance is implemented by means of a hierarchical name space. Every name within this name space corresponds biuniquely to a topological position in the object world (scene). The name of an instance results from the following rule:

```
Name        : Name(Father) '.' LocalName
            | LocalName

LocalName : Character
            | LocalName Character

Character   : 'A' - 'z' | '0' - '9' | '_'
```

Consequently, the name of an instance results from the interlinking of the name of the father, if one exists, via a point-to-point operator with the local name.

Example:

- *env* – is the name of a fatherless root object.
- *env.sky* – is the name of a child of *env*. The local name is *sky*.
- *env.sky-1* – is an invalid name.
- *env.sky_1* – is a valid name.
- *env.env* – is the name of a child of *env* and designates a sibling object of *env.sky* at the same time.
- *top* – is the name of another fatherless root object.
- _ – is not allowed, neither as global nor as local name.

The following absolute names are predefined:

- t – is the root object that combines the planning hierarchy.
- e – is the root object that combines the environment hierarchy, if necessary.
- m – is the root object that combines the dimensioning hierarchy, if necessary.

At the same time, additional root objects can be defined for specific applications.

Restriction: The (local) names in the form $e<n>$, whereby n is a natural number, are reserved and may not be assigned explicitly. These names are assigned automatically during the generation of elements.

2.2.3 Instance Variables

A type (in combination with its super types) defines a set of instance variables of which each instance owns its own copy. The conventions dictate that the name of an instance variable consists of the prefix *m* plus a non-empty set of words that each start with a capital letter. In addition, the name of an instance variable is a valid designator as defined by the basic syntax (Chapter 3). Examples for valid names of instance variables include: *mWidth* and *mIsCutable*.

An instance variable that is defined in a type, may not be re-defined in a derived type. In addition, an instance variable must at least be initialized in the type in which it was defined. Direct access to an instance variable is permitted only within the defined type. An external access can be accomplished only via respective methods.

Instance variables may also be defined via interfaces.

Example: An instance variable could be used to define whether a roll container features an espagnolette or not.

2.3 Property

A property (*property*) is a special instance variable that represents an implicit interface of an instance to the (graphical) user interface. A property has a type, a symbolic designator, and an actual value. In most cases, a discrete value range is assigned to a property. Additional optional features of a property include the initial assignment as well as usually for geometric properties the minimum value and the maximum value.

The current embodiment of the set of properties of an instance generally corresponds to a concrete article number.

Properties are read out by a (*property editor*) and can interactively be set by this editor.

The concept of properties allows for combining any large set of configurations that correspond to exactly one article number each by using a type that covers all possible configurations, while also considering dependencies between individual properties.

Example: The (interactive) configurability of a carcass cabinet can be implemented using the three properties *width*, *height*, and *depth*. In general, a manufacturer-specific discrete value range is defined for each of these properties, e.g., for the width: *600 mm*, *800 mm*, *1000 mm*, and *1200 mm*.

2.4 Methods

A type (in combination with a super type) defines a set of methods or type-specific functions (Section 3.8.4). The name of a method results from a non-empty set of words that all start with a capital letter, except for the first one. In addition, the name of a method is a valid designator as

defined by the basic syntax (Chapter 3). Examples for valid names of methods include: *selectable()* and *isSelectable()*.

A method that is defined in a type, may be redefined in a derived type only if it features the same signature. In the case of OFML this means that number, format, and semantics of the parameters must be identical.

Methods may also be defined via interfaces.

Example: The stop change of a door can be implemented via a corresponding method. This method then implements the stop change without the internal design of the door being known to the outside.

2.5 Rules

A type (in combination with its super types) defines a set of rules. A rule is a procedural construct that is defined analogous to a method within the range of a type. A rule differs from a method through the following features:

- A rule is a type-dependent construct whose signature consists of a rule reason in form of a predefined or user-defined symbols, an optional specific rule parameter, and a formal parameter.
- The return value of a rule is of type *Int*. The value 0 signals the successful processing of the rule. The value -1 denotes a failed rule. The user can be informed about the failure of a rule, if required, through the use of a corresponding text message.
- Several rules may exist for one and the same rule reason within a type or a hierarchy of a type.
- A rule cannot be overwritten, for example, by a rule with identical reason in a derived type.
- A rule is classified as anterior rule or posterior rule. An anterior rule is called before an action is performed. The failure of an anterior rule prevents the corresponding action from being performed. A posterior rule is called after an action was performed. Consequently, this action cannot be prevented. However, the effect of the action can be reversed by applying a suitable counter-action.

For an action that was performed or still needs to be performed and a given instance, a list is first compiled that contains the rules defined by the type and its super type for the respective reason. The order of the rules in the list corresponds to the derivative hierarchy of the respective type. That is, a rule defined by a certain type is located in the list ahead of a rule defined by a derived type. The list of rules is then processed sequentially. Processing is interrupted provided that a rule has failed. In this case, and if the rule was an anterior rule, the corresponding action is not performed.

The rule reasons predefined in OFML are documented in chapter 5.

Example: Inserting any object, e.g. in a carcass cabinet, can be controlled by a corresponding anterior rule. For example, the carcass cabinet can ensure using this rule that only shelves of a certain type and a certain number can be inserted.

Moving an object can be controlled by a corresponding posterior rule. For example, if the move results in a collision, the move should subsequently be corrected accordingly.

2.6 Categories

A category is a classification of types or entities that results from a certain philosophy.

Categories represent an extension to the concept of types: types that belong to a common category do not have to be derived from a common type. In addition, a type can be assigned to several categories.

The association with a category is determined by each type itself. It can be determined for an instance whether its type or super types belong to a certain category (Section 4.1).

The concept of categories can be used to circumvent the limitation of simple inheritance of types in the classification of entities based on orthogonal categorization criteria. It is also useful if *rolls* must be modeled.

Examples Material and planning categories (see Appendix H).

2.7 Initialization

The initialization of an instance is carried out via the *initialize()* procedure. The functions of initialization are essentially the initialization of instance variables and the generation of child objects. The following properties refer to the initialization:

- Exactly one initialization function exists for each type. It is labeled *initialize()*.
- Within the implementation of the initialization function, the initialization function of the direct super type is called first.

The standard signature for the initialization function is as follows:

$$\textit{initialize(pFather(MObject), pName(Symbol))} \rightarrow \textit{MObject}$$

Where *pFather* is the father object and *pName* the local name of the new object to be created. The return value of the initialization function is a reference to the created object.

If required, additional random parameters can be defined for the initialization function of a type. However, this is only allowed for abstract types or internal components. All types that can be instantiated interactively must conform with the standard signature of the initialization function.

Example: The initialization function of a carcass cabinet must create and parameterize the corresponding children (stringer, base, back wall, etc.). However, the creation of shelves can be done interactively at a later time.

2.8 Interactors

In OFML, interactors represent a special type that, in contrast to most other OFML objects, does not represent an object of the real world. Interactors are objects that exist only at runtime and in a simple way allow the user actions that go beyond elementary manipulations such as translation and rotation. Corresponding examples are the marking of connection points or "handles" for interactively changing the size of an object.

Interactors distinguish themselves with respect to other objects through the following features:

- They are not stored persistently.
- They cannot be selected directly. The attempt to select an interactor triggers the *INTERACTOR* rule at the father (Section 5.5).
- Interactors cannot cause a collision.
- They are ignored during photorealistic output and export into an external data format.

Example: Designs can be mounted to an organizational wall at different positions. If interactors are defined for these positions, the user can interactively select the desired mounting point.

Chapter 3

Basic Syntax and Semantics

3.1 Introduction

This chapter describes the programming language fundamentals of OFML whose syntax is oriented to the programming languages C, C++, and Java. From a semantics point of view, OFML is similar to Smalltalk or Python since it is based on a dynamic type concept.

3.1.1 Syntax Representation

A slightly modified version of the familiar Backus-Naur form is used in this document to represent syntax. The following typographical conventions apply: reserved identifiers, characters and character combinations are represented in `Schreibmaschine`. All other grammatical symbols are written in *kursiv*. Multiple alternatives for the right side of a production are separated either by a linebreak and indent or by "‘|"’ within a line. Optional symbols are identified by a subscript "‘*opt*"’:

`{` $stmt_{opt}$ `}`

3.1.2 Implementation

The language definition of OFML assumes that an OFML program is converted into a processible form by a compiler[1]. This takes place in two phases:

1. The translation o f al definitions to module and class levels. In this step, executable statements and definitions within compound statements are translated only partially or not at all.

[1] This can be, for example, bytecode, machine code or vectored graphs.

2. The translation of all executable statements and definitions within compound statements. Depending on implementation, this step can be delayed for each compound statement until just before it is first processed.

The purpose of this division is to handle translation units that reference each other through variables, functions or classes defined by them. Translation units that form a loop based on the super-classes they reference are not permitted.

Another reason for the division is to partially distribute the time needed to translate the program to the runtime, which can be achieved through delayed translation of compound statements.

3.1.3 Program Structure

An OFML program consists of one or more translation units. Each unit represents a sequence of characters of the character set (see Section3.2.1), which can exist in the form of a file and string. Each translation unit is conceptually closed by the *EOF* (end of file) character. This character is not a part of the source text, but instead is used only to represent the end of the input stream in the syntax description.

3.2 Lexical Structure

The first pass during processing reads a sequence of input characters and produces as the result a series of lexical symbols (*Token*)[2].

3.2.1 Character Set

The character set processed by the compiler is the set of printable ASCII characters, i.e. 8-bit characters with an integer value from 32 to 126 and the control characters mentioned in Section 3.2.1. Exceptions are permitted only in comments and literal characters and character string constants. In the latter case, the programmer is responsible for ensuring that the corresponding characters are processed correctly by the runtime environment. In the following, non-printable ASCII characters are represented by hexadecimal numbers in the manner common to C; the grammatical *any-chars* symbol denotes any sequence of characters from the entire character set of the implementation.

Alphanumeric Characters

The following productions define letters (*alpha*), numbers (*num*) and alphanumeric characters (*alnum*). Note that the underscore also belongs to the letters.

[2] The English term is used here to avoid mix ups with OFML symbols.

alpha:

`A | B | C | D | E | F | G | H | I | J | K | L | M`
`N | O | P | Q | R | S | T | U | V | W | X | Y | Z`
`a | b | c | d | e | f | g | h | i | j | k | l | m`
`n | o | p | q | r | s | t | u | v | w | x | y | z`
`_`

num:

`1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0`

alnum:

alpha | *num*

Spaces

The following characters, as sequences or combined with comments (see Section 3.2.1) form *Zwischen-r"aume* (*white-space*): horizontal tabs (HT), linebreaks (NL), vertical tabs (VT), formfeeds (FF), carriage returns (CR) and spaces (SP). If an identifier or keyword follows an identifier, a keyword or a symbol, both have to be separated by a white space. The same applies to integer constants (excluded character constants) and floating-point constants. Otherwise, white spaces have no meaning, but are used only to improve program readability.

white-space:

HT | NL | VT | FF | CR | SP | *comment*

Comments

Comments begin with the `//` character combination and end with a linebreak (NL), carriage return (CR) or a combination of the both.

comment:

`//` *any-chars eol*

eol:

CR | NL | CR NL | NL CR

The `#` sign is different: If it occurs at the start of the first line of a file, the rest of the line is interpreted as a comment.

3.2.2 Token

There are various classes of token: keywords, identifiers, literal constants, operators and delimiters.

3.2.3 Identifiers

ident identifiers begin with a letter, which can be followed by any number of alphanumeric characters in sequence.

ident:
    alpha alnum-seq
alnum-seq:
    alnum alnum-seq$_{opt}$

The keywords mentioned in the next section cannot be used as identifiers.

3.2.4 Keywords

The following keywords are reserved and cannot be used as identifiers:

```
abstract  break      case        catch   class
continue  default    do          else    final
finally   for        foreach     func    goto
if        import     instanceof  native  operator
package   private    protected   public  return
rule      self       static      super   switch
throw     transient  try         var     while
```

3.2.5 Literal Constants

OFML includes literal constants of the following types (see Section 3.3): integers, floating-point numbers, character strings and symbols.

constant:
    integer-constant
    float-constant
    string-constant
    symbol-constant

Integer Constants

Integer constants (*integer-constant*) can be specified in three different numerical systems: decimal, octal and hexadecimal. Because OFML does not distinguish character types, character constants (*character-constant*) are also interpreted as integers.

integer-constant:
    dec-constant
    oct-constant
    hex-constant
    character-constant

Decimal numbers begin with a digit unequal to `0`, followed by any sequence of digits:

dec-constant:
    dec-start $dec\text{-}rest_{opt}$
dec-start:
    `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
dec-rest:
    num $dec\text{-}rest_{opt}$

Octal numbers begin with the digit `0`, followed by any sequence of digits from `0` to `7`:

oct-constant:
    `0` $oct\text{-}rest_{opt}$
oct-rest:
    oct-num $oct\text{-}rest_{opt}$
oct-num:
    `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`

Hexadecimal numbers begin with the `0x` or `0X` character string, followed by any sequence of digits and the letters `A` to `Z` and `a` to `z`:

hex-constant:
    hex-start hex-rest
hex-start:
    `0X` | `0x`
hex-rest:
    hex-num $hex\text{-}rest_{opt}$
hex-num:
    num | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`

An integer constant must be smaller or equal to the largest representable value in the implementation. Otherwise, an error is generated during the translation.

Character constants consist of a character enclosed in single quotation marks "'":

char-constant:
    ' *char-char* '

The number of characters allowed in character constants is indicated by *char-char* . The single quotation mark itself is not allowed in character constants, nor are linebreaks. To represent these and other special characters, use the escape sequences described in Section 3.2.5.

The value of a character constant is the numerical value of the character in the character set of the runtime environment.

Floating-point Constants

Floating-point constants begin with an integer part, followed by a decimal point, the broken part and the exponents. The exponent consists of the `E` or `e` character, an optional +/- sign and an integer value. Either the integer or broken part, but not both, can be omitted. Furthermore, either the decimal point or the exponent can be omitted.

float-constant:
    dec-rest `.` *dec-rest* *float-exp*$_{opt}$
    dec-rest `.` *float-exp*$_{opt}$
    `.` *dec-rest* *float-exp*$_{opt}$
    dec-rest *float-exp*
float-exp:
    exp-char *sign*$_{opt}$ *dec-rest*
exp-char:
    `E` | `e`
sign:
    `+` | `-`

If an underflow occurs during conversion of the floating-point constants in the Internal representation, the value of the constants is `0.0`. If an overflow occurs, it becomes `Float::HUGE_VAL`. If the accuracy of the floating-point constants is greater than supported by the internal representation, excess positions are ignored.

Constant Strings

Constant strings (*string-constant*) consist of a sequence of characters enclosed in single quotation marks ("""). The quotation mark itself is not allowed in character strings. To represent this and other certain special characters, use the following escape sequences:

`\a`	bell character (BEL)
`\b`	backspace (BS)
`\t`	horizontal tab (HT)
`\n`	linebreak (NL)
`\v`	vertical tab (VT)
`\f`	formfeed (FF)
`\r`	carriage return (CR)
`\"`	quotation mark
`\'`	single quotation mark
`\\`	backslash
`\`*oct-rest*	octal character code
`\x`*hex-rest*	hexadecimal character code

The number of constant character string allowed is indicated by *string-char* .

The *oct-rest* octal character code consists of a sequence of up to three octal digits and ends with the first not-octal character. The *hex-rest* hexadecimal character code consists of a sequence of any number of hexadecimal digits and ends with the first not-hexadecimal character.

If an overflow in a character occurs during translation while converting an octal or hexadecimal character code, an error is generated.

string-constant:
    `"`*string-char-seq*$_{opt}$`"`
string-char-seq:
    string-char string-char-seq$_{opt}$

Literal Symbols

Literal symbols in OFML always begin with the special character "'@"', directly followed by a character string, which passes the rules for identifiers.[3]

symbol-constant:
    `@`*ident*

3.2.6 Operators

The following tokens are handled by OFML as operators:

[3] By using the symbol(...) constructor, it is possible to generate symbols from any character string; see Section 3.3.3.

operator:

```
.   | (  | [  | ++ | -- | !  | !! | ~  | $
*   | /  | %  | +  | -  | << | >> | >>>| <
<=  | >= | >  | == | != | ~= | <? | >? | &
^   | |  | && | || | => | ?  | :  | *= | /=
%=  | += | -= | &= | ^= | |= | <<=| >>=| >>>=
=   | ,  | @( | :: | instanceof
```

3.2.7 Delimiters

The following tokens in OFML represent delimiters:

delimiter:

```
::  |  {  |  }  |  ;  |  )  |  ]
```

3.3 Types

OFML is a dynamically typified language, meaning that the type of a variable or expression generally is not known until runtime.

Apart from the class definition, there are no special syntactical constructs for types OFML. Types are objects and, as such, are also stored in variables like all other objects. Within the framework of the operations defined for types, they can be handled like any other object. Mainly this means that they can be assigned, passed to functions and called.

The two basic kinds of types in OFML are the simple types and the reference types. Simple types are the numerical types, the symbol type and the `Void` type. The reference types are predefined reference types or user-defined classes.

3.3.1 Objects and Variables

An object is an instance of a class. It is generated by the calling of the corresponding class. Objects are accessed via references.

A variable is a memory region where the value of a simple type or the reference to an object of a reference type is stored.

There are two kinds of variables, named and unnamed. Named variables are all the variables that can be specified by an identifier. Unnamed variables have to be accessed using an operator (such as the `[]` index operator).

3.3.2 Operations for all Types

All types inherit from the `Object` root type. This makes the following operations available to all types:

- The constructor. This is a function (see Section 3.6.3) that requires a type-specific number of parameters and, for simple types, returns a new value of the type or, for reference types, a reference to a newly generated object.
- The assignment via the `=` operator (see Section 3.6). Here, the variables on the left side of the assignment operator are assigned the value from the result of the expression on the right side. If the result has a reference type, the reference is assigned, without a new instance of the referenced object being created.
- The passing as argument to a function. This takes place according to the rules of assignment by the `=` operator, where the argument is assigned the corresponding, formal parameters of the function.
- The comparison using the `==` or `!=` operator. For simple types, the values themselves are compared, while, if not otherwise defined, for reference types, object identity is verified.
- The verification of the type using the `instanceof` operator.

3.3.3 Simple Types

All simple types are defined in the `::cobra::lang` package.

The `Void` Type

The `Void` type is always used if a variable is to have a non-concrete value. The only possible value for the `Void` type is `NULL`.

Integers

Integers are represented by the `Int` type and have a size determined by the machine[4]. The available value range can be found using the static `Int::MIN_VALUE` member (the amount being the largest representable negative value) and `Int::MAX_VALUE` (largest representable positive value).

The `Int()` constructor can be called either without arguments (in which case the value of 0 is returned) or with an argument with one of the following types:

- `Int`: The value of the argument is copied.
- `Float`: A conversion from `Float` to `Int` is carried out and any fractional part is cut off. If the available value range is exceeded, the result is undefined.

[4]With most currently distributed architectures, these are 32-bit numbers in with complement of two. The available value range is $[-2147483648, 2147483647]$.

- `Symbol`: A number that is unambiguously assigned to the symbol is returned.
- `String`: An attempt is made to interpret the string as an integer constant. If, in doing so, the rules specified in Section 3.2.5 are violated, an exception is triggered (see Section 3.5.3).

The following operators (see Section 3.6) can be applied to the `Int` type.

- The arithmetical operators: the `+` and `-` operators in prefix and infix form, the `++` and `--` operators in prefix and postfix form and the `*`, `/` and `%` infix operators.
- The relational operators: `==`, `!=`, `<`, `>`, `<=`, `>=`, `<?` and `>?`.
- The logical operators: `!` and `!!`.
- The bitwise operators: `&`, `|`, `^`, `~`, `<<`, `>>` and `>>>`.
- All combined assignments that can be formed using the above operators.

Floating-point Numbers

Floating-point numbers are represented by the `Float` type and have a size determined by the machine[5]. The available value range can be found using the static `Float::MIN_VALUE` member (the amount being the largest representable negative value) and `Float::MAX_VALUE` (largest representable positive value).

Depending on implementation, the static `Float::HUGE_VAL` member is either infinite positive or the largest representable positive value. It is used by arithmetical operations on floating-point values, sometimes with a minus sign, to signalize an overflow.

The `Float()` constructor can be called either without arguments (in which case the value of 0.0 is returned) or with an argument with one of the following types:

- `Float`: The value of the argument is copied.
- `Int`: A conversion from `Int` to `Float` is carried out.
- `String`: An attempt is made to interpret the string as a floating-point constant. If, in doing so, the rules specified in Section 3.2.5 are violated, an exception is triggered (see Section 3.5.3).

The following operators (see Section 3.6) can be applied to the `Float` type.

- The arithmetical operators: the `+` and `-` operators in prefix and infix form, the `++` and `--` operators in prefix and postfix form and the `*`, `/` and `%` infix operators.
- The relational operators: `==`, `!=`, `<`, `>`, `<=`, `>=`, `<?` and `>?`.
- The logical operators: `!` and `!!`.
- All combined assignments that can be formed using the above operators.

[5] With most currently distributed architectures, the amount of the smallest representable number is $\pm 2.2 \cdot 10^{-308}$, the largest is $\pm 1.8 \cdot 10^{308}$ and the accuracy is 15 decimal places.

Arithmetic and Type Conversion

Depending on the types of the operands, arithmetical calculations are carried out either in `Int` or `Float`. `Float` is used if at least one of the operands is a `Float` type, except for combined assignments, in which an `Int` type value is located on the left side.

Implicit type conversions for numerical types occur under the following conditions:

- If one of the operands is of the `Int` type and the calculation takes place in `Float`, the operand is converted to `Float`.
- If one of the operands is of the `Float` type and the calculation takes place in `Int`, the operand is converted to `Int`. Any fractional part is cut off. If an overflow occurs during the conversion, the result is undefined.

The following rules apply to calculations in `Int`:

- The complement of two is used for the internal representation of integer values.
- The result is undefined if it is not representable in a value range of `Int`. Addition and subtraction operations represent exceptions, for which the result comes from the lowest-value bit of an integer value of sufficient size.
- Division by `0` triggers an exception.

The following rules apply to calculations in `Float`:

- If the exact result cannot be represented, either the next higher or next lower representable value is applied, depending on implementation[6].
- The amount of the result is `Float::HUGE_VAL` if it is not representable in the value range of `Float` due to an overflow. The +/- sign corresponds to the +/- sign of the correct value.
- The result is `0` if it is no longer representable in the value range of `Float` due to an underflow. Whether the +/- sign is preserved is dependent on implementation.
- An exception is triggered if the operand is not within the range of definition of the operation.

Symbols

Symbols represent the dynamic counterpart to numeration constants in statically typified languages. Internally, they are represented by unique integers, which, using the `Int()` function, are also available externally (see Section 3.3.3). With this representation, very fast comparison of symbols is attainable (in contrast to string comparison).

[6] The direction of rounding can differ from operation to operation and is not dependent on the amount of the difference to the next lower or next higher representable value.

In various instances of an OFML program, the conversion of the same string to a symbol can lead to variously applied integers for the internal representation. Due to this, the outcome of comparisons on symbols that are based on an order is not reproducible in various instances of an OFML program.

The `Symbol()` constructor demands an argument with one of the following types:

- `Symbol`: The value of the argument is copied.
- `String`: The string (without the leading `@`) is converted to a symbol. The `@foo` and `Symbol("foo")` expressions are thereby equivalent. This method also makes it possible to convert strings into symbols that do not meet the requirements for identifiers, such as `Symbol("500 Motels")`.

The following operators (see Section 3.6) can be applied to the `Symbol` type.

- The `==`, `!=`, `<`, `>`, `<=`, `>=`, `<?` and `>?` relational operators.

3.3.4 Reference Types

Automatic Garbage Collection

The language definition of OFML requires the implementation of an automatic garbage collection. Objects of reference types are generated implicitly when the constructor is called (see exception below). There is no way of releasing objects explicitly. Instead, they can be released automatically by the system as soon as no more references to the object exist. However, when and if objects that are no longer referenced are released is not fixed[7].

The language definition makes the manner of implementation of the automatic garbage collection optional.

Operators to Reference Types

The behavior of operators, which can be used within expressions, is firmly defined for the simple types. If the operand of a unary operator or the left operand of a binary operator delivers a reference type, an instance-oriented method, specific to the operator, is called for the reference type. These methods are freely definable for classes. Exceptions are the `$` (symbol resolution operator), `!` (logical negation), `instanceof` (type verification), `>?` (maximum), `<?` (minimum),

[7] This differs greatly from the algorithm used for automatic garbage collection. When using reference counters, objects are generally released as soon as there are no more references to them. However, a garbage collection based only on reference counters does not release data structures with cycles. Due to this, the programmer has to break such cycles before the last reference to such a data structure is released.

Other methods, for which, based on a known number of referenced objects, all reachable (thus referenced) objects are determined, delay the release of objects that are no longer referenced and, under certain circumstances, do not release all objects if conservative algorithms are used.

A combination of both methods is also conceivably.

`&&` (logical AND), `||` (logical OR), `?:` (conditional expression), `=` (assignment) and `,` (comma operator) operators, whose behaviors either are firmly preset for reference types, are mapped to other operators or fundamentally cannot be applied to reference types.

The operator methods to be used in class definitions are described in Section 3.6 below their corresponding operators.

Sequence Types

Sequence types are all of the reference types that can be seen as sequences of objects. For this to be the case, they have to meet the following conditions:

- The `size()` method must be defined and return a nonnegative *size* `Int` value.
- The *operator[](pIdx(Int))* and *operator[](pIdx(Int), pValue(Object))* index operators must be defined for each *pIdx* integer index within the range of $[0, size)$.
- The sequential access via the index operators, forward or backward, should require constant time.

Of the predefined types in OFML, `String`, `Vector` and `List` are sequence types.

3.4 Predefined Reference Types

The following sections describe the predefined reference types in OFML; user-defined classes are described in Section 3.8.

All predefined reference types are defined in the `::cobra::lang` package.

3.4.1 The `Type` Metatype

All types, including the `Type` type, are instances of the `Type` type.

Type names OFML are variables having a reference to an instance of `Type`.

The following operators (see Section 3.6) and methods (see Section 3.8.4) can be applied to instances of `Type`:

operator()(parameters) → *Object*
: The function call operator defined for all types is called a constructor. The constructor generates an instance of the type for which it is called and calls the `initialize()` method for the instance if it exists. The arguments passed to the constructor are forwarded to the initialization method.

getName() → *Symbol*
: The $getName()$ method returns the simple name of the type as a `Symbol`.

getFullName() → String

The *getFullName*() method returns the fully qualified name of the type as a string.

subClassOf(pType(Type)) → Int

The *subClassOf*() method returns 1 if the type for which it was called is either identical to the type passed as an argument or is derived from this type. Otherwise, the value returned is 0. If the argument is not of the `Type` type, an exception is triggered.

3.4.2 Functions

Functions are represents in OFML by the `Func` and `CFunc` types. `Func` is the type for functions defined in OFML, while `CFunc` is the type for predefined functions. In addition to the operators that are available to all types (see Section 3.3.2), `Func` and `CFunc` implement the ”‘`()`”’ function call operator.

3.4.3 Character Strings

Character strings are represented by the `String` type and are represented Internally by a sequence of 8-bit values, where each value corresponds to one character. Whether the null character (`'\0'`) can be a component of a string depends on implementation.

The `String()` constructor can be called either without arguments (in which case the empty string, `""`, is returned) or with an argument with one of the following types:

- `String`: A copy of the string passed as an argument is created.
- `Symbol`: A new string is generated, the content of which is equivalent to the string represented by the symbol.
- `Int`, `Float`: A new string is generated, which contains the result of the conversion of numbers in a string.

A string constant in an expression causes an implicit call of the `String()` constructor, for which the string is passed as an argument.

The following operators (see Section 3.6) and methods (see Section 3.8.4) can be applied to the `String` type:

operator==(pString(String)) → Int
operator!=(pString(String)) → Int
operator<(pString(String)) → Int
operator<=(pString(String)) → Int
operator>=(pString(String)) → Int
operator>(pString(String)) → Int

The result is 1 if the character-to-character comparison of both strings turns out identical. Otherwise the result is 0. the character-to-character comparison of strings is described on page 35 under the *compare*() function.

operator+(pString(String)) → String

The `+` addition operator anticipates a string on the right side. Otherwise, an exception is triggered. It in turn creates a new string consisting of the linked strings on the left and right sides of the operator.

operator+=(pString(String)) → String

The `+=` addition operator anticipates a string on the right side. Otherwise, an exception is triggered. It in turn appends the string on the right side of the operator onto the string on the left side. The result is the combined string.

operator[](pIdx(Int)) → Int
operator[](pIdx(Int), pChar(Int))

The index operators anticipate a value of the `Int` type as the $pIdx$ index. Otherwise, an exception is triggered. Assume the length of the string is len. If $pIdx < 0$, $pIdx$ is set to $pIdx + len$. If afterwards, $pIdx < 0 \vee pIdx \geq len$, an exception is triggered. Otherwise, the character at the $pIdx$ position is returned as a positive `Int`.

If the index operator is used on the left side of the assignment operator (the second form of the index operator), the expression on the right side of the assignment operator must return a value of the `Int` type. Otherwise, an exception is triggered. This value, modulo 2^8, is assigned the to the string at the $pIdx$ position.

operator[:](pBegin(Int), pEnd(Int)) → String
operator[:](pBegin(Int), pEnd(Int), pChar(Int))
operator[:](pBegin(Int), pEnd(Int), pString(String))

The `[:]` range operator anticipates both the $pBegin$ and $pEnd$ indexes of the `Int` type. Assume the length of the string is len. If $pBegin < 0$, $pBegin$ is set to $pBegin + len$. Likewise, $pEnd$ is set to $pEnd + len$ if $pEnd < 0$. If, afterwards, $pBegin < 0 \vee pBegin > len$ or $pEnd < 0 \vee pEnd > len$ or $pBegin > pEnd$, an exception is triggered. Otherwise, a substring is returned, starting with the $pBegin$ position and ending with the $pEnd - 1$ position.

If the range operator is used on the left side of the assignment operator (the second and third form of the range operator), the value of the expression on the right side of the assignment operator must be either an `Int` value or a string of any length. Otherwise, an exception is triggered. The substring specified by the $pBegin$ and $pEnd$ indexes is replaced by the character, modulo 2^8, specified by the integer value, or by the string.

operator!!() → Int

The `!!`test operator returns 1 if the length of the string is not null. Otherwise, it returns 0.

getAt(pIdx(Int)) → Int

If $pIdx < 0$, $pIdx$ is set to $pIdx + size()$. If afterwards, $pIdx < 0 \vee pIdx \geq size()$, an exception is triggered. Otherwise, the character at the $pIdx$ position is returned as a positive `Int`.

setAt(pIdx(Int), pChar(Int))

If $pIdx < 0$, $pIdx$ is set to $pIdx + size()$. If, afterwards, $pIdx < 0 \vee pIdx \geq size()$ or $pChar < 0 \vee pChar \geq 2^8$, an exception is triggered. Otherwise, $pChar$ is assigned to the character at the $pIdx$ position.

size() $\rightarrow$ *Int*
: returns the number of characters in the string.

empty() $\rightarrow$ *Int*
: returns 1 if the length of the string is null. Otherwise, it returns 0.

resize(pSize(Int), pChar(Int) = ' ')
: If $pChar < 0 \lor pChar \geq 2^8$, an exception is triggered. Otherwise, the new length of the string is set to $pSize$. If the new value is greater than the old length, the $pChar$ character is used to fill it in.

append(pString(String), pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE)
: If $pPos < 0 \lor pPos > pString.size()$ or $pLen < 0$, an exception is triggered. Otherwise, $pLen = \min(pLen, pString.size() - pPos)$. The substring of $pString$ with the $pLen$ length is then, starting at the $pPos$ position appended to the string.

append(pNum(Int), pChar(Int) = ' ')
: If $pNum < 0$ or $pChar < 0 \lor pChar \geq 2^8$, an exception is triggered. Otherwise, the $pChar$ character is appended to the string $pNum$ number of times.

assign(pString(String), pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE)
: If $pPos < 0 \lor pPos > pString.size()$ or $pLen < 0$, an exception is triggered. Otherwise, $pLen = \min(pLen, pString.size() - pPos)$. The string is then set to the substring of $pString$, which begins at the $pPos$ position and has a length of $pLen$.

assign(pNum(Int), pChar(Int) = ' ')
: If $pNum < 0$ or $pChar < 0 \lor pChar \geq 2^8$, an exception is triggered. Otherwise, the string is set to a sequence of $pLen$ times the $pChar$ character.

insert(pPos1(Int), pString(String), pPos2(Int) = 0, pLen(Int) = Int::MAX_VALUE)
: If $pPos1 < 0 \lor pPos1 > size()$ or $pPos2 < 0 \lor pPos2 > pString.size()$ or $pLen < 0$, an exception is triggered. Otherwise, $pLen$ is set to $\min(pLen, pString.size() - pPos2)$. Then, the substring from $pString$, beginning at the $pPos2$ position and having a length of $pLen$, is inserted at position $pPos1$.

insert(pPos(Int), pNum(Int), pChar(Int) = ' ')
: If $pPos < 0 \lor pPos > size()$ or $pNum < 0$ or $pChar < 0 \lor pChar >= 2^8$, an exception is triggered. Otherwise, the $pChar$ character is inserted at the $pPos$ position $pNum$ number of times.

remove(pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE)
: If $pPos < 0 \lor pPos > size()$ or $pLen < 0$, an exception is triggered. Otherwise, $pLen$ is set to $\min(pLen, size() - pPos)$ Then, $pLen$ characters are removed starting at position $pPos$.

replace(pPos1(Int), pLen1(Int), pString(String), pPos2(Int) = 0, pLen2(Int) = Int::MAX_VALUE)
: If $pPos1 < 0 \lor pPos1 > size()$ or $pPos2 < 0 \lor pPos2 > pString.size()$ or $pLen1 < 0$ or $pLen2 < 0$, an exception is triggered. Otherwise, $pLen1$ is set to $\min(pLen1, size() - pPos1)$ and $pLen2$ is set to $\min(pLen2, pString.size() - pPos2)$. Then, $pLen1$ characters starting at position $pPos1$ are replaced by a substring from $pString$, which begins at position $pPos2$ and is $pLen2$ characters long.

replace(pPos(Int), pLen(Int), pNum(Int), pChar(Int) = ' ')

  If $pPos < 0 \vee pPos > size()$ or $pLen < 0$ or $pNum < 0$ or $pChar < 0 \vee pChar \geq 2^8$, an exception is triggered. Otherwise, $pLen$ is set to $\min(pLen, size() - pPos)$ Then, $pLen$ characters starting at position $pPos$ are replaced by $pNum$ number of new $pChar$ characters.

swap(pString(String))

  swaps the contents of two strings.

find(pString(String), pPos(Int) = 0) → Int

  If possible, the smallest res value is returned for which these are valid:
  $res \geq pPos \wedge res + pString.size() \leq size()$ and
  $getAt(res + i) = pString.getAt(i)$ for all $i \geq 0 \wedge i < pString.size()$
  Otherwise, -1 is returned.

find(pChar(Int), pPos(Int) = 0) → Int

  If $pChar < 0 \vee pChar \geq 2^8$, an exception is triggered. Otherwise, if possible, the smallest res value is returned for which these are valid:
  $res \geq pPos \wedge res < size()$ and $getAt(res) = pChar$
  Otherwise, -1 is returned.

rfind(pString(String), pPos(Int) = Int::MAX_VALUE) → Int

  If possible, the largest res value is returned for which these are valid:
  $res \leq pPos \wedge res + pString.size() \leq size()$ and
  $getAt(res + i) = pString[i]$ for all $i \geq 0 \wedge i < pString.size()$
  Otherwise, -1 is returned.

rfind(pChar(Int), pPos(Int) = Int::MAX_VALUE) → Int

  If $pChar < 0 \vee pChar \geq 2^8$, an exception is triggered. Otherwise, if possible, the largest res value is returned for which these are valid:
  $res \leq pPos \wedge res < size()$ and $getAt(res) = pChar$
  Otherwise, -1 is returned.

findFirstOf(pString(String), pPos(Int) = 0) → Int

  If possible, the smallest res value is returned for which these are valid:
  $res \geq pPos \wedge res < size()$ and
  $getAt(res) = pString.getAt(i)$ for at least one $i \geq 0 \wedge i < pString.size()$
  Otherwise, -1 is returned.

findFirstOf(pChar(Int), pPos(Int) = 0) → Int

  If $pChar < 0 \vee pChar \geq 2^8$, an exception is triggered. Otherwise, if possible, the smallest res value is returned for which these are valid:
  $res \geq pPos \wedge res < size()$ and $getAt(res) = pChar$ Otherwise, -1 is returned.

findLastOf(pString(String), pPos(Int) = Int::MAX_VALUE) → Int

  If possible, the largest res value is returned for which these are valid:
  $res \leq pPos \wedge pPos < size()$ and
  $getAt(res) = pString.getAt(i)$ for at least one $i \geq 0 \wedge i < pString.size()$
  Otherwise, -1 is returned.

findLastOf(pChar(Int), pPos(Int) = Int::MAX_VALUE) → Int

  If $pChar < 0 \vee pChar \geq 2^8$, an exception is triggered. Otherwise, if possible, the largest res

value is returned for which these are valid:
$res \leq pPos \land pPos < size()$ and $getAt(res) = pChar$
Otherwise, -1 is returned.

findFirstNotOf(pString(String), pPos(Int) = 0) → Int
If possible, the smallest res value is returned for which these are valid:
$res \geq pPos \land res < size()$ and
$getAt(res) = pString.getAt(i)$ for no $i \geq 0 \land i < pString.size()$
Otherwise, -1 is returned.

findFirstNotOf(pChar(Int), pPos(Int) = 0) → Int
If $pChar < 0 \lor pChar \geq 2^8$, an exception is triggered. Otherwise, if possible, the smallest res value is returned for which these are valid:
$res \geq pPos \land res < size()$ and $getAt(res) \neq pChar$
Otherwise, -1 is returned.

findLastNotOf(pString(String), pPos(Int) = Int::MAX_VALUE) → Int
If possible, the largest res value is returned for which these are valid:
$res \leq pPos \land pPos < size()$ and
$getAt(res) = pString.getAt(i)$ for no $i \geq 0 \land i < pString.size()$
Otherwise, -1 is returned.

findLastOf(pChar(Int), pPos(Int) = Int::MAX_VALUE) → Int
If $pChar < 0 \lor pChar \geq 2^8$, an exception is triggered. Otherwise, if possible, the largest res value is returned for which these are valid:
$res \leq pPos \land pPos < size()$ and $getAt(res) \neq pChar$
Otherwise, -1 is returned.

substr(pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE) → String
If $pPos < 0 \lor pPos > size()$ or $pLen < 0$, an exception is triggered. Otherwise, $pLen$ is set to $\min(pLen, size() - pPos)$ Then, a new string is created and returned whose contents are equivalent to the substring beginning at $pPos$ and having a length of $pLen$.

toUpper(pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE)
If $pPos < 0 \lor pPos > size()$ or $pLen < 0$, an exception is triggered. Otherwise, $pLen$ is set to $\min(pLen, size() - pPos)$ Then, if $pLen > 0$, all lowercase letters from position $pPos$ up to and including position $pPos + pLen - 1$ are converted to uppercase letters.

toLower(pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE)
If $pPos < 0 \lor pPos > size()$ or $pLen < 0$, an exception is triggered. Otherwise, $pLen$ is set to $\min(pLen, size() - pPos)$ Then, if $pLen > 0$, all uppercase letters from position $pPos$ up to and including position $pPos + pLen - 1$ are converted to lowercase letters.

compare(pPos1(Int), pLen1(Int), pString(String), pPos2(Int) = 0, pLen2(Int) = Int::MAX_VALUE) → Int

runs a character-to-character comparison on the $pStr1 = substr(pPos1, pLen1)$ and $pStr2 = pString.substr(pPos2, pLen2)$ strings. The result is -1 if $pStr1$ is smaller than $pStr2$. It is $+1$ if $pStr1$ is larger than $pStr2$. And it is 0 if $pStr1$ and $pStr2$ are the same.

When two strings are compared character-to-character, the characters of both strings, starting at position 0, are compared to each other in pairs. The comparison is terminated as soon

as a pair of unidentical characters or the end of at least one string is reached. In the first case, the result of the comparison is -1 if the code of the character in the first (or left) string is less than the code of the character in the second (or right) string. Accordingly, the result is $+1$ if the code of the character in the first string is greater than the code of the character in the second string. In the second case, the result is 0 if the ends of both strings are reached simultaneously. It is -1 if the end of the first sting was reached and $+1$ if the end of the second string was reached.

compare(pString(String), pPos(Int) = 0, pLen(Int) = Int::MAX_VALUE) → Int
: Corresponds to calling $compare(0,$ `Int::MAX_VALUE`$, pString, pPos, pLen)$.

getHashValue() → Int
: The $getHashValue()$ method returns a hash value for the string. Like the `==` operator, it always returns the same hash value for identical strings, but can also return the same hash value for unidentical strings.

3.4.4 Vectors

The `Vector` type represents one-dimensional vectors. Multidimensional fields can be formed from vectors of vectors, whereby the dimensions of the individual vectors do not have to be identical.

Random access to individual vector elements through their indexes requires constant time, as do insert and delete operations at the ends of vectors. For insert and delete operations at the start or in the middle of the vector, the required time is proportional to the number of subsequent vector elements.

Insert operations might require additional time for reallocation of the vector.

Vectors can be created in two ways:

- By calling the `Vector` constructor. *Vector(pSize(Int), ...)* creates a vector with *pSize* elements, which are initialized with `NULL`. The entry of a second $pSize2$ argument of the `Int` type initializes the vector with vectors of size $pSize2$, thus creating a two-dimensional field. This can be continued recursively by entering three and more arguments of the `Int` type to create three and higher multidimensional fields.
- By entering the elements in brackets, separated by commas. For every element there can be any type of assignment expression, the result of which is to be used to initialize the element.

special-ctor:
    `[` *arg-expr-list*$_{opt}$ `]`
arg-expr-list:
    assign-expr
    arg-expr-list `,` *assign-expr*

The following operators (see Section 3.6) and methods (see Section 3.8.4) can be applied to the `Vector` type:

operator==(pSeq(Object)) $\rightarrow$ *Int*
operator!=(pSeq(Object)) $\rightarrow$ *Int*

The `==` and `!=` relational operators with a *vec* vector on the left side anticipate a *pSeq* instance of a sequence type (see Section 3.3.4) on the right side. The *vec* vector and the *pSeq* sequence are the same if:

- The length of *vec* is equal to the length of *pSeq*.
- For every *idx* integer index in the range of $[0, vec.size())$, the comparison of $vec[idx]$ to $pSeq[idx]$ using the `==` *true* operator yields ($\neq 0$). The first comparison of elements that triggers an exception or does not yield *true* terminates the comparison of the *vec* vector to the *pSeq* sequence.

operator[](pIdx(Int)) $\rightarrow$ *Object*
operator[](pIdx(Int), pObj(Object))

The `[]` index operator anticipates a value of the `Int` type as index $pIdx$. Assume the length of the vector is *len*. If $pIdx < 0$, $pIdx$ is set to $pIdx + len$. If afterwards, $pIdx < 0 \vee pIdx \geq len$, an exception is triggered. Otherwise, the vector element indexed by $pIdx$ is returned.

If the index operator is used on the left side of the assignment operator (the second form of the index operator), the result of the expression on the right side of the assignment operator is assigned the vector element indexed by $pIdx$.

operator[:](pBegin(Int), pEnd(Int)) $\rightarrow$ *Vector*
operator[:](pBegin(Int), pEnd(Int), pSeq(Object))

The `[:]` range operator anticipates both the *pBegin* and *pEnd* indexes of the `Int` type. Assume the length of the vector is *len*. If $pBegin < 0$, *pBegin* is set to $pBegin + len$. Likewise, *pEnd* is set to $pEnd + len$ if $pEnd < 0$. If, afterwards, $pBegin < 0 \vee pBegin > len$ or $pEnd < 0 \vee pEnd > len$ or $pBegin > pEnd$, an exception is triggered. Otherwise, a vector is returned that consists of the elements of the original vector that are indexed by *pBegin* to $pEnd - 1$.

If the range operator is used on the left side of the assignment operator (the second form of the range operator), the result of the expression on the right side of the assignment operator must be a sequence (see Section 3.3.4) of any length. The elements of the vector indexed by *pBegin* to $pEnd - 1$ are replaced by all of the elements of the sequence.

operator!!() $\rightarrow$ *Int*
The `!!`test operator returns 1 if the length of the vector is not null. Otherwise, it returns 0.

getAt(pIdx(Int)) $\rightarrow$ *Object*
If $pIdx < 0$, $pIdx$ is set to $pIdx + size()$. If afterwards, $pIdx < 0 \vee pIdx \geq size()$, an exception is triggered. Otherwise, the element with index $pIdx$ is returned.

setAt(pIdx(Int), pObject(Object))
If $pIdx < 0$, $pIdx$ is set to $pIdx + size()$. If afterwards, $pIdx < 0 \vee pIdx \geq size()$, an exception is triggered. Otherwise, *pObject* is assigned the element with the $pIdx$ index.

size() $\rightarrow$ *Int*
The number of elements of the vector is returned.

empty() $\rightarrow$ *Int*
: If $size() = 0$, 1 is returned;otherwise 0.

front() $\rightarrow$ *Object*
: If $size() = 0$, an exception is triggered. Otherwise, the first element of the vector is returned.

back() $\rightarrow$ *Object*
: If $size() == 0$, an exception is triggered. Otherwise, the last element of the vector is returned.

pushBack(pObject(Object))
: As the last element, $pObject$ is appended to the vector.

popBack() $\rightarrow$ *Object*
: If $size() = 0$, an exception is triggered. Otherwise, the last element of the vector is removed from this and returned.

insert(pPos(Int), pNum(Int) = 1, pObj(Object))
: If $pPos < 0 \vee pPos > size()$ or $pNum < 0$, an exception is triggered. Otherwise, $pObj$ is inserted$pNum$ number of times at the $pPos$ index.

erase(pBegin(Int), pEnd(Int) = pBegin + 1)
: If $pBegin < 0 \vee pBegin > size()$ or $pEnd < 0 \vee pEnd > size()$ or $pBegin > pEnd$, an exception is triggered. Otherwise, if $pBegin < pEnd$, the elements indexed by $pBegin$ to $pEnd - 1$ are deleted from the vector.

swap(pVec(Vector))
: If $pVec$ is not an instance of the `Vector` type, an exception is triggered. Otherwise, the contents of both vectors are swapped.

3.4.5 Lists

The `List` type represents double-chained lists.

Sequential access to individual elements of a list, both forwards and backwards, as well as insert and delete operations at any position, require constant time. The most required time for random access to list elements is proportional to the minimum distance to the start or end of the list.

Lists can be created in two ways:

- By calling the `List` constructor. This can be called with zero or more arguments. The arguments form the individual elements of the list.
- By entering the elements in `@()`, separated by commas. For every element there can be any type of assignment expression, the result of which is to be used to initialize the element.

special-ctor:
: `@(` *arg-expr-list*$_{opt}$ `)`

arg-expr-list:
: *assign-expr*
: *arg-expr-list* **,** *assign-expr*

The following operators (see Section 3.6) and methods (see Section 3.8.4) can be applied to the `List` type:

operator==(pSeq(Object)) $\rightarrow$ *Int*
operator!=(pSeq(Object)) $\rightarrow$ *Int*

The `==` and `!=` relational operators with a *list* list on the left side anticipate a *pSeq* instance of a sequence type (see Section 3.3.4) on the right side. The *list* list and the *pSeq* sequence are the same if:

- The length of *list* is equal to the length of *pSeq*.
- For every *idx* integer index in the range of $[0, list.size())$, the comparison of $list[idx]$ to $pSeq[idx]$ using the `==` *true* operator yields ($\neq 0$). The first comparison of elements that triggers an exception or does not yield *true* terminates the comparison of the *list* list to the *pSeq* sequence.

operator[](pIdx(Int)) $\rightarrow$ *Object*
operator[](pIdx(Int), pObject(Object))

The `[]` index operator anticipates a value of the `Int` type as index $pIdx$. Assume the length of the list is *len*. If $pIdx < 0$, $pIdx$ is set to $pIdx + len$. If afterwards, $pIdx < 0 \vee pIdx \geq len$, an exception is triggered. Otherwise, the list element indexed by $pIdx$ is returned.

If the index operator is used on the left side of the assignment operator (the second form of the index operator), the result of the expression on the right side of the assignment operator is assigned the list element indexed by $pIdx$.

operator[:](pBegin(Int), pEnd(Int)) $\rightarrow$ *List*
operator[:](pBegin(Int), pEnd(Int), pSeq(Object))

The `[:]` range operator anticipates both the *pBegin* and *pEnd* indexes of the `Int` type. Assume the length of the list is *len*. If $pBegin < 0$, *pBegin* is set to $pBegin + len$. Likewise, *pEnd* is set to $pEnd + len$ if $pEnd < 0$. If, afterwards, $pBegin < 0 \vee pBegin > len$ or $pEnd < 0 \vee pEnd > len$ or $pBegin > pEnd$, an exception is triggered. Otherwise, a list is returned that consists of the elements of the original list that are indexed by *pBegin* to $pEnd - 1$.

If the range operator is used on the left side of the assignment operator (the second form of the range operator), the result of the expression on the right side of the assignment operator must be a sequence (see Section 3.3.4) of any length. The elements of the list indexed by *pBegin* to $pEnd - 1$ are replaced by all of the elements of the sequence.

operator!!() $\rightarrow$ *Int*
The `!!`test operator returns 1 if the length of the list is not null. Otherwise, it returns 0.

getAt(pIdx(Int)) $\rightarrow$ *Object*
If $pIdx < 0$, $pIdx$ is set to $pIdx + size()$. If afterwards, $pIdx < 0 \vee pIdx \geq size()$, an exception is triggered. Otherwise, the element with index $pIdx$ is returned.

setAt(pIdx(Int), pObject(Object))
If $pIdx < 0$, $pIdx$ is set to $pIdx + size()$. If afterwards, $pIdx < 0 \vee pIdx \geq size()$, an exception is triggered. Otherwise, *pObject* is assigned the element with the $pIdx$ index.

size() $\rightarrow$ *Int*
: The number of elements of the list is returned.

empty() $\rightarrow$ *Int*
: If $size() = 0$, 1 is returned;otherwise 0.

front() $\rightarrow$ *Object*
: If $size() = 0$, an exception is triggered. Otherwise, the first element of the list is returned.

back() $\rightarrow$ *Object*
: If $size() = 0$, an exception is triggered. Otherwise, the last element of the list is returned.

pushFront(pObject(Object))
: As the first element, $pObject$ is appended to the list.

pushBack(pObject(Object))
: As the last element, $pObject$ is appended to the list.

popFront() $\rightarrow$ *Object*
: If $size() = 0$, an exception is triggered. Otherwise, the first element of the list is removed from this and returned.

popBack() $\rightarrow$ *Object*
: If $size() = 0$, an exception is triggered. Otherwise, the last element of the list is removed from this and returned.

insert(pPos(Int), pNum(Int) = 1, pObj(Object))
: If $pPos < 0 \vee pPos > size()$ or $pNum < 0$, an exception is triggered. Otherwise, *pObj* is inserted$pNum$ number of times at the $pPos$ index.

erase(pBegin(Int), pEnd(Int) = pBegin + 1)
: If $pBegin < 0 \vee pBegin > size()$ or $pEnd < 0 \vee pEnd > size()$ or $pBegin > pEnd$, an exception is triggered. Otherwise, if $pBegin < pEnd$, the elements indexed by $pBegin$ to $pEnd - 1$ are deleted from the list.

swap(pList(List))
: If $pList$ is not an instance of the `List` type, an exception is triggered. Otherwise, the contents of both lists are swapped.

splice(pPos(Int), pList(List), pBegin(Int) = 0, pEnd(Int) = pBegin + 1)
: If $pPos < 0 \vee pPos > size()$ or $pBegin < 0 \vee pBegin > pList.size()$ or $pEnd < 0 \vee pEnd > pList.size()$ or $pBegin > pEnd$, an exception is triggered. Otherwise, if $pBegin < pEnd$, the elements from $pList$ that are indexed by $pBegin$ to $pEnd - 1$ are removed from $pList$ and inserted in the same order starting at $pPos$.

remove(pObj(Object))
: Compares each element of the list, starting with the first and ascending sequentially until the last, using the `==` operator with $pObj$, whereby the list element appears on the left and *pObj* on the right of the relational operator. If an exception is triggered by the comparison, the function turns back immediately. Otherwise, if the comparison resulted in *true* ($\neq 0$), it removes the current list element from the list.

removeIf(pPred(Func))

The *pPred* argument has to be a function that expects an argument and returns either *true* ($\neq 0$) or *false* (0) (unary predicate).

The $removeIf()$ method calls the*pPred* function for every element of the list, starting with the first and ascending sequentially until the last, whereby the list element is passed as an argument to the function. If an exception is triggered by the function, *removeIf()* turns back immediately. If the return value of the function is not an `Int` type, an exception is triggered. Otherwise, if the return value of the function is *true* ($\neq 0$), it removes the current list element from the list.

unique()

Removes from each sequence all identical, consecutive elements except the first one. To do so, it compares each current element to its directly successive element using the `==` operator, whereby the current element appears on the left and the successive element on the right of the relational operator. If an exception is triggered by the relational operator, the function turns back immediately. Otherwise, the successive element is deleted if the comparison results in *true*, or the successive element is made the current element if the comparison results in *false*.

unique(pPred(Func))

The *pPred* argument has to be a function that expects two arguments and returns *true* ($\neq 0$) if both arguments are the same, or *false* (0) if they are different (binary predicate). If the return value is not an `Int` type, *unique*() triggers an exception.

The *unique*() method with *pPred* as an argument behaves exactly as it does without an argument except that, instead of the `==` operator, the *pPred* function is called, to which the current element is passed as the first argument and the successive element as the second.

merge(pList(List))

The *merge*() method merges two lists, sorted in ascending order, into a single sorted list. It uses the `<` operator, to the left side of which is passed an element from *pList* and to the right an element from $self$. If an exception is triggered by the relational operator, *merge*() turns back immediately and the content of each list is undefined. The *pList* argument list is empty after *merge*() comes back. If elements are equivalent in both lists, the elements from $self$ are placed before those from *pList* in the result list. The order of elements in a list remains unchanged in the result list.

merge(pList(List), pPred(Func))

The second *pPred* argument has to be a function that expects two arguments and returns *true* ($\neq 0$) if the first argument is smaller than the second argument or *false* (0) otherwise (binary predicate). If the return value is not an `Int` type, *merge*() triggers an exception.

The *merge*() method with two arguments behaves exactly as it does with only one argument except that, instead of the `<` operator, the *pPred* function is called.

sort()

Sorts the list using the relational `<` operator, which can be called for any of the elements in a list. If an exception is triggered by *sort*() either directly or indirectly, the content of the list is undefined. The order of same elements in the unsorted list remains intact in the sorted list. The complexity of *sort*() is approximately $size() \cdot \log(size())$ relational operations.

sort(pPred(Func))

The *pPred* argument has to be a function that expects two arguments and returns *true* ($\neq 0$) if the first argument is smaller than the second argument or *false* (0) otherwise (binary predicate). If the return value is not an `Int` type, *sort*() triggers an exception.

The *sort*() method with one argument behaves exactly as it does without an argument except that, instead of the `<` relational operator, the *pPred* function is called.

reverse()

The *reverse*() method reverses the order of the elements in the list.

3.4.6 Hash Tables

The `Hash` type makes hash tables available. A hash table contains a set of entries in pairs. Each entry consists of a key and a value. The key is used to access the value for read or write operations.

Values of the simple types, `Int`, `Float` and `Symbol`, as well as all reference types that define the instance-oriented `getHashValue()` method, can be used as keys. The `getHashValue()` method must return a value of the `Int` type, which is the same for two keys for which the equality operator, `==`, when applied to them, yields *true*.

Keys of different types can be used in a hash table. Two keys are considered the same if their types are identical and the equality operator, `==`, when applied to both keys, yields *true*.

The *Hash()* constructor creates an empty hash table. The initial size of the hash table is dependent on implementation. It grows with the amount of values stored in the hash table, whereby the time for hash table enlargement is distributed to consecutive read or write accesses, while the additional time taken for an access is on average independent of the size of the hash table.

The following operators (see Section 3.6) and methods (see Section 3.8.4) can be applied the `Hash` type:

operator[](pKey(Object)) → *Object*
operator[](pKey(Object), pValue(Object))

The index operator anticipates a key as an index value that meets the key requirements listed above. If the hash table contains an entry with this key, the value stored for this entry is returned. Otherwise, an exception is triggered.

If the index operator is used on the left side of the assignment operator, the result of the expression on the right side of the assignment operator is stored as a value under the specified key in the hash table. If no entry yet exists for this key, a new entry is created.

operator!!() → *Int*

The test operator returns 1 if the hash table contains at least one entry. Otherwise, it returns 0.

getAt(pKey(Object)) → *Object*

The *getAt*() method anticipates as an argument a key that meets the above-mentioned requirements. If no entry exists for this key, an exception is triggered. Otherwise, the value of the entry is returned.

setAt(pKey(Object), pValue(Object))

The *setAt*() method anticipates as the first argument a key that meets the above-mentioned requirements and, as the second, an object of any type. If no entry exists for this key, a new one is created. Then, the value of the second argument is stored in this entry as a value.

size() $\rightarrow$ *Int*

The number of entries in the hash table is returned.

empty() $\rightarrow$ *Int*

If *size*() =, 1 is returned, otherwise 0.

hasKey(pKey(Object)) $\rightarrow$ *Int*

The *hasKey*() method anticipates as the argument a key that meets the above-mentioned requirements. It returns 1 if an entry with this key exists in the hash table; otherwise it returns 0.

keys() $\rightarrow$ *Vector*

The *keys*() method returns a `Vector` whose individual elements are the keys of all entries in the hash table.

values() $\rightarrow$ *Vector*

The *values*() method returns a `Vector` whose individual elements are the values of all entries in the hash table.

swap(pHash(Hash))

If the argument is not of the `Hash` reference type, an exception is triggered. Otherwise, the entries of both hash tables are swapped.

remove(pKey(Object))

The argument must be a key that meets the above-mentioned requirements. If no entry exists for this key, an exception is triggered. Otherwise, the corresponding entry is deleted.

The identical order of the keys and values returned by the `keys()` and `values()` methods can only be guaranteed if no other methods of the hash table, including index operators, are called between the execution of the two methods.

3.5 Statements

The translation unit (*translation-unit*) shapes the entry symbol of the OFML grammar (see Section3.1.3). Every translation unit consists of an optional package statement, a (potentially empty) sequence of import statements (*import-stmts*) and a (potentially empty) sequence of other statements (*stmt-list*). The syntax and semantics of package and import statements are described in Section 3.7.

translation-unit:
  package-stmt$_{opt}$ *import-stmts*$_{opt}$ *stmt-list*$_{opt}$
import-stmts:
  import-stmts$_{opt}$ *import-stmt*
stmt-list:
  stmt-list$_{opt}$ *stmt*

An OFML statement can contain the following: a definition (*definition-stmt*), an expression (*expr-stmt*), a control statement (*ctrl-stmt*) or a compound statement (*compound-stmt*).

stmt:
    definition-stmt
    expr-stmt
    ctrl-stmt
    compound-stmt

Definitions are handled by the compiler. All other statements are executed in the order in which they appear textually at runtime.

In some cases, either a semicolon or the end of file is expected at the end of a statement[8].

eox:
    ; | EOF

3.5.1 Definitions

The following elements can be introduced by definitions: variables (*var-def*), named functions (*named-func-def*), classes (*class-def*), the name of the package to which the translation unit belongs (*package-stmt*) and the packages imported by the translation unit (*import-stmt*). Package and import statements are described in Section 3.7, class definition in Section 3.8.

definition-stmt:
    var-def
    named-func-def
    class-def

Variable Definitions

A variable definition starts with an optional sequence of modifiers and the `var` keyword, followed by one or more initialization expressions (*init-expr*) separated by commas. The last expression is ended with a semicolon or the end of file (*eox*). Every initialization expression consists of an identifier (*ident*), optionally followed by an assignment operator and an expression (*expr*) evaluated in the value context (see Section 3.6.1). The latter is used to set the initial valued of variables. If neither the assignment operator nor expression (*expr*, see Section3.6) are present, the variable is given the `NULL` value. The identifier becomes valid immediately after the initialization expression contained within it.

[8] The end of file is allowed to terminate a statement so that the semicolon can be dropped in interactive mode.

var-def:
    global-modifiers$_{opt}$ `var` *init-expr-list eox*
init-expr-list:
    init-expr
    init-expr-list `,` *init-expr*
init-expr:
    ident
    ident `=` *expr*

Modifiers are described in Sections 3.7.6 and 3.8.

Named Function Definitions

The definition of a named function begins with an optional sequence of modifiers and the `func` keyword, followed by the name of the function that, as such, becomes valid in the current namespace (see Section3.7). A pair of parentheses follows, which encloses any existing parameters and compound statement, which represents the function body.

named-func-def:
    global-modifiers$_{opt}$ `func` *ident* `(` *param-list*$_{opt}$ `)` *compound-stmt*
    global-modifiers$_{opt}$ `func` *ident* `(` *param-list* `,` `...` `)` *compound-stmt*
    `native` *global-modifiers*$_{opt}$ `func` *ident* `(` `)` `;`
param-list:
    ident
    param-list `,` *ident*

Modifiers are described in Sections 3.7.6 and 3.8.

The second form of the function definition, for which an ellipse (`...`), separated by it with a comma, follows a non-empty parameter list, defines a function with a variable number of arguments. If the function defined in this manner has n parameters in its parameter list, it is to be called with at least $n - 1$ arguments. A vector, which receives all further arguments, is created for the nth parameter.

The third form of the function definition, which is introduced by the `native` keyword, does not contain any parameter declarations[9] or any function body. Instead, its definition is closed with a semicolon.

A function that is defined as `native` is implemented in platform-dependent code. This is usually another programming language, such as C, C++ or Assembler.

[9] This does not mean that no arguments can be passed to a function defined as `native`. The parameters are not declared, since it is the task of the platform-dependent code to verify the number of arguments (and their types).

3.5.2 Expressions as Statements

Most statements in OFML consist of an expression (*expr*, which is evaluated in secondary context (see Section 3.6.1) and is closed with a semicolon or end of file.

expr-stmt:
    $expr_{opt}$ *eox*

If the expression is not present, the statement is an empty statement, which can be used in situations where the syntax requires a statement but no action is desired (for example in the body of an empty loop).

3.5.3 Control Statements

Control statements are used to control the course of a program dynamically and are divided roughly into three categories: selection statements (*select-stmt*), loop statements (*loop-stmt*), jump statements (*jump-stmt*) and exception statements *exception-stmt*). The latter are described in Section 3.5.3.

ctrl-stmt:
    select-stmt
    loop-stmt
    jump-stmt
    exception-stmt

Selection Statements

Selection statements select one or more program sequences.

select-stmt:
    `if ( ` *expr* `)` $stmt_1$
    `if ( ` *expr* `)` $stmt_1$ `else` $stmt_2$
    $label_{opt}$ `switch ( ` *expr* `) {` *switch-stmt-list* `}`

For both forms of the `if` statement, the *expr* expression is assessed in test context (see Section 3.6.1). If the expression yields *true*, the $stmt_1$ statement is executed. In the second form, $stmt_2$ is executed if the expression yields *false*. The syntactical ambiguity for `else` is resolved by always assigning an `else` to the last occurring `if` without `else` on the same block nesting level.

The $stmt_1$ and $stmt_2$ statements of the `if` statement cannot be definitions (*definition-stmt*).

The `switch` statement evaluates the `switch` expression *expr* in value context and branches, depending on the result, to a label (*switch-label*) within the subsequent statement list (`switch-stmt-list`),

which is enclosed in curly brackets. Optionally, it can include a label to which the `break`and `continue`statements can refer within the `switch` statement list (see Section 3.5.3).

switch-stmt-list:
    switch-stmt-list$_{opt}$ *switch-stmt*
switch-stmt:
    expr-stmt
    ctrl-stmt
    compound-stmt
    switch-label
switch-label:
    `case` *expr* `:`
    `default :`

Here, the expressions (*expr*) of the `case` labels are evaluated in the order in which they occur and compared for equality to the result of the `switch` expression, whereby the result of the `switch` expression appears on the left of the relational operator. If the comparison yields *true*, processing continues with the statement (*switch-stmt*) that directly follows the `case` label. Otherwise, processing continues at the next `case` label.

If all `case` labels have been processed without the occurrence of equality, processing continues with the statement following a `default` label if one is present. If none is present, no statement in the statement list is processed.

No more than one `default` label may occur within the statement list of a `switch` statement.

Exceptions that have been triggered by the `switch` expression, the `case` expressions or the `==` relational operator, which is applied to the results of both expressions, are not caught.

Loop Statements

Loop statements are used to repeat the execution of statements. Optionally, a loop statement can include a label to which, within the body of the loop, the `break`and `continue`statements can refer (see Section 3.5.3).

The *stmt* statement that forms the body of the loop cannot be a definition (*definition-stmt*).

labeled-loop-stmt:
    label$_{opt}$ *loop-stmt*
label:
    ident `:`
loop-stmt:
    `while (` *expr* `)` *stmt*
    `do` *stmt* `while (` *expr* `)`
    `for (` $expr1_{opt}$; $expr2_{opt}$; $expr3_{opt}$ `)` *stmt*
    `foreach (` *name* `;` *expr* `)` *stmt*

The *expr* expressions of the `while` or `do–while` statements and the second $expr_2$ expression of the `for` statement are evaluated in test context (see Section 3.6.1).

Using the `while` statement, the *stmt* statement is repeated until the *expr* expression yields *false*. The evaluation of the expression takes place **before** the first execution of the statement.

The `do–while` statement is similar to the `while` statement, except that the expression is evaluated **after** the execution of the *stmt* statement. In this case, the statement is executed at least once no matter what.

With the `for` statement, the first expression ($expr_1$) is evaluated first in secondary context. It is used (in general) to initialize the loop. The second expression ($expr_2$) is evaluated before each processing of the loop body. If it yields *false*, the `for` loop is terminated. Otherwise, the body of the loop is processed and then the third expression ($expr_3$), which (in general) is used to reinitialize the loop, is processed in secondary context.

All three expressions of the `for` statement may be omitted. If the second expression ($expr_2$) is not present, this is equivalent to the test result of *true*.

If the statement does not contain a `continue` statement, the `for` statement is equivalent to:

```
expr1;
while ( expr2 ) {
    stmt
    expr3;
}
```

The `foreach` statement is used for iterations through a sequence. In this case, the first expression must be a (if necessary, qualified) name. The result of the second expression processed in value context must meet the requirements for a sequence type (see Section 3.3.4). Otherwise, an exception is triggered (potentially after one or more iterations).

The implementation must behave as if creating a temporary *idx* variable that is assigned the `Int` value of −1 before the loop is processed and is increased by 1 before each pass of the loop. The second expression is evaluated once prior to processing the loop and its result is stored in the temporary *seq* variable. The loop is terminated if *idx*, after being increased by 1, is greater than or equal to the current length of the sequence stored in *seq*. Otherwise, the element of the sequence indexed by *idx* is assigned the value determined by the first expression. Then the body of the loop is completely processed.

If the `foreach` statement does not contain a `continue` statement, it is equivalent to[10]:

```
seq = expr;
for (idx = 0; idx < seq.size(); idx++) {
    name = seq[idx];
    stmt
}
```

[10]The identifiers are selected only for demonstration; in principle, OFML generates internal variables that cannot come into conflict with user-defined variables.

Jump Statements

Jump statements unconditionally continue processing of the program at another position.

jump-stmt:
  continue-stmt
  break-stmt
  return-stmt

The `continue` statement may occur only within a `while`, `do–while`, `for` or `switch` statement. For `while` and `do–while` loops, it continues program processing with the evaluation of the test expression, for the `for` loop, with the evaluation of the reinitialization expression, and for the `switch` statement, by restarting the entire `switch` statement.

continue-stmt:
  `continue ;`
  `continue` *ident* `;`

A `continue` statement without identifier passes control to the innermost of the statements listed above. If such a statement is does not exist, a translation error occurs.

A `continue` statement with an *ident* identifier passes control to the innermost of the statements listed above that has the same identifier as a label. If such a statement is does not exist, a translation error occurs.

The `break` statement may occur only within a `while`, `do–while`, `for` or `switch` statement.

break-stmt:
  `break ;`
  `break` *ident* `;`

A `break` statement without identifier continues program processing directly after the innermost of the statements listed above. If such a statement is does not exist, a translation error occurs.

A `break` statement with an *ident* identifier continues program processing directly after the innermost of the statements listed above that has the same identifier as a label. If such a statement is does not exist, a translation error occurs.

The identifiers specified for `continue` and `break` statements and used as labels before `while`, `do–while`, `for` and `switch` statements are located in a separate namespace, in which they can be applied with any frequency.

The `return` statement ends the execution of a function. If the statement contains an expression (optional), it is processed in value context (see Section 3.6.1) and its value is returned as a return value of the function. If there is no expression or if the end of the function is reached without an occurrence of the `return` statement, the function returns the `NULL` value.

A `return` statement outside of a function causes a translation error.

return-stmt:
    `return ;`
    `return` *expr* `;`

Exceptions

Exceptions can be triggered either by internal errors (such as errors while loading translation units or division by zero) or by the explicit execution by the programmer of the `throw` statement. They cause a nonlocal[11] pass of the program process from the position where the exception was triggered to the position at which it is caught. The latter is determined during program runtime.

exception-stmt:
    try-stmt
    throw-stmt

The `try` statement allows exceptions to be handled in a user-defined manner. Using several optional `catch` components, it is possible to handle various types of exceptions separately. Here *name* is the name of a type and *ident* is the name of a local variable that contains the value of the exception and that is valid only within the `catch` block.

try-stmt:
    `try` *compound-stmt catch-stmts*$_{opt}$
catch-stmts:
    catch-stmt catch-stmts$_{opt}$
catch-stmt:
    `catch (` *name ident* `)` *compound-stmt*

Only reference types are permitted as type names in the `catch` statement. Other types lead to a translation error. Using a `catch` statement, all of the exceptions that are instances of the class specified by the type names or one of the classes derived from this are caught.

If a `try` statement has several `catch` statements, the body of the first matching `catch`statement is executed even if a subsequent `catch` statement of the same `try` statement would yield a more exact match between the type of the `catch` parameter and the class of the exception.

A `try` statement without `catch` statement catches all exceptions. If no match between at least on type of the `catch` parameter and the class of the exception can be found in a `try` statement with at least one `catch` statement, the exception is not caught by this `try` statement.

The `throw` statement allows the programmer to trigger exceptions. Here, the value of the *expr* expression processed in value context (see Section 3.6.1) is passed as the value of the exception.

[11] Nonlocal pass means that the catching `try` statement can be located in a direct or indirect caller of the exception-triggering function.

throw-stmt:
    `throw` *expr eox*

The result of the expression of a `throw` statement must have a reference type. Otherwise, another exception is triggered. The `throw` statement passes the program processing on to the `try` statement that dynamically encloses it and that either contains one matching `catch` statement or none. If a `try` statement of this sort is not present, the exception is, dependent on implementation, handled by the runtime system, for example by outputting an error message and possibly terminated program execution.

Compound Statements

Compound statements are used to insert sequences of several statements at positions where, syntactically, only one statement is permitted, such as in the body of a loop.

compound-stmt:
    `{` *stmt-list* `}`

Compound statements make a new namespace available where variables defined within the compound statement can be entered. When binding an identifier to a variable, a search is carried out from inside to outside, one after the other, in the namespaces of the statically enclosing compound statements.

Variables with identical identifier cannot be defined more than once in a compound statement. Compound statements cannot contain any function or class definitions.

3.6 Expressions

The following section describes the operators of OFML, sorted by precedence. Precedence, associativity and evaluation order of operands are fixed conditions. Unless otherwise stated, operands are evaluated from left to right, while the evaluation of one operand with all side-effects must be completed before the evaluation of the next can be begun. This also applies to arguments of functions and methods. With few exceptions, which are explicitly mentioned, all operands of an operator are evaluated always.

The behavior of unary operators is oriented to the type of the result of the operand. For binary operators, it is oriented to the type of the result of the left operand. If the corresponding result has a predefined reference type, the exact behavior of each operator, if defined, is described in Section 3.4.

3.6.1 Value, Test and Secondary Context

Expressions and subexpressions are processed in three different contexts:

Wert–Kontext The expression must supply a value of one of the simple types or a reference type. If the result of the expression is a logical value, it is converted to the `Int` value of 1 if it is *true* and to the `Int` value of 0 if it is *false*.

Test–Kontext The expression must deliver a logical value, i.e. either *true* or *false*. If the result of the expression is a value of a reference type, the `operator!!()` operator function is called up for it and then takes into account its return value. If the result now is not an `Int` or `Float`, an exception is triggered. Otherwise, the result of the expression becomes *true*, if the `Int` or `Float` does not equal null and, otherwise, *false*.

Nebenwirkungs–Kontext The expression is evaluated to achieve a side effect. The result of the expression that presents both a logical value and a value of a simple type or a reference type is ignored.

If not otherwise stated, operators process their operands in value context.

3.6.2 Primary Expressions

Primary expressions are identifiers (see Sections3.7.2 and 3.7.5), literal constants (see Section3.2.5), special constructors for vectors (see Section3.4.4) and lists (see Section3.4.5) or bracketed expressions:

primary-expr:
    name
    constant
    special-ctor
    `(` *expr* `)`
special-ctor:
    `[` *arg-expr-list*$_{opt}$ `]`
    `@(` *arg-expr-list*$_{opt}$ `)`
arg-expr-list:
    assign-expr
    arg-expr-list `,` *assign-expr*

names (*name*) are described in Section 3.7.2, constants (*constant*) in Section 3.2.5 and special constructors (*special-ctor*) in Sections 3.4.4 and 3.4.5. The value of a bracketed expression is equal to the value of the *expr* expression within the brackets.

3.6.3 Postfix Expressions

Postfix expressions are left-associative.

postfix-expr:

primary-expr
postfix-expr `[` *expr* `]`
postfix-expr `[` $expr1_{opt}$ `:` $expr2_{opt}$ `]`
postfix-expr `(` $arg\text{-}expr\text{-}list_{opt}$ `)`
postfix-expr `.` *ident*
postfix-expr `++`
postfix-expr `--`

The operator for accessing "'`.`"' attributes is described in Section 3.8.

Index Expressions

The *postfix-expr* in the index expression, *postfix-expr* `[` *expr* `]`, must deliver a reference type. Otherwise, an exception is triggered.

For reference types, two operator methods can be defined, which are used to request and set an object in a sequence based on an index:

`operator[](`*idx*`)` is called for the result of the *postfix-expr* if the indexed value is to be read. The result of *expr* is passed to the *idx* parameter. The return value of the operator method is the value of the index expression.

`operator[](`*idx*`,` *value*`)` is called for the result of the *postfix-expr* if the indexed value is to be written[12]. The result of *expr* is then passed to the *idx* parameter and the value to be written is passed to the *value* parameter. Any return value is ignored.

Range Expressions

The *postfix-expr* in the range expression, *postfix-expr* `[` $expr1_{opt}$ `:` $expr2_{opt}$ `]`, must deliver a reference type. Otherwise, an exception is triggered.

If $expr_1$ is been specified, the `Int` value of 0 is passed to the range operator as the start of the range. Similarly, if $expr_2$ is not specified, the return value of the *size()* method, applied to the result of the *postfix-expr*, is passed to the range operator as the end of the range. An exception is triggered if the *size()* method does not exist.

For reference types, two operator methods can be defined, which are used to request and set a range of a sequence based on a start and end index:

`operator[:](`*beginend*`)` is called for the result of the *postfix-expr* if the range is to be read. The result of $expr_1$ is passed to the *begin* parameter and the result of $expr_2$ to the *end* parameter. The return value of the operator method is the value of the range expression.

`operator[:](`*beginendvalue*`)` is called for the result of the *postfix-expr* if the range is to be written. The result of $expr_1$ is passed to the *begin* parameter, the result of $expr_2$ to the *end* parameter and the write value to the *value* parameter. Any return value is ignored.

[12] This is the case, for example, if the index operator is used on the left side of the assignment operator. The result of the expression on the right side of the assignment operator is then passed as *value* on to the index operator.

Function Calls

For function calls, the first expression (*postfix-expr*) has to deliver an object of a reference type that implements the function call operator (`operator()`), such as the predefined function types, `Func` and `CFunc`. An object of this sort is referred to as a function in the following.

For function calls, the two following cases can be distinguished in regard to the called function:

- The called function is a common function or a class-oriented (static) method.
- The called function is an instance-oriented method.

The difference when calling an instance-oriented method compared to calling a class-oriented (static) method or a common function is that the object for which the method is called is implicitly passed to an instance-oriented method as a `self` parameter.

If the function to be called is an instance-oriented method and the expression delivering the method is in the form of *postfix-expr*$_2$ `.` *ident*, the result of *postfix-expr*$_2$ is passed as a `self` parameter. Otherwise, the caller must be an instance-oriented method and the `self` of the calling method is passed as the `self` parameter of the instance-oriented method being called.

An exception is triggered if no object can be passed as `self` for an instance-oriented method[13] or if the class of the object passed as `self` is not equal to the class or or one of its derived classes for which the called instance-oriented method was defined.

The passing of arguments is analogous to the assignment as value for simple types (*call by value*) and as reference for reference types (*call by reference*). Exactly the same number of arguments as specified in the function definition must be passed unless the function is defined as a function with a variable number of arguments. In this case, the number of passed arguments may be no more than one less that the number of declared parameters. All further arguments are assigned to the last parameter in the form of a vector.

The return value of a function call is the value that was passed in the called function to the `return` statement or `NULL` (see Section 3.5.3).

The function call operator can be defined for classes as follows:

`operator()`(*parameters*) is called for the instance of a class if the instance is the result of the *postfix-expr*. The arguments of the function call are passed in the manner described above to the parameters of the function call operator declared in the place of *parameters*. The return value of the function call operator method is the result of the function call.

Postfix Incrementation and Decrementation

The operand of a postfix increment or decrement operator must be a variable, an index expression or a range expression. Otherwise, a translation error occurs.

The postfix increment and decrement operators, `++` and `--`, behave as follows depending on the type of the operand:

[13] This is the case if the calling function is a common function or a class-oriented (static) method.

If the value of the operand is a simple type, it has to be an `Int` or `Float`. Otherwise, an exception is triggered. Then, the following equivalence applies to the processing of the operator:

$$expr\oplus\oplus \equiv \texttt{(}tmp \texttt{ = } expr\texttt{, } expr \texttt{ = } tmp \oplus \texttt{ 1, } tmp\texttt{)},$$

where *tmp* is an unnamed variable created dynamically for the length of processing this subexpression and subexpressions of the *expr* expression are only processed once. The addition or subtraction follows the rules listed in Section 3.3.3.

If the value of the operand is a reference type, the `operator++(`*value*`)` or `operator--(`*value*`)` operator method is called for the postfix increment or decrement operator for this reference type. `NULL` is passed to the *dummy* parameter. It is used only to distinguish from the corresponding prefix increment or decrement operator. The return value of the operator method is the result of the operator.

3.6.4 Unary Operators

Unary expressions are right-associative.

unary-expr:
  postfix-expr
  unary-op unary-expr
unary-op:
  `+` | `-` | `++` | `--` | `~` | `!` | `!!` | `$`

Unary Plus and Minus Operator

The unary plus and minus operators, `+` and `-`, behave as follows depending on the type of the operand:

If the value of the operand is a simple type, it has to be an `Int` or `Float`. Otherwise, an exception is triggered. In the case of the plus operator, the value of the operand is equal to the result of the operator. In the case of the minus operator (arithmetical negation operator), the result of the operator is equal to the value of the operand multiplied by -1[14].

If the value of the operand is a reference type, the `operator+(`*value*`)` or `operator-(`*value*`)` operator method is called for the unary plus or minus operator for this reference type. The value of the operand is passed as a *value* parameter to this method, the return value of which is the result of the operator.

[14] Due to the use of the complement of two for representing integer numbers, the arithmetical negation of the greatest-valued representable negative value is equal to this value.

Prefix Incrementation and Decrementation

The operand of a prefix increment or decrement operator must be a variable, an index expression or a range expression. Otherwise, a translation error occurs.

The prefix increment and decrement operators, `++` and `--`, behave as follows depending on the type of the operand:

If the value of the operand is a simple type, it has to be an `Int` or `Float`. Otherwise, an exception is triggered. Then, the following equivalence applies to the processing of the operator:

$\oplus\oplus$*expr* $\equiv$ `(`*expr* `=` *tmp* `=` *expr* $\oplus$ `1,` *tmp*`)`,

where *tmp* is an unnamed variable created dynamically for the length of processing this subexpression and subexpressions of the *expr* expression are only processed once. The addition or subtraction follows the rules listed in Section 3.3.3.

If the value of the operand is a reference type, the)`operator++()` or)`operator--()` operator method is called for the prefix increment or decrement operator for this reference type. Any return value from these methods is ignored. The value of the operand is equal to the result of the operator.

Bitwise Negation

The bitwise negation operator, `~`, behaves as follows depending on the type of the operand:

If the value of the operand is a simple type, it has to be an `Int` . Otherwise, an exception is triggered. The result of the operator is then equal to the bitwise negation of the value of the operand.

If the value of the operand is a reference type, the `operator~()` operator method is called for bitwise negation for this reference type. The return value of this method is equal to the result of the operator.

Logical Negation

The operand of the logical negation operator, `!`, is evaluated in test context. Its result is *true* if the value of the operand is *false* and *false* if the value of the operand is *true*.

The Test Operator

The operand of the test operator, `!!`, is evaluated in test context. Its result is identical to the value of the operand.

The Symbol Resolution Operator

The symbol resolution operator, `$`, requires an argument of the `Symbol` type. Otherwise, an exception is triggered. It cannot be redefined for reference types.

The symbol resolution operator dynamically binds the symbol that its operands deliver to a variable. This takes place according to the rules for binding identifiers, which are specified in Section 3.7.5.

3.6.5 Multiplicative Operators

Multiplicative expressions are left-associative.

mul-expr:
unary-expr
mul-expr mul-op unary-expr
mul-op:
`*` | `/` | `%`

The multiplicative operators, `*`, `/` and `%`, behave as follows depending on the type of the left operand:

If the value of the left operand is a simple type, it has to be an `Int` or `Float`. The value of the right operand, then, must also be an `Int` or `Float`. If either of these conditions are violated, an exception is triggered. Otherwise, the operation takes place in `Int` if both operands are of the `Int` type, or in `Float` if at least one of the operands are of the `Float` type. In the second case, an operand of the `Int` type is converted to `Float` before the operation.

The `*` operator multiplies the two operands.

The `/` operator divides the two operands, where the left operand is the dividend and the right operand, the divisor. If both operands are integers, the result is rounded towards 0.

The `%` operator determines the remainder of an implicit division for which the left operand is the dividend and the right, the divisor.

If the remainder operation is carried out in `Int`, $(a/b)*b+(a\%b) = a$ applies to the value calculated by the remainder operator. It therefore follows, that the +/- sign of the remainder is the same as the +/- sign of the dividends. Furthermore, the value of the remainder is always less than the value of the divisor.

If the remainder operation is executed in `Float`, the result is the value of $a-i*b$, where the integer value of i is selected so that the result carries the same +/- sign as a and the value of the result is less than the value of b.

The calculations are carried out according to the rules listed in Section 3.3.3.

If the value of the left operand has a reference type, one of the following operator methods is called for the left operand:

`operator*(`*rhs*`)` (multiplication),

`operator/(`*rhs*`)` (division)

`operator%(`*rhs*`)` (remainder)

Here, the value of the right operand is passed as an *rhs* parameter. The return value of the operator method is the result of the operator.

3.6.6 Additive Operators

Additive expressions are left-associative.

add-expr:
  mul-expr
  add-expr add-op mul-expr
add-op:
  `+` | `-`

The additive operators, `+` and `-`, behave as follows depending on the type of the left operand:

If the value of the left operand is a simple type, it has to be an `Int` or `Float`. The value of the right operand, then, must also be an `Int` or `Float`. If either of these conditions are violated, an exception is triggered. Otherwise, the operation takes place in `Int` if both operands are of the `Int` type, or in `Float` if at least one of the operands are of the `Float` type. In the second case, an operand of the `Int` type is converted to `Float` before the operation.

The `+` operator adds the two operands.

The `-` operator subtracts divides the two operands, where the left operand is the minuend and the right operand, the subtrahend.

If the value of the left operand has a reference type, one of the following operator methods is called for the left operand:

`operator+(`*rhs*`)` (addition)

`operator-(`*rhs*`)` (subtraction)

Here, the value of the right operand is passed as an *rhs* parameter. The return value of the operator method is the result of the operator.

3.6.7 Bitwise Shifts

Expressions for bitwise shifts are left-associative.

shift-expr:
  add-expr
  shift-expr shift-op add-expr
shift-op:
  `<<` | `>>` | `>>>`

The shift operators, `<<` (left shift), `>>` (signed right shift) and `>>>` (unsigned right shift) behave as follows depending on the type of the left operand:

If the value of the operand is a simple type, both operands must be of the `Int` type and the right operand must be nonnegative. If these conditions are not met, an exception is triggered.

Otherwise, the left operand is interpreted as a bit sequence, which is shifted by the number of positions specified by the right operand either left (`<<`) or right (`>>` and `>>>`). The `<<` and `>>>` operators fill vacated positions with 0-bits, while the `>>` operator fills vacated positions with the value of the highest bit before the operation.

If the value of the left operand has a reference type, one of the following operator methods is called for the left operand:

> `operator<<(`*rhs*`)` (left shift),
>
> `operator>>(`*rhs*`)` (signed right shift)
>
> `operator>>>(`*rhs*`)` (unsigned right shift)

Here, the value of the right operand is passed as an *rhs* parameter. The return value of the operator method is the result of the operator.

3.6.8 Relational Operators

Relational expressions are left-associative[15].

comp-expr:
  shift-expr
  comp-expr comp-op shift-expr
  comp-expr `instanceof` *shift-expr*
comp-op:
  `<` | `>` | `<=` | `>=`

The relational operators, `<` (less than), `<=` (less than or equal to), `>=` (greater than or equal to) and `>` (greater than), behave as follows depending on the type of the left operand:

If the value of the left operand is a simple type, it has to be an `Int`, `Float` or `Symbol`. The value of the right operand must be an `Int` or `Float` if the left operand is an `Int` or `Float`, or it must be a `Symbol` if the left operand is a `Symbol`. If any of these conditions is violated, an exception is triggered.

If none of the operands is `Symbol`, the relational operation takes place in `Int` if both operands are of the `Int` type, or in `Float` if at least one of the operands are of the `Float` type. In the second case, an operand of the `Int` type is converted to `Float` before the operation.

If the operands are of the `Symbol` type, a comparison of the internal representation of both symbols takes place. The result of this comparison can only be guaranteed reproducible in an instance of an OFML program.

The `<` operator yields *true* if the value of the left operand is less than the value of the right operand.

The `<=` operator yields *true* if the value of the left operand is less than or equal to the value of the right operand.

[15] Note that consecutively written relational expressions do not follow common mathematical syntax: `0 < x < 5` is interpreted as `(0 < x) < 5` and always returns the value of 1.

The `>=` operator yields *true* if the value of the left operand is greater than or equal to the value of the right operand.

The `>` operator yields *true* if the value of the left operand is greater than the value of the right operand.

If the relational operator does not yield *true*, it yields *false*.

If the value of the left operand has a reference type, one of the following operator methods is called for the left operand:

> `operator<(`*rhs*`)` (less than),
>
> `operator<=(`*rhs*`)` (less than or equal to),
>
> `operator>=(`*rhs*`)` (greater than or equal to),
>
> `operator>(`*rhs*`)` (greater than)

Here, the value of the right operand is passed as an *rhs* parameter. The return value of the operator method is interpreted in test context and converted to a logical value as described in Section 3.6.1[16].

Type Verification

As the value of the right expressions, the `instanceof` operator expects a type derived from the `Type` type (see Section 3.4.1). Otherwise, an exception is triggered. The result of the `instanceof` operator is *true* if

- the left expression returns a value of a simple type whose type is identical to the type returned by the right expression, or
- the left expression returns a value of a reference type which is either identical to the type returned by the right expression or has been derived from this.

Otherwise, the result of the operator is *false*.

3.6.9 Equality Comparisons

Equality comparisons are left-associative.

equiv-expr:
  comp-expr
  equiv-expr equiv-op comp-expr
equiv-op:
  `==` | `!=` | `~=`

[16]Subsequently, the return value should though is not required to be an `Int` value.

If the value of the right operand of the relational operators, `==` (equality) and `!=` (inequality), is `NULL`, both operands are considered equal if the value of the left operand is also `NULL`. The `==` operator returns *true* in case of equality, otherwise it returns *false*. The `!=` operator returns *false* in case of equality, otherwise *true*.

Otherwise, the relational operators, `==`, `!=` and `˜=` (pattern match), behave as follows depending on the type of the left operand:

If the value of the left operand has a reference type, one of the following operator methods is called:

> `operator==(`*rhs*`)` (equality),
>
> `operator!=(`*rhs*`)` (inequality),
>
> `operator =(`*rhs*`)` (pattern match)

Here, the value of the right operand is passed as an *rhs* parameter. The return value of the operator method is interpreted in test context and converted to a logical value as described in Section 3.6.1.

If the value of the left operand has a simple type, an exception is triggered in the case of the `˜=` operator. In the case of the `!=` operator, the result is the logical negation of the result of the `==` operator, applied to the same operands. The `==` operator behaves as follows:

If the value of the left operand is `NULL`, the result is *true* if the value of the right operand is also `NULL`, or *false* if the value of the right operand is not `NULL`.

If the value of the left operand is of the `Symbol` type, the value of the right operand also has to be of the `Symbol` type. Otherwise, an exception is triggered. The result is *true* if both symbols embody the same string, otherwise it is, *false*.

If the value of the left operand is of the `Int` or `Float` type, the value of the right operand also has to be of the `Int` or `Float` type. Otherwise, an exception is triggered. The comparison takes place in `Int` if both operands are of the `Int` type, otherwise it takes place in `Float`. In the second case, any `Int` type operand is converted to `Float`. The result is *true* if both operands (also after conversion, if one takes place) have the identical value, otherwise it is *false*.

3.6.10 Minimum and Maximum

Minimum and maximum operators are left-associative.

minmax-expr:
    equiv-expr
    minmax-expr minmax-op equiv-expr
minmax-op:
    `<?` | `>?`

For the `<?` (minimum) and `>?` (maximum) operators, the following equivalencies apply for processing the minimum and maximum operators:

$$a \texttt{ <? } b \equiv \texttt{(}tmp_1 \texttt{ = } a\texttt{, } tmp_2 \texttt{ = } b\texttt{, } tmp_1 \texttt{ < } tmp_2 \texttt{ ? } tmp_1 \texttt{ : } tmp_2\texttt{)}$$

$$a \texttt{ >? } b \equiv \texttt{(}tmp_1 \texttt{ = } a\texttt{, } tmp_2 \texttt{ = } b\texttt{, } tmp_1 \texttt{ > } tmp_2 \texttt{ ? } tmp_1 \texttt{ : } tmp_2\texttt{)}$$

Here, tmp_1 and tmp_2 are unnamed variables created dynamically for the length of processing this subexpression.

3.6.11 Bitwise Links

Expressions for bitwise links are left-associative.

bit-and-expr:
  minmax-expr
  bit-and-expr `&` *minmax-expr*
bit-xor-expr:
  bit-and-expr
  bit-xor-expr `^` *bit-and-expr*
bit-or-expr:
  bit-xor-expr
  bit-or-expr `|` *bit-xor-expr*

The bitwise link operators, `&` (bitwise AND), `^` (bitwise exclusive OR) and `|` (bitwise OR), behave as follows depending on the type of the left operand:

If the value of the operand is a simple type, both operands must be of the `Int` type. Otherwise, an exception is triggered. The result is of the `Int` type.

If the value of the left operand has a reference type, one of the following operator methods is called for the left operand:

`operator&(`*rhs*`)` (bitwise AND),

`operator^(`*rhs*`)` (bitwise exclusive OR),

`operator|(`*rhs*`)` (bitwise OR)

Here, the value of the right operand is passed as an *rhs* parameter. The return value of the operator method is the result of the operator.

3.6.12 Logical Links

Expressions for logical links are left-associative.

logic-and-expr:
  bit-or-expr
  logic-and-expr `&&` *bit-or-expr*
logic-or-expr:
  logic-and-expr
  logic-or-expr `||` *logic-and-expr*

The `&&` (logical AND) and `||` (logical OR) operators evaluate their left operands in test context. If the value for the `&&` operator is *false*, the right operand is not evaluated and the result of the operator is *false*. Accordingly, the right operand is not evaluated and the result of the operator is *true* for the `||`, operator if the left operand yields *true*. Otherwise, both operators evaluate their right operands in test context and their result is equal to the value of the right operand.

Principally, the right operand is not evaluated if the result of the operator is determined by the result of the left operand.

3.6.13 Conditional Expression

Conditional expressions are right-associative.

cond-expr:
    logic-or-expr
    logic-or-expr ? *expr* : *cond-expr*

For the conditional expression, the first operand (*logic-or-expr*) is evaluated in test context. If the evaluation yields *true*, the second operand (*expr*) is evaluated and the result of the conditional expression is equal to the value of the second operand. If the evaluation of the first operand yields *false*, the third operand (*cond-expr*) is evaluated and the result of the conditional expression is equal to the value of the third operand.

Either the second or the third operand is evaluated, never both.

3.6.14 Assignment Operators

All assignment operators are right-associative.

assign-expr:
    cond-expr
    unary-expr assign-op assign-expr
assign-op:
    `=` | `+=` | `-=` | `*=` | `/=` | `%=` | `<<=` | `>>=` | `&=` | `^=` | `|=`

The left operand of an assignment must be a variable, an index expression or a range expression. Otherwise, a translation error occurs.

If the left operand is a variable, the value of the right operand is calculated by the `=` assignment operator and the variable is assigned. This value is the result of the assignment operator.

If the left operand is an index or range expression, the value of the right operand is calculated first by the assignment operator. Then, the subexpressions (sequence, index or indices) of the left operand are calculated and the index or range operator is called to set a value to the value of right operand as an argument. The value of the right operand is the result of the assignment operator.

The combined assignment operators, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=` and `|=` first calculate the value of the right operand. Then, the value of the left operand is calculated. Depending on its type, processing continues as follows:

If the value of the left operand has a simple type, the following equivalence applies to the processing of the combined assignment operator:

$$lhs \oplus\texttt{=}\ rhs \equiv \texttt{(}tmp_1\ \texttt{=}\ rhs\texttt{,}\ lhs\ \texttt{=}\ tmp_2\ \texttt{=}\ lhs \oplus tmp_1\texttt{,}\ tmp_2\texttt{)}$$

Here, tmp_1 and tmp_2 are unnamed variables created dynamically for the length of processing this subexpression. Subexpressions of the left operand (a) are calculated only once.

If the value of the left operand has a reference type, one of the following operator methods is called for the left operand:

`operator*=(`*rhs*`)` (`*=` operator),

`operator/=(`*rhs*`)` (`/=` operator),

`operator%=(`*rhs*`)` (\%= operator),

`operator+=(`*rhs*`)` (`+=` operator),

`operator-=(`*rhs*`)` (`-=` operator),

`operator<<=(`*rhs*`)` (`<<=` operator),

`operator>>=(`*rhs*`)` (`>>=` operator),

`operator>>>=(`*rhs*`)` (`>>>=` operator),

`operator&=(`*rhs*`)` (`&=` operator),

`operator^=(`*rhs*`)` (`^=` operator),

`operator|=(`*rhs*`)` (`|=` operator)

Here, the value of the right operand is passed as an *rhs* parameter. The return value of the operator method is the result of the combined assignment operator.

3.6.15 The Comma Operator

The comma operator is left-associative.

expr:
  assign-expr
  expr **,** *assign-expr*

The left operand is evaluated in secondary context. Then, the right operand is evaluated. Its value is the result of the comma operator.[17]

Table 3.1 summarizes once more the precedence and associativity for all operators. Here, the lowest number represents highest precedence.

3.7 Packages and Namespaces

3.7.1 Module

Every translation unit forms a module. A module belongs to a package, which is optionally specified at the beginning of the module with the `package` statement (see Section 3.7.3).

[17] Note that, based on the grammar defined here, comma expressions in contexts in which the comma has another syntactic meaning (such as in argument lists of function calls) must be placed within brackets in order to achieve the targeted effect.

<table>
<tr><th>Operators</th><th>Precedence</th><th>Associativity</th></tr>
<tr><td><code>::</code></td><td>1</td><td>left</td></tr>
<tr><td><code>() @() [] .</code></td><td>2</td><td>left</td></tr>
<tr><td><code>! !! ~ ++ -- + - $</code></td><td>3</td><td>right</td></tr>
<tr><td><code>* / %</code></td><td>4</td><td>left</td></tr>
<tr><td><code>+ -</code></td><td>5</td><td>left</td></tr>
<tr><td><code><< >> >>></code></td><td>6</td><td>left</td></tr>
<tr><td><code>< <= > >= instanceof</code></td><td>7</td><td>left</td></tr>
<tr><td><code>== != ~=</code></td><td>8</td><td>left</td></tr>
<tr><td><code>>? <?</code></td><td>9</td><td>left</td></tr>
<tr><td><code>&</code></td><td>10</td><td>left</td></tr>
<tr><td><code>^</code></td><td>11</td><td>left</td></tr>
<tr><td><code>|</code></td><td>12</td><td>left</td></tr>
<tr><td><code>&&</code></td><td>13</td><td>left</td></tr>
<tr><td><code>||</code></td><td>14</td><td>left</td></tr>
<tr><td><code>?:</code></td><td>15</td><td>right</td></tr>
<tr><td><code>= *= /= %= += -= <<= >>= >>>= &= ^= |=</code></td><td>16</td><td>right</td></tr>
<tr><td><code>,</code></td><td>17</td><td>left</td></tr>
</table>

Table 3.1: Operators

A module forms a namespace, which implicitly contains all names defined within its package as `public` or private to the package (i.e. without `public` or `private`). In addition to these, other names from other packages can be implicitly or explicitly imported (see Section 3.7.4), and new names can be defined within a module.

Qualified access to names in the namespace is not possible.

If an attempt is made to explicitly reimport or redefine an explicitly imported name within a module, a translation error occurs. Implicitly imported names can be imported implicitly more than once and imported explicitly or defined no more than once.

An explicitly imported or defined name obscures all implicitly imported names of the same name.

If multiple identical names are implicitly imported from different packages, these implicitly Import names are no longer visible.

A module is considered *loaded* if it has been translated to the point where all the definitions on the module and class levels in it have been processed and the corresponding names can be referenced by the compiler while translating other modules. Compound statements do not yet have to have been translated.

3.7.2 Packages and Namespaces

The following namespaces exist In OFML: package (see Section 3.7.3), module (see Section 3.7.1), class (see Section 3.8) and compound statement (see Section 3.5.3).

The namespaces of packages and classes from a hierarchy, where the individual components of a name are separated by double colons `::`. If such a name begins with `::`, the search begins in the root package, otherwise it begins in the package to which the translated module belongs. If the `::` operator is located in the middle of a name, the identifier specified to the right of it is searched for in the package or class specified to its left. Searching for names if `::` is not present is subject to other rules (see Section 3.7.5). The syntax for names is:

name:
  ident
  qualified-name
  fully-qualified-name
fully-qualified-name:
  `::` *ident*
  `::` *qualified-name*
qualified-name:
  name-qualifier `::` *ident*
name-qualifier:
  ident
  name-qualifier `::` *ident*

If using a not fully qualified name, the first component must be a class that has been define in the package to which the translated module belongs or a direct subpackage of this package.

For (qualified) access to a name that has not yet been loaded, an attempt is made to load it into the corresponding package. For unqualified names, this is the package to which the accessing module belongs. For qualified names, it is the package that is specified by the qualifier. For loading, the name is mapped through an implementation-dependent mechanism to the name of a module, which is then loaded.

3.7.3 Package Statement

The syntax of a package statement is:

package-stmt:
  `package` *fully-qualified-name* `;`

The module is translated in the specified package. If a package statement is not present, the module is translated in the root package (or default package).

During the translation of the package statement, all names of the package defined within the specified package as `public` or private to the package (i.e. without `public` or `private`) are imported implicitly into the translated module. If a package statement is not present, the names of the root package are imported implicitly.

3.7.4 Import Statement

The syntax of the import statement is:

import-stmt:
  `import` *fully-qualified-name* `;`
  `import :: * ;`
  `import` *fully-qualified-name* `:: * ;`

The first form of the import statement checks whether the specified name has already been defined. If not, the module that defines the name is determined through an implementation-dependent mechanism. This module is loaded. Then, the name in the namespace of the importing module is carried over. It is an error to import names whose last components are identical or are defined within the importing module using this form of the import statement.

The second and third forms of the import statement first check that all modules of the specified package have been loaded. Then, all of the names of this package defined as `public` are copied into the importing module.

Importing a name that belongs to the same package as the importing module is not possible.

Note that the import statement imports the names into the namespace of the module and not into that of the package to which the module belongs. This is necessary to prevent modules from influencing each other reciprocally via import statements.

3.7.5 Static and Dynamic Bindings

In principle, names (both simple and (fully) qualified) are bound statically.

The point operator (see Sections 3.6 and 3.8.3) can be understood as a binary operator that anticipates an object on the left and an identifier on the right. The identifier is bound to the corresponding attribute of the object dynamically during runtime.

The definition of OFML allows the implementation to generate attributes dynamically during runtime. To check for undefined names, such attributes must be accessed via `self` (see Section3.8.3).

Binding a simple (unqualified) name takes place in the following order:

1. Within functions and methods, the name is searched for in the namespace of the innermost compound statement. If the name is not found there, it is searched for from the inside outwards until the namespace that continues the compound statement that forms the function body is reached.

   If the name cannot be found within a method (either instance-oriented or class-oriented), the search continues in the namespace of the class to which the method belongs.

   If the name cannot be found within a (common) function, the search continues within the module in which the function was defined.

2. Within classes, the name is searched for in the namespace of the class. This contains all of the names inherited from super-classes. If an instance-oriented method or variable from a class-oriented method or a class-oriented initializer is found, a translation error is triggered.

   If the name is not found within the namespace of the class, the search continues in the module in which the class was defined.

3. The search in the module takes place only in the namespace of the module. This contains all imported names[18].

3.7.6 Visibility and Accessibility

A simple name is visible if it can be bound according to the rules described in Section 3.7.5.

A qualified name is visible if it has not been defined as `private` or if a simple name consisting only of the last component of the qualified name refers to the same definition and is visible.

A name to the right of the point operator is visible if it exists in the namespace of the class of the object on the left side of the point operator and either has not been defined as `private` or access takes place from within the same class.

The visibility of a simple name can be limited if it is covered by the same name in another namespace, where the search is more likely to take place, as is described in Section 3.7.5.

A name is accessible if it is visible and access allows it. For unqualified access, every visible name is also accessible. For qualified access or for access by means of the point operator, a visible name may be inaccessible under some circumstances.

Accessibility and visibility are controlled by modifiers at the start of a definition. This section describes only the modifiers for the definition of variables, functions and classes on the level of modules. Modifiers for class attributes are described in Section 3.8.

global-modifiers:
    global-modifiers$_{opt}$ *global-modifiers*
global-modifier:
    `final` | `public` | `private`

The effect of the `final` keyword on variables is that no new values can be assigned to them after the initialization requested within the variable definition. With OFML, however, variables as well as functions and classes on the level of modules can be redefined by different modules or by

[18] To conserve memory, the implementation does not by requirement have to adopt every single name at the time of translation. It is necessary, however, when removing a name, to search through all imported modules for the name to exclude the possibility of ambiguity. For names for which this has already been done, an entry can be made in the symbol table of the module to minimize the amount of processing necessary the next time the same name is used.

The same applies to names that belong to the package of the module but which were not defined by the module. Names that were defined by the module are located in the module's symbol table anyway.

The search order for access to an unqualified name is as then as follows: 1. symbol table of the module, 2. symbol table of the package, 3. all imported modules.

retranslating the (possibly modified) same module, in which case the value of the variables defined with `final` can also change.

Classes defined as `final` cannot be used as super-classes of other classes.

For function definitions, `final` cannot be applied.

Variables in which functions and classes are stored are defined implicitly as `final`.

Variables, functions and classes defined as `public` are accessible in all modules, even in those of other packages.

Variables, functions and classes defined as `private` are visible and accessible only within their defining modules.

If a variable, function or class is defined as neither `public` nor `private`, it is handled as private to the package. Names defined in this manner are generally visible, but accessible only from modules belonging to the same package[19].

3.8 Classes

3.8.1 Class Definitions

Class definitions define new reference types and describe their implementation.

class-def:
        global-modifiers$_{opt}$ `class` *ident super-class$_{opt}$ class-body*

3.8.2 Super-classes

super-class:
        : *ident*

Optionally, the name of a super-class can be specified in a class definition. The specified super-class cannot be defined as `final`. If a super-class is not specified, the class automatically inherits from the `Object` root class (see Section 3.3.2).

If the super-class is defined within the same translation unit, its definition must appear before the definition of the derived class. Furthermore, the constraint on using super-classes, which is described in Section 3.1.2, must be observed.

A class inherits all attributes not defined as `private` from its super-class. These are placed in the namespace of the subclass and are thus accessible in the subclass. Attributes of the super-class defined as `private` are not visible in the subclass.

[19] Even though these names are visible to import statements for importing all the names of a package (in asterisk form), these import statements do not try to access them (import them).

3.8.3 Attribute

The body of a class definition consists of a sequence of attribute definitions and class-oriented initializers.

class-body:
    `{` *member-def-stmts*$_{opt}$ `}`
member-def-stmts:
    member-def-stmts$_{opt}$ *member-def-stmt*
member-def-stmt:
    field-def
    method-def
    static-initializer

3.8.4 Data Fields

field-def:
    field-modifiers$_{opt}$ `var` *init-expr-list* `;`
field-modifiers:
    field-modifiers$_{opt}$ *field-modifiers*
field-modifier:
    `public` | `protected` | `private` | `final` | `static`

The syntax for defining data fields is the same as the syntax for defining variables (see Section 3.5.1) except that the `protected` and `static` modifiers are additionally allowed.

The initialization of class-oriented data fields takes place immediately after the module that contains the class definition is loaded. The initialization of instance-oriented data fields, including the evaluation of the initialization expression, takes place in the order in which they occur immediately after a new instance is created and before any `initialize()` method is called.

Data fields defined as `static` describe class-oriented variables that can exist only once per class. Otherwise, they are instance-oriented variables, which are created new for every instance of the class.

Data fields defined as `final` cannot be assigned new values after the initialization requested within the variable definition.

Data fields defined as `public` are accessible from all modules, even those from other packages.

Data fields defined as `protected` are generally visible, but are accessible only from methods, class-oriented initializers and initialization expressions from data fields of the same class or from classes derived from this class.

Data fields defined as `private` are visible and accessible only in methods, class-oriented initializers and initialization expressions from data fields of the same class or from classes derived from this class.

Data field defined as private to the package (i.e. without one of the keywords, `public`, `protected` or `private`), are generally visible, but are accessible only from methods, class-oriented initializers and initialization expressions from data fields of the same class or from classes derived from this class as well as from all modules belonging to the same package.

Methods

method-def:
    method-modifiers$_{opt}$ `func` *method-name* `(` *param-list*$_{opt}$ `)` *compound-stmt*
    method-modifiers$_{opt}$ `func` *method-name* `(` *param-list* `, ...` `)` *compound-stmt*
    `native` *method-modifiers*$_{opt}$ `func` *method-name* `( ) ;`
method-modifiers:
    method-modifier method-modifiers$_{opt}$
method-modifier:
    `public | protected | private | final | static`
method-name:
    ident
    `operator` *method-operator*
method-operator:
    `++ | -- | !! | ~ | * | / | % | + | - | << | >> | >>>`
    `< | <= | >= | > | == | != | ~= | & | ^ | |`
    `*= | /= | %= | += | -= | <<= | >>= | >>>= | &= | ^= | |=`

The syntax for defining methods is the same as the syntax for defining named functions (see Section 3.5.1) except that modifiers and special names for redefining operators are additionally allowed.

Methods defined as `static` describe class-oriented methods that cannot access instance-oriented variables. Class-oriented methods are called in conjunction with the type (see below). Otherwise, it is an instance-oriented method, which is called in reference to a specific object that is an instance of the class or an instance of a class derived from this class.

Methods defined as `final` cannot be redefined in subclasses.

The modifiers for controlling access have the same meaning as for data fields (see above).

Redefinition of Operators

Most of the operators supported by the OFML grammar can be redefined for reference types. To do so, instance-oriented methods having names combined from the `operator` keyword, followed by the operator being redefined, must be defined in the corresponding class definition. The number of parameters to be declared for operator methods is defined in Section 3.6.

If an operator method is defined as `static` (class-oriented) or defined with an unallowed number of parameters, a translation error occurs.

Constructors

Constructors are never defined explicitly, but instead are always created automatically. User-defined operations for initializing instances can take place within the special `initialize()` instance method, which is passed to the arguments that are passed to the constructor. If an `initialize()` method is defined, it is called in the constructor automatically. If an `initialize()` method is not defined, no arguments may be passed to the constructor. `initialize()` methods from super-classes are **not** called automatically. Such calls must take place explicitly in the `initialize()` methods of their corresponding subclasses!

Class-oriented Initializers

static-initializer:
    `static` *compound-stmt*

Class-oriented initializers consist of the `static` keyword, followed by a compound statement. Class-oriented initializers are processed in the order in which they occur within the module together with other executable statements on the module level and initializations of class-oriented variables after the module that contains the class definition is loaded.

The following example illustrates this concept:

```
public class MyInt {
  private var value = 0;          // instance variable
  private static var num;         // class variable
  public func initialize() {      // initialization method
    numInts++;
  }
  public func getValue() {        // normal instance method
    return (value);
  }
  public func incr(i) {
    value += i;
  }
  public static func getNum() {  // class method
    return (num);
  }
  static {                        // class-oriented initializer
    num = 0;
  }
}
```

Access to Attributes

Within instance methods of the same class, access to attributes is in general direct, meaning it takes place within the current namespace. Alternatively, instance variables and methods can be

accessed using the special `self` keyword and of the "'.'' access operator. If instance variables are created dynamically by the implementation, as is the case with child objects in *OFML*, `self` must be used to access them. Accordingly, in the example above, `return (self.value);` could have been written instead of `return (value);`.

Access to instance methods and variables takes place via the "'.'' operator. In this case, the left operand is any expression, the type of which must be shared by the attribute, and the right operand is the name of the attribute, e.g. `i.getValue()`.

Access to class methods and variables takes place via the "'::'' operator according to the rules for qualified names (see Section3.7.2), where the class name is used as a qualifier, e.g. `MyInt::getNum()`.

3.9 Predefined Functions

3.9.1 Standard Functions

Standard functions are defined in the `::cobra::lang` package.

typeOf(pObject(Object)) $\rightarrow$ *Type*

The *typeOf*() function anticipates any value of a simple type or reference type as an argument and returns the type of the argument.

3.9.2 Numerical Standard Functions

All predefined numerical standard functions are defined in the `::cobra::math` package.

Error Handling

Range Errors: If an argument of a numerical standard function is outside the definition range of the function, an exception is triggered.

Overflow and Underflow Errors: An overflow or underflow error occurs if the result of a function cannot be represented as `Float`. If an overflow occurs (meaning the amount of the results is so great that it cannot be represented in a `Float`), the function returns the value `Float::HUGE_VAL` with the same +/- sign as the correct value of the function (except in the case of `tan()`). In the case of an underflow, the result is `0`.

Argument Conversion

If an `Int` value is passed to one of the numerical standard functions instead of a `Float` value, the `Int` value is converted implicitly by the function into a `Float` value.

Trigonometrical Functions

acos(x(Float)) $\rightarrow$ *Float*

The $acos()$ function computes the arc cosine of x in radians. An exception is triggered if x is not in the interval $[-1, +1]$. The result is in the interval $[0, \pi]$.

asin(x(Float)) $\rightarrow$ *Float*

The $asin()$ function computes the arc sine of x in radians. An exception is triggered if x is not in the interval $[-1, +1]$. The result is in the interval $[-\pi/2, +\pi/2]$.

atan(x(Float)) $\rightarrow$ *Float*

The $atan()$ function computes the arc tangent of x in radians. The result is in the interval $(-\pi/2, +\pi/2)$.

atan2(y(Float), x(Float)) $\rightarrow$ *Float*

The $atan2()$ function computes the arc tangent of y/x in radians. It uses the sign of both arguments to compute the quadrants of the return value. If both arguments are 0, an exception is triggered. The result is in the interval $[-\pi, +\pi]$.

cos(x(Float)) $\rightarrow$ *Float*

The $cos()$ function computes the cosine of x (specified in radians).

sin(x(Float)) $\rightarrow$ *Float*

The $sin()$ function computes the sine of x (specified in radians).

tan(x(Float)) $\rightarrow$ *Float*

The $tan()$ function computes the tangent of x (specified in radians).

acosh(x(Float)) $\rightarrow$ *Float*

The $acosh()$ function computes the hyperbolic arc cosine of x. If x is not in the interval $[1, \infty)$, an exception is triggered.

asinh(x(Float)) $\rightarrow$ *Float*

The $asinh()$ function computes the hyperbolic arc sine of x.

atanh(x(Float)) $\rightarrow$ *Float*

The $atanh()$ function computes the hyperbolic tangent of x. If x is not in the interval $(-1, +1)$, an exception is triggered.

cosh(x(Float)) $\rightarrow$ *Float*

The $cosh()$ function computes the hyperbolic cosine of x.

sinh(x(Float)) $\rightarrow$ *Float*

The $sinh()$ function computes the hyperbolic sine of x.

tanh(x(Float)) $\rightarrow$ *Float*

The $tanh()$ function computes the hyperbolic tangent of x.

Exponential Functions and Logarithmic Functions

exp(x(Float)) $\rightarrow$ *Float*
: The $exp()$ function computes the exponential function of x (i.e. e^x).

frexp(x(Float)) $\rightarrow$ *[Float, Int]*
: The $frexp()$ function breaks a floating-point number into a normalized fractions ($frac$) and an integral power of 2 (exp), where $x = frac \cdot 2^{exp}$. Both values are returned as vector $[frac, exp]$.

  If x is 0, both parts of the result are 0.

ldexp(x(Float), exp(Int)) $\rightarrow$ *Float*
: The $ldexp()$ function multiplies the floating-point number x by the integral power exp of 2.

log(x(Float)) $\rightarrow$ *Float*
: The $log()$ function computes the natural logarithm of x. If the argument is negative, an exception is triggered. If it is 0, the result is $-Float :: HUGE_VAL$.

log10(x(Float)) $\rightarrow$ *Float*
: The $log10()$ function computes the base 10 logarithm of x. If the argument is negative, an exception is triggered. If it is 0, the result is $-Float :: HUGE_VAL$.

modf(x(Float)) $\rightarrow$ *[Float, Float]*
: The $modf()$ function breaks the argument into an integer part (int) and a fractional part ($frac$), of which both have the same sign as the argument. Both values are returned as vector $[int, frac]$.

Exponential Functions

pow(x(Float), y(Float)) $\rightarrow$ *Float*
: The $pow()$ function computes x to the power of y. An exception is triggered if x is negative and y is not an integer, or x is 0 and y is negative. The result is 1.0 if both x and y are 0.

sqrt(x(Float)) $\rightarrow$ *Float*
: The $sqrt()$ function computes the nonnegative square root of x. If x is negative, an exception is triggered.

Rounding, Absolute Value and Remainder

ceil(x(Float)) $\rightarrow$ *Float*
: The $ceil()$ function computes the smallest integer not smaller than x.

fabs(x(Float)) $\rightarrow$ *Float*
: The $fabs()$ function computes the absolute value of x.

floor(x(Float)) $\rightarrow$ *Float*
: The $floor()$ function computes the largest integer not larger than x.

fmod(x(Float), y(Float)) $\rightarrow$ *Float*

For y is not equal to 0, the $fmod()$ function computes the value $x - i \cdot y$ so that the result for an integer i has the same sign as x and a magnitude less than the magnitude of y. If y is 0, an exception is triggered.

Chapter 4

Basic Interfaces

The basic interfaces described below implement fundamental concepts on which the actual types of the OFML standard are based. Such a type implements one or several of these basic interfaces.

Each interface is assigned an interface category (Appendix H). By means of the general method for determining the category association of a type or an instance described below, it is possible to determine as an alternative to determining the type identity whether a type implements a specific interface or whether an instance of the type provides the functionality of the interface.

4.1 MObject

The *MObject* interface defines the fundamental interfaces of all OFML types. Consequently, every OFML type implements at least this interface.

4.1.1 Type Identity and Category Association

- *getType()* → *Type*

  The function provides the direct type of the implicit instance.

- *getClass()* → *String*

  The function provides the name of the direct type of the implicit instance.

  Note: Equivalent to `String(getType().getName())`.

- *isA(pType(Type))* → *Int*

  The function verifies the association to a transferred type *pType*. *isA()* furnishes 1 if *pType* is the direct type of the implicit instance or a super type of it. Otherwise, the result is 0.

- *isCat(pCat(Symbol))* → *Int*

  The function furnishes 1 if the implicit instance belongs to the transferred category.

  Note: As a rule, a type inherits the association to categories from its direct super type. For this reason, attention should generally be paid in overwriting the function that the inherited implementation of the function is called for the transfer of a category that is not defined by the concrete type itself.

4.1.2 Instance Identity and Hierarchy

- *final getName()* → *String*

  The function returns the absolute name of the implicit instance.

- *final getFather()* → *MObject*

  The function provides a reference to the father object. If the implicit instance does not have a father, the result is $NULL$.

- *final getRoot()* → *MObject*

  The function furnishes a reference to the root instance of the hierarchy in which the implicit instance is located.

- *final getChildren()* → *MObject[]*

  The function returns a list of object references that represent the direct children of the implicit instance. If no children are available, an empty list is returned.

- *final getElements()* → *MObject[]*

  The function returns a list of object references that represent those direct children of the implicit instance that are also elements. If no elements are available, an empty list is returned.

- *final add(pType(Type) ...)* → *MObject*

  The function creates a child of the implicit instance of type *pType* and continues to register it as element. The local name of the child is selected automatically. Should a type require additional parameter for its instantiation, they must be specified with the *add()* call after *pType*.

  The return value of the function is a reference to the created object or $NULL$.

- *final remove(pChild(MObject))* → *self*

  The function removes the specified object, which is a child of the implicit instance, from the list of children of the implicit instance. If it is an element at the same time, it is also removed from the list of elements.

4.2 Base

As an extension of the *MObject* interface, the *Base* interface also represents a fundamental interface of the OFML types that is implemented by most of the OFML types.

Every type that implements the *Base* interface **also** implements the *MObject* interface.

4.2.1 Instance Variables

- *mIsCutable(Int)*

  The variable specifies the independence of the instance with respect to the cut operation of the clipboard (*setCutable()* and *isCutable()* functions in section 4.2.2).

- *[static] eps(Float) = 0.005*

  The static variable *eps* must be used for geometric relation operations due to the limited presentation accuracy of floating point numbers in OFML. Neither this variable nor the following ones may be redefined. Non-redefinable variables can be designated with *final*.

- *[static] sPi4(Float)* $= \frac{\pi}{4}$
- *[static] sPi2(Float)* $= \frac{\pi}{2}$
- *[static] sPi(Float)* $= \pi$
- *[static] s2Pi(Float)* $= 2\pi$

4.2.2 Selectability

- *final selectable()* $\rightarrow$ *self*

  The function allows the selection of the implicit instance.

- *final notSelectable()* $\rightarrow$ *self*

  The function prohibits the selection of the implicit instance. In the case of an attempted selection of the implicit instance, the first instance that is selectable within the scope of an upward traversing is selected.

- *final hierSelectable()* $\rightarrow$ *self*

  The function allows the selection of all entities of the subhierarchy whose root object is the implicit instance. Whether a single instance can actually be selected is determined by the status that was set via *selectable()* or *notSelectable()*.

- *final notHierSelectable()* $\rightarrow$ *self*

  The function prohibits the selection of all entities of the subhierarchy whose root object is the implicit instance. The prohibition applies to all entities of the subhierarchy, even if the selection of an individual instance via *selectable()* is allowed in principle. Thus, *notHierSelectable()* in reference to a single instance takes precedence over *selectable()*.

- *final isSelectable()* → *Int*

  The function returns True if the implicit instance can be selected. This is the case if *selectable()* was called for the instance and was not called for an object in the hierarchy via *notHierSelectable()* instance.

  Entities can be selected initially.

4.2.3 Cuttability

- *setCutable(pMode(Int))* → *Void*

  The function determines the independence of the implicit instance with respect to the cut operation of the clipboard and saves the transferred mode in the *mIsCutable* instance variable. Possible values are:

-1	In general, the implicit instance may not be deleted.
0	The implicit instance itself may not be deleted, but it can be deleted within the scope of a higher-level instance. In the case of an attempted cut operation of the implicit instance, the operation is applied to the first instance in the course of an upward traversing for which *isCutable()* furnishes 1.
1	The implicit instance can be deleted and copied to the clipboard. This is the initial state.
2	The implicit instance can be deleted, but it may not be copied to the clipboard.

  Example: Mode 2 is used for objects such as cornice profiles that are constructed with regard to a set of base objects and, consequently cannot readily be copied to another spatial or topological position.

- *isCutable()* → *Int*

  The function queries the independence of the implicit instance with respect to the cut operation of the clipboard. It furnishes the value of the *mIsCutable* instance variable that can be described using the *setCutable()* function.

- *removeValid()* → *Int*

  The function returns True if the implicit instance may be deleted.

  Entities can be deleted initially.

  In contrast to the *isCutable()* function, which specifies the fundamental ability of deleting an object and is called by the application prior to a cut operation on the selected object, the *removeValid()* function is used for modeling dynamic aspects of the ability of deleting and is called by father entities within the scope of REMOVE_ELEMENT rules.

4.2.4 Visibility

- *final hide() → self*

  The function hides the implicit instance, including its children and grandchildren. Hiding entities does not have any influence on collision recognition.

- *final show() → self*

  The function makes the implicit instance visible again if it was hidden.

- *final isHidden() → Int*

  The function indicates through its return value whether the implicit instance is visible (0) or hidden (1).

4.2.5 Resolution

The following functions can be used to set or query the object space resolution of an object. In general, this applies to the mapping of an analytical or parametric primitive to a piece by piece linear approximation. The direct conversion is done for geometric elementary types only (Chapter 7). All other types or entities merely pass on the resolution. Imported polygonal data records are not effected.

Normally, the resolution should only be set directly for the root of an object hierarchy. However, direct setting of the object space resolution for a non-root object is allowed.

The resolution is indicated by a floating point number r in the range $0.0 \leq r \leq 1.0$. Where 0.0 represents the minimum resolution and 1.0 the maximum resolution. If the resolution 0.0 is specified for a parametric primitive, its representation corresponds to the polygonal body that results from the corresponding connection of the defining vertices. The initial resolution is 0.1.

- *final setResolution(pRes(Float)) → self*

  The function sets the object space resolution for the subtree preset by the implicit instance. This resolution continues to be inherited in the subtree. If an ancestor already contains an explicitly assigned object space resolution, the recursive inheritance is ended at this position and for this path of the subtree.

- *final getResolution() → Float*

  The function returns the valid object space resolution for the implicit instance.

4.2.6 Change Status

- *final setChanged() → self*

  The function explicitly marks the implicit instance as changed with respect to the instant immediately after executing the initialization. An explicit call of *setChanged()* is necessary if instance variables are directly written without other changing operations being performed (e.g., creation of children, move). The change status is evaluated to enable an efficient storing of instance hierarchies which is applied to clipboard and persistence operations.

- *final setUnchanged() → self*

  The function resets the change status of the implicit instance to the status immediately after executing the initialization. That is, the instance is now considered as unchanged with respect to the instant immediately after the initialization.

4.2.7 Collision Detection

- *final disableCD() → Void*

  The function deactivates the collision detection for the implicit instance. Afterwards, the implicit instance, including its children, are ignored by the collision detection.

- *final enableCD() → Void*

  The function (re-)activates the collision detection for the implicit instance.

- *final isEnabledCD() → Int*

  The function furnishes 0, if the implicit instance is excluded from collision detection, otherwise it furnishes 1.

4.2.8 Dimensioning

- *measure(pMode(Symbol)) → Void*

  The function activates the dimensioning of the implicit instance. If necessary, different types of dimensioning can be selected using the implementation-dependent value *pMode*. If only one type of dimensioning exists, the parameter may be ignored.

  The following symbols are predefined for the dimensioning:

  - *@ISO* The dimensioning is done in meter.
  - *@INCH* The dimensioning is done in inch.

- *unMeasure() → Void*

  The function deactivates the dimensioning of the implicit instance.

4.2.9 Spatial Modeling

- *final setPosition(pPosition(Float[3])) → self*

  The function unconditionally sets the local position[1] of the implicit instance, i.e., no rules are called and the degrees of translation freedom are ignored to the position *pPosition*. At the same time, this position represents the move of the implicit instance compared with the father of the implicit instance. Initially, an instance does not have a move with respect to its father. If no father exists, the world coordinate system serves as reference.

  The function is used for the explicit positioning within functions.

[1] Actually, the local coordinate system is moved to the respective position relative to the father.

- *final getPosition() → Float[3]*

  The function furnishes the current move of the implicit instance with respect to its father, provided that he exists, or with respect to the world coordinate system.

- *final translate(pVector(Float[3])) → self*

  The function conditionally moves the implicit instance by the vector *pVector* defined in the world coordinate system. The conditionality of the move results from a possible rasterization and snapping functionality, provided that it is supported by the OFML runtime environment, translational degrees of freedom (*setTrAxis()*), as well as through the presence of *TRANSLATE* rules of the reason (Chapter 5). If *TRANSLATE* rules are defined for the implicit instance, they are directly called after executing the translation.

  The function is used for interactive positioning via direct manipulation or user interface. The *pVector* vector is transformed from the world coordinate system to the local coordinate system of the implicit instance under consideration of the current inherited and local modeling of the implicit instance. This ensures an intuitive modeling via the *translate()* function.

- *final moveTo(pPosition(Float[3])) → self*

  The function conditionally moves the implicit instance to the *pPosition* position defined in the world coordinate system. The semantics of the function completely corresponds to the call of *translate()* with the *pPosition* - *getWorldPosition()* vector.

- *final setTrAxis(pAxis(Int)) → self*

  The function permits or prohibits the movability of the implicit instance for individual axes of the local coordinate system. The *pAxis* parameter results from the addition of allowed axes, whereby x, y and z-axis are represented by 1, 2 and 4. If *pAxis* features the value 0, the object cannot be moved.

- *final getTrAxis() → Int*

  The function furnishes the current movability of the implicit instance.

- *final rotate(pAxis(Symbol), pArc(Float)) → self*

  The function conditionally rotates the implicit instance by the *pArc*angle[2] defined in the radiant measure with respect to the *pAxis* local coordinate axis. The conditionality of the rotation results from a possible rasterization and snapping functionality, provided that it is supported by the OFML runtime environment, rotational degrees of freedom (*setRtAxis()*), as well as through the presence of *ROTATE* rules of the reason (Chapter 5). If *ROTATE* rules are defined for the implicit instance, they are directly called after executing the rotation.

  pAxis is either *@PX*, *@PY* or *@PZ* as long as a rotation about the (positive) x, y or z-axis occurs. Alternatively, a rotation about an opposite axis may be carried out. The respective symbols are *@NX*, *@NY* and *@NZ*. The rotation about random axes is achieved by corresponding consecutive rotations about the elementary axes.

  In contrast to the translation, there is no function for unconditional setting for the rotation. The setting of a certain orientation is carried out either initially, i.e., if no rotation compared to the father has taken place, or through a subtraction of the actual orientation from the orientation to be set. However, this does not invalidate the conditionality described above.

[2] Actually, the local coordinate system is rotated accordingly.

Within the scope of rules, the correction of orientations is carried out through a new application of the *rotate()* function. In this case, the OFML runtime system must ensure that *ROTATE* rules are not called again.

A general issue concerning the rotation about cartesian axes consists of the overlay of the three elementary rotations. For this reason and to ensure the correct functioning of the rotations, it is recommended to release only one rotational axis at a time (*setRtAxis()*).

The function is used for the interactive positioning via direct manipulation or user interface as well as for the explicit positioning within functions.

- *final getRotation(pAxis(Symbol)) → Float*

  The function furnishes the current rotation in the radiant measure by the rotational axis specified through *pAxis*.

  Attention: If an instance was rotated about more than one axis, *getRotation()* could furnish unexpected results. This is due to the principal problem of overlay of the three elementary cartesian rotations.

- *final setRtAxis(pAxis(Int)) → self*

  The function permits or prohibits the ability of rotation of the implicit instance for individual axes of the local coordinate system. The *pAxis* parameter results from the addition of allowed axes, whereby x, y and z-axis are represented by 1, 2 and 4. If *pAxis* features the value 0, the object cannot be rotated.

  It should be possible to rotate entities about a maximum of one rotation axis. However, this axis may be changed over time.

- *final getRtAxis() → Int*

  The function furnishes the current ability of rotation of the implicit instance.

- *final getLocalBounds() → Float[2][3]*

  The function furnishes the minimum axis-orthogonal delimiting volume of the implicit instance in reference to its local coordinate system. The delimiting volume includes children and the origin of the local coordinate system.

  The return value is a vector consisting of two elements. The first element is the minimum coordinate within the local delimiting volume. The second element is the maximum coordinate within the local delimiting volume.

  The OFML runtime environment must ensure that the local delimiting volume is always in a consistent state.

- *final getLocalGeoBounds() → Float[2][3]*

  The function furnishes the minimum axis-orthogonal delimiting volume of the implicit instance with reference to its local coordinate system. In contrast to *getLocalBounds()*, the delimiting volume does not include the origin of the local coordinate system and children with empty geometry.

- *final getWorldBounds() → Float[2][3]*

The function furnishes the minimum axis-orthogonal delimiting volume of the implicit instance in reference to the world coordinate system. The delimiting volume includes the children.

The return value is a vector consisting of two elements. The first element is the minimum coordinate within the global delimiting volume. The second element is the maximum coordinate within the global delimiting volume.

The OFML runtime environment must ensure that the global delimiting volume is always in a consistent state.

- *final getWorldGeoBounds() → Float[2][3]*

  The function furnishes the minimum axis-orthogonal delimiting volume of the implicit instance in reference to the world coordinate system. In contrast to *getWorldBounds()*, the delimiting volume does not include children with empty geometry.

- *final getDistance(pDirection(Symbol)) → Float*

  The function determines the shortest distance of the implicit instance along one of six directions, starting with the local delimiting volume to another instance in the scene. The direction indicated by *pDirection* features one of the following values: *@NX*, *@PX*, *@NY*, *@PY*, *@NZ*, *@PZ*.

  The return value is the distance, provided that another instance could be determined, or −1.

4.2.10 Rule Call

- *final callRules(pReason(Symbol), pArg(Any)) → Int*

  The function triggers the execution of the rules defined for the reason *pReason*. *pReason* is either a predefined rule (Chapter 5) or a user-defined rule.

  The explicit call of a predefined rule reason can be used, for example, to explicitly request a snapping behavior that is implemented by means of a *TRANSLATE* rule, following the initial positioning of the instance. If a predefined rule reason is called, *pArg* must correspond to the specification in Chapter 5.

  The explicit call of a user-defined rule reason via *callRules()* is the only possibility to bring the corresponding rules to be executed. A principal application of user-defined rule reasons consist of enabling a communication between entities that is more flexible and robust than the communication via the functions of types. In this case, the necessity for checking type compatibility is not required; if no rules are defined in a type for a certain reason, no error will occur. However, calling a function for a type that does not define this function, will always result in an error.

  Example: Spotlights can be planned for a system, but they can normally not be moved. However, as children of a very few types they can be moved in a specific way. In the *TRANSLATE* rule of the spotlight, the move is reset so that the spotlight is not moved. This is followed by a call of the father with *callRules()*, whereby the user-defined reason *MOVE_SPOT*, the desired new position of the spotlight, and the spotlight itself are transferred. If the father permits a movement of the spotlight, it can control it with a corresponding *MOVE_SPOT* rule. Otherwise, the spotlight remains unchanged.

The return value is -1 if a rule of the reason *pReason* failed. Otherwise, the return value is 0.

4.2.11 Dynamic Properties

Features of an instance whose current characteristics are stored in a corresponding instance variable (and that can be assigned or queried using corresponding set and get functions), are referred to as *static* features or properties of the instance. In contrast, it is sometimes necessary to assign dynamic properties and values to an instance for the duration of its existence. For this purpose, the *Base* interface manages an internal hash table for each instance, in which such properties can be set up. A property is defined and addressed via its unique key of the *Symbol* type. The value for the key entered in the table can come from a simple type or from the reference types *String*, *Vector*, *List*, and *Hash*.

- *getDynamicProps() → Hash*

  The function furnishes the (reference to the) hash table for dynamic properties.

4.2.12 2D Representation and ODB

Objects that implement the *Base* interface can be equipped with a 2D representation. It is created by means of the *getOdbInfo()*, *getPictureInfo()*, *invalidatePicture()* methods and the methods for creating primitive 2D objects, as described in Appendix B.

2D symbols created via *getOdbInfo()* and *getPictureInfo()* cannot be used simultaneously. If a hash table is returned by *getOdbInfo()*, a symbol that may be specified by means of getPictureInfo() will not be represented.

- *getOdbInfo() → Hash*

  The function is called by the core system at random times to query the current ODB information that is required for the creation of a 2D symbol or the 3D geometries. The function returns the ODB information in form of a hash table or *NULL* if no ODB information is available for the object. The use of ODB is described in [ODB].

- *getPictureInfo() → Vector*

  The function is called by the core system at random times to query information about the 2D symbol that is to be used for this object. The return value is a vector consisting of three elements:

  – The first element is either *NULL*, in which case this object does not feature a 2D symbol, or the fully qualified name of the symbol. The name is used to search for a corresponding EGM, DMP, or FIG symbol. The first symbol to be found is used for the representation in 2D mode. If no symbol can be found, a symbol is created automatically based on the 3D geometries for the object and all its current child objects.

- The second element is either *@TRAVERSAL_STOP* or *@TRAVERSAL_CONT*. It determines whether possibly available symbols of child objects should be represented (*@TRAVERSAL_CONT*) or not (*@TRAVERSAL_STOP*) for the representation in 2D mode. If a symbol is automatically generated for the object, this value should always be set to *@TRAVERSAL_STOP*.
- The third element is either *@SHARE_ON* or *@SHARE_OFF*. It determines whether symbols with identical names from different objects should be used jointly (*@SHARE_ON*) or whether the symbol for each object is loaded and generated again (*@SHARE_OFF*). In general, *@SHARE_ON* should be specified in this case for symbols that are loaded from files. For objects whose 2D symbol is automatically generated, *@SHARE_ON* can be specified if different entities with identical symbol name are always equipped with the same 3D geometry, otherwise *@SHARE_OFF* should be used. But it should be observed that child objects (e.g., accessories) become part of the symbol, so that for the joint use of automatically generated symbols for objects, that may possibly contain additional children, these children are visible either for all objects or for none.

By default, *getPictureInfo()* returns the type of class of the implicit instance as symbol name, allows the traversing of child objects with representation in 2D mode, and prevents the joint utilization of symbols.

Note: A change in the child objects does not automatically cause a matching of an automatically generated symbol.

If possible, the automatic generation of 2D symbols should be abandoned since it can lead to noticeable delays, especially with repeated application, and the result is usually unsatisfactory for an effective planning in 2D mode.

- *invalidatePicture()* → *Void*

  The function must be called after properties of the object that affect the 2D or 3D geometry have changed. The core system discards all saved information (return values of *getOdbInfo()* and *getPictureInfo()* as well as 2D symbols (ODB, EGM, DMP, FIG, and generated ones)). If required, this information is queried again and the 2D symbols are generated.

- *createOdbObjects(pUpdate(Int))* → *Void*

  The function generates child objects according to the specification in the ODB. If the *pUpdate* parameter is 0 (false), all currently existing child objects generated by the ODB are deleted and then recreated. If the *pUpdate* parameter is 1 (true), a matching of the existing child objects generated by the ODB is carried out.

Note: In the current OFML implementation of EasternGraphics GmbH, all child objects generated by the ODB are deleted independent of the Parameter *pUpdate* parameter, and then recreated by the ODB.

4.3 Material

The *Material* interface defines the functions for processing material properties (surface properties) on the basis of material categories. All types whose entities can be assigned material properties

must implement this interface.

All furniture or furniture components whose materials should be processed similarly due to functional and/or aesthetic viewpoints are combined in a material category. An instance may belong to one or several material categories. Material categories are designated through symbols.

Predefined material categories are listed in Appendix H.

Example: Typical material categories are *corpus*, *front*, *base*, *tabletop*. Entities of a cabinet type with doors and/or drawers would then belong to the categories *corpus*, *front* and *base* while the child entities, for example, that implement the corpus belong only to the *corpus* category. The corresponding OFML material categories could be: *@CORPUS*, *@FRONT*, *@BASE* and *@TOP*.

Note: The universally valid *@ANY* material category is predefined (function *setCMaterial()*) and may not be used in a different capacity.

For each material category, a limited set of possible materials is specified that is also designated by symbols (*getMatCategories()* function). Material designators are unique across all material categories. The visual properties of a material are specified in a separate file whose format is described in Appendix D.2. Each material designator must be assigned a material name (*getMatName()* function) to be able to read the corresponding material description file during runtime by using the name.

Example: The materials "gray laminate" and "light beech veneer" are intended for the *corpus* category, and only the "light beech veneer" material for the *front* category. Possible designators for these materials are *@LGray* or *@VBlight*[3]. Corresponding material names would be "gray laminate" or "light beech" and the corresponding material description files `graylaminate.mat` or `lightbeech.mat`.

4.3.1 Material Categories

- *getMatCategories()* → *Symbol[]*

  It furnishes the list of material categories that are currently defined for the implicit instance. An instance for which no material categories are defined furnishes either an empty list or *Void*. In the latter case, the defined material categories for the father instance should be used for the implicit instance.

  The number of material categories defined for the implicit instance can change dynamically and, therefore, differentiate themselves from the set of all potentially possible material categories (*getAllMatCats()* function).

- *isMatCat(pCat(Symbol))* → *Int*

  It returns 1 if the transferred material category belongs to the material categories currently defined for the implicit instance, otherwise 0.

[3]The material codes already in place in manufacturing companies are ideal for use as symbolic material designators.

- *getAllMatCats() → Symbol[]*

  It furnishes the list of all material categories that are potentially definable for the implicit instance (see also *getMatCategories()* function).

- *getCMaterials(pCat(Symbol)) → Symbol[]*

  It furnishes the list of all materials that are applicable within the transferred material category for the implicit instance. The return value is of type *Void* if the transferred material category does not belong to the material categories currently defined for the implicit instance.

4.3.2 Materials

- *setCMaterial(pCat(Symbol), pMat(Symbol)) → Int*

  It assigns the specified material to the implicit instance in the transferred material category. The operation is recursively applied to all children and grandchildren. The function is without effect if neither the implicit instance nor one of the children belongs to the transferred material category. The return value is 1 if the material could be assigned to the implicit instance or at least to one of its ancestors (child, grandchild, etc.) 0.

  The predefined universally valid material category *@ANY* can be used of explicitly assigning a material without considering the association of the implicit instance to a concrete material category.

- *getCMaterial(pCat(Symbol)) → Symbol*

  The function furnishes the material currently assigned to the implicit instance in the transferred material category or a value of the *Void* type if the implicit instance does not currently belong to the transferred material category.

- *getMatName(pMat(Symbol)) → String*

  The function furnishes the material name to the transferred material or a value of the *Void* type for the implicit instance if the material is unknown. The standard implementation calls the function of the father of the same name.

4.4 Property

Properties are object features that can be changed interactively by the system user with the help of suitable dialogs (property editors).The *Property* interface defines the functions for handling properties. Properties can be associated with features in product databases (Chapter 9).

4.4.1 Specifying Properties

- *setupProperty(pKey(Symbol), pDef(Any[5]), pPos(Int)) → Void*

  The function creates a property with the specified key (identifier) and the transferred specification. If a property with the specified key is already registered, its specification is overwritten by the parameter values.

  The definition of a property (*pDef* parameter) is a vector made up of five values:

pName(String) the name of the property (appears in the property editor). This can be a wildcard that is resolved via an external resource file (Appendix D).

pMin(Any) lower (inclusive) limit of the value range

pMax(Any) upper (inclusive) limit of the value range or maximum length fir String properties

pFmt(String) desired special input/output format (syntax and meaning according to Appendix E.1)

pType(String) the type of property:

- *b* boolean value
- *i* integer
- *f* real number
- *s* string
- *ch* choice list (*choice list*)
  The type specification is followed by a space and then by the list of choice values. Each choice value is either a string ID designated (language-neutral) by a preceding @ character, or by a pair made up of string ID separated by spaces and language-dependent designation (Appendix D). The choice values are separated by spaces. If no language-dependent designation is specified for a value, it is read from language-dependent designation files by means of the string ID.
- *chf* Choice list via function
  The type information is followed by the name of a function which, when called for the implicit instance, furnishes the list of choice values in the same form as the explicit information in a property of type *ch*.
- u Special type (*user defined*)
  The type information is followed by a space and then by the ID of the required special editor and (after an additional space) additional information for the special editor, if required.

  Note: It is not guaranteed that the special editor is implemented in the OFML runtime environment used at the time.

Besides the actual property definition, the desired position in the property list can be specified in the *pPos* parameter. The same specification applies to the setting of the position as of the *setPropPosOnly()* function which can be used to individually set the position for an existing property.

The value range limits, format, and position are optional. Missing information are designated by a parameter of type *Void*.

In the type of the implicit instance, a *set* and a *get* method can be defined for each property:

– *set<Key>(pValue(Any))* → *Void*
  The function is called if the value of the <Key> property was changed.

Note: Generally, an assignment of the value to a corresponding instance variable is performed in this function. Any additional semantics, such as the regeneration of geometry or corresponding collision tests, is reserved for the *propsChanged()* function.

– *get<Key>() → Any*
  The function furnishes the value for the <Key> property currently stored in the implicit instance.

  Note: Generally, the function furnishes the contents of a corresponding instance variable. A return value of type *Void* designates a non-specified property, e.g., with optional features.

- *setPropPosOnly(pKey(Symbol), pPos(Int)) → Int*

  The function specifies the desired position in the property list for the property with the specified key. If no property with the specified key is defined for the implicit instance, the function is without effect and the return value is of type *Void*. If a property with the specified key is defined for the implicit instance, the old position information is overwritten. If *pPos* is an integer greater than or equal to 0 and the desired position was already assigned to another property, then this and all the following properties in the position list are moved back by one position. If *pPos* is of type *Void* or features the value −1, no special position is required for the property. It is then filed in the property list according to the properties for which a position was explicitly requested. The new position of the property is the return value or −1 if no special position is required.

- *setExtPropOffset(pOffset(Int)) → Void*

  This function is used to assign an offset to the implicit instance for positions of externally defined properties, i.e., of properties that are defined for the implicit instance by another instance besides the implicit instance. The offset indicates the smallest position number that may be used for externally defined properties.

  Example: A typical example of externally defined properties are those that are defined for the representation of product features from the product database for the implicit instance by a global product data manager instance (Section 9.1).

- *removeProperty(pKey(Symbol)) → Void*

  The function removes the property specified by the indicated key from the property list. If no property with the indicated key is defined for the implicit instance, the function is without effect.

- *clearProperties() → Void*

  The function removes all properties from the property list.

4.4.2 Querying Properties

- *hasProperties() → Int*

  The function furnishes 1 if properties are defined for the implicit instance, otherwise it returns 0.

- *hasProperty(pKey(Symbol))* → *Int*

  The function furnishes 1 if a property with indicated key is defined for the implicit instance, otherwise it returns 0.

- *getPropertyDef(pKey(Symbol))* → *Any[]*

  The function furnishes the definition of the property with indicated key. The structure of the returned vector corresponds to the structure of the *pDef* parameter that was transferred as property definition to the *setupProperty()* function. If no property with the indicated key is defined for the implicit instance, the return value is of type *Void*.

- *getPropertyPos(pKey(Symbol))* → *Int*

  The function furnishes the position of the property with indicated key. If no special position was requested for the property, the return value is −1. If no property with the indicated key is defined for the implicit instance, the return value is of type *Void*.

- *getExtPropOffset()* → *Int*

  This function is used to furnish the for positions of externally defined properties, i.e., of properties that are defined for the implicit instance by another instance besides the implicit instance. The offset indicates the smallest position number that may be used for externally defined properties. This offset should be called by an external instance before the definition of a property for the implicit instance and should be taken into consideration for the assignment of explicit positions.

  If no other value was assigned using *setExtPropOffset()*, the default return value is equal to 0.

- *getPropertyKeys()* → *Symbol[]*

  The function furnishes a list of the keys of all properties currently defined for the implicit instance.

  At the same time, the properties are sorted in ascending order according to their explicit positions. The properties without explicit position appear at the end of the list in an undefined order.

- *getProperties()* → *String*

  The function furnishes a description of all properties currently defined for the implicit instance. The format of this description is explained in Appendix E.2.

- *getPropTitle()* → *String*

  The function furnishes a brief description of the instance for use in the header line of property editors.

Note: The two functions described beforehand are used by the property editors to build up a dialog window.

4.4.3 Property Values

- *getPropValue(pKey(Symbol))* → *Any*

  The function furnishes the value currently stored in the implicit instance for the property with the indicated key. If no property with the indicated key is defined for the implicit instance, the return value is of type *Void*

  Note: The function utilizes the get method of the property (see *setupProperty()* function). If the type of the implicit instance does not feature such a method, the value is determined from the hash table of the dynamic properties (see *getDynamicProps()* function at the *Base* interface).

- *setPropValue(pKey(Symbol), pValue(Any))* → *Int*

  The function assigns the implicit instance a new value for the property with the indicated key.

  If the property is associated with a feature in a product database, the global product data manager (Chapter 9) evaluates relationships between properties and property values next (consequently, other properties or their values may change). Next, the *propsChanged()* function (see below) for performing special processings is called. True is transferred for the *pDoChecks* parameter. If the value assignment of the product manager or *propsChanged()* was rejected, all properties are reset to the state saved at the start of the function and the *propsChanged()* function is called again, whereby False is now transferred for the *pDoChecks* parameter.

  The return value of the function is True if the definition of one or several properties changed or if properties were added or removed.

  Note: The function uses the set method of the property (see *setupProperty()* function) for the actual assignment of the new value to the corresponding instance variable. If the type of the implicit instance does not feature such a method, the value under the key of the property is written in the hash table of the dynamic properties (see *getDynamicProps()* function at the *Base* interface).

- *propsChanged(pPKeys(Symbol[]), pDoChecks(Int))* → *Int*

  The function performs special processings and checks after property values were changed. The properties whose values changed are specified by their keys. The *pDoChecks* parameter indicates whether checks need to be performed or whether it is only necessary to respond to the change of property values, e.g., through geometry matching. The return value is 1 if the new property values are valid, otherwise it is 0.

  Note: The function is called at the end of the *setPropValue()* function. In general, matchings of the geometry or the material properties of the implicit instance are carried in the function.

- *changedPropList()* → *Symbol[]*

  The function delivers the reference to the list of properties whose values changed during the processing of the *setPropValue()* function. The properties are recorded in the list based on their keys.

Note: In general, the function is used only by product data managers (Chapter 9) during the evaluation of knowledge on product data relationships within the *setPropValue()* function.

The list is emptied at the start of each execution of *setPropValue()*.

4.4.4 Activation Status

A property can be *active* or not. For an active property, its value can be changed interactively. For non-active properties, only their current values are displayed and they cannot be changed interactively. The initial state following the definition of a property is "active."

- *setPropState(pKey(Symbol), pState(Int)) → Void*

  The function sets the activation status of the property with the indicated key for the implicit instance to the transferred value. If no property with the indicated key is defined for the implicit instance, the function is without effect.

- *getPropState(pKey(Symbol)) → Int*

  The function furnishes 1 if the implicit instance features a property with the indicated key and if it is active. The function furnishes 0 if the implicit instance features a property with the indicated key and if it is not active. If no property with the indicated key is defined for the implicit instance, the return value is -1.

4.4.5 Information about Properties and Property Values

- *getPropInfo(pKey(Symbol), pPropValue(Any), pInfoType(Symbol)) → Any*

  The function furnishes the information of the requested type for the specified property value for the implicit instance. The return value is of type *Void* if the instance does not feature the specified property or if no information of the requested type is available.

  Default implementations of this function delegate the call to the *getPropInfo4Obj()* method of the OiProgInfo instance (if available) responsible for the instance, see Chapter 8.

  The following standard information types are predefined:

  @Picture
  : Name of the graphics file that represents the property value (String)

  @Text
  : text description (String, can be text resource)

  @HTML
  : URL of the HTML description (String)

4.5 Complex

The *Complex* interface describes the necessary functionality of complex objects, i.e., of objects that are composed of one or several accessible subobjects (children). In principle, this applies to all types whose entities can be combined, expanded or disassembled at runtime.

4.5.1 Spatial Model

On the one hand, the functions of this group serve the more effective access to the spatial dimensions of objects that would otherwise have to be determined by the more time-consuming *getLocalBounds()* function of the *Base* interface. On the other hand, they allow for using dimensions that deviate from the exact geometric dimensions according to *getLocalBounds()*.

- *getWidth() → Float*

  The function furnishes the width of the implicit instance.

- *getHeight() → Float*

  The function furnishes the height of the implicit instance.

- *getDepth() → Float*

  The function furnishes the depth of the implicit instance.

4.5.2 Dynamic Creation and Management of Children

- *checkAdd((pType(Type), pObj(MObject), pPosRot(Any[2]), pParams(Any)) → Float[3]*

  The function checks whether an instance of the indicated type can be attached to the implicit instance as child and, if positive, furnishes a valid position for the child instance (in the local coordinate system of the implicit instance). If no instance of the indicated type can be attached as child or if no open valid position can be determined, the function returns a value of type *Void*.

  If the *pObj* argument is not of type *Void*, it specifies an already existing instance that should be enlisted to locate a position. If the *pPosRot* argument is not of type *Void*, it specifies a suggested position and rotation with respect to the local coordinate system of the implicit instance. The first element of the parameter vector contains the suggested position (Float[3]) and the second element the suggested rotation with respect to the positive Y axis. If the *pParams* argument is not of type *Void*, it contains additional parameters for the initialization function of the type *pType*.

  To check whether an instance of the transferred type should be added, it may be necessary to generate a temporary instance of the type during the execution of the function, e.g., to be able to make statements about the child to be generated by using function calls on this instance. The way in which such a temporary child instance is generated, is controlled by the so-called *Paste Mode* which is assigned by means of the *setPasteMode()* function before *checkAdd()* is called by the client. If the instance to be inserted represents an article (see *Article* interface), a simple instantiation of the transferred type may sometimes not be sufficient; instead, the temporary child instance must also accept the configuration of the article to be inserted. For this purpose, the desired article specification is transferred by the client by calling the *setTempArticleSpec()* function before calling *checkAdd()* of the implicit instance.

  So far as the type to be inserted defines planning categories (Appendix H), they can be taken into consideration during the implementation of *checkAdd()* functions.

Note: In general, this function is called by the runtime environment if the user has entered the command for inserting an object of a selected type in the scene or in a selected object. If the function furnishes a valid position, the runtime environment generates an instance of the indicated type in the next step and places it at the determined position. If the new object cannot be inserted into the selected object, an attempt is made to insert it in its father instance, etc.

- *setPasteMode(pMode(Symbol))* → *Void*

  The function sets the *Paste* mode for inserting temporary child entities into the implicit instance. The following modes are possible:

  `@CR` The child instance must be re-generated as instance of the type that was transferred to the *checkAdd()* function. This is the default setting.

  `@PA` The child instance should be created as a copy of an already existing object whose representation can be found on the clipboard of the application. In this case, the child instance is generated by means of evaluating the clipboard using the global *oiApplPaste()* function.

- *getPasteMode()* → *Symbol*

  The function furnishes the current *Paste* mode for inserting temporary child entities into the implicit instance.

- *setTempArticleSpec(pArticle(Vector[2]))* → *Void*

  The function assigns the article specification to the implicit instance which must be assigned to the temporary child instance after its creation (see *setXArticleSpec()* function of the *Article* interface, Section 4.6). The *pArticle* parameter contains a vector whose first element lists the base article number, while the second specifies the variant code of the article.

- *getTempArticleSpec()* → *Vector[2]*

  The function returns the article specification for the temporary child instance that was assigned with the *setTempArticleSpec()* function.

- *setMethod(pMethod(String))* → *Void*

  The function sets the method call, including the parameters according to the basic syntax (Chapter 3), which should be executed after generating and initially positioning a child instance following an execution of the *checkAdd()* function for this child instance.

- *getMethod()* → *String*

  The function provides the code according to the basic syntax (Chapter 3), which should be executed after generating and initially positioning a child instance following an execution of the *checkAdd()* function for this child instance. If no method is to be executed, an empty string is returned.

Note: The method call to be executed is provided by the *checkAdd()* function which is executed beforehand. It contains actions that go beyond the positioning of the child instance, e.g., rotating the child instance by a required angle.

- *clearMethod()* $\rightarrow$ *Void*

  After generating a child instance, the function resets a method call to be executed for the child instance, if necessary. In this case, an empty string is set as method call.

- *addPart(pType(Type), pParams(Any))* $\rightarrow$ *MObject*

  The function adds an instance of the specified type as a child to the implicit instance, if possible. If the *pParams* argument is not of type *Void*, it contains additional parameters for the initialization function of the type *pType*. If no instance of the specified type can be added as child, the function returns a value of type *Void*.

  Note: The function utilizes the *checkAdd()* function for determining a valid position and upon the return of such a position after the initial positioning performs the code specified by the *getMethod()* function, if necessary.

- *checkElPos(pEl(MObject), pOldPos(Float[3]))* $\rightarrow$ *Int*

  The function checks the validity of the current local position of the transferred child instance. The function furnishes 1 if the current position is allowed, otherwise it furnishes 0.

  Note: The function is used primarily for checking the new position of a child instance after a translation or rotation of the instance. Generally, a collision check is performed. Additional, type-dependent checks are possible, e.g., monitoring for compliance of a specified grid. If necessary, a correction of the position may be performed before the transformation using the position transferred in the *pOldPos* parameter, e.g., a setting to the next grid position.

4.5.3 Collision Check

- *disableChildCD()* $\rightarrow$ *Void*

  The function deactivates the collision detection for children of the implicit instance which is performed via *checkChildColl()*.

- *enableChildCD()* $\rightarrow$ *Void*

  The function (re-)activates the collision detection for children of the implicit instance which is performed via *checkChildColl()*.

- *isEnabledChildCD()* $\rightarrow$ *Int*

  The function furnishes 1, if the collision detection for the implicit instance is activated, otherwise it furnishes 0.

- *isValidForCollCheck(pObj(MObject))* $\rightarrow$ *Int*

  The function furnishes 1 if the specified (child) instance should be considered during the collision check, otherwise it furnishes 0.

  Note: The function is a hook function which is called by the *checkChildColl()* function. Standard implementations of this function always deliver 1.

- *checkChildColl(pObj(MObject), pExclObj(MObject[])) → MObject*

  The function checks whether a collision of the transferred (child) instance with other objects is present. If the *pExclObj* argument contains a non-empty set of objects, they are excluded from the collision check.

  The function first checks for collision with the children of the implicit instance. The check only takes place if the following conditions are met:

  - *isEnabledChildCD()* of the implicit instance delivers True
  - *isValidForCollCheck()* of the implicit instance delivers True for the transferred (child) instance
  - *isEnabledCD()* of the transferred (child) instance delivers True

  The following children are excluded from the collision check:

  - children for which the *isValidForCollCheck()* function of the implicit instance delivers False
  - children whose *isEnabledCD()* function delivers False
  - children that are listed in the *pExclObj* argument

  If *isEnabledCD()* of the implicit instance delivers True, the function of the father instance of the same name is called next (if it exists and if its type implements the *Complex* interface).

  The return value is the first located object with which the transferred instance collides or a value of type *Void* if no collision was detected or if the collision detection is deactivated.

4.6 Article

The *Article* interface includes a set of functions that provide the necessary information about a planning object from a commercial point of view.

4.6.1 Program Access

- *getProgram() → Symbol*

  The function delivers the ID of the program (Appendix I) to which the implicit instance belongs.

4.6.2 Structure of Order Lists

- *setOrderID(pID(Symbol)) → Void*

  The function assigns a unique ID to the implicit article instance that is used in structures of order lists for assigning an article item of the order list to the instance that represents the article in planning.

The order ID is assigned to the article instance immediately following its generation and is not changed as long as the article instance exists. If the position of the article in the planning hierarchy changes (e.g., in grouping actions), the order ID is transferred from the destroyed instance to the newly generated clone instance in the cut-and-paste operation that takes place.

- *getOrderID() → Symbol*

  The function delivers the unique order ID of the implicit article instance.

4.6.3 Product Data

- *getArticleSpec() → String*

  The function delivers the name of the article (base article number) to which the implicit instance corresponds or a value of type *Void* if no article specification is available for the implicit instance.

  If the result of the function is a value of type *Void*, no entry is generated for the instance in the order lists.

- *getXArticleSpec(pType(Symbol)) → String*

  The function delivers the specification of the requested type for the article to which the implicit instance corresponds or a value of type *Void* if no article specification of the required type is available for the implicit instance.

  The following specification types are predefined:

  `@Base`
  : base article number, designates the model of the article without reference to a concrete implementation/configuration (corresponds to the return value of *getArticleSpec()*)

  `@VarCode`
  : variant code, describes the concrete implementation/configuration of the article with respect to the base article number.

  `@Final`
  : final article number, designates the model of the article and describes its concrete implementation/configuration

  Note: Usually, the final article number consists of the base article number and the variant code. However, this depends upon the underlying product data system. If it does not allow for such a strict definition, variant code and final article number are identical.

- *setArticleSpec(pSpec(String)) → Void*

  The function assigns a new base article number to the implicit instance.

  Note: The function applies only for types whose entities can represent different article (numbers). Assigning a new article number generally leads to a change of certain properties of the instance and, if necessary, also to a new geometric representation.

- *setXArticleSpec(pType(Symbol), pSpec(String))* → *Void*

  The function assigns a new article specification of the specified type to the implicit instance.

  The possible specification types are described under the *getXArticleSpec()* function. With a transfer of an article specification of type @Base, the function behaves like the *setArticleSpec()* function above.

  Note: Assigning a new final article number or a new variant code (specification types @Final or @VarCode) generally leads to a change of certain properties of the instance and, if necessary, to a new geometric representation.

- *getArticleParams()* → *Any*

  The function furnishes the parameters of the implicit instance that should be used for determining the article number (see *getArticleSpec()* function) in addition to the type of the instance. The return value is a vector with the parameter values or a string that already contains the parameter values that were converted into the respective storage format. If no parameters are required for determining the article number, the function furnishes a value of type *Void*.

- *getArticlePrice(pLanguage(String), ...)* → *Any[]*

  The function delivers price information for the implicit instance in the specified language. If an additional optional parameter is given, it specifies the desired currency. However, the price information does not have to be furnished in this currency by the function (if, for example, the underlying product database cannot supply prices in this currency). In this case, the client of the function must perform a conversion into the desired currency by means of conversion rates.

  The return value is a list that contains the individual price components. Every list entry is a vector consisting of three elements:

  1. a description (*String*) that specifies the type or existential reason of the price component, e.g. the reason for a surcharge.
  2. the selling price of the price component (*Float*)
  3. the purchase price of the price component (*Float*)

  The first entry represents an exception since it contains the applied currency (*String*) instead of the prices. The last entry of the list specifies the (accumulated) final price. The optional entries in between specify the individual price components (base price, extra charges, discounts, etc.). If such a price component contains the designator `"@baseprice"`, then it is explicitly designated as base price.

  Note: The explicit designation of the base price component can be used by the respective application to treat the base price differently for the presentation of order lists.

  The function furnishes a value of type *Void*, if no price information is available for the implicit instance.

- *getArticleText(pLanguage(String), pForm(Symbol))* → *String[]*

The function furnishes a text description of the desired form in the specified language for the article that is represented by the implicit instance.

The *pForm* parameter may take on the following values:

– `@s` short description
– `@l` long description

The return value is a list of strings that contain the individual lines of the description or a value of type *Void* if no article description is available for the implicit instance.

Note: The article description furnished by this function contains (typically in long form) only information about the fixed features of the article. A description of the concrete current implementations of the changeable/configurable features of the article is furnished by the *getArticleFeatures()* function.

- *getArticleFeatures(pLanguage(String))* → *Any*

  The function furnishes a description in the specified language for the article represented by the implicit instance, of the current implementation of the product properties that can be changed/configured for the article.

  The return value is a list of two-digit vectors whose first element (*String*) labels the feature, while the second element contains the current value (as character string) of the feature. If the *pLanguage* parameter contains a value of type *Void*, language-independent designators are furnished for feature and value. The function furnishes a value of type *Void*, if no feature description is available for the implicit instance.

  Calls of the function immediately following each other with different parameters for the language furnish lists of identical length and contain the features in the same order. If no language-independent designator is available for a value with a language parameter of type *Void* for a feature, the corresponding entry in the return list is not a vector, but a value of type *Void*.

  Note: The language-independent designators (codes) furnished by the function with a language parameter of type *Void* are generally used by export routines of the application to generate a complete description of an article that can be exported to an external PPS, e.g., for order processing.

4.6.4 Consistency Check

- *checkConsistency()* → *Int*

  The function checks the consistency and completeness of the planning element. If necessary, corrections or additions are performed or error messages are generated.

  If the higher-order instance that initiated the consistency check of the implicit instance, created an *error log*, the error messages should be written into this error log; otherwise they can be issued directly to the user by means of *oiOutput()*. The error log to be used must be called using the *getErrorLog()* function of the global planning instance (*OiPlanning* type, Section 8.1). The data structure of the error log specified for *checkConsistency()* is a

hash table, in which the corresponding messages for each article instance are entered as a code under their order ID (see *getOrderID()* function). The value for this code is a list of three-digit vectors:

1. the error message (String)
2. the name of the object that reported the error (String)
3. the name of the method by which the error was detected (String)

Note: If required, detailed reports for error analyses can be generated with the last two entries.

Chapter 5

Predefined Rule Reasons

This chapter contains a description of predefined rule reasons. The properties of the predefined rule reasons are:

- They correspond to the fundamental basic interactions in their entirety, such as selecting, moving, copying, inserting, etc.
- They are called automatically by the runtime environment, if a corresponding action occurred (implicit call).
- They can also be called explicitly.

In addition, there may be user-defined rule reasons. The properties of user-defined rule reasons are:

- They are always called explicitly.
- The definition of user-defined rule reasons does not violate the compatibility of OFML data.

5.1 Element Rules

CREATE_ELEMENT

The rules of the *CREATE_ELEMENT* reason are called for an O object *before* an E object of T_E type is generated as element of O. The corresponding interaction is the generation of objects in general, e.g., by inserting an object from the clipboard. The parameter of the rules is the T_E type. Rules of this reason can be used to control the aggregation dynamically and dependent upon the state of the O object. Reasons for the failure of such rules can be:

- Entities of the T_E type cannot be aggregated in O.

  Example: Tabletop lamps cannot be planned in a carcass cabinet.

- Entities of the T_E type can be aggregated only in O if certain (geometric) rules are adhered to, e.g., a linear dependency between the width of O and the width of the instance of T_E. If such a condition is violated, the rule will necessarily fail.

  Example: In general, only shelves with the corresponding width can be planned for a carcass cabinet of a certain width.

- On principle, entities of the T_E type can be aggregated in O; however, an insertion would create a conflict with already existing children.

  Example: No more shelves can be planned for a carcass cabinet that already contains a shelf at every grid position.

In these cases further processing of the list of rules is interrupted, and no instance of T_E as element of O is generated.

NEW_ELEMENT

The rules of the *NEW_ELEMENT* reason are called for an O object *before* an E child of O is accepted in the list of elements of O. According to Chapter 2, an element is a special child in so far as elements from outside are accessible by O, i.e., they can be generated or deleted. The corresponding interaction is the generation of objects in general, e.g., by inserting an object from the clipboard. The rules of the *NEW_ELEMENT* reason are called after the rules of the *CREATE_ELEMENT* reason have been called. The generation of an instance cannot be prevented by a *NEW_ELEMENT* rule. Instead, the *NEW_ELEMENT* reason offers expanded possibilities for the derivation of functionality within the corresponding rules compared to the *CREATE_ELEMENT* reason. Since an actual instance is transferred as a parameter instead of a type, queries can be implemented that go beyond comparing types, e.g., the query of type compatibility to abstract super types, the query of geometric parameters, and other (type-dependent) queries.

The rule parameter is an already existing child of O that is to be incorporated as element of O. If a rule fails, E is not incorporated.

Example: The automatic generation of components such as mounting rails that are required for fastening add-on parts, can be implemented via *NEW_ELEMENT* or *CREATE_ELEMENT*.

REMOVE_ELEMENT

The rules of the REMOVE_ELEMENT reason are called for an O object *before* an E element is deleted. The corresponding interaction is the removal of objects in general, e.g., by operations such as cutting or deleting.

The rule parameter is a reference to the already existing element E of O that is to be deleted. The rule can fail if other elements in O depend upon E. If a rule fails, E is not deleted.

Example: A mattress box as an element of a bed cannot be deleted as long as it contains bed frame, mattresses, head sections, back panels, etc.

5.2 Selection Rules

PICK

The rules of the PICK reason are called *after* an object was chosen or selected. The corresponding interaction is the selection of an object in general, e.g., in a direct-manipulative way (2D/3D interaction) or via a graphical user interface. The rule parameter is of type *Float[3]* und indicates the local coordinates at which the object was selected.

PICK rules can be defined to generate a special feedback, e.g., the change in material characteristics or the geometry. Such a feedback is independent from the general feedback which is provided by the OFML runtime environment. In addition, random actions can be triggered by a PICK rule, e.g., the display of object properties within the graphical user interface or the change of the global state.

UNPICK

The rules of the UNPICK reason are called *after* an object was deselected, e.g., by selecting another object. The rule parameter is not defined.

UNPICK rules are generally a reversal of the corresponding PICK rules. For example, the feedback generated by the PICK rule can be reset.

5.3 Move Rules

TRANSLATE

The rules of the *TRANSLATE* reason are called *after* an O object was moved. The corresponding interaction is the translatory move of objects via direct or indirect manipulation. The rule parameter is the local position of O before the move.

The translatory move of an object can be controlled at random through the definition of the *TRANSLATE* rules, e.g.:

- homogenous or inhomogenous rasterization,
- limitation to a range,
- initiation of a collision detection with corresponding correction of the position,
- snapping to objects or positions.

The different possibilities can be combined within a single rule, for example, to enable multidimensional moves. In addition, the father can be called within the rule and the rule functionality can be delegated to it.

Moreover, O can adapt itself to the new position at random. This can refer to local properties such as geometry or the properties of children, e.g., the position of a child relative to its O father. The vector that results from the new and old position can be used to derive directional information and, if required, applied.

Example: The shelves of a carcass cabinet can be moved in a grid of 32 mm, starting at a height of 80 mm. The maximum fitting height results from the inner height of the carcass cabinet minus 80 mm.

ROTATE

The rules of the *ROTATE* reason are called *after* an O object was rotated. The corresponding interaction is the rotary move of objects via direct or indirect manipulation. The rule parameter is the local orientation of O along the employed rotary axis before the rotation.

The rotary move of an object can be controlled at random through the definition of the *ROTATE* rules, e.g.:

- homogenous or inhomogenous rasterization,
- limitation to a range,
- initiation of a collision detection and corresponding correction of the orientation,
- snapping to objects or positions.

The different possibilities can be combined within a single rule, for example, to enable a rasterized rotation within a certain range. In addition, the father can be called within the rule and the rule functionality can be delegated to it.

Analogous to the *TRANSLATE* rule, an object can adapt itself at random to the new orientation.

Example: The door of a carcass cabinet can be opened at an angle from 0 to 90 degrees. At angles of 10 degrees or less, a snapping to 0 degrees is performed automatically. At angles of 80 degrees or more, a snapping to 90 degrees is performed automatically. The snapping behavior can be used to simulate the latching at the end positions.

SPATIAL_MODELING

The rules of the SPATIAL_MODELING reason are called *after* an O object was moved indirectly, i.e., shifted or rotated. An indirect move takes place if an ancestor (father, grandfather, etc.) was panned or rotated. A match of O can take place again. The rule parameter is undefined.

Example: The door handles of a construction kit within a shelf plan are placed dependent upon the fitting height of the construction kit. At a height of less than 1.40 m, it is placed at the top end of the door. Otherwise, it is placed at the bottom end. This adjustment can be implemented automatically by using a *SPATIAL_MODELING* rule.

5.4 Persistence Rules

The persistence rules serve for the conversion of the instance variables from a representation that is used at runtime to a persistent representation and vice versa. This includes, above all, the conversion of object references to values such as *String* or *Int* that can be stored and restored. Furthermore, especially the **_EVAL* rules can be used for adapting stored scenes by initializing instance variables accordingly that did not exist so far.

The definition of persistence rules is required only in exceptional cases.

The rule parameter of persistence rules is undefined.

START_DUMP

The rules of the START_DUMP reason are called *before* the generation of a persistent representation of the *O* object, e.g., within the scope of scenes/object saving or a clipboard operation (e.g., cutting, copying). After processing the rules, the instance variables must be available in a storable representation.

FINISH_DUMP

The rules of the FINISH_DUMP reason are called *after* the generation of a persistent representation of the *O* object and its children. After processing the rules, the instance variables must be available again in the representation that is required for the normal operating mode.

START_EVAL

The rules of the START_EVAL reason are called *before* the processing of a persistent representation of the *O* object, e.g., within the scope of loading a scenes/object saving or a clipboard operation (e.g., inserting). The call is performed immediately following the generation of the *O* object and before the assignment of attributes, children, etc.

FINISH_EVAL

The rules of the FINISH_EVAL reason are called *after* the generation of a persistent representation of the *O* object and its children. After processing the rules, the instance variables must be available in the representation that is required for the normal operating mode.

Example: With a certain roll-container type, the new version can be used to optionally configure an espagnolette. Consequently, this type defines an additional instance variable in the new version that describes by means of a symbol whether the espagnolette is desired or not. A *FINISH_EVAL* rule can be used to ensure that saves of the old version can be post-initialized in this connection.

5.5 Other Rules

SENSOR

The rules of the *SENSOR* reason are called if any M object was moved directly. The rule parameter is a reference to M.

Sensory objects, i.e., objects with at least a *SENSOR* rule, can autonomously respond to changes of the environment.

Example: The door of a room opens automatically if an object is located in a circumcircle of 5 m.

TIMER

The rules of the *TIMER* reason are called if the time interval defined in the respective rule signature expired at least once. The number of passed intervals (typically 1 for time intervals that are not too small) is passed on to the rule(s) as parameter. An instance (or a type) with at least a *TIMER* rule is time-dependent. By generating and removing time-dependent children, a dynamic indirect time dependence of an object can be implemented.

Example: An instance of type clock shows the current time. A *TIMER* rule is used for updating.

INTERACTOR

The rules of the *INTERACTOR* reason are called for the father of the interactor if an attempt to select an interactor is made. They typically serve to activate the interactor (Section F.1).

The selected interactor is transferred as reference as the rule parameter.

Example: Designs can be mounted to an organizational wall at different positions. If interactors are defined for these positions, the user can interactively select the desired mounting point.

Chapter 6

Global functions

6.1 Formatted Output

Some of the functions described below use special character strings to control the formatting. The format character string contains two types of components: regular characters that are accepted in the output without change, and formatting sequences that control the conversion of one of the following arguments in each case. Every formatting sequence begins with the character `%` and ends with a formatting character. The following optional characters can be used between the character `%` and the conversion character in the sequence indicated here:

- Control characters (in random order) that modify the conversion:

  `-` The converted argument is left-aligned.

  `+` The number is always indicated with a sign.

  space If the first character is not a sign, a space is used as prefix.

  `0` Numbers are filled with zeros up to the width of the field.

  `\#` Generates an alternative form of the conversion, dependent upon the formatting character (see below). For `o`, the first character is a zero. For `x` or `X`, `0x` or `0X` are prefixed for a result different than zero. For `e`, `E`, `f`, `g`, and `G`, the output always contains a decimal point; for `g` and `G`, zeros at the end are not suppressed.

- A number that specifies the minimum field width. The number of characters output is at least equal to the number of characters indicated, and more if required (i.e., characters will never be cut off). If the converted argument is shorter, it is filled up to the width of the field. Alignment and fill characters are dependent upon the formatting and control characters.
- A period that separates the field width and the accuracy.
- A number with the following meaning: For `e`, `E` or `f` the number of places behind the decimal point. For `g` or `G` the number of significant digits. For integer values, the minimum number of digits to be output. In the remaining cases, the number indicates the maximum number of characters that are output by a character string.

In each case, * can be indicated as field width or accuracy, so that the value is determined by the next or the next two arguments that must be of type `Int`.

Table 6.1 explains the formatting characters. A character that follows % and is not a formatting character, represents an error.

Character	Argument type	Formatting
d, i	Int	decimal with sign
o	Int	octal without sign, leading zero optional
x, X	Int	hexadecimal without sign, 0x, 0X optional, for abcdef for x or ABCDEF for X
u	Int	decimal without sign
c	Int	single character (Section 3.2.1)
s	String	character string
f	Float	decimal as [-]*mmm.ddd*, accuracy determines the number of *d*, default: 6, no decimal point for 0
e, E	Float	decimal as [-]*mmm.ddd*e±*xx* or [-]*mmm.ddd*E±*xx* accuracy determines the number of *d*, default: 6, no decimal point for 0
g, G	Float	corresponds to %e, %E if exponent is smaller than −4 or not smaller than accuracy, otherwise %f. zero and decimal point are not issued at the end.
%	-	issues %

Table 6.1: Formatting Character

6.2 oiApplPaste()

- *oiApplPaste(pFather(MObject), pName(Symbol)) → Int*

  The function evaluates the clipboard of the application and generates a new object as child of *pFather*. The local name of the new object is specified by *pName*. If an object with the resulting global name already exists or if the clipboard of the application is empty, a runtime error occurs. If *NULL* is set for *pName*, a valid name is automatically selected. The return value of the function is 1 if an object could be generated, otherwise 0.

  The state of the clipboard does not change.

  Note: The clipboard of the application is implemented by the runtime environment and does not have any reference to the global OFML clipboard which can be manipulated or evaluated using the *oiCopy()*, *oiCut()*, and *oiPaste()* function.

6.3 oiClone()

- *oiClone(pSrc(MObject), pDest(String)) → MObject*

The function generates an identical copy of the *pSrc* object under the global name *pDest* and returns the corresponding object reference. If an immediately preceding object under the name *pDest* exists, it causes a runtime error.

The state of the OFML clipboard does not change.

6.4 oiCollision()

- *oiCollision(pObject1(MObject), pObject2(MObject))* → *Int*

  The function checks the collision between two objects *pObject1* and *pObject2*. The polygons of the geometric basic primitives are the atomic element of the collision check. In the case of the parametric primitives *OiRotation*, *OiSweep* and *OiSurface*, these polygons result from the definition coordinates or areas. That is, the actual mapping to a piece by piece linear approximation is not taken into account.

  In the case of a collision, 1 is returned, otherwise 0.

  The function always delivers 1 if *pObject1* is an ancestor (father, grandfather, etc.) or a successor (child, grandchild, etc.) of *pObject2* and vice versa.

6.5 oiCopy()

- *oiCopy(pObject(MObject))* → *Void*

  The function writes an adequate description of *pObject* to the global OFML clipboard.

  The existing state of the clipboard is lost.

  Since the OFML clipboard is a global data structure, corresponding operations must follow each other directly. Otherwise, the correctness of the operations cannot be guaranteed.

6.6 oiCut()

- *oiCut(pObject(MObject))* → *Void*

  The function writes an adequate description of *pObject* to the global OFML clipboard and then deletes object *pObject*.

  The existing state of the clipboard is lost.

  Since the OFML clipboard is a global data structure, corresponding operations must follow each other directly. Otherwise, the correctness of the operations cannot be guaranteed.

6.7 oiDialog()

- *oiDialog(pDialog(Symbol), pIcon(Symbol), pMessage(String))* → *Symbol*

This function causes the reaction of the user to a modal dialog that is generated by the OFMLruntime environment.

The *pDialog* parameter specifies the dialog through one of the following symbols. The possible return values are listed in parentheses.

- *@OK* - Confirmation (*@OK*).
- *@OK_CAN* - Confirmation or Cancel (*@OK*, *@CANCEL*).
- *@ABT_IGN* - Abort or Ignore (*@ABORT*, *@IGNORE*).
- *@YES_NO_CAN* - Yes or No or Cancel (*@YES*, *@NO*, *@CANCEL*).
- *@YES_NO* - Yes or No (*@YES*, *@NO*).

If no valid value is transferred for *pDialog*, no dialog is started and the return value is *@INVALID_DIALOG*.

In addition, the *pIcon* parameter specifies the visual representation of the dialog. It can be executed through a corresponding icon and is always binding, independent of the value of *pDialog*. The value range of *pIcon* is specified as follows:

- *@NONE* - No display of a special character.
- *@STOP* - Display of a stop character.
- *@QUESTION* - Display of a question mark.
- *@WARNING* - Display of an exclamation mark.
- *@INFO* - Display of an information sign (a small *i* in a circle).

If no valid value is transferred for *pIcon*, no dialog is started and the return value is *@INVALID_ICON*.

The *pMessage* parameter specifies the message to be output. If the first character of the *pMessage* string is a `@`, the string is considered a reference that is triggered by an access to an external database (Appendix D).

pMessage must be either a valid string according to the basic syntax (Chapter 3) or a vector. In the first case, umlauts are not permitted. Specifying a vector is used for the formatted output. In this case, the first element is the format character string (Section 6.1), the remaining elements are the arguments to be formatted. If the first element of the vector starts with `@`, the format character string is read from the external database, as indicated above. If no valid value is transferred for *pMessage*, an empty character string is output in the dialog.

The return value is a symbol that describes the selected answer in accordance with the aforementioned alternatives.

6.8 oiDump2String()

- *oiDump2String(pObj(MObject))* $\rightarrow$ *String*

  The function delivers the (implementation-dependent) dump representation of the transferred instance.

It can be used together with the *oiReplace()* function to store and restore object states which may be used, for example, in problem cases for implementing undo-capable operations.

6.9 oiExists()

- *oiExists(pName(String))* $\rightarrow$ *Int*

  The function checks the existence of the object whose absolute name is transferred as string in the *pName* parameter. If the object exists, the return value is 1, otherwise 0. The existence check may be necessary since accessing a non-existing object will cause a runtime error.

6.10 oiGetDistance()

- *oiGetDistance(pPosition(Float[3]), pDirection(Float[3]))* $\rightarrow$ *Float*

  The function determines the first point of intersection of a beam whose origin lies in the world coordinate point *pPosition* and runs alongside the normed vector *pDirection*, with the objects of the scene. The return value is the distance along the beam to the first intersection or -1 if no intersection is found.

6.11 oiGetNearestObject()

- *oiGetNearestObject(pPosition(Float[3]), pDirection(Float[3]))* $\rightarrow$ *MObject*

  The function determines the first encountered object while tracing a beam whose origin lies in the world coordinate point *pPosition* and runs alongside the normed vector *pDirection*. The return value is a reference to the first encountered object or *NULL* if no object was found.

6.12 oiGetRoots()

- *oiGetRoots()* $\rightarrow$ *MObject[]*

  The function determines the root objects available in the scene.

6.13 oiGetStringResource()

- *oiGetStringResource(pStr(String), pLanguage(String), ...)* $\rightarrow$ *String*

  The function delivers the text stored in an external resource file for the transferred text resource in the specified language or the text resource if no text could be found for the resource or an invalid value was transferred for the language.

  If an additional optional parameter is given, it specifies an instance. The text is searched in the name space of this instance if the text resource is not fully qualified (see Appendix D).

6.14 oiLink()

- *oiLink(pURL(String)) → Void*

  The function loads the file specified by the string *pURL*. The current scene can be replaced by a new scene or another document in the result.

6.15 oiOutput()

- *oiOutput(pLevel(Symbol), pMessage(String)) → Void*

  This function causes the output of a text message through the OFML runtime environment. The output should be implemented through a modal dialog. The *pLevel* symbol describes the category of the output as follows:

  - *@MESSAGE* - Output of a message.
  - *@WARNING* - Output of a warning.
  - *@ERROR* - Output of an error message.
  - *@FATAL* - Output of an error message. After quitting the modal dialog, the runtime environment must terminate.

  If the first character of the *pMessage* string is a `@`, the string is considered a reference that is triggered by an access to an external database (Appendix D).

  pMessage must be either a valid string according to the basic syntax (Chapter 3) or a vector. In the first case, umlauts are not permitted. Specifying a vector is used for the formatted output. In this case, the first element is the format character string (Section 6.1), the remaining elements are the arguments to be formatted. If the first element of the vector starts with `@`, the format character string is read from the external database, as indicated above.

  If `"::ofml::app::@none"` is transferred as the message, the application does not output a message.

  Note: It can be used, for example, to indicate an "error condition" via `oiOutput(@ERROR, "::ofml::app::@none")` of the application for which the OFML already performed a dialog (*oiDialog()* function) (e.g., a Cancel dialog during *checkAdd()*, see *Complex* interface), and no additional message is desired.

6.16 oiPaste()

- *oiPaste(pFather(MObject), pName(Symbol)) → MObject*

  The function evaluates the global OFML clipboard and generates a new object as child of *pFather*. The local name of the new object is specified by *pName*. If an object with the resulting global name already exists, a runtime error occurs. If *NULL* is set for *pName*, a

valid name is automatically selected. The return value of the function is a reference to the created object.

The state of the clipboard does not change.

Since the OFML clipboard is a global data structure, corresponding operations must follow each other directly. Otherwise, the correctness of the operations cannot be guaranteed.

6.17 oiReplace()

- *oiReplace(pObj(MObject), pDump(String))* → *Void*

  The function replaces the transferred instance by an object whose dump representation is contained in the transferred buffer.

  An (implementation-dependent) dump representation can be created with the *oiDump2String()* function.

6.18 oiSetCheckString()

- *oiSetCheckString(pString(String)* → *Void*

  The function sets a string that must be verified by the respective OFML runtime environment. This string, which is usually set in persistent OFML scene representations, can be used for checking the consistency or validity of a scene representation. An incorrect string in this sense must result in the cancellation of the read operation of the persistent scene representation.

6.19 oiTable()

- *oiTable(pRequest(Symbol), pArgs(List))* → *List*

  The *oiTable* function implements the read access to data from an external relational database (Appendix D).

  The desired table operation is specified via *pRequest* parameter and the corresponding arguments via *pArgs*. The following listing shows the possible operations and corresponding arguments:

```
@openTbl     List of TableEntry
@closeTbl    List of TableID
@readTE      List of TableEntry
```

  A *TableEntry* is transferred as `[tableID, attributeList]` vector, where *tableID* is indicated as string and *attributeList* as a list of *TableAttributen.*

  A *TableAttribute* is transferred as `[name, isPrimKey, isKey, type, value, format]` vector, where *name* is indicated as string, *isPrimKey* and *isKey* as (boolean) Int, *type* as symbol, *value* as object according to *type* and *format* as string.

  The following attribute types and corresponding format strings are defined:

```
Int:    type = @i, format = maximum number of places
Float:  type = @f, format = total number of places.number of places
                                                behind decimal point
String: type = @s, format = maximum length
```

A TableID string consists of three components separated by spaces:

- the designator for the type of database which is always `"FTXT"` (text file with fixed field length) in OFML,
- the localization path for the database to be used, and
- the actual name of the concerned table.

A table must be opened before the first access. The `@openTbl` operation opens a table for reading where its structure is defined via the list of attributes. The *isPrimKey* and *isKey* attributes of *TableAttribute* are evaluated only during the table definition in `@openTbl`.

The `@closeTbl` operation requests a list of TableIDs of the tables to be closed as parameter. All system resources used for the management of these tables are released; afterwards, an access is no longer possible.

The `@readTbl` operation is used for reading table rows. The TableEntries transferred in the list specify which rows should be read. `@readTbl` is the only operation where the transferred TableEntries do not have to feature a complete row description according to the table definition. Rather, only those TableAttributes must be given that should serve as key for the access. All rows whose values in the specified columns correspond to the specified values in the transferred TableAttributes are delivered for a transferred TableEntry. Thus, the `@readTbl` operation presents a simple, table-specific search function. Only if the TableEntry contains a TableAttribute marked as primary key during the table definition can a unique result be expected. If several TableEntry objects are transferred to `@readTbl`, the corresponding query is performed for each object and the results are linked in the returned list.

Chapter 7

Geometric types

This chapter describes the hierarchy of the geometrically oriented types. A geometric instance can be viewed directly through its geometry and, if required, through its children. The entities of geometric types are generally located at the lowest level in hierarchical product models.

7.1 OiGeometry

Description

- The abstract type *OiGeometry* is the base type for the geometrically-oriented types described below. *OiGeometry* may not be instantiated directly. The derivation of application-specific types of *OiGeometry* is allowed. An implementation of an application-specific derived type is carried out here through parameterization and aggregation of one or several *OiGeometry*-compatible entities.

  The entities of the *OiGeometry* type can feature a child with the local name *geo* for the implementation of the geometry. This name may not be assigned elsewhere. In addition, the potential existence of *geo* should be observed with iterations on the list of children.

- **Interface(s):** Base, material

Initialization

- *OiGeometry(pFather(MObject), pName(Symbol))*

  The function initializes an indirect instance of *OiGeometry* type. Initially, the selection option is deactivated. The initial material category is *@ANY*. In the normal case, it must be changed accordingly via *setMatCat()*. The initial alignment is not defined uniformly and is determined by the respective primitive.

Methods

- *setMatCat(pCat(Symbol))* $\rightarrow$ *Void*

  The function overwrites the initial material category *@ANY* or a category previously set with the value of *pCat*.

- *setAlignment(pAlignment(Symbol[3]))* $\rightarrow$ *Void*

  The function allows for the alignment of the geometry with respect to the local axes. The following symbols for element of *pAlignment* are supported (in each case with respect to one of the three axes):

  - *@C* – The origin of the object is located in the middle of the local delimiting volume.
  - *@I* – The origin of the object is located in the minimum of the local delimiting volume.
  - *@A* – The origin of the object is located in the maximum of the local delimiting volume.

  A subsequent change of the geometry does not lead to adaptation in accordance with the specified alignment. Children that may exist can lead to unexpected results when the alignment is set.

7.2 OiBlock

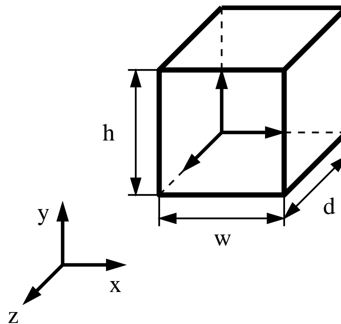

Figure 7.1: The geometric type *OiBlock*

Description

- *OiBlock* represents an orthogonal quboid that begins in the origin of the local coordinate system and expands accordingly along the positive axes of the local coordinate system. The dimensions of the quboid can be changed after its generation.
- **Super type:** *OiGeometry*

Initialization

- *OiBlock(pFather(MObject), pName(Symbol), pDimensions(Float[3]))*

  The function initializes an instance of the *OiBlock* type. The initial dimensions of the quboid are indicated by a vector of three positive numbers.

Methods

- *setDimensions(pDimensions(Float[3]))* $\rightarrow$ *Void*

  The function sets the dimensions of the quboid. *pDimensions* must be a vector of three positive numbers.

  If required, an adaptation of the alignment must be performed afterwards (*setAlignment()*).

- *getDimensions()* $\rightarrow$ *Float[3]*

  The function delivers the current dimensions of the quboid.

7.3 OiCylinder

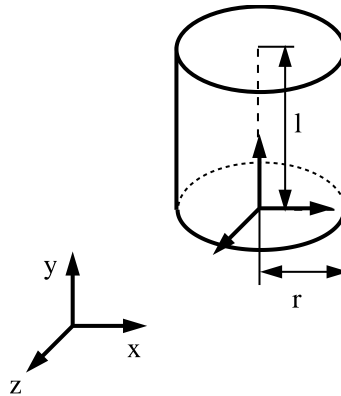

Figure 7.2: The Geometric Type *OiCylinder*

Description

- *OiCylinder* represents a closed homogenous cylinder that begins in the origin of the local coordinate system and expands centered along the positive y-axis of the local coordinate system. The dimensions of the cylinder can be changed after its generation.

- **Super type:** *OiGeometry*

Initialization

- *OiCylinder(pFather(MObject), pName(Symbol), pLength(Float), pRadius(Float))*

  The function initializes an instance of the *OiCylinder* type. The initial dimensions of the cylinder are indicated by the parameters length and radius. Only positive numbers are allowed.

Methods

- *setLength(pLength(Float))* → *Void*

  The function sets the length of the cylinder. *pLength* must be a positive number.

  If required, an adaptation of the alignment must be performed afterwards (*setAlignment()*).

- *getLength()* → *Float*

  The function delivers the current length of the cylinder.

- *setRadius(pRadius(Float))* → *Void*

  The function sets the radius of the cylinder. *pRadius* must be a positive number.

  If required, an adaptation of the alignment must be performed afterwards (*setAlignment()*).

- *getRadius()* → *Float*

  The function delivers the current radius of the cylinder.

7.4 OiEllipsoid

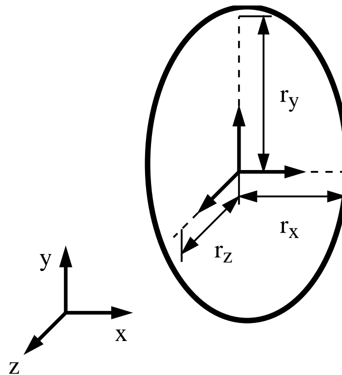

Figure 7.3: Der geometric type *OiElliposid*

Description

- *OiEllipsoid* represents a homogenous ellipsoid whose center is located in the origin of the local coordinate system and expands accordingly to all six sides of the local coordinate system. The dimensions of the ellipsoid can be changed after its generation.
- **Super type:** *OiGeometry*

Initialization

- *OiEllipsoid(pFather(MObject), pName(Symbol), pDimensions(Float[3]))*

  The function initializes an instance of the *OiEllipsoid* type. The initial dimensions of the ellipsoid are indicated by a vector of three positive numbers.

Methods

- *setDimensions(pDimensions(Float[3]))* → *Void*

  The function sets the dimensions of the ellipsoid. *pDimensions* must be a vector of three positive numbers.

  If required, an adaptation of the alignment must be performed afterwards (*setAlignment()*).
- *getDimensions()* → *Float[3]*

  The function delivers the current dimensions of the ellipsoid.

7.5 OiFrame

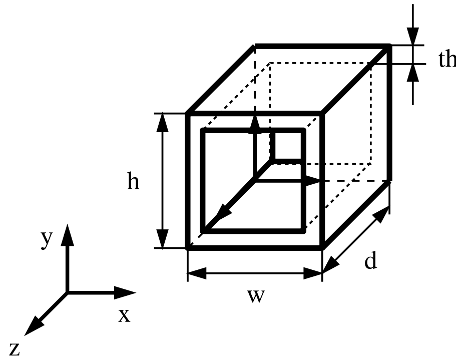

Figure 7.4: Der geometric type *OiFrame*

Description

- *OiFrame* represents a frame that begins in the origin of the local coordinate system and expands accordingly along the positive axes of the local coordinate system. An orthogonal volume is subtracted from the body in the local x-y plane. The thickness of the frame in x and y direction is identical. The following must always apply to the dimensions w, h in x and y direction and the x/y thickness th: $w, h > 2 * th$. The dimensions of the frame can be changed after its generation.
- **Super type:** *OiGeometry*

Initialization

- *OiFrame(pFather(MObject), pName(Symbol), pDimensions(Float[3]), pThickness(Float))*

  The function initializes an instance of the *OiFrame* type. The initial outer dimensions of the frame are indicated by a vector of three positive numbers. The initial thickness of the frame in the local x and y direction is indicated by a positive number.

Methods

- *setDimensions(pDimensions(Float[3])) → Void*

  The function sets the outer dimensions of the frame. *pDimensions* must be a vector of three positive numbers.

  If required, an adaptation of the alignment must be performed afterwards (*setAlignment()*).
- *getDimensions() → Float[3]*

  The function delivers the current outer dimensions of the frame.
- *setThickness(pThickness(Float)) → Void*

  The function sets the frame thickness in the local x and y direction. *pThickness* must be a positive number.
- *getThickness() → Float*

  The function delivers the actual frame thickness in the local x and y direction.

7.6 OiHole

Description

- *OiHole* implements circular or rectangular openings in circular or rectangular areas. This allows for simulating boolean operations, especially the subtraction in special cases. However, no actual subtraction in the sense of a boolean operation takes place. The real purpose of *OiHole* consists of generating the areas for the combination of circular outlines rectangular hole and rectangular outline circular hole. *OiHole* does not implement outside areas along

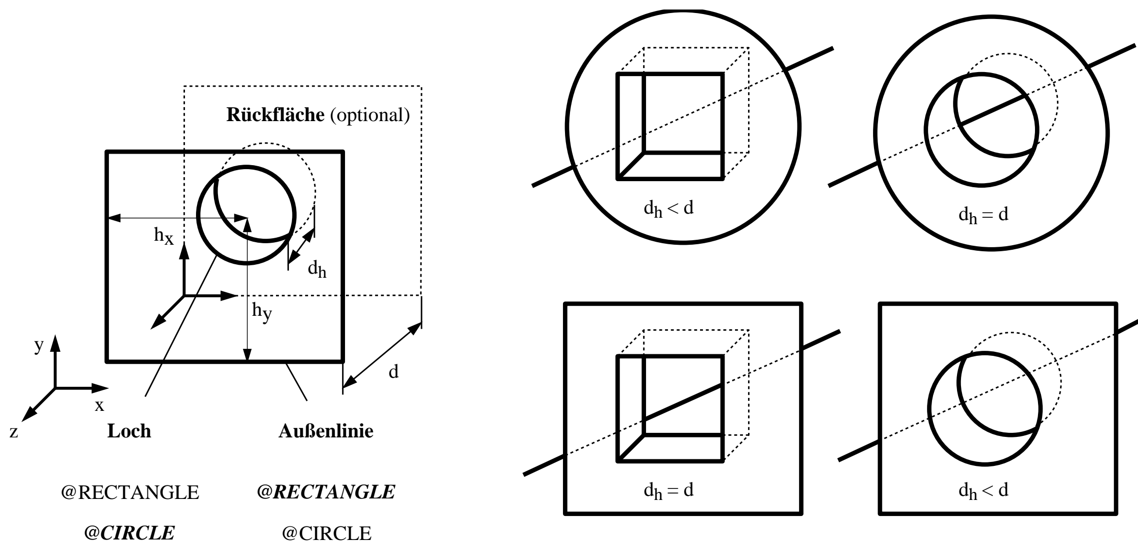

Figure 7.5: The geometric type *OiHole*

the outline in the local z direction. *OiHole* entities begin in the origin of the local coordinate system and expand according to the outer dimensions along the positive x-, y- and z-axis.

- **Super type:** *OiGeometry*

Initialization

- *OiHole(pFather(MObject), pName(Symbol), pOMode(Symbol), pODim(Float[3]), pBack(Int), pHMode(Symbol), pHDim(Float[3]), pHOffset(Float[2])*

  The function initializes an instance of the *OiHole* type. The following specific parameters must be supplied:

  – The *pOMode* parameter indicates the mode of the outline. Permissible implementations for *pOMode* are:
    * *@RECTANGLE* – The outline corresponds to a rectangle.
    * *@CIRCLE* – The outline corresponds to a circle.

  – The *pODim* parameter determines the outer dimensions of the body, consisting of width w, height h and depth d. All dimensions must be positive numbers. In the case of a circular outline, the width also determines the height.

  – The *pBack* parameter indicates whether the outer back plane is generated (*pBack == 1*) or not (*pBack == 0*).

  – The *pHMode* parameter indicates the mode of the hole. Permissible implementations for *pHMode* are:
    * *@RECTANGLE* – A rectangular hole is generated.

    * *@CIRCLE* – A circular hole is generated.

  – The *pHDim* determines the dimensions of the hole consisting of width w_h, height h_h and depth d_h. All dimensions must be positive numbers. In the case of a circular hole, the width also determines the height. The hole width w_h must be smaller than total width w. The hole height h_h must be smaller than the total height h. The hole depth d_h may not be larger than the total depth d. If it is smaller, a hole back area is generated automatically.

  – The *pHOffset* parameter defines the offset from the center of the hole to the local origin of the primitive. The hole may not go beyond the area of the outer volume.

7.7 OiHPolygon

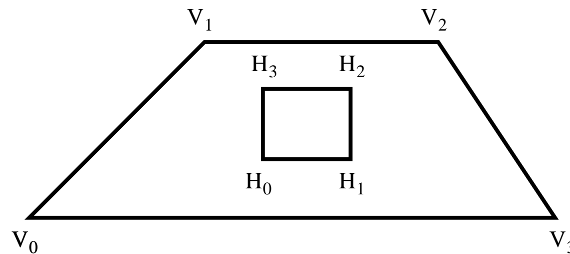

Figure 7.6: The geometric type *OiHPolygon*

Description

- *OiHPolygon* represents a one-sided, simple, planar and convex polygon from which a number of simple, planar and convex polygons can be cut out.

- **Super type:** *OiGeometry*

Initialization

- *OiHPolygon(pFather(MObject), pName(Symbol), pMosaic(Int), pOutline(Float[3][]), pHoles(Float[3][][]))*

  The function initializes an instance of the *OiHPolygon* type. The *pMosaic* parameter controls the tessalation of the resulting polygon net. If *pMosaic* obtains the value 0, it results in a triangulation. Otherwise, the number of internal polygons is minimized. *pOutline* describes the outer polygon in clockwise direction. *pHoles* is an optional empty vector of polygons that each describes a cutout. These polygons must be defined in counterclockwise direction.

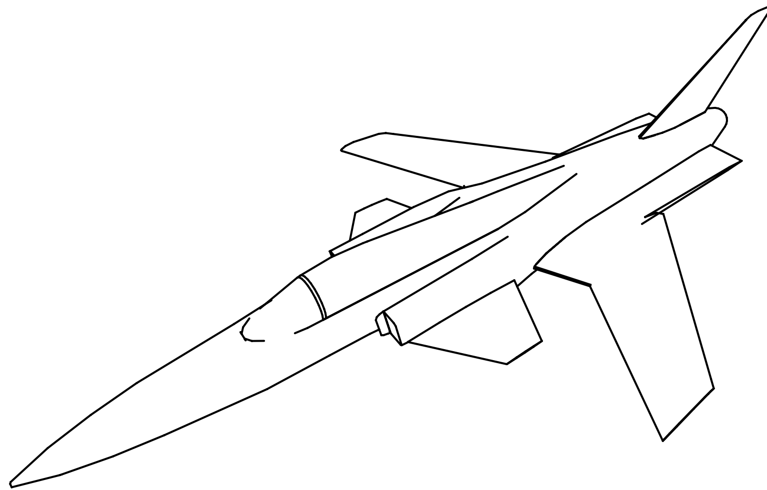

Figure 7.7: The geometric type *OiImport*

7.8 OiImport

Description

- *OiImport* imports an external file in a geometric format. Provided that it does not contain any materials, a material can be set via the *Material* interface.

  OiImport optionally supports exactly one heavily resolution-reduced geometry next to the actual geometry. If it is present, it can be used for the speed-optimized presentation. However, its use is dependent upon the respective presentation software.

- **Super type:** *OiGeometry*

Initialization

- *OiImport(pFather(MObject), pName(Symbol), pMode(Symbol), pGeometry(String))*

  The function initializes an instance of the *OiImport* type. *pGeometry* describes the name of a geometry file in form of a simple string without path and extension information, e.g., "wheel." The file type is determined by the *pMode* parameter. Permissible implementations for *pMode* are:

  – *@OFF* – The geometry features the Object File Format.

– *@G3DS* – The geometry features the 3D Studio format.

The optional resolution-reduced geometry file also contains an underscore character at the beginning of the name, e.g., "_wheel."

The data record is loaded in accordance with the definitions for external data (Chapter D) and must be fully qualified.

Methods

- *setScale(pFactor(Float[3]))* → *Void*

  The function allows for the scaling of *OiImport* objects. The elements of the vector *pFactor* must be real, positive numbers. The initial scaling is 1.0 in all three dimensions.

  If required, an adaptation of the alignment must be performed afterwards (*setAlignment()*).

- *getScale()* → *Float[3]*

  The function furnishes the current scaling of the implicit instance.

7.9 OiPolygon

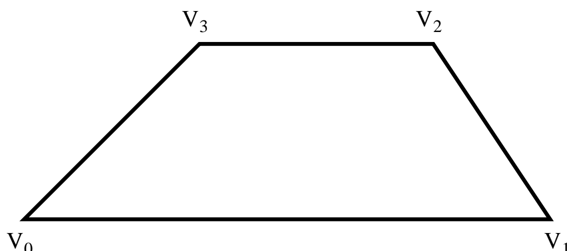

Figure 7.8: The geometric type *OiPolygon*

Description

- *OiPolygon* represents a one-sided, simple, planar and convex polygon. This primitive should be used in exceptional cases only since a number of *OiPolygonen* is extremely inefficient compared to other polygon sets (e.g., on the basis of *OiImport*). In addition, a singular *OiPolygon* does not describe a body which contradicts the general intention of OFML.
- **Super type:** *OiGeometry*

Initialization

- *OiPolygon(pFather(MObject), pName(Symbol), pPoints(Float[3][]))*

  The function initializes an instance of the *OiPolygon* type. The *pPoints* parameter defines a one-sided, simple, planar and convex polygon. The last and first point are automatically connected. The visibility results by means of the right-hand rule. If the curvature of the right hand follows the vertex line, the thumb of the right hand indicates the visible side.

7.10 OiRotation

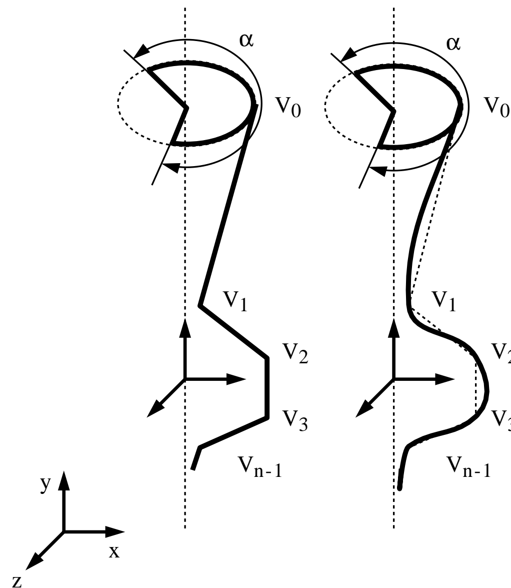

Figure 7.9: The geometric type *OiRotation*

Description

- *OiRotation* describes a solid body that is defined by the planar rotation of a three-dimensional planar curve around an axis. The curve must be defined as follows, according to the right-hand rule: If the thumb of the right hand points towards the direction of rotation, the remaining fingers of the right hand indicate the orientation. Otherwise, an inversion must take place.
- **Super type** *OiGeometry*

Initialization

- *OiRotation(pFather(MObject), pName(Symbol), pMode(Symbol), pAxis(Float[3]), pPoints(Float[3][]), pArc(Float), pUWMode(Symbol[2]), pCMode(Symbol[2]), pFlip(Int))*

  The function initializes an instance of the *OiRotation* type. This requires indicating the following specific parameters:

  - *pMode* specifies whether the body along the definition curve should be smooth (*@SMOOTH*) or not (*@LINEAR*).
  - *pAxis* defines the rotation axis with respect to the local coordinate system.
  - *pPoints* describes the definition curve. However, points on the rotation axis are not allowed.
  - *pArc* sets the angle of rotation of the definition curve. *pArc* must be positive and smaller than or equal to 2π.
  - *pUWMode* defines the openness (*@OPEN*) or compactness (*@CLOSED*) of the body along two curves. *pUWMode[0]* specifies whether the body along the rotation axis is closed. In general, this is the case for bodies with $pArg = 2\pi$. *pUWMode[1]* specifies whether a compactness of the body results with respect to the definition curve (by joining the first and last point). In general, this is not the case.
  - *pCMode* defines the openness (*@OPEN*) or compactness (*@CLOSED*) of the body with respect to two areas. *pCMode[0]* specifies whether possibly resulting interfaces of the body should be closed. This is only necessary for bodies with $pArg = 2\pi$. *pCMode[1]* specifies whether the cap areas of the body should be generated or not.
  - *pFlip* forces an inversion of the sequence in *pPoints* if it features the value 1. Otherwise, the value must be 0.

7.11 OiSphere

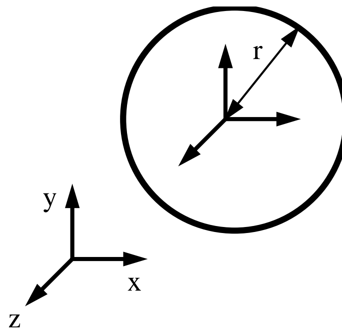

Figure 7.10: The geometric type *OiSphere*

Description

- *OiSphere* represents a homogenous sphere that is centered about the origin of the local coordinate system. The radius of the sphere can be changed after its generation.
- **Super type** *OiGeometry*

Initialization

- *OiSphere(pFather(MObject), pName(Symbol), pRadius(Float))*

  The function initializes an instance of the *OiSphere* type. The initial radius of the sphere is indicated by the positive number *pRadius*.

Methods

- *setRadius(pRadius(Float))* → *Void*

  The function sets the radius of the sphere. *pRadius* must be a positive number.

  If required, an adaptation of the alignment must be performed afterwards (*setAlignment()*).
- *getRadius()* → *Float*

  The function delivers the current radius of the sphere.

7.12 OiSweep

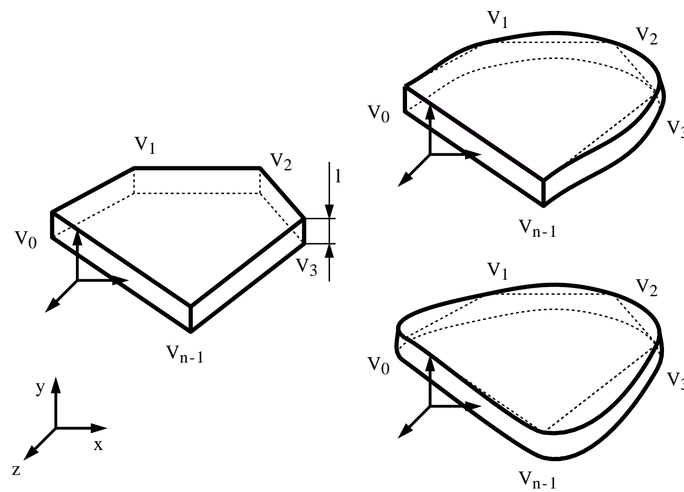

Figure 7.11: The geometric type *OiSweep*

Description

- *OiSweep* describes a solid body that is defined by the planar move of a three-dimensional planar curve along an axis. The curve must be defined as follows, according to the right-hand rule: If the thumb of the right hand points towards the direction of move, the remaining fingers of the right hand indicate the orientation. Otherwise, an inversion must take place.
- **Super type:** *OiGeometry*

Initialization

- *OiSweep(pFather(MObject), pName(Symbol), pMode(Symbol), pAxis(Float[3]), pLength(Float), pPoints(Float[3][]), pUMode(Symbol), pCMode(Symbol[2]), pFlip(Int))*

  The function initializes an instance of the *OiSweep* type. This requires indicating the following specific parameters:

  - *pMode* specifies whether the body along the definition curve should be smooth (*@SMOOTH*) or not (*@LINEAR*).
  - *pAxis* defines the move axis with respect to the local coordinate system.
  - *pLength* sets the length of the body along the move axis. *pLength* must be a positive number.
  - *pPoints* describes the definition curve.
  - *pUMode* defines the openness (*@OPEN*) or compactness (*@CLOSED*) of the body along the definition curve. If *pUMode* = *@OPEN*, end point and start point of *pPoints* are connected by a straight line. Otherwise, an appropriate soft connection occurs.
  - *pCMode* defines the openness (*@OPEN*) or compactness (*@CLOSED*) of the body with respect to two areas. *pCMode[0]* specifies whether the side faces of the body should be closed or not. *pCMode[1]* specifies whether the connection of the last point with the first point should be closed or not.
  - *pFlip* forces an inversion of the sequence in *pPoints* if it features the value 1. Otherwise, the value must be 0.

Methods

- *setLength(pLength(Float))* → *Void*

  The function sets the length of the object along the move axis. *pLength* must be a positive number.

  If required, an adaptation of the alignment must be performed afterwards (*setAlignment()*).

- *getLength()* → *Float*

  The function delivers the current length of the object along the move axis.

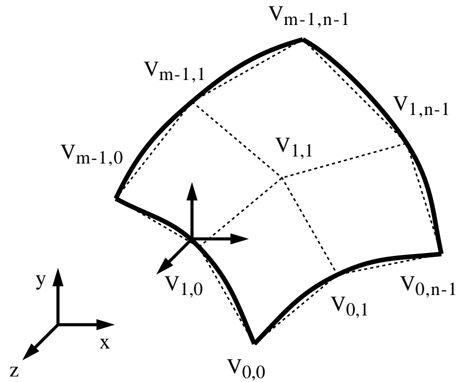

Figure 7.12: The geometric type *OiSurface*

7.13 OiSurface

Description

- *OiSurface* describes a primitive that is defined by a two-dimensional net of three-dimensional supporting points. Here, u and w are the dimensions of the net.
- **Super type:** *OiGeometry*

Initialization

- *OiSurface(pFather(MObject), pName(Symbol), pUDim(Int), pWDim(Int), pPoints(Float[3][pUDim(pWDim]), pUWMode(Symbol[2])*

  The function initializes an instance of the *OiSurface* type. This requires the following specific parameters:

  - *pUDim* defines the u dimension of the net.
  - *pWDim* defines the w dimension of the net.
  - *pPoints* describes an array with the definition points. Within a patch, the right-hand rule indicates the orientation, i.e., if the thumb of the right hand sits on the patch at a right angle, the remaining fingers of the hand indicate the orientation.
  - *pUWMode* defines the openness (*@OPEN*) or compactness (*@CLOSED*) of the primitive along the u and w dimension. If *pUWMode[0] = @OPEN*, no connection of the net in the u direction is made. If *pUWMode[1] = @OPEN*, no connection of the net in the w direction is made.

Chapter 8

Global Planning Types

This chapter describes global, higher-level planning base types. These base types are independent of concrete planning elements (pieces of furniture) and, therefore, also independent of concrete geometric implementations.

The types described here are based on the conceptual model shown in Figure 8.1.

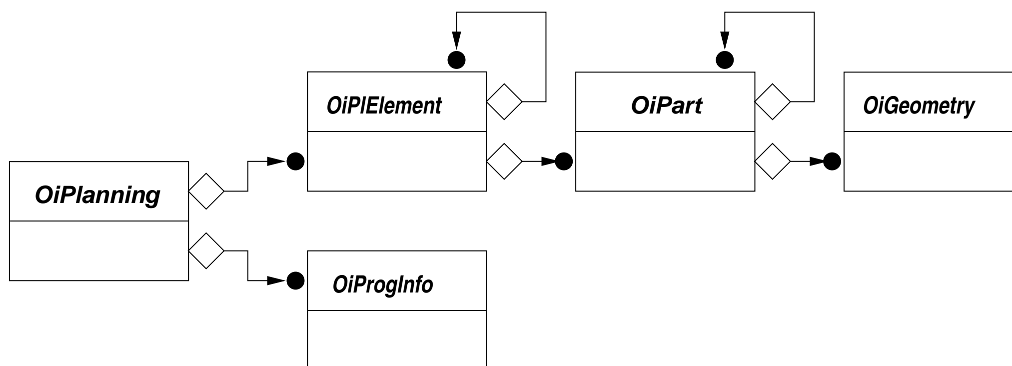

Figure 8.1: Conceptual model of the global planning types

8.1 OiPlanning

Description

- An instance of this type functions as root object of a complete planning hierarchy and implements global planning logics for the elements of the planning (*OiPlElement* type).
- Additional tasks of the global planning object include:

- The definition of a *planning limit* that specifies the space within which the planning elements can be placed.
- The monitoring and handling of the transformation of planning elements for the purpose of avoiding collisions and exceeding planning limits.
- The management and utilization of information about characteristics and requirements of the (furniture) programs to which the elements belong that are represented in the planning (*OiProgInfo* type).
- Use of a product data manager for accessing product data (see also Chapter 9).

- **Interface(s):** Base, Complex, Property, Material

Initialization

- *OiPlanning(pFather(MObject), pName(Symbol))*

  The function initializes an instance of the *OiPlanning* type. Initially, the selection option is deactivated. The initial planning limit is infinite.

Methods

General Methods

- *setLanguage(pLang(String))* → *Void*

  The function specifies the language to be used for subsequent messages and labels. The *pLang* parameter describes the national language through a string in accordance with ISO 639 guidelines.

  Examples:

  - `"de"` – German
  - `"en"` – English
  - `"nl"` – Dutch

- *getLanguage()* → *String*

  The function delivers the language that is currently used for messages and labels.

- *setRegion(pRegion(String))* → *Void*

  The function specifies the sales region, i.e., generally a country, for which the current planning is created. The *pRegion* parameter describes the sales region through a string in accordance with ISO 3166 guidelines (ISO Code 2) or extensions in this connection for the designation of federal states, etc.

  Examples:

  - `"DE"` – Germany

  - "UK" – United Kingdom
  - "NL" – Netherlands

- *getRegion() → String*

  The function delivers the current sales region. If no sales region was specified, the return value is of type *Void*.

- *setProgram(pProgr(Symbol)) → Void*

  The function specifies the currently relevant program.

  Note: Programs are distinguished based on their ID (identification symbol). The program ID must be unique across all manufacturers. For this reason, it starts with a two-digit manufacturer code, followed by the code for the actual program.

  The program ID is used with certain operations for the delegation of functionality to program-specific information or similar objects. The currently relevant program can be determined externally through the runtime environment, or even internally out of the certain context, e.g. out of the association of the currently processed planning element to a program.

- *delegationDone() → Void*

  The function signals the implicit planning instance the successful execution of a functionality delegated to another instance (*Delegat*).

  Note: The function is called by the delegation instance upon successful execution of the delegated functionality.

Error Log

Complex test algorithms that are executed on an object structure can lead to error messages concerning various objects of the structure. Instead of issuing an error message from every test method for the corresponding objects, it is generally desirable to collect these messages and view them together in a dialog. For this purpose, the global planning instance manages a so-called *error log*. The instance or method that initiates a global testing process, generates the data structure to be used for the log of the testing process and passes it on to the *setErrorLog()* function before the execution of the test is delegated to another instance or inherited implementations are called. Implementations of the testing algorithm must first be checked with the *getErrorLog()* function whether a higher-level log was created, in which case the generated messages must be entered in this log and no separate dialog for display may be started. If no higher-level log exists, it must be created before possible delegations can take place, and a dialog for displaying the messages from the log must be started at the end of the testing process. The data structure used for the log can be defined separately for each testing algorithm.

Note: An application example of an error log is located in the *Article* interface under the performance of consistency checks.

- *setErrorLog(pLog(Any))* → *Void*

  The function assigns the implicit planning instance a new data structure for the error log.

- *getErrorLog()* → *Any*

  The function returns the reference to the last data structure for the error log that was assigned by means of the *setErrorLog()* function.

Instance Hierarchy

- *getEnvironment()* → *MObject*

  The function delivers the root object of the hierarchy of the planning environment. The function furnishes a value of the type *Void* if no planning environment exists.

- *getPlElementUp(pObj(MObject))* → *MObject*

  The function traverses the instance hierarchy, beginning with the transferred instance and upward to the root object and delivers the first instance which is of the *OiPlElement* type. If no such instance was located on the traversing path, the function delivers a value of the type *Void*.

- *getTopPlElement(pObj(MObject))* → *MObject*

  The function traverses the instance hierarchy, beginning with the transferred instance and upward to the root object and delivers the top instance which is of the *OiPlElement* type. If no such instance was located on the traversing path, the function delivers a value of the type *Void*.

- *getPropObj(pObj(MObject))* → *MObject*

  The function traverses the instance hierarchy, beginning with the transferred instance and upward to the root object and delivers the first instance that features properties. If no such instance was located on the traversing path, the function delivers a value of the type *Void*.

Planning Limit

- *setBorder(pBorder(Float[2][3])* → *Void*

  The function assigns the implicit planning instance a new value for the (axis-orthogonal) planning limit volume.
  The *pBorder* parameter is a vector consisting of two vectors with three *Float* values each. The first *Float* vector specifies the origin of the acceptable planning volume in world coordinates. The second *Float* vector determines the maximum expansion of the acceptable planning volume along the x-, y- and z-axes. If the spatial location of the planning is not important, a value of the type *Void* may also be transferred instead of the first vector.

- *getBorder()* → *Float[2][3]*

  The function delivers the specification for the planning limit volume that is currently used in the implicit planning instance. Structure and semantics of the return value correspond to the parameter for the *setBorder()* function.

- *checkBorder() → String*

  The function checks whether the planning limit is maintained by the planning elements currently contained in the implicit planning instance.
  In the case of a limit violation, the function delivers a string that contains the corresponding error message. If the limit is maintained, the function delivers a value of the type *Void*.

Management of Program Information

- *addInfoObj(pType(Type), pID(Symbol)) → Void*

  The function adds an instance of the indicated type to the set of program information objects and registers them under the indicated program ID. If a program information object with the indicated program ID already exists, it is removed before the new object is inserted.

  Note: The program information objects are inserted into the instance hierarchy of the planning as non-graphical (thus, not visible) objects. After storing and reloading the planning, these information objects are available immediately.

- *delInfoObj(pID(Symbol)) → Void*

  The function removes the program information object with the indicated program ID.

- *clearInfoObjs() → Void*

  The function removes all program information objects.

- *getInfoIDs() → Symbol[]*

  The function delivers the program IDs of all registered program information objects.

- *getInfo(pID(Symbol)) → MObject*

  The function delivers the program information object with the indicated program ID or a value of the type *Void* if no program information object is registered under the indicated program ID.

Materials

- *Material::getMatCategories() → Symbol[]*
- *Material::getCMaterials(pCat(Symbol)) → Symbol[]*
- *Material::getCMaterial(pCat(Symbol)) → Symbol*
- *Material::setCMaterial(pCat(Symbol), pMat(Symbol)) → Int*
- *Material::getMatName(pMat(Symbol)) → String*

  The standard implementation performs a string conversion of the symbol.

These functions implement the corresponding functions of the *Material* interface by means of delegation to the functions of the program information object under the same name (*OiProgInfo* type) of the currently relevant program (*setProgram()* function).

Element Management and Collision Detection

- *Complex::checkAdd((pType(Type), pObj(MObject), pPos(Float[3]), pParams(Any)) → Float[3]*

  The function checks whether an instance of the indicated type can be inserted as element into the planning and, if positive, delivers a valid position for the element.
  (For more information about the semantics of the function or its parameter, see the *Complex* interface.)
  First, the function calls the function of the program information object under the same name (*OiProgInfo* type) for the program to which the instance belongs that was transferred in the *pObj* parameter. Afterwards, a program-independent check is performed in accordance with a global planning logic, if required. This requires a call to the *doCheckAdd()* hook function.

- *doCheckAdd(pType(Type), pObj(MObject), pPos(Float[3]), pParams (Any)) → Float[3]*

  The function checks independent of concrete furniture programs whether an instance of the indicated type can be inserted as element into the planning and, if positive, delivers a valid position for the element. The semantics of the parameters corresponds to the *checkAdd()* function. The standard implementation achieves an attaching of the new element to the right of the existing planning.

  Note: The function is called by *checkAdd()* and, in contrast to *checkAdd()*, can be redefined in subtypes where the function of the same name of the immediate super type should be called in the case of non-applicability of the special planning logic that is implemented by the subtype.

- *Complex::checkChildColl(pObj(MObject), pExclObj(MObject)) → MObject*

  The function checks whether a collision of the transferred (child) instance with other objects is present. If the *pExclObj* argument contains a non-empty set of objects, they are excluded from the collision check. The function first checks for collision with the children of the implicit instance. Objects for which the *isValidForCollCheck()* hook function delivers the value 0 are excluded from the collision check. Before and after this check, the *startCollCheck()* or *finishCollCheck()* functions of the program information object are called for the program to which instance belongs that is transferred in the *pObj* parameter. Afterwards, the function of the same name of the root object of the hierarchy of the planning environment is called (if it exists and its type implements the *Complex* interface). The return value is the first located object with which the transferred instance collides or a value of type *Void* if no collision was detected or if the collision detection is deactivated.

- *Complex::isValidForCollCheck(pObj(MObject)) → Int*

  This function implements the corresponding function of the *Complex* interface by means of delegation to the function of the same name of the program information object (*OiProgInfo* type) for the program to which the instance belongs that is transferred in the *pObj* parameter.

- *Complex::checkElPos(pEl(MObject), pOldPos(Float[3])) → Int*

  The function implements the corresponding function of the *Complex* interface by means of collision detection and planning limit monitoring (*checkChildColl()* and *checkBorder()* functions).

Element Transformations

- *elemTranslation(pEl(MObject), pOldPos(Float[3]))* → *Void*

  The function handles an (already performed) translation of the indicated planning element in the following way.

  - First, the general acceptability of the translation of the planning element is checked (see also *translateValid()* function of the *OiPlElement* type).
  - If the translation is acceptable on principle, the *translated()* function of the transferred planning element is now called (see also the *OiPlElement* type).
  - If the *translated()* function returned the value 0, the implicit instance now checks the validity of the current position of the planning element (collision detection, adherence to planning limit, and others). If necessary, a correction of the current position may occur before the translation with the aid of the position of the planning element transferred in the *pOldPos* parameter.

  If the indicated object is not an instance of the *OiPlElement* type, the function is without effect.

  Note: The function is called from the *TRANSLATE* rule of the transferred planning element (*OiPlElement* type).

- *elemRotation(pEl(MObject), pOldRot(Float))* → *Void*

  The function handles the rotation of the indicated planning element in the following way.

  - First, the general acceptability of the rotation of the planning element is checked (see also *rotateValid()* function of the *OiPlElement* type).
  - If the rotation is acceptable on principle, the *rotated()* function of the transferred planning element is now called (see also the *OiPlElement* type).
  - If the *rotated()* function returned the value 0, the implicit instance now checks the validity of the current rotary angle of the planning element (collision detection, adherence to planning limit, and others). If necessary, a correction of the current angle may occur before the rotation with the aid of the rotary angle of the planning element transferred in the *pOldRot* parameter.

  If the indicated object is not an instance of the *OiPlElement* type, the function is without effect.

  Note: The function is called from the *ROTATE* rule of the transferred planning element (*OiProgInfo* type).

- *checkPosition(pEl(MObject), pPos(Float[3]), pAngles(Float[3]))* → *Float[2][3]*

  The function checks whether the indicated position and the indicated rotary angle (per axis) are allowed for the transferred planning element.

  The return value is a vector consisting of two vectors of three *Float* values each. The first vector specifies an acceptable position, the second the rotary angle (per axis). The returned

values may deviate from the desired values transferred in the parameters to a certain extent to prevent collisions and other conflicts. If the planning element cannot be placed at the desired position on principle (or in its vicinity), the return vector contains a value of the *Void* type instead of a position information.

Note: The function is called by the runtime environment during a dialog for explicit positioning of a planning element.

Product Data Management

- *setPDManager(pType(Type))* → *Void*

  The function generates an instance of the indicated type to be used as global product data manager (*OiPDManager* type). If a product data manager instance already exists, it is removed first.

- *getPDManager()* → *MObject*

  The function delivers the global product data manager instance or a value of the type *Void* if such an instance is not registered.

- *article2Class(pArticle(String))* → *String*

  The function delivers the name of the type that models the article which was specified based on its article number, or a value of the type *Void* if no assignment could be found for the article. If a global product data manager instance is registered, the query to this instance is delegated.

- *addProductDB(pType(Type), pID(Symbol), pPath(String), pProgList(Symbol[])* → *MObject*

  The function generates an instance of the transferred type (subtype of *OiProductDB*) and registers it with the global product data manager under the indicated ID. The file system path of the directory that contains the files of the product database is transferred in the *pPath* parameter (relative to the root directory of the runtime environment). The additional *pProgList* parameter specifies the programs (IDs) that are represented in the database. If a product database is already registered under the indicated ID, the list of programs of the product database is expanded, if required.

  The return value is the reference to the (generated) product database instance.

  Note: The function achieves the same effect as the function of the *OiPDManager* type of the same name.

Miscellaneous

- *checkConsistency()* → *Int*

  The function checks the consistence and completeness of the planning. If required, corrections or additions are performed or error messages are generated. The function delivers True if the planning is consistent, otherwise False.

First, the function calls the function of the same name on all registered entities of *OiProgInfo* and then on all children of the *OiPlElement* type. The result of the check is False if the check was not successful for at least one instance. The check uses the error log written with the *checkConsistency()* function in the *Article* interface.

Note: The function is usually called by the runtime environment before the creation of an order list.

- *checkObjConsistency(pObj(MObject) → Int*

  The function performs a consistence check on the transferred instance of the *OiPlElement* or *OiPart* type. Besides the call of the *checkConsistency()* method on the transferred instance, the function can perform additional actions, e.g., displaying or removing a visual feedback with incorrect articles or adding or removing an entry in the global error log.

- *doSpecial(pPID(Symbol), pOp(Symbol), pArgs(Any)) → Any*

  Using the transferred arguments, the function performs the indicated operation concerning the program specified by the transferred ID. If a program information object is registered for the program, the function is delegated to it (*OiProgInfo* type). The return value is dependent upon the operation.

  Note: The function can be used for expanding the functionality of a planning system without having to expand the interface between runtime environment and global planning instance.

Rules

- *REMOVE_ELEMENT(pValue(Symbol)) → Int*

  The rule prevents the removal of planning elements whose *removeValid()* function delivers the value 0.

8.2 OiProgInfo

Description

- Entities of this type manage information about a (furniture) program (Appendix I) or implement program-specific functions if requested by the global planning instance (*OiPlanning* type).
- **Interface(s):** *MObject*, *Property*

Initialization

- *OiProgInfo(pFather(MObject), pName(Symbol), pPID(Symbol))*

  The function initializes an instance of the *OiProgInfo* type with the indicated program ID. The program ID cannot be changed later.

Methods

General Methods

- *getID() → Symbol*

  The function delivers the ID of the program for which the implicit instance responsible is.

- *getPlanning() → MObject*

  The function delivers the root object of the planning hierarchy (t) if this is an instance of the *OiPlanning* type, otherwise a value of the type *Void*.

- *checkConsistency() → Int*

  The function performs a program-specific consistence check. It is called by the global planning instance of the *OiPlanning* type with a global consistence check before the check on the planning elements is performed.

- *doSpecial(pOp(Symbol), pArgs(Any)) → Any*

  Using the transferred arguments, the function performs the indicated operation (see also the function of the same name of the *OiPlanning* type).

Materials

- *getMatCategories() → Symbol[]*
- *getCMaterials(pCat(Symbol)) → Symbol[]*
- *setCMaterial(pCat(Symbol), pMat(Symbol)) → Int*
- *getCMaterial(pCat(Symbol)) → Symbol*
- *getMatName(pMat(Symbol)) → String*

  The standard implementation performs a string conversion of the symbol.

These functions represent program-specific versions of the corresponding functions of the *Material* interface and are called by the functions of the same name of the global planning instance (*OiPlanning* type).

Element Management and Collision Detection

- *Complex::checkAdd((pType(Type), pObj(MObject), pPos(Float[3]), pParams(Any)) → Float[3]*

  The function checks whether an instance of the indicated type can be inserted as neighboring element of the program element that is transferred in the *pObj* parameter, into the planning and, if positive, delivers a valid position for the element.
  The semantics of the function or its parameter correspond to the function of the same name of the global planning instance (*OiPlanning* type) and is called by it.

- *isValidForCollCheck(pObj(MObject))* → *Int*

  The function delivers 1 if the indicated program element should be considered in the collision check, otherwise 0. The function is called by the function of the same name of the global planning instance (*OiPlanning* type).

- *startCollCheck(pObj(MObject))* → *Void*

  The function performs required actions before the indicated program element is checked for collision with other planning elements.
  It is called by the *checkChildObj()* function of the global planning instance (*OiPlanning* type).
  The standard implementation of the function does not perform any actions.

- *finishCollCheck(pObj(MObject))* → *Void*

  The function performs required actions after the indicated program element was checked for collision with other planning elements.
  It is called by the *checkChildObj()* function of the global planning instance (*OiPlanning* type).
  The standard implementation of the function does not perform any actions.

8.3 OiPlElement

Description

- Entities of the *OiPlElement* type represent independent elements of a planning.
- Planning elements cooperate in a defined way with the global planning instance (*OiPlanning* type).
- **Interface(s):** Base, Complex, Material, Property, Article

Initialization

- *OiPlElement(pFather(MObject), pName(Symbol))*

  The function initializes an instance of the *OiPlElement* type.

  The initialization function of concrete subtypes must define the properties of the planning element. This is accomplished either by means of the *setupProperty()* function of the *Property* interface or through delegation to the *setupProps()* function of the global product data manager (*OiPDManager* type) if such an instance exists.

Methods

General Methods

- *getPlanning()* → *MObject*

  The function delivers the root object of the planning hierarchy (*t*) if this is an instance of *OiPlanning*, otherwise a value of the type *Void*.

- *setPlProgram()* → *Void*

  The function assigns the inherent program specified through the *getProgram()* function as the currently relevant program by means of the *setProgram()* function (*OiPlanning* type) to the global planning instance.

Spatial Model

- *setWidth(pWidth(Float))* → *Void*

  The function assigns an explicit value for the width expansion to the implicit instance.

- *Complex::getWidth()* → *Float*

  The function furnishes the width of the implicit instance. If a value was assigned for the width during or after the initialization by means of the *setWidth()* method, it is returned, otherwise the width of the delimiting volume determined by the *getLocalBounds()* method (*Base* interface).

- *setHeight(pHeight(Float))* → *Void*

  The function assigns an explicit value for the height expansion to the implicit instance.

- *Complex::getHeight()* → *Float*

  The function furnishes the height of the implicit instance. If a value was assigned for the height during or after the initialization by means of the *setHeight()* method, it is returned, otherwise the height of the delimiting volume determined by the *getLocalBounds()* method (*Base* interface).

- *setDepth(pDepth(Float))* → *Void*

  The function assigns an explicit value for the depth expansion to the implicit instance.

- *Complex::getDepth()* → *Float*

  The function furnishes the depth of the implicit instance. If a value was assigned for the depth during or after the initialization by means of the *setDepth()* method, it is returned, otherwise the depth of the delimiting volume determined by the *getLocalBounds()* method (*Base* interface).

- *setOrigin(pOrigin(Float[3]))* → *Void*

  The function assigns an offset of the reference origin with respect to the minimum of the local delimiting volume to the implicit instance.

- *getOrigin()* → *Float[3]*

  The function delivers the offset of the reference origin of the implicit instance with respect to the minimum of the local delimiting volume. If a value was assigned for the offset during or after the initialization by means of the *setOrigin()* method, it is returned, otherwise it is determined with the help of the *getLocalBounds()* method of the *Base* interface.

Materials

In each of the following functions, a call of *setPlProgram()* is performed at the beginning.

- *Material::getMatCategories() → Symbol[]*

  It delivers the list of material categories currently defined for the implicit instance (for detailed specifications see the *Material* interface). The standard implementation delivers a value of type *Void*.

- *Material::getAllMatCats() → Symbol[]*

  It furnishes the list of *all* material categories that are potentially definable for the implicit instance. The standard implementation delivers the return value of the *getMatCategories()* function.

- *Material::getCMaterials(pCat(Symbol)) → Symbol[]*

  It delivers the list of all materials that are applicable within the transferred material category for the implicit instance (for detailed specifications see the *Material* interface). The standard implementation delivers the return value of the function of the same name of the global planning instance if its type is *OiPlanning*, otherwise a value of type *Void*.

- *Material::getCMaterial(pCat(Symbol)) → Symbol*

  The function furnishes the material currently assigned to the implicit instance in the transferred material category or a value of the *Void* type if the implicit instance does not currently belong to the transferred material category. The standard implementation delivers the return value of the function of the same name of the global planning instance if its type is *OiPlanning*, otherwise a value of type *Void*.

  Note: Concrete subclasses must overwrite this method in such a way that the material currently set in the object for this category is delivered. The standard implementation (call of the father object) must be performed only if a material in this category has not been assigned with explicit assignment to the object.

- *Material::getMatName(pMat(Symbol)) → String*

  The function furnishes the material name to the transferred material or a value of the *Void* type for the implicit instance if the material is unknown. The standard implementation delivers the return value of the function of the same name of the global planning instance if its type is *OiPlanning*, otherwise the return value of the function of the same name of the father instance if its type implements the *Material* interface.

Element Generation

- *isElemCatValid(pCat(Symbol)) → Int*

  The function delivers 1 if instances of the indicated category can be added to the implicit instance as elements, otherwise 0.

  The standard implementation delivers 0.

Example: The *isElemCatValid()* function of a type of table on which instances of the *@TOP_ELEM* category can be placed, must deliver 1 for this category.

Note: After checking for special categories during an overwriting of the function in derived types, the inherited function must be called so that 1 is also delivered for the categories which are allowed by super types.

- *Complex::checkAdd((pType(Type), pObj(MObject), pPos(Float[3]), pParams(Any)) → Float[3]*

  The function checks whether an instance of the indicated type can be inserted as element into the planning and, if positive, delivers a valid position for the element.

  The standard implementation implements the placement of elements of the *@TOP_ELEM* category if the *isElemCatValid()* function delivers 1 for this category. The *getWidth()*, *getHeight()*, *getDepth()*, and *getOrigin()* functions are used for this purpose.

- *getPDistance() → Float*

  The function delivers the desired initial distance to the previous element.
  The standard implementation delivers the minimum x-value of the local delimiting volume.

  Note: The function can be used within the *checkAdd()* function of the father instance.
  The value delivered by the function can be queried by the user through a dialog before the new element is inserted. The subtypes must make a corresponding set function available for this purpose.

- *getWallOffset() → Float*

  The function delivers the desired initial distance to a wall element in front of which the implicit instance should be placed.
  The standard implementation delivers 0.01 minus the minimum z-value of the local delimiting volume.

  Note: The function can be used within the *checkAdd()* function of the father instance.
  The value delivered by the function can be queried by the user through a dialog before the new element is inserted. The subtypes must make a corresponding set function available for this purpose.

- *onCreate(pRot(Float), pObj(MObject), pParams(Any)) → Void*

  The function can be called after the generation of the implicit instance and ends the overall process of interactively inserting the instance into the planning. The requested rotation with respect to the y-axis in positive direction, the neighboring element to which the implicit instance was added, and an additional random parameter are transferred. The standard implementation implements the required rotation with respect to the y-axis in positive direction.

  Note: The function is used for setting object properties that cannot be performed during the object generation (in the *initialize()* function) for lack of knowledge of the planning context. The function is

usually set together with the appropriate arguments during the *checkAdd()* of *OiPlanning* by means of calling *setMethod()* (*Complex* interface).

Element Control

- *elRemoveValid(pObj(MObject))* → *Int*

  The function returns True if the transferred child instance can be removed. The function is called in REMOVE_ELEMENT rules in addition to the *removeValid()* function of the *Base* interface for the instance that is to be removed.

  Example: A cupboard unit subplanning can use this, for example, to remove elements from the left and right side only.

  The standard implementation delivers True.

- *isElOrderSubPos(pObj(MObject))* → *Int*

  The function delivers True if the transferred child instance may not appear as a subitem in an order list.

  Example: The function can be used in algorithms for generating order lists to move certain items to specific positions in the order list.

  The standard implementation delivers True.

Product Data

- *Article::getArticleSpec()* → *String*

  The standard implementation of the function delegates the query to the *class2Article()* function of the global product data manager (*OiPDManager* type), if such an instance exists.

- *Article::setArticleSpec(pSpec(String))* → *Void*

  The standard implementation does not perform any actions.

- *Article::getArticleParams()* → *Any*

  The standard implementation delivers a value of type *Void*.

- *Article::getArticlePrice(pLanguage(String))* → *Any[]*

  The standard implementation of the function delegates the query to the function of the same name of the global product data manager (*OiPDManager* type), if such an instance exists.

- *Article::getArticleText(pLanguage(String), pForm(Symbol))* → *String[]*

  The standard implementation of the function delegates the query to the function of the same name of the global product data manager (*OiPDManager* type), if such an instance exists.

- *Article::getArticleFeatures(pLanguage(String))* → *Any*

  The standard implementation of the function delegates the query to the function of the same name of the global product data manager (*OiPDManager* type), if such an instance exists.

Consistence Check

- *Article::checkConsistency()* → *Int*

  The function checks the consistence and completeness of the planning element and is called by *OiPlanning::checkConsistency()*. If required, corrections or additions are performed or error messages are generated.

  The standard implementation delegates to the function of the same name of the global product data manager (*OiPDManager* type).

Child Transformations

- *elemTranslation(pEl(MObject), pOldPos(Float[3]))* → *Void*

  The function handles a (completed) translation of the transferred child instance of the *OiPlElement* or *OiPart* type.

  For instances of the *OiPlElement* type, this is accomplished in the same way as the *OiPlanning* function of the same name. For instances of the *OiPart* type, the *onTranslate()* function is called.

  Note: The function is called from the *TRANSLATE* rule of the transferred child instance.

- *elemRotation(pEl(MObject), pOldRot(Float))* → *Void*

  The function handles the (completed) rotation of the transferred child instance of the *OiPlElement* or *OiPart* type.

  For instances of the *OiPlElement* type, this is accomplished in the same way as the *OiPlanning* function of the same name. For instances of the *OiPart* type, the *onRotate()* function is called.

  Note: The function is called from the *ROTATE* rule of the transferred child instance.

Translation and Rotation

- *translateValid(pOldPos(Float[3]))* → *Int*

  The function delivers 1 if the planning element can be moved from the transferred old position to the new current position, otherwise it delivers 0.
  The standard implementation delivers 1.

  Note: The function is used within the *elemTranslation()* function of the global planning instance (*OiPlanning* type).

- *translated(pOldPos(Float[3]))* → *Int*

  The function is called by the *elemTranslation()* function of the global planning instance to enable the planning element to individually react to its translation. The return value is 1 if the function has already checked the validity of the new position, otherwise it is 0.

- *rotateValid(pOldPos(Float)) → Int*

  The function delivers 1 if the planning element can be rotated from the transferred old rotary angle to the new current rotary angle, otherwise it is 0.
  The standard implementation delivers 1.

  Note: The function is used within the *elemRotation()* function of the global planning instance (*OiPlanning* type).

- *rotated(pOldPos(Float)) → Int*

  The function is called by the *elemRotation()* function of the global planning instance to enable the planning element to individually react to its rotation. The return value is 1 if the function has already checked the validity of the new rotary angle, otherwise it is 0.

Rules

- *REMOVE_ELEMENT(pValue(Symbol)) → Int*

  The rule prevents the removal of child instances whose *removeValid()* function delivers False or for which the *elRemoveValid()* function delivers False.

- *TRANSLATE(pValue(Float[3])) → Int*

  The rule delegates the handling of the translation to the *elemTranslation()* function of the global planning instance (*OiPlanning* type).

- *ROTATE(pValue(Float)) → Int*

  The rule delegates the handling of the rotation to the *elemRotation()* function of the global planning instance (*OiPlanning* type).

8.4 OiPart

Description

- The *OiPart* type is the basic type for functional base types that are used as components in planning elements (*OiPlElement* class).
- **Interface(s):** Base, Complex, Material, Property, Article

Initialization

- *OiPart(pFather(MObject), pName(Symbol))*

  The function initializes an instance of the *OiPart* type.

Methods

General Methods

- *getPlanning() → MObject*

  The function delivers the root object of the planning hierarchy (*t*) if this is an instance of *OiPlanning*, otherwise a value of the type *Void*.

Spatial Model

- *setWidth(pWidth(Float)) → Void*

  The function assigns an explicit value for the width expansion to the implicit instance.

- *Complex::getWidth() → Float*

  The function furnishes the width of the implicit instance. If a value was assigned for the width during or after the initialization by means of the *setWidth()* method, it is returned, otherwise the width of the delimiting volume determined by the *getLocalBounds()* method (*Base* interface).

- *setHeight(pHeight(Float)) → Void*

  The function assigns an explicit value for the height expansion to the implicit instance.

- *Complex::getHeight() → Float*

  The function furnishes the height of the implicit instance. If a value was assigned for the height during or after the initialization by means of the *setHeight()* method, it is returned, otherwise the height of the delimiting volume determined by the *getLocalBounds()* method (*Base* interface).

- *setDepth(pDepth(Float)) → Void*

  The function assigns an explicit value for the depth expansion to the implicit instance.

- *Complex::getDepth() → Float*

  The function furnishes the depth of the implicit instance. If a value was assigned for the depth during or after the initialization by means of the *setDepth()* method, it is returned, otherwise the depth of the delimiting volume determined by the *getLocalBounds()* method (*Base* interface).

- *setOrigin(pOrigin(Float[3])) → Void*

  The function assigns an offset of the reference origin with respect to the minimum of the local delimiting volume to the implicit instance.

- *getOrigin() → Float[3]*

  The function delivers the offset of the reference origin of the implicit instance with respect to the minimum of the local delimiting volume. If a value was assigned for the offset during or after the initialization by means of the *setOrigin()* method, it is returned, otherwise it is determined with the help of the *getLocalBounds()* method of the *Base* interface.

Materials

- *Material::getMatCategories() → Symbol[]*

  It delivers the list of material categories currently defined for the implicit instance (for detailed specifications see the *Material* interface). The standard implementation delivers a value of type *Void*.

- *Material::getAllMatCats() → Symbol[]*

  It furnishes the list of *all* material categories that are potentially definable for the implicit instance. The standard implementation delivers the return value of the *getMatCategories()* function.

- *Material::getCMaterials(pCat(Symbol)) → Symbol[]*

  It delivers the list of all materials that are applicable within the transferred material category for the implicit instance (for detailed specifications see the *Material* interface). The standard implementation delivers the return value of the function of the same name of the father instance if its type implements the *Material* interface, otherwise a value of type *Void*.

- *Material::getCMaterial(pCat(Symbol)) → Symbol*

  The function furnishes the material currently assigned to the implicit instance in the transferred material category or a value of the *Void* type if the implicit instance does not currently belong to the transferred material category. The standard implementation delivers the return value that was delivered by the function of the same name from the father instance.

- *Material::getMatName(pMat(Symbol)) → String*

  The function furnishes the material name to the transferred material or a value of the *Void* type for the implicit instance if the material is unknown. The standard implementation delivers the return value of the function of the same name of the global planning instance if its type is *OiPlanning*, otherwise the return value of the function of the same name of the father instance if its type implements the *Material* interface.

Element Generation

- *isElemCatValid(pCat(Symbol)) → Int*

  The function delivers 1 if instances of the indicated category can be added to the implicit instance as elements, otherwise 0.

  The standard implementation delivers 0.

  Example: The *isElemCatValid()* function of a type of table on which instances of the *@TOP_ELEM* category can be placed, must deliver 1 for this category.

  Note: After checking for special categories during an overwriting of the function in derived types, the inherited function must be called so that 1 is also delivered for the categories which are allowed by super types.

- *Complex::checkAdd((pType(Type), pObj(MObject), pPos(Float[3]), pParams(Any)) → Float[3]*

  The function checks whether an instance of the indicated type can be inserted as element into the planning and, if positive, delivers a valid position for the element.

  The standard implementation implements the placement of elements of the *@TOP_ELEM* category if the *isElemCatValid()* function delivers 1 for this category. The *getWidth()*, *getHeight()*, *getDepth()*, and *getOrigin()* functions are used for this purpose.

Element Control

- *elRemoveValid(pObj(MObject)) → Int*

  The function returns True if the transferred child instance can be removed. The function is called in REMOVE_ELEMENT rules in addition to the *removeValid()* function of the *Base* interface for the instance that is to be removed.

  Example: A cupboard unit subplanning can use this, for example, to remove elements from the left and right side only.

  The standard implementation delivers True.

- *isElOrderSubPos(pObj(MObject)) → Int*

  The function delivers True if the transferred child instance may not appear as a subitem in an order list.

  Example: The function can be used in algorithms for generating order lists to move certain items to specific positions in the order list.

  The standard implementation delivers True.

Product Data

- *Article::getArticleSpec() → String*

  The standard implementation of the function delegates the query to the *class2Article()* function of the global product data manager (*OiPDManager* type), if such an instance exists.

- *Article::setArticleSpec(pSpec(String)) → Void*

  The standard implementation does not perform any actions.

- *Article::getArticleParams() → Any*

  The standard implementation delivers a value of type *Void*.

- *Article::getArticlePrice(pLanguage(String)) → Any[]*

  The standard implementation of the function delegates the query to the function of the same name of the global product data manager (*OiPDManager* type), if such an instance exists.

- *Article::getArticleText(pLanguage(String), pForm(Symbol)) → String[]*

  The standard implementation of the function delegates the query to the function of the same name of the global product data manager (*OiPDManager* type), if such an instance exists.

- *Article::getArticleFeatures(pLanguage(String)) → Any*

  The standard implementation of the function delegates the query to the function of the same name of the global product data manager (*OiPDManager* type), if such an instance exists.

Consistence Check

- *Article::checkConsistency() → Int*

  The function checks the consistence and completeness of the planning element and is called by *OiPlanning::checkConsistency()*. If required, corrections or additions are performed or error messages are generated.

  The standard implementation delegates to the function of the same name of the global product data manager (*OiPDManager* type).

Translation and Rotation

- *onTranslate(pOldPos(Float[3])) → Void*

  The function is called by the translation rule and is used in derived classes to implement a specific behavior for a move.

- *onRotate(pOldRot) → Void*

  The function is called by the rotation rule and is used in derived classes to implement a specific behavior for a rotation.

Rules

- *REMOVE_ELEMENT(pValue(Symbol)) → Int*

  The rule prevents the removal of child instances whose *removeValid()* function delivers False or for which the *elRemoveValid()* function delivers False.

- *TRANSLATE(pValue(Float[3])) → Int*

  If the father instance is of *OiPlElement* type, the handling of the translation is delegated to its *elemTranslation()* function, otherwise to the *onTranslate()* function of the implicit instance.

- *ROTATE(pValue(Float)) → Int*

  If the father instance is of *OiPlElement* type, the handling of the rotation is delegated to its *elemRotation()* function, otherwise to the *onRotate()* function of the implicit instance.

8.5 OiUtility

Description

- The *OiUtility* type is the basic type for types that are used for specific tasks, e.g., for the representation and storage of the global data of a program.
- **Interface(s):** MObject

Initialization

- *OiUtility(pFather(MObject), pName(Symbol))*

  The function initializes an instance of the *OiUtility* type.

8.6 OiPropertyObj

Description

- The *OiPropertyObj* type is the basic type for types that are used for specific tasks and feature properties.
- **Super type:** OiUtility
- **Interface(s):** MObject (inherited), Property

Initialization

- *OiPropertyObj(pFather(MObject), pName(Symbol))*

  The function initializes an instance of the *OiPropertyObj* type.

Methods

General Methods

- *getPlanning()* → *MObject*

  The function delivers the root object of the planning hierarchy (t) if this is an instance of *OiPlanning*, otherwise a value of the type *Void*.
- *isCutable()* → *Int*

  See the function of the same name of the *Base* interface.
- *removeValid()* → *Int*

  See the function of the same name of the *Base* interface.

8.7 OiOdbPlElement

Description

- The *OiOdbPlElement* type is the basic type for planning elements whose geometries are generated by the ODB.
- **Super type:** OiPlElement
- **Interface(s):** Base, Complex, Material, Property, Article
- The most important function of the *OiOdbPlElement* class consists of providing the ODB information in form of a hash table returned by *getOdbInfo()*. It contains an entry for the ODB name and an additional entry for each property, where the property key is also used as key in the hash table. Thus, the values of the properties are available in the ODB for the parameterization of the geometries.

Initialization

- *OiOdbPlElement(pFather(MObject), pName(Symbol))*

  The function initializes an instance of the *OiOdbPlElement* type analogous to *OiPlElement*. In addition, the translation in x- and z-direction is enabled and blocked in y-direction.

Methods

General Methods

- *setOdbType(pArticle(String))* → *Void* (protected)

  The function sets the ODB name on the basis of the transferred article. The ODB name is determined by calling the *article2ODBType()* method on the PD manager. If this method does not deliver an ODB name, a name is generated by default. This name consists of the series of the article and the article designation, where all characters in the article designation except for letters, numbers and underscore sign are replaced by `_XX`. `XX` represents the hexadecimal representation of the code of the respective character. The underscore sign is replaced by two succeeding underscores.

- *setArticleSpec(pArticle(String))* → *Void*

  The function assigns a new base article number to the implicit instance. This causes the initialization of the ODB information and the subsequent generation of the geometries by means of calling *createOdbChildren(@NEW)*.

- *getArticleSpec()* → *String*

  The function delivers the name of the article (base article number) to which the implicit instance corresponds or a value of type *Void* if no article specification is available for the implicit instance.

  The name of the article is determined by means of the ODB name.

- *propsChanged(pPKeys(Symbol[]), pDoChecks(Int)) → Int*

  If the list of property keys *pPKeys* is not empty, the *createOdbChildren(@INCR)* function is called to regenerate the geometries for this article. In the current implementation, the *pDoChecks* parameter is ignored and the return value is always 1.

- *setPropValue(pKey(Symbol), pValue(Any)) → Int*

  First, the *setPropValue()* method of the *OiPlElement* top class is called. Next, an iteration is performed on all direct children of the implicit instance and the *setPropValue()* method is called for every child that is either an *OiPlElement* or an *OiPart*.

- *getOdbInfo() → Hash*

  The function returns a hash table with the ODB parameters. It contains the ODB name determined from the base article number and the current property values.

- *createOdbChildren(pVal(Symbol)) → Void*

  The function controls the generation of the child objects via ODB. Dependent upon the *pVal* parameter, the child objects are either generated completely new (*@NEW* and *@RULE*) or adapted to the new ODB information (*@INCR*). Usually, either *@NEW* or *@INCR* is transferred as argument. The *@RULE* argument is intended for use in the *FINISH_EVAL* rule.

  Note: The current implementation always performs a complete regeneration of the child objects.

  With a regeneration of the child objects, those child objects that are not part of the article, such as accessories, are deleted and not displayed again. Since the regeneration of child objects can lead to random changes in the geometry of the article, a general discussion of this problem is not possible.

Translation and Rotation

- *translated(pOldPos(Float[3])) → Int*

  The function is called following every translation and checks whether the object at the current position causes a collision. If this is the case, the function attempts to determine a new position on the line between old and current position that is as close as possible to the current position and on which the object does not collide.

Chapter 9

Types for Product Data Management

On principle, *OFML* allows the complete description of logics and dependencies of types without external data records. Still, a specification of product properties via external data records could be desirable for various reasons, e.g., to be able to use an existing data array directly in *OFML*.

For this purpose, *OFML* defines a powerful, generic product data management interface. The concept of a product data management (see also Figure 9.1) conceives that there are a number of external product databases (possibly in different data formats), but they are managed by a global product data manager and communicate with this manager via a uniform generic interface. For each concrete data format (but not for each external product database), a special interface type must be implemented (subtypes of *OiProductDB*) that takes over the interpretation of the data format on the *OFML* level.

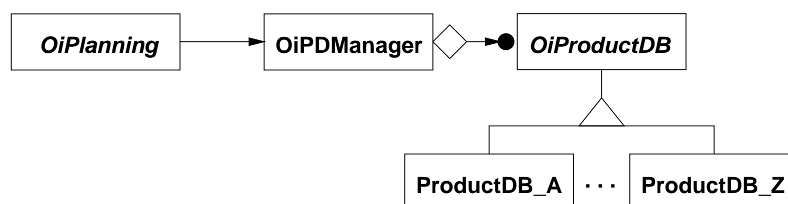

Figure 9.1: Conceptual model of the product data management types

Example: A concrete example is the data format that is generated from a SAP/R3 system while maintaining the physical basic format. The data is distributed over several tables that are interlinked. The relational knowledge is stored in expressions of the ABAP/4 language. Using the implementation of a respective subtype of *OiProductDB*, this format can now be read in on the *OFML* level. This includes the implementation of an ABAP/4 parser.

9.1 OiPDManager

Description

- An instance of the *OiPDManager* type manages a set of external product databases (*OiProductDB* type) and allows access to the product data stored in these databases.
- Exactly one instance of this type exists for each planning. This instance is referred to as product data manager. It is generated by means of the *setPDManager()* function of the *OiPlanning* type.
- The product data manager also manages the assignment of types to articles and vice versa.
- **Interface(s):** *MObject*

Initialization

- *OiPDManager(pFather(MObject), pName(Symbol))*

  The function initializes an instance of the *OiPDManager* type.

Methods

Management of the Product Databases

- *addProductDB(pType(Type), pID(Symbol), pPath(String), pProgList(Symbol[]) → MObject*

  The function generates an instance of the transferred type (subtype of *OiProductDB*) and registers it under the indicated ID. The file system path of the directory that contains the files of the product database is transferred in the *pPath* parameter (relative to the root directory of the runtime environment). The additional *pProgList* parameter specifies the programs (IDs) that are represented in the database. If a product database is already registered under the indicated ID, the list of programs of the product database is expanded, if required.

  The return value is the reference to the (generated) product database instance.

- *delProductDB(pID(Symbol)) → Void*

  The function deletes and removes the product database with the indicated ID from the set of registered product databases.

- *clearProductDBs() → Void*

  The function deletes and removes all registered product databases.

- *getPDB_IDs() → Symbol[]*

  The function delivers the IDs of all registered product databases.

- *getProductDB(pID(Symbol)) → MObject*

  The function delivers the product database instance that is registered under the indicated ID or a value of type *Void*, if such an ID is not available.

- *getPDBFor(pObj(MObject))* → *MObject*

  The function delivers the product database instance that is responsible for the transferred planning element or a value of type *Void*, if such a product database is not found. The responsibility results from the program association of the planning element.

- *getProgPDB(pPID(Symbol))* → *MObject*

  The function delivers the product database instance that is responsible for the program specified by the transferred ID or a value of type *Void*, if such a product database is not found.

Assignment of Types to Articles

- *article2Class(pArticle(String))* → *String*

  The function delivers the name of the type that models the article which was specified based on its article number, or a value of the type *Void* if no assignment could be found.

- *article2Params(pArticle(String))* → *String*

  The function delivers the parameter values for the type that models the article specified by its article number. The return value is a string that contains the presentation of the parameter values stored in the product data. The function delivers a value of type *Void* if no entry for the article was found in the product data.

- *object2Article(pObj(MObject))* → *String*

  The function delivers the name of the article (article number) to which the transferred planning element corresponds.

  The assignment is composed of the program association, the immediate type, and the relevant parameters (see *OiPlElement::getArticleParams()* function) of the planning element.

- *class2Articles(pObj(MObject))* → *String[]*

  The function delivers the list of article (numbers) that are represented by the class of transferred planning elements. The return value is a value of type *Void* if no article assignment for the class exists.

Properties

- *setupProps(pObj(MObject))* → *Void*

  The function defines the initial properties for the indicated planning element based on the product data for the article that corresponds to the type of the planning element and its program association. The language currently selected in the global planning instance (see *OiPlanning* type) is used for designations (labels, values).

- *evalPropValue(pObj(MObject), pPKey(Symbol), pValue(Any), pOldValue(Any), pOldArticle(String))* → *Int*

  The function evaluates the relational knowledge in the product data after the property of the indicated planning element that was specified by its key was set to the transferred new

value. In addition, the old property value and the base article number are transferred before the value assignment. The evaluation of the value assignment can lead to changes of the definition (value ranges) or current values of other properties of the planning element. In this case, the function delivers 0, otherwise 1.

Note: The function is called from the *setPropValue()* function of the *Property* interface.

- *checkConsistency(pObj(MObject)) → Int*

  The function checks the correctness of the product data of the article that is represented by the transferred instance. The global error log is used for error messages (see the function of the same name of the *Article* interface). The standard implementation delegates to the function of the same name of the product database that is responsible for the article (*OiProductDB* type).

Article Information

- *getXArticleSpec(pObj(MObject), pType(Symbol)) → String*

  The function delivers the specification of the requested type for the article that is represented by the transferred instance or a value of type *Void* if no article specification of the required type is available for the implicit instance. Semantics and return value of the function correspond to the function of the same name of the *Article* interface, where only the @VarCode and @Final specification types are allowed. The standard implementation of the function delegates the query with the @VarCode specification type to the *getVarCode()* function and with the @Final specification type to the *getFinalArticleSpec()* function of the product database that is responsible for the article instance (*OiProductDB* type), if such an instance exists. Instead of the article instance, its base article number and a list of its current property values are transferred.

- *setXArticleSpec(pObj(MObject), pType(Symbol), pSpec(String)) → Void*

  The function assigns a new article specification of the specified type to the transferred article instance. Semantics and return value of the function correspond to the function of the same name of the *Article* interface, where only the @VarCode specification type is allowed. The standard implementation of the function uses the *varCode2PValues()* function of the product database that is responsible for the article instance (*OiProductDB* type) to determine the product properties that match the transferred variant code. If the obtained values differ from the current values of the respective properties, they will be reassigned by means of the *setPropValue()* function (*Property* interface) of the transferred article instance.

- *getArticlePrice(pObj(MObject), pLanguage(String), ...) → Any[]*

  The function delivers price information for the transferred planning element in the specified language. Semantics and return value of the function correspond to the function of the same name of the *Article* interface. The standard implementation of the function delegates the query to the function of the same name of the product database that is responsible for the planning element (*OiProductDB* type), if such an instance exists. Instead of the planning element, its base article number and a list of its current property values are transferred.

- *getArticleText(pObj(MObject), pLanguage(String), pForm(Symbol))* → *String[]*

  The function delivers describing article information for the transferred planning element in the specified language and in the specified form. Semantics and return value of the function correspond to the function of the same name of the *Article* interface. The standard implementation of the function delegates the query to the function of the same name of the product database that is responsible for the planning element (*OiProductDB* type), if such an instance exists. Instead of the planning element, its base article number is transferred.

- *getArticleFeatures(pObj(MObject), pLanguage(String))* → *Any*

  The function delivers a description of the configurable product properties for the transferred planning element in the specified language. Semantics and return value of the function correspond to the function of the same name of the *Article* interface. The standard implementation of the function delegates the query to the *getPropDescription()* function of the product database that is responsible for the planning element (*OiProductDB* type), if such an instance exists. Instead of the planning element, its base article number and a list of its current property values are transferred.

9.2 OiProductDB

Description

- An instance of the *OiProductDB* type manages exactly one product database and offers services for access and evaluation of information about articles and their properties.
- **Interface(s):** *MObject*

Initialization

- *OiProductDB(pFather(MObject), pName(Symbol), pID(Symbol))*

  The function initializes an instance of the *OiProductDB* type with the indicated ID. The ID cannot be changed later.

Methods

Article Configuration

Some of the functions described below expect a *pPValues* parameter that contains the current article configuration. This parameter is a list that contains a vector made up of the following elements for each product property:

1. the feature class (*String* or *Void*, unless relevant)
2. the (language-independent) designator of the feature (*String*)

3. the value of the feature (*Any*)
4. the list of the currently possible values (*List* or *Void*, unless relevant)
5. the activation state of the property that is assigned to the feature (*Int*)

General Methods

- *getID() → Symbol*

  The function delivers the ID of the product database.

- *setPrograms(pProgList(Symbol[])) → Void*

  The function assigns the number of programs (IDs) that are represented in the product database to the implicit instance.

- *getPrograms() → Symbol[]*

  The function delivers the number of programs (IDs) that are represented in the product database.

- *setDataBasePath(pDir(String)) → Void*

  The function assigns the root directory of the product data to the implicit instance.

- *getDataBasePath() → String*

  The function delivers the root directory of the product data.

- *getPDManager() → MObject*

  The function delivers the reference to the global product data manager.

Features and Relational Knowledge

- *hasProductKnowledge() → Int*

  The functions delivers True if the product database contains relational knowledge which must be evaluated with a change of feature values.

- *getArticlePropClasses(pArticle(String)) → Any*

  The function delivers a list with feature classes to which the indicated article (base article number) is assigned.

- *getPropDefs(pArticle(String), pPropOffset(Int), pLanguage(String), pChangedProp(Any[]), pPValues(Vector[])) → Any*

  The function delivers the property definitions for all features of the transferred article (base article number).

  The *pPropOffset* parameter specifies the number at which positions can be assigned for the properties. The specified language (ISO code) is used for designations (label, values). If no language is specified (*Void*), English is used. If the *pChangedProp* parameter is not a value of type *Void*, it specifies a feature whose value was changed so that the function is called.

In this case the parameter contains a three-digit vector consisting of (language-independent) designator of the feature, new and old feature value. The *pPValues* parameter describes the current article configuration (see above) or is a value of type *Void* if the function is called for an article that has not yet been initialized.

The return value is a list of seven-digit vectors. Each vector describes a feature and consists of the following:

1. Feature class (*String* or *Void*, unless relevant).
2. (Language-independent) designation of the feature (*String*).
3. Specification of the associated property (*Any[5]*) in accordance with the *setupProperty()* function of the *Property* interface.
4. (Initial) value of the feature or *Void* if no value is (pre)defined.
5. List of all possible values in so far as several values (*List* or *Void* are defined for the feature, unless relevant).
   The entries are two-digit vectors that contain the value and the language-independent description of the value. For optional features, the list must contain the value "not selected" which must be specified as `[@VOID, "@VOID"]`.
6. Position in the property list (*Int*).
7. Activation status for the property (*Int*).

- *checkConsistency(pArticle(String), pPValues(Vector[]), pLanguage(String), pErrorList(String[]))* → *Int*

  The function delivers True if the transferred article configuration for the indicated article (base article number) is correct from a product point of view. Error messages are attached to the list that is transferred in the *pErrorList* parameter. Here, the language specified in the *pLanguage* parameter is used.

Article Information

- *getVarCode(pArticle(String), pPValues(Any[]), ...)* → *String*

  The function delivers the variant code for the transferred article (base article number) and the transferred article configuration. If an additional optional parameter is indicated, it specifies whether the feature values contained in the article configuration are OFML values of the associated property (True) or whether they are given in the form used by the product database (False). Without any information, True is assumed.

- *varCode2PValues(pArticle(String), pVarcode(String))* → *Any[]*

  The function delivers the feature values to the transferred variant code for the indicated article (base article number).

  The return value is a list that contains a vector consisting of the following elements for each product feature:

  1. the feature class (*String* or *Void*, unless relevant)
  2. the (language-independent) designator of the feature (*String*)

3. the value of the feature (*Any*)

- *getFinalArticleSpec(pArticle(String), pPValues(Any[])) → String*

  The function delivers the final article number for the transferred article (base article number) and the transferred article configuration.

- *getArticlePrice(pArticle(String), pPValues(Any[]), pLanguage(String), ...) → Any[]*

  The function delivers price information for the transferred article (base article number) and the transferred article configuration in the specified language. If an additional optional parameter is given, it specifies the desired currency.

  The return value corresponds to the function of the same name of the *Article* interface.

- *getArticleText(pArticle(String), pLanguage(String), pForm(Symbol)) → String[]*

  The function delivers the article description for the transferred article (base article number) in the specified language and in the specified form. The *pForm* parameter may take on the following values:

  - @`short` short description
  - @`long` long description

  The return value is a list of strings that contain the individual lines of the description or a value of type *Void* if no article description is available for the implicit instance.

- *getPropDescription(pArticle(String), pPValues(Any[]), pNeedSymbols(Int), pLanguage(String)) → Any*

  The function delivers a description of the transferred article configuration for the specified article (base article number) in the specified language.

  The return value is a list of two-digit vectors whose first element (*String*) labels the feature, while the second element contains the current value (as character string) of the feature. If the *pLanguage* parameter contains a value of type *Void*, language-independent designators are furnished for feature and value.

  If the *pNeedSymbols* parameter has the value 1, the list entries consist of four-digit vectors with the following fields in the indicated order:

  1. language-independent symbol of the feature
  2. language-independent designation of the feature
  3. language-independent symbol of the current value of the feature
  4. language-independent designation of the current value of the feature

  The function delivers a value of type *Void* if no descriptions for the features are available.

Chapter 10

Types of the Planning Environment

10.1 The Wall Interface

Wall defines the interface of a wall or some of its parts (e.g., sides) for furniture planning.

- *getWallParams())* $\rightarrow$ *[Float, Float, Float[3]]*

  The function delivers the geometric parameter to be able to place furniture at the wall in the course of the planning process. The return value is a vector with three elements.

  1. Width.
  2. Rotary angle (in positive orientation about the y-axis).
  3. Position (origin of the local coordinate system).

10.2 OiLevel

Description

- *OiLevel* models one story of a building that can consist of one or several rooms.
- **Interface(s):** Base, Complex

Initialization

- *OiLevel(pFather(MObject), pName(Symbol))*

  The function initializes an instance of the *OiLevel* type.

Methods

- *setDefaultHeight(pHeight(Float))* → *Void*

  The function sets a default for the height of walls to be created. This value is effective only if there are no walls on the story, otherwise the height of the planning wall (see below) is used as default.

- *Complex::getHeight()* → *Float*

  The function delivers the maximum wall height within a story or, if no walls exist, the specified height.

- *setPlanningWall(pWall(MObject))* → *MObject*

  The function selects a wall to which furniture is to be added in the following. *pWall* must be an object that implements the *Wall* interface (Section 10.1). *NULL* is allowed as a special value for *pWall*. In this case, a possibly existing setting is deleted. The function delivers the new planning wall as return value.

- *getPlanningWall()* → *MObject*

  The function delivers the specified planning wall (see below). If no planning wall was explicitly set, the wall generated last is used.

- *setPlanningMode(pMode(Int))* → *Void*

  The function sets the planning mode. As a minimum requirement, the values 0 (activates furniture planning) and 1 (switches to the base mode of floor space planning) must be detected. Values > 1 are acceptable depending upon the implementation.

- *Complex::checkAdd(pType(Type), pObj(MObject), pPos(Float[3]), pParams(Float[]))* → *Float[3]*

  The function checks the insertion of a new wall. *pType* must be a subtype of OiWall. *pObj* must be instance of a subtype of *OiWall* to which the new wall should be attached. If *NULL* is transferred for *pObj*, attaching is performed at the preset planning wall. (Section 10.2). *pPos* is ignored and may be *NULL*. *pParams* is *NULL* or contains an optional list of default parameters. If available, these parameters are interpreted as follows:

  1. Width.
  2. Attaching angle.
  3. Thickness.

  If the given parameters can be inserted, the attaching position is returned, otherwise *NULL*.

- *objInLevel(pObj(MObject))* → *Int*

  The function delivers 1 if the *pObj* object is located within the story, otherwise 0. Simplified tests (e.g., limitation to the surrounding rectangle or bounding box) are possible, and collision must not be observed.

10.3 OiWall

Description

- *OiWall* represents a wall as a component of a story. This may be an outside wall or a dividing wall. Windows, doors, etc. can be inserted in a wall as children.
- **Interface(s):** Base, Complex, Properties, Material, Wall

Initialization

- *OiWall(pFather(MObject), pName(Symbol))*
  The function initializes an instance of the *OiWall* type.

10.4 OiWallSide

Description

- A single side of a wall at which furniture can be placed in the course of the planning process.
- **Interface(s):** Base, Properties, Material, Wall

Initialization

- *OiWallSide(pFather(MObject), pName(Symbol))*
  The function initializes an instance of the *OiWallSide* type.

Appendix A

Product Data Model

This appendix describes the underlying data model for the types of the product data management (Chapter 9). Figure A.1 shows a graphic representation of the model which illustrates the major concepts and terms[1].

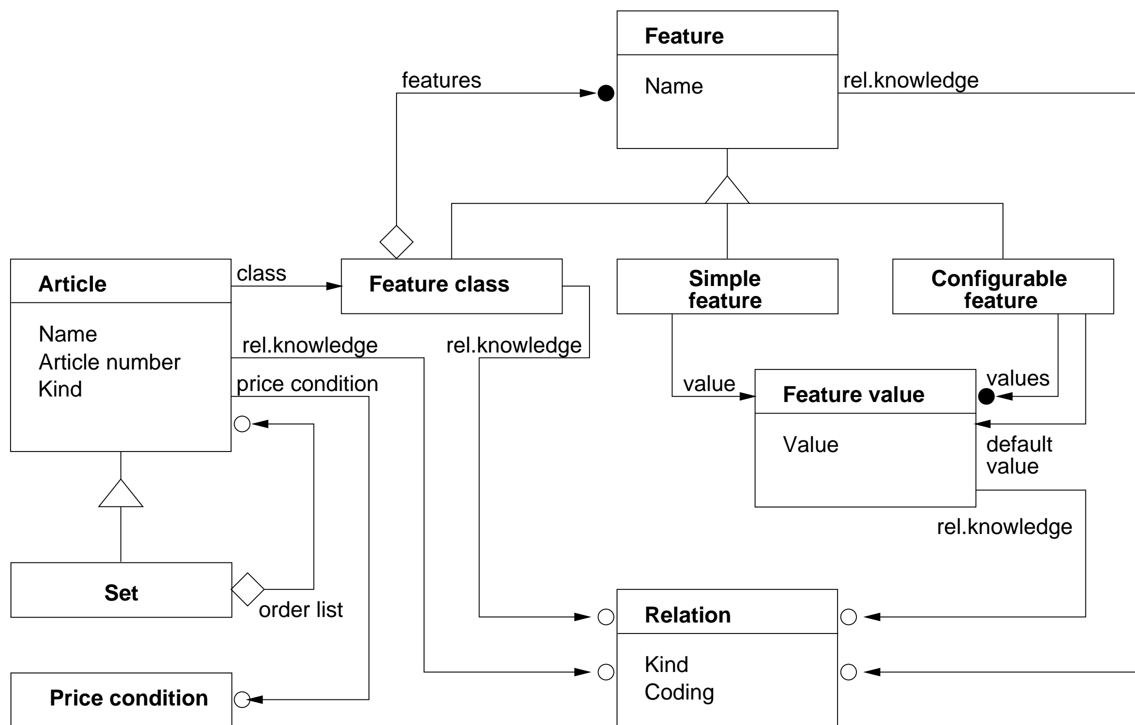

Figure A.1: Product data model

[1]The notational conventions used here is explained in Appendix G.

Additional remarks and explanations

Each article is assigned to a certain article type that specifies which actions are allowed for the article or which meaning certain model properties have. The major article types are "configurable article," "assembly unit" and "commercial article."

Features describe the properties of articles and are combined to feature classes. A feature in a class can be another class. Each article is assigned one or several feature classes.

Price terms contain the definitions for the base price as well as extra charges and discounts for configurable articles via variant terms. Relational knowledge must be used to establish the relation to the corresponding features or feature values.

Relational knowledge is shown by means of five types of relations:

- *Conditions*
  determine whether a feature may be evaluated or whether a feature value may be set.
- *Selection criteria*
  specify that a feature must be evaluated or that a parts list position must be selected.
- *Actions* and *Procedures*
  serve for derivation of feature values and are executed if a feature value is selected or a feature is evaluated. For this purpose, actions have declarative character and are independent of the order of the evaluation. Procedures, on the other hand, implement more complex algorithms and are executed only at certain times.
- *Constraints*
  serve for monitoring the consistence of a configuration and, therefore, can only be bound to a configurable article via the configuration profile.

Appendix B

The 2D Interface

B.1 Introduction

The 2D interface described in this chapter allows for programming of 2D objects. Altogether, the generation of 2D objects in OFML can be accomplished in the following ways:

- through generation based on a (3D) OFML geometry,
- through description via the OFML database [ODB],
- through import of an external 2D vector data record (Chapter C), and
- through programming.

A specialty of this 2D programming interface is the fact that the generated 2D objects cannot be stored persistently. Thus, they must be restored in the appropriate rules (Chapter 5), if required.

B.2 The 2D Object Hierarchy

The 2D objects are generally arranged in a tree where the nodes of the tree are of the `G2DCompound` type and the leaves are consequently of a type derived from `G2DPrimitive`. The root of the tree is always bound to an OFML object that supports the *MObject* interface so that each 2D object can directly or indirectly be assigned to an *MObject* object.

From OFML, the 2D objects are referenced via integer ID's. An *MObject* object is not assigned two 2D objects with the same ID, that is, the ID's below an *MObject* object are unique. Assigned ID's do not grow monotonously, that is, a new object can receive the ID of an old object that was deleted.

The object with the ID 0 always exists[1] and is of the `G2DCompound` type.

[1] In fact, it is generated if required.

B.3 Coordinates

All coordinates are indicated in the rectangular X/Y coordinate system, where the positive X axis points to the right and the positive Y axis up. In principle, angular dimensions are radiant measures and mathematically positive (counterclockwise). The zero angle shows in the direction of the positive X axis.

B.4 Methods

The manipulation of 2D objects is carried out via the methods listed in the following subsections.

B.4.1 `new2DObj`

`t.new2DObj(`*parent_id*`,` *object_type*`, ...)`

All 2D objects are generated with the `new2DObj` method. Their first argument is the ID of the father object which must be of the `G2DCompound` type. The second argument is a symbol which determines the type of 2D object to be generated. The remaining arguments are dependent upon the type of the object to be generated. The return value is the ID of the newly generated object.

The exact form of the calls of `new2DObj` is described in the section B.5 in the respective object-specific subsections.

B.4.2 `delete2DObj`

`t.delete2DObj(`*obj_id*`)`

The `delete2DObj` method removes the object with the indicated ID and, if required, recursively all existing child objects of this object.

B.4.3 `set2DObjAttr`

`t.set2DObjAttr(`*obj_id*`,` *attr_type*`, ...)`

The `set2DObjAttr` method sets the attributes of existing objects. The first argument is the ID of the object of which an attribute is to be set. The second argument is a symbol which determines the type of the attribute to be set. The remaining attributes are dependent upon the attribute type.

The exact form of the calls of `set2DObjAttr` is described in the B.6 section in the respective attribute-specific subsections.

B.4.4 `translate2DObj`

`t.translate2DObj(`*obj_id*`, [`x`, `y`])`

The `translate2DObj` method moves the `G2DCompound` object with the ID *obj_id* relative to the current position in the coordinate system of the father object by $x; y$.

B.4.5 `rotate2DObj`

`t.rotate2DObj(`*obj_id*`, `*angle*`)`

The `rotate2DObj` method rotates the `G2DCompound` object with the ID *obj_id* relative to the current rotation by the angle *angle*. The rotation is carried out around the origin of the coordinate system of the father object.

B.5 Object Types

The following subsections list the available 2D object types with their attributes. Besides the specified attributes, every type has the *Pickable* and *Snapable* attribute.

B.5.1 `G2DCompound`

A `G2DCompound` object differs from all other objects that are derived from `G2DPrimitive` in that it

- can have additional 2D objects as children and
- can be translated, rotated and scaled.

A new `G2DCompound` object is generated with the method

`t.new2DObj(`*parent_id*`, @COMPOUND)`

B.5.2 `G2DPoints`

An object of the `G2DPoints` type consists of a list o f X/Y coordinates that describe the center of the individual points. In addition, it features the attributes *Color*, *PointSize*, and *PointSmooth*.

A new `G2DPoints` object with n points is generated with the method

`t.new2DObj(`*parent_id*`, @POINTS, [[`x_0`, `y_0`], ..., [`x_{n-1}`, `y_{n-1}`]])`

B.5.3 `G2DLines`

An object of the `G2DLines` type consists of individual line segments that are defined in the X/Y coordinate system by means of their start and end points. It features the attributes *Color*, *LineWidth*, and *LineStyle*.

A new `G2DLines` object with n lines is generated with the method

`t.new2DObj(`*parent_id*`, @LINES, [[`x_0`, `y_0`], ..., [`x_{2n-1}`, `y_{2n-1}`]])`

where $x_{2i}; y_{2i}$ specifies the start point and $x_{2i+1}; y_{2i+1}$ the end point of a line.

B.5.4 `G2DLineStrip`

An object of the `G2DLineStrip` type consists of a series of at least two points that are connected with each other by means of individual line segments in the specified order where, in contrast with `G2DLineLoop`, the last point is not connected with the first point. It features the attributes *Color*, *LineWidth*, and *LineStyle*.

A new `G2DLineStrip` object with n points is generated with the method

`t.new2DObj(`*parent_id*`, @LINE_STRIP, [[`x_0`, `y_0`], ..., [`x_{n-1}`, `y_{n-1}`]])`

B.5.5 `G2DLineLoop`

An object of the `G2DLineLoop` type consists of a series of at least two points that are connected with each other by means of individual line segments in the specified order where, in contrast with `G2DLineStrip`, the last point is also connected with the first point. It features the attributes *Color*, *LineWidth*, and *LineStyle*.

A new `G2DLineLoop` object with n points is generated with the method

`t.new2DObj(`*parent_id*`, @LINE_LOOP, [[`x_0`, `y_0`], ..., [`x_{n-1}`, `y_{n-1}`]])`

B.5.6 `G2DConvexPolygon`

An object of the `G2DConvexPolygon` type is described by a series of at least three points that must result in a convex polygon when connected with each other, including the last point with the first point. It features the attributes *Color*, *PointSize*, *PointSmooth*, *LineWidth*, *LineStyle*, *FillStyle*, and *PolygonMode*. Depending upon *PolygonMode*, only one subset of the attributes is used in each case.

A new `G2DConvexPolygon` object with n corner points is generated with the method

`t.new2DObj(`*parent_id*`, @POLYGON, [[`x_0`, `y_0`], ..., [`x_{n-1}`, `y_{n-1}`]])`

B.5.7 `G2DRectangle`

An object of the `G2DRectangle` type is described by two cornerpoints lying diagonally opposite each other. It features the attributes *Color*, *PointSize*, *PointSmooth*, *LineWidth*, *LineStyle*, *FillStyle*, and *PolygonMode*. Depending upon *PolygonMode*, only one subset of the attributes is used in each case.

A new `G2DRectangle` object is generated with the method

`t.new2DObj(`*parent_id*`, @RECTANGLE, [[`x_0`, `y_0`], [`x_1`, `y_1`]])`

where the rectangle has the four cornerpoints $x_0; y_0$, $x_0; y_1$, $x_1; y_0$ and $x_1; y_1$.

B.5.8 `G2DText`

An object of the `G2DText` type consists of an ASCII text positioned relative to a reference point. It features the attributes *Color*, *Text*, *Position*, *Height*, *AspectRatio*, and *Alignment*.

A new `G2DText` object is generated with the method

`t.new2DObj(`*parent_id*`, @TEXT, [`x`, `y`], `*text*`)`

where $x; y$ is the position and *text* a character string with the text to be represented.

B.5.9 `G2DArc`

An object of the `G2DArc` type represents the arc of a circle that is described by means of its center, radius, start and end angle. The circle segment is drawn in mathematically positive direction of rotation from start to end angle. `G2DArc` objects feature the attributes *Color*, *LineWidth*, and *LineStyle*.

A new `G2DArc` object is generated with the method

`t.new2DObj(`*parent_id*`, @ARC, [`x_{center}`, `y_{center}`], `*radius*`, `*start*`, `*end*`)`

Using the method

`t.new2DObj(`*parent_id*`, @CIRCLE, [`x_{center}`, `y_{center}`], `*radius*`)`

generates the special case of an arc of a circle where $start = 0$ and $end = 2\pi$.

B.5.10 `G2DEllipse`

An object of the `G2DEllipse` type represents an ellipse that is described by means of its center, radius in x and y direction, and rotary angle about the center.

A new `G2DEllipse` object is generated with the method

`t.new2DObj(`*parent_id*`, @ELLIPSE, [`x_{center}`, `y_{center}`], [`x_{radius}`, `y_{radius}`], `*angle*`)`

B.6 Attributes

The following subsections describe the attributes supported for 2D objects.

B.6.1 *Color*

Using the call

```
t.set2DObjAttr(obj_id, @COLOR, [red, green, blue])
```

sets the color of the object specified by the ID *obj_id* to the RGB value *red*; *green*; *blue*, where the three color components must be given as floating point values in the interval $[0.0, 1.0]$.

B.6.2 *PointSize*

Using the call

```
t.set2DObjAttr(obj_id, @POINT_SIZE, point_size)
```

sets the size of a point to the floating point value *point_size*. For the screen display, a value of 1.0 corresponds to a point size of one pixel. For printout, 1.0 should correspond to a size of 1pt (1/72in).

The standard value of the point size is 1.0

B.6.3 *PointSmooth*

Using the call

```
t.set2DObjAttr(obj_id, @POINT_SMOOTH, smooth)
```

specifies for points whose size is not 1.0 whether the point should be represented as a square ($smooth = 0$) or as a filled circle with a smooth edge ($smooth \neq 0$). This setting is only relevant for screen display with OpenGL. The screen display with other drivers or the printout can ignore the *PointSmooth* flag if the point is always represented as a filled circle.

The standard value for the *PointSmooth* flag is 0.

B.6.4 *LineWidth*

Using the call

```
t.set2DObjAttr(obj_id, @LINE_WIDTH, line_width)
```

sets the line width to *line_width* pixels (screen display) or points (1pt = 1/72in) (print). *line_width* is given as floating point value.

The standard value for the line width is 1.0.

B.6.5 *LineStyle*

Using the call

`t.set2DObjAttr(`*obj_id*`, @LINE_STYLE, `*factor*`, `*pattern*`)`

sets the line style. In this case, *pattern* is a symbol that can accept the values listed in table B.1.

Sample	Description
`@DEFAULT`	The preset line style is used.
`@SOLID`	A continuous line is drawn.
`@DASHED`	A dashed line is drawn. The *factor* factor determines the length of the line segments displayed and not displayed.
`@DOTTED`	A dotted line is drawn. The *factor* factor determines the distance between the centers of two neighboring points.
`@DASH_DOTTED`	A dot-dash line is drawn. The *factor* factor determines the length of the displayed line segment and half the length of the non-displayed line segments.
`@DASH_DOUBLE_DOTTED`	A dash double-dotted line is drawn. The *factor* factor determines the length of the displayed line segment and one-third the length of the non-displayed segments.
`@DASH_TRIPLE_DOTTED`	A dash triple-dotted line is drawn. The *factor* factor determines the length of the displayed line segment and on-fourth the length of the non-displayed segments.

Table B.1: Line styles

The *factor* factor is given as floating point value. Its unit is one pixel for screen display and a dot (1pt = 1/72in) for the printout.

The standard value for *factor* is 4.0 and for *pattern* `@DEFAULT`.

B.6.6 *FillStyle*

Using the call

`t.set2DObjAttr(`*obj_id*`, @FILL_STYLE, `*style*`)`

sets the fill pattern for polygons. In this case, *style* is a symbol that can accept the values listed in table B.2.

The horizontal distance between diagonal or vertical lines in the fill pattern or the vertical distance between horizontal lines should measure eight pixels for the screen display and eight points (1pt = 1/72in) for the printout.

By default, a polygon is shown completely filled.

Fill pattern	Description
`@LEFT_30`	diagonal lines from top left to bottom right at an angle of 30 degrees to the horizontal
`@RIGHT_30`	diagonal lines from bottom left to top right at an angle of 30 degrees to the horizontal
`@CROSS_30`	crossing diagonal lines from top left to bottom right and from bottom left to top right, both at an angle of 30 degrees to the horizontal
`@LEFT_45`	diagonal lines from top left to bottom right at an angle of 45 degrees to the horizontal
`@RIGHT_45`	diagonal lines from bottom left to top right at an angle of 45 degrees to the horizontal
`@CROSS_45`	crossing diagonal lines from top left to bottom right and from bottom left to top right, both at an angle of 45 degrees to the horizontal
`@H_LINES`	horizontal lines
`@V_LINES`	vertical lines
`@CROSS`	crossing horizontal and vertical lines

Table B.2: Fill pattern

B.6.7 *PolygonMode*

Using the call

```
t.set2DObjAttr(obj_id, @POLYGON_MODE, mode)
```

and the *mode* parameter specifies for polygons whether the corner points (*mode* = `@POINT`), the boundary lines (*mode* = `@LINE`) or the area (*mode* = `@FILL`) should be displayed. The attributes used dependent on *PolygonMode* are listed in table B.3.

PolygonMode	attributes used
`@POINT`	*Color, PointSize, PointSmooth*
`@LINE`	*Color, LineWidth, LineStyle*
`@FILL`	*Color, FillStyle*

Table B.3: Attributes used dependent on *PolygonMode*

The standard value for *PolygonMode* is `@FILL`.

B.6.8 *Text*

Using the call

```
t.set2DObjAttr(obj_id, @TEXT, text)
```

sets the *text* text to be displayed for an object of the `G2DText` type. The *text* argument must be a character string whose characters are from the ASCII character set.

B.6.9 *Position*

Using the call

`t.set2DObjAttr(`*obj_id*`, @POSITION, [`x`, `y`])`

sets the position (the reference point) for an object of the `G2DText` type.

B.6.10 *Height*

Using the call

`t.set2DObjAttr(`*obj_id*`, @HEIGHT, `*height*`)`

sets the height of a capital letter without descender for an object of the `G2DText` type.

The standard value for the text height is 0.1.

B.6.11 *AspectRatio*

Using the call

`t.set2DObjAttr(`*obj_id*`, @ASPECT_RATIO, `*ratio*`)`

determines the extension or compression in x direction for an object of the `G2DText` type. The floating point value of *ratio* must be greater than 0.0. With the default value of 1.0, the text appears within normal proportions that are predefined by the font used. With values greater than 1.0, it is extended in the x direction, with values smaller than 1.0 and greater than 0.0 it is compressed in the x direction.

B.6.12 *Alignment*

Using the call

`t.set2DObjAttr(`*obj_id*`, @ALIGNMENT, `*align*`)`

determines the alignment of the text relative to the reference point for an object of the `G2DText` type.

If *width* is the width of the text, the move Δx of the left side of the first letter on the base line relative to the reference point is determined as follows:

$\Delta x = -(alignment + 1.0) \times (width/2.0)$

The standard value for *align* is -1.0, which left-aligns the text.

B.6.13 *Pickable*

Using the call

`t.set2DObjAttr(`*obj_id*`, @PICKABLE, `*pickable*`)`

can specify for each object whether it should be considered in 2D mode in determining the object that is to be selected with the mouse. If the integer argument *pickable* is zero, the corresponding 2D object is not considered; if it is not zero, it is considered.

If the *pickable* flag of an object of the `G2DCompound` type is not set, none of the child objects is taken into account. If it is set, then the *pickable* flag of the respective child object is decisive.

The standard value for *pickable* is 1.

B.6.14 *Snapable*

Using the call

`t.set2DObjAttr(`*obj_id*`, @SNAPABLE, `*snapable*`)`

can determine for each object whether it should be taken into account in determining a trap point. If the integer argument *snapable* is zero, the corresponding 2D object is not taken into account. If it is not zero, it is taken into account.

If the *snapable* flag of an object of the `G2DCompound` type is not set, none of the child objects is taken into account. If it is set, then the *snapable* flag of the respective child object is decisive.

The standard value for *snapable* is 1.

B.6.15 *Exportable*

Using the call

`t.set2DObjAttr(`*obj_id*`, @EXPORTABLE, `*exportable*`)`

can determine for each object whether it should be exported during the export to a 2D vector format[2]. If the integer argument *exportable* is zero, the corresponding 2D object is not exported. If it is not zero, it is exported.

If the *exportable* flag of an object of the `G2DCompound` type is not set, none of the child objects is exported. If it is set, then the *exportable* flag of the respective child object is decisive.

The standard value for *exportable* is 1.

[2] Since the print command also uses the export to a 2D vector format, the *exportable* flag also influences the objects appearing during printout.

B.6.16 *Layer*

Using the call

`t.set2DObjAttr(`*obj_id*`, @LAYER, `*layer*`)`

can determine the layer for each object which is used to control its visibility. Here, the *layer* argument is a symbol that should solely consist of letters, numbers and underscore sign.

If the layer was set for an object of the `G2DCompound` type, this layer is used for all direct and indirect child objects for which a layer was not explicitly set.

Appendix C

The 2D vector file format

C.1 Introduction

The EasternGraphics Metafile (EGM) is an expandable file format for saving graphic and non-graphic data. Due to its expandability it allows the integration of data in other file formats.

The EGM format is structured in such a way that an EGM parser doesn't have to be able to interpret all EGM elements to read until the end of the file. It should be able to generally ignore unknown elements.

The EGM format supports hierarchical saving of data. This makes it possible, for example, to save graphic 2D symbols on the top level as well as 2D symbols embedded in a scene element within an EGM. The first case is used when EGM describes only a single 2D symbol for use in the GF. The second case may become interesting when a whole scene is described in EGM format which, among other things, also contains user-defined 2D symbols.

EGM is specified in binary as well as text format. Binary format helps to save data efficiently while text format may be used primarily during the developmental phase.

C.2 data types

This section describes the data types used in EGM, in particular their representation in EGM binary and text format.

Both formats – binary and text – are defined to be platform-independent making easy data exchange between different computer platforms possible.

All numeric values are saved in binary format so that the byte with the highest value comes first, followed by all other bytes in descending value order. This is also known as "Network Byte Order" or "Big-Endian."

The text format character set must be an aggregate of the ASCII character set. This is the case for most, if not all, ISO–8859–x character sets. To enable easy conversion between text format and binary format, the same limitation also applies to the binary format *String* data type.

C.2.1 Simple Data Types

Byte

A *Byte* is an integer 8 bit value that can be interpreted either as an unsigned value in the 0 to $2^8 - 1$ ($[0, 255]$) range or as a two's complement signed value in the -2^7 to $2^7 - 1$ ($[-128, 127]$) range. Type *Byte* values are saved in EGM binary format as a single byte, as shown in figure C.1, with I_7 being the highest value bit and I_0 the lowest value bit.

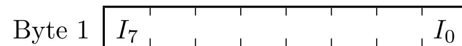

Figure C.1: Type *Byte* values

In text format, a *Byte* is displayed as a decimal, octal, or hexadecimal number, which can optionally be preceded by a minus sign. An octal number is designated by a leading `0` and a hexadecimal number by a leading `0x` or `0X`, with only digits ranging from `0` to `7` being permitted for octal numbers and digits ranging from `0` to `9`, `A` to `F`, and `a` to `f` being permitted for hexadecimal numbers.

In this EGM specification, the UINT8 identifier is used as a type specification for unsigned *Byte* values; for signed *Byte* values, the INT8 identifier is used.

Word

A *Word* is an integer 16 bit value that is either interpreted as an unsigned value in the 0 to $2^{16} - 1$ ($[0, 65535]$) range or as a two's complement signed value in the -2^{15} to $2^{15} - 1$ ($[-32768, 32767]$) range. Type *Word* values are saved in EGM binary format as two successive bytes as shown in figure C.2, with I_{15} being the highest value bit and I_0 the lowest value bit.

Figure C.2: Type *Word* values

In text format, a *Word* is displayed as a decimal, octal or hexadecimal number, which can optionally be preceded by a minus sign. An octal number is designated by a leading `0` and a hexadecimal number by a leading `0x` or `0X`, with only digits ranging from `0` to `7` being permitted for octal numbers and digits ranging from `0` to `9`, `A` to `F`, and `a` to `f` being permitted for hexadecimal numbers.

In this EGM specification, the UINT16 identifier is used as a type specification for unsigned *Word* values and the INT16 identifier is used for signed *Word* values.

Double Word

A *Double Word* is an integer 32 bit value that is interpreted as an unsigned value in the 0 to $2^{32} - 1$ ($[0, 4294967295]$) range or as a two's complement signed value in the -2^{31} to $2^{31} - 1$ ($[-2147483648, 2147483647]$) range. Type *Double Word* values are saved in EGM binary format as four successive bytes as shown in figure C.3, with I_{31} being the highest value bit and I_0 the lowest value bit.

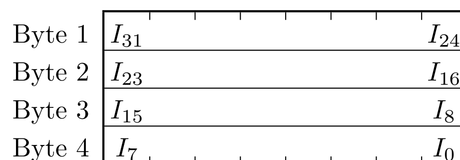

Figure C.3: Type *Double Word* values

In text format, a *Double Word* is displayed as a decimal, octal or hexadecimal number that is optionally preceded by a minus sign. An octal number is designated by a leading `0` and a hexadecimal number by a leading `0x` or `0X`, with only digits ranging from `0` to `7` being are permitted for octal numbers and digits ranging from `0` to `9`, `A` to `F`, and `a` to `f` being permitted for hexadecimal numbers.

In this EGM specification, the UINT32 identifier is used as a type specification for unsigned *Double Word* values and the INT32 identifier is used for signed *Double Word* values.

Single Precision Floating Point

Type *Single Precision Floating Point* values are displayed according to the IEEE 754 standard. The absolute value lies in the range between $1.17549435 \times 10^{-38}$ and $3.40282347 \times 10^{38}$ with a minimum of 6 significant decimals in the mantissa. In EGM binary format, single precision floating point values are saved as four successive bytes as displayed in figure C.4.

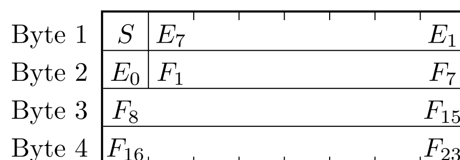

Figure C.4: Single precision floating point number

The value is 0.0, if the exponent and the mantissa are 0. Otherwise it is calculated according to $(-1)^s \times 1.f \times 2^{e-127}$. S is the sign bit, f is the mantissa ($F_1 \ldots F_{23}$ with F_1 being the highest value bit) and e is the exponent ($E_7 \ldots E_0$ with E_7 being the highest value bit).

In text format, a single precision floating point number consists of an optional leading minus or plus sign, an integer decimal quantity, a decimal point, a fractional quantity, and an optional

exponent. The exponent consists of one of the numbers `e` or `E`, followed by an optional minus or plus sign which in turn is followed by an integer decimal . The integer or fractional quantity can be dropped, but both cannot. The decimal point can be dropped, if the fractional quantity is dropped with an exponent present.

In this EGM specification, the FLOAT32 identifier is used as a type specification for single precision floating point values.

Double Precision Floating Point

Type *Double Precision Floating Point* values are displayed according to the IEEE 754 standard. The absolute value lies in the range between $2.2250738585072014 \times 10^{-308}$ and $1.7976931348623157 \times 10^{308}$ with a minimum of 15 significant decimals in the mantissa. In EGM binary format, double precision floating point values are saved as eight successive bytes as displayed in figure C.5.

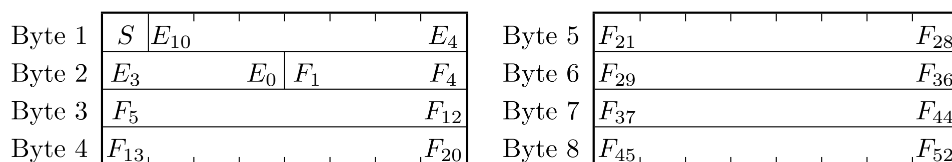

Figure C.5: double precision floating point

The value is 0.0, if the exponent and the mantissa are 0. Otherwise it is calculated according to $(-1)^s \times 1.f \times 2^{e-1023}$. S is the sign bit, f is the mantissa ($F_1 \ldots F_{52}$ with F_1 being the highest value bit) and e is the exponent ($E_{10} \ldots E_0$ with E_{10} being the highest value bit).

The display of a double precision floating point number in text format corresponds to the display of a single precision floating point number as described above.

In this EGM specification, the FLOAT64 identifier is used as a type specification for double precision floating point values.

Symbol

A symbol is a series of characters and is saved in EGM binary format, as shown in figure C.6. The length ($U_{15} \ldots U_0$ with U_{15} being the highest value bit) that is saved in the first two bytes as an unsigned value neither includes itself nor the NUL character terminating the symbol.

In text format, a symbol consists of a series of ASCII letters, digits and underscores; the first character must not be a digit. This symbol is case sensitive.

In this EGM specification, the SYMBOL identifier is used as a type specification for symbols.

String

A string is saved in EGM binary format, as shown in figure C.7. The length ($U_{15} \ldots U_0$ with U_{15} being the highest value bit) that is saved in the first two bytes as an unsigned value neither includes itself or the NUL character terminating the string.

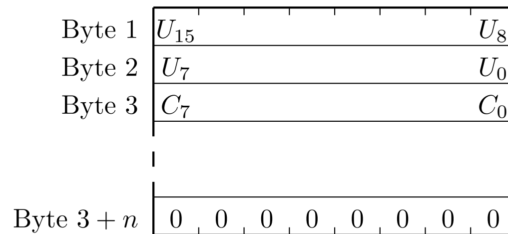

Figure C.6: Symbol

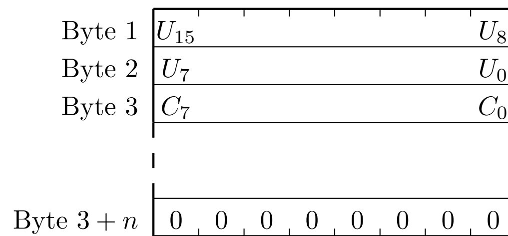

Figure C.7: Character string

In EGM text format, character strings are displayed as a result of any number of characters (including none) that are surrounded by quotation marks. Non-printable characters are represented by escape sequences that can be used in the applications listed in table C.1. Please note that the octal coding represented in parentheses can vary between platforms. For example, in MacOS(R) the coding of `\n` is `\r` exchanged.

`\a`	Klingelzeichen	(`\7`)	`\\`	backward slash	(`\`)
`\b`	backspace	(`\8`)	`\?`	question mark	(`?`)
`\f`	form feed	(`\14`)	`\'`	apostrophe	(`'`)
`\n`	line separator	(`\12`)	`\"`	quotation marks	(`"`)
`\r`	carriage return	(`\15`)	`\`*ooo*	octal number	
`\t`	tab character	(`\9`)	`\x`*hh*	hexadecimal number	
`\v`	vertical tabulation character	(`\13`)			

Table C.1: escape sequences for character strings

In the escape sequence `\`*ooo* *ooo* stands for a series consisting of one to three octal digits $(0 \ldots 7)$ and in `\x`*hh* *hh* stands for a series consisting of one or more hexadecimal digit $(0 \ldots 9, \mathrm{A} \ldots \mathrm{F}, \mathrm{a} \ldots \mathrm{f})$. You should preferably use the format `\`*ooo* with three octal digits, since only then a correct coding of the character can be ensured without any consideration for the subsequent character.

In this EGM specification, the STRING identifier is used as a type specification for character strings.

C.2.2 Structured data types

Structured data types consist of a series of simple data types. Each structured data type is described using a combination consisting of type class and object type. The type class is used for classifying object types; the object types are used for defining the structure and the meaning of structured data types. For example, a type class can combine all object types defined for describing graphic 2D primitives, with each object type describing the structure of the respective 2D object within the EGM format.

In the binary format, every structured data type consists of the structure header and the structure body. The structure header is eight bytes long and contains information regarding the type of structure and its length. The structure body contains the actual data. Within the structure body, every piece of data is oriented to a multiple of its own size relative to the structure beginning, with the exception of strings that are oriented to a multiple of two.

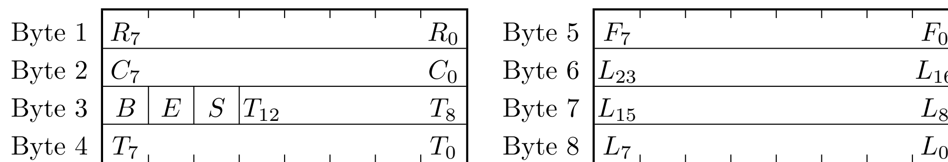

Figure C.8: Structure header

Figure C.8 shows the organization of the structure header.

Bits $R_7 \dots R_0$ are reserved for future use and should be set to 0.

Bits $C_7 \dots C_0$ contain the type class, bits $T_{12} \dots T_0$ contain the object type. The object types must be unique within the type class.

The B– and E–bits are used to label the beginning and end of a compound object as described in section C.2.3.

The S bit indicates whether single precision floating point parameters (S–Bit is set) or double precision floating point parameters (S–Bit is reset) are present [1]. Not every object type that has floating point parameters must support single and double precision.

Bits $F_7 \dots F_0$ do not have a predefined meaning and, depending on object type, can be used for flag bits.

Finally, bits $L_{23} \dots L_0$ contain the entire length of the structure.

The structural header itself is always oriented to a multiple of eight relative to the beginning of the file and the entire length of the structure also is a multiple of eight. This requires the end of the structure to be filled with null bytes, if necessary. This ensures that the EGM can be mapped directly to memory and that no orientation problems arise when individual data are directly accessed, which could cause a bus error.

[1] For type attributes that use either FLOAT32 or FLOAT64 depending on S bit, only FLOAT is written.

In text format, every structured data type consists of one or more lines, with all but the last line having to have a backward slash (\) directly before the line end character(s). These lines are combined into a data record. The backward slashes and the subsequent line end are removed.

Every single line cannot have more than 2047 characters, including the line end character. The number of characters per line without the line end character should not exceed 2045, since two characters are used for designating the end of a line on some platforms. Theoretically, the length of the entire data record is unlimited.

Lines are ended using either `\x0A`, `\x0D`, or `\x0D\x0A`.

The individual single data within a data record are delimited by one or more separators, using the blank space (`\40`) and the tab character (`\t`) as separator.

At the beginning of the data record, there are type class and object type, delimited by one or several separators and followed by flags F_0 to F_7 indicated as unsigned decimal, octal, or hexadecimal number in the range between 0 and 255, with F_7 being the highest value bit. Type class as well as object type may be indicated as identifier that is case sensitive or as decimal, octal, or hexadecimal numbers. The structural remainder of the data record is defined by the object type whose identifier and coding only must be unique within the type class.

An octal number is marked by a leading `0` and a hexadecimal number by a leading `0x` or `0X`, with only digits ranging from `0` to `7` being permitted for octal numbers and digits ranging form `0` to `9`, `A` to `F`, and `a` to `f` being permitted for hexadecimal numbers.

C.2.3 Compound types

Compound types consist of a series of structured data types that are enclosed by a *Begin* and a *End* object of the same structured data type. These *Begin* and *End* objects must always occur in pairs.

In EGM binary format, the *Begin* object is marked by a set B bit in the structure header; the *End* object is marked by a set E bit.

In EGM text format, the data record of the *Begin* object starts with the `begin` identifier, which is followed by the type class, delimited by one or several separators. The data record of the *End* object starts analogically with the `end` identifier, followed by one or several separators and the type class. Instead of `begin`, there can also be a single plus sign (`+`), and instead of `end`, there can be a single minus sign (`-`).

C.3 File header

The EGM binary format starts with the following structure:

major is the main version number and *minor* is the sub version number. The version described in this document of EGM is 1.0 ($major = 1; minor = 0$).

In EGM text format, the first line contains the `EGM` and `version` identifiers, with `EGM` being directly at the beginning of the line and `version` being delimited from `EGM` by a blank space. This is followed

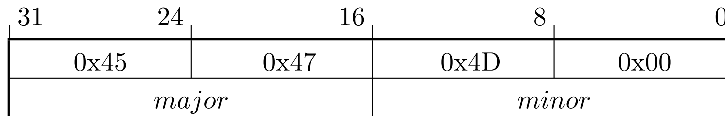

Figure C.9: binary EGM header

by the main version number and the sub version number, delimited by the "usual" separators. The first EGM line in version 1.0 is as follows:

```
EGM version 1 0
```

C.4 General structured data types

This section describes general structured data types that have no relation to concrete type classes. These types have the type class 1 with the `common` identifier.

C.4.1 Comment

Type class: 1 / `common`

Object type: 1 / `comment`

Parameter:

Offset	Type	Parameter
8	STRING	*comment*
roundup($11 + len, 8$)	End of structure	

Comments are ignored during reading.

In a special case, comments can consist of a data record in the text format that starts with a pound sign (`#`) followed directly by the actual comment and delimited from the pound sign by one or more separators, if necessary.

C.4.2 EGM type

Type class: 1 / `common`

Object type: 2 / `egm_type`

Parameter:

Offset	Type	Parameter
8	STRING	*egmtype*
roundup($11 + len, 8$)	End of structure	

The file header described in section C.3 can be followed directly by a type *EGMType* object. The *egmtype* character string describes the EGM type. Currently, the types listed in table C.2 have been defined.

Identifier	Description
x2DSYMBOL	The EGM describes a 2D symbol. It should only contain objects of type classes common and gr2dobj. Objects of a different type class are ignored while they are being read . Type x2DSYMBOL EGM files contain exactly one 2D object. If this object is made up of several primitive 2D objects, they must be encapsulated by a *Compound* object.

Table C.2: EGM types

C.5 Graphic 2D objects

The graphic 2D objects are combined into one type class carrying the number 2 and the gr2dobj identifier .

The 2D objects are described using a x/y-coordinate system, with the x-axis pointing to the right and the y-axis pointing up. Angle information is given by radian measure, they are mathematically positive (counterclockwise) and are relative to the positive x-axis, if not otherwise specified.

All coordinates are given as single or double precision floating point values. The precision used is specified in each individual 2D object by the S bit of the structure header.

C.5.1 Compound

Graphic 2D objects are nested by the *Compound* compound type. The nesting can also be applied recursively.

A *Compound* does not possess a geometric representation. Ordinarily, it includes at least one object of type class gr2dobj. Other included objects are not permitted.

As an option, the *Compound* object can contain a geometric transformation that is to be applied to the enclosed objects. The transformation is specified either as a rotation of the enclosed objects with subsequent translation or as a 3×3 transformation matrix plus optional inverse transformation matrix.

Type class: 2 / `gr2dobj`

Object type: 1 / `compound`

Flags:
$F_1F_0 = 00$: no transformation
$F_1F_0 = 01$: Rotation and translation
$F_1F_0 = 10$: Transformation matrix
$F_1F_0 = 11$: Transformation matrix and inverse transformation matrix

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
with $F_1F_0 = 01$:			
8	8	FLOAT	α
12	16	FLOAT	x_{offs}, y_{offs}
24	32	End of structure	
with $F_1F_0 = 10$:			
8	8	FLOAT	$mat_{2\times3}$
32	56	End of structure	
with $F_1F_0 = 11$:			
8	8	FLOAT	$mat_{2\times3}$
32	56	FLOAT	$mat^{-1}_{2\times3}$
56	104	End of structure	

If the transformation is entered using rotation and translation, the coordinates of the encapsulated in the coordinate system of the *Compound* object are calculated as follows:

$$x' = x\cos\alpha - y\sin\alpha + x_{offs}$$
$$y' = x\sin\alpha + y\cos\alpha + y_{offs}$$

If the transformation is entered using the transformation matrix, the coordinates of the encapsulated objects in the coordinate system of the *Compound* object are calculated as follows:

$$x' = m_{11}x + m_{12}y + m_{13}$$
$$y' = m_{21}x + m_{22}y + m_{23}$$

The transformation matrices as well as their inverse are saved in EGM format line by line, beginning with the element in the left upper corner (m_{11}). The third line is not saved, since it always reads $0.0, 0.0, 1.0$.

The inverse matrix may be included in the EGM format in order to eliminate the need to determine it while reading the EGM .

C.5.2 Graphic primitives

This section describes all the 2D objects with a graphic representation.

The object definitions contain only the geometric information, such as point coordinates and angles. Attributes, such as color, are saved in a *Attribut–Set* and are set using special attribute objects within it. The graphic primitives use the attributes that are relevant to them at the time of their occurrence in the attribute set. For each individual graphic primitive, these attributes are listed under "Attributes:"

Lines

Type class: 2 / `gr2dobj`

Object type: 256 / `lines`

Attributes: *Color, LineWidth, LineStyle*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	UINT32	n
12	16	FLOAT	$x1_0, y1_0, x2_0, y2_0$
$12 + (n-1) \times 16$	$16 + (n-1) \times 32$	FLOAT	$x1_{n-1}, y1_{n-1}$, $x2_{n-1}, y2_{n-1}$
$16 + n \times 16$	$16 + n \times 32$	End of structure	

Type *Lines* objects represent one or several separated line segments. The n parameter specifies the number of line segments. Its value must be greater than or equal to 1. Every single line segment starts at $x1_i, y1_i$ and ends at $x2_i, y2_i$.

Polyline

Type class: 2 / `gr2dobj`

Object type: 257 / `polyline`

Attributes: *Color, LineWidth, LineStyle*

Flags: $F_0 = 0$: open
$F_0 = 1$: closed

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	UINT32	n
12	16	FLOAT	x_0, y_0
$12 + (n-1) \times 8$	$16 + (n-1) \times 16$	FLOAT	x_{n-1}, y_{n-1}
$16 + n \times 8$	$16 + n \times 16$	End of structure	

Depending on flag F_0, type *Polyline* objects represent an open or closed line. The n parameter is bigger than the number of line segments by 1. It must be greater than or equal to 2. If flag F_0 is set, the last point (x_{n-1}, y_{n-1}) is connected to the first point (x_0, y_0).

Points

Type class: 2 / `gr2dobj`

Object type: 258 / `points`

Attributes: *Color, PointSize, PointType*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	UINT32	n
12	16	FLOAT	x_0, y_0
$12 + (n-1) \times 8$	$16 + (n-1) \times 16$	FLOAT	x_{n-1}, y_{n-1}
$16 + n \times 8$	$16 + n \times 16$	End of structure	

Type *Points* objects represent one or several points. The parameter n specifies the number of points. Its value must be greater than or equal to 1.

Circle

Type class: 2 / `gr2dobj`

Object type: 259 / `circle`

Attributes: *Color, LineWidth, LineStyle*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	FLOAT	x_{center}, y_{center}
16	24	FLOAT	r
24	32	End of structure	

Circle type objects represent a circle with a radius of r, whose center is determined by x_{center}, y_{center}. The radius r must be greater than 0.0.

Arc

Type class: 2 / `gr2dobj`

Object type: 260 / `arc`

Attributes: *Color, LineWidth, LineStyle*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	FLOAT	x_{center}, y_{center}
16	24	FLOAT	r
20	32	FLOAT	$\alpha_{start}, \alpha_{end}$
24	48	End of structure	

Arc type objects represent an arc with a radius of r, whose center is determined by x_{center}, y_{center}. The arc is drawn from angle α_{start} to angle α_{end} in mathematically positive direction.

Ellipsis

Type class: 2 / `gr2dobj`

Object type: 261 / `ellipse`

Attributes: *Color, LineWidth, LineStyle*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	FLOAT	x_{center}, y_{center}
16	24	FLOAT	x_{radius}, y_{radius}
24	40	FLOAT	α
32	48	End of structure	

Type *Ellipse* objects represent an ellipsis whose center is determined by x_{center}, y_{center}. The radius of the not rotated ellipsis in the direction of the x-axis is x_{radius}, the radius in the direction of the y-axis is y_{radius}. The rotation angle of the ellipsis around its center is α.

Text

Type class: 2 / `gr2dobj`

Object type: 262 / `text`

Attributes: *Color, Font, FontHeight, FontAspectRatio, FontAlignment*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	FLOAT	x_{origin}, y_{origin}
16	24	FLOAT	α
20	32	FLOAT	*width*
20	32	STRING	*text*
roundup$(23 + len, 8)$	roundup$(35 + len, 8)$	End of structure	

Type *Text* objects represent the *text* text whose baseline goes through the reference point x_{origin}, y_{origin} and around which the reference point is rotated by the angle α. The *width* parameter indicates the width of the text that was not rotated. The position of the left side of the first letter on the baseline is determined as described in section C.5.3.

A system that uses the same fonts as EGR GF can ignore the *width* parameter when reading, since the width of the text is determined by the *FontAspectRatio* attribute. Other systems must ignore the *FontAspectRatio* attribute and must instead modify the text to match the width specified in the *width* parameter.

Convex Polygon

Type class: 2 / `gr2dobj`

Object type: 263 / `cvx_polygon`

Attributes: *Color, FillStyle*

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	UINT32	n
12	16	FLOAT	x_0, y_0
$12 + (n - 1) \times 8$	$16 + (n - 1) \times 16$	FLOAT	x_{n-1}, y_{n-1}
$16 + n \times 8$	$16 + n \times 16$	End of structure	

Type *Convex Polygon* objects represent a convex polygon. In a convex polygon, no edges intersect and all interior angles are smaller than or equal to π.

The n parameter containing the number of corner points of the polygon must be greater than 2.

C.5.3 Attributes

As described in section C.5.2, the graphic 2D primitives contain no attributes, but only pure geometry information. They use the attributes relevant to them that are specified in the current attribute set at the time of their occurrence in the EGM format.

Since the EGM permits the hierarchical structuring of data in tree form, it must also support a hierarchy of attribute sets. This hierarchy is built in tree form when the EGM is read, with all

nodes of the tree that are located on the path from the root to the current leaf existing. This is implemented by a stack of Attribute sets.

To keep the stack of the attribute sets consistent with the EGM hierarchy, every compound type *End* object removes the necessary number of attribute sets from the stack, until the number of attribute sets on the stack is the same as the number that the corresponding *Begin* object found on the stack. Similarly, an error occurs when an attribute set is removed from the stack, so that the number of attribute sets on the stack becomes smaller than the number of the attribute sets that the most interior compound type *Begin* object found on the stack.

Attribute values

The coordinate values given with the graphic 2D primitives generally (i.e. if there was no scaling) can be interpreted as meters. The dimensions of the display on screen, on the printer or the plotter depend on the scale used.

Compared with this, many attribute values for the graphic 2D object are given independently from the scale, since on the one hand this complies with the capabilities of common output devices and on the other hand scaling is oftentimes not desired. The basic unit in this case is the point. The following interrelations are valid for the size of one point:

$$1\text{pt} = 0.0352\overline{7}\text{cm} \qquad 1\text{pt} = 0.352\overline{7}\text{mm} \qquad 1\text{pt} = 0.013\overline{8}\text{in}$$
$$1\text{cm} = 28.346457\text{pt} \qquad 1\text{mm} = 2.8346457\text{pt} \qquad 1\text{in} = 72\text{pt}$$

This definition of a point is compatible with the definition of a PostScript point, but somewhat differs from the definition used during letterpress printing. This definition stated 1in = 72.27pt and 1in = 72bp, where bp stands for *Big Point*.

When outputting the point on a printer or plotter, the size of the point should be observed exactly. When outputting on the screen it is acceptable to display a point as a pixel using the customary resolution of 75 to 100 dpi.

Push Attrib

Type class: 2 / `gr2dobj`

Object type: 512 / `push_attr`

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	End of structure	

When an *PushAttrib* object is being read, the current status of the attribute set is put on the attribute stack.

Pop Attrib

Type class: 2 / `gr2dobj`

Object type: 513 / `pop_attr`

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	End of structure	

When a *PopAttrib* object is being read, the top attribute set is copied into the current attribute set and is removed from the stack. An error occurs when the number of the attribute sets on the stack is subsequently smaller than when the most interior compound type *Begin* object was read.

Init Attrib

Type class: 2 / `gr2dobj`

Object type: 514 / `init_attr`

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	End of structure	

When a *InitAttrib* object is read, the current attribute set is reset to the default values. The default values are specified with the individual attributes.

Color

Type class: 2 / `gr2dobj`

Object type: 515 / `color`

Default: $r/g/b = 0/0/0$

Parameter:

Offset	Type	Parameter
8	UINT16	*red*
10	UINT16	*green*
12	UINT16	*blue*
16	End of structure	

The specification of numbers is accomplished in the RGB system. Here, an unsigned integer value ranging from 0 to 65535 is specified for each color component. 0 Corresponds to minimum intensity and 65535 corresponds to maximum intensity.

Line Width

Type class: 2 / `gr2dobj`

Object type: 516 / `line_width`

Default: 1.0

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	FLOAT	*linewidth*
16	16	End of structure	

Type *LineWidth* objects set the line width attribute in the current attribute set. The line width is specified in points (pt). The value must be greater than 0.0. Invalid values are interpreted as 1.0.

Line Style

Type class: 2 / `gr2dobj`

Object type: 517 / `line_style`

Default: -1

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	UINT32	$linestyle$
12	16	FLOAT	$factor$
16	24	End of structure	

Type *LineStyle* objects set the line type in the current attribute set.

The following values are predefined for the *linestyle* parameter:

Constant	line type
-1	default value
0	solid line
1	dashed line
2	dotted line
3	dash–point line
4	dash–point–point line
5	dash–point–point–point line

Table C.3: Predefined line types

If the *LineStyle* is -1, the *LineStyle* set by the parent object will apply.

The parameter is specified in points (pt) and determines how the line is stretched. Depending on the line type, the parameter affects the display of the line as follows:

Line type	meaning of $factor$ factor
dashed	length of the displayed and hidden segments
dotted	distance between the center points of two neighboring points
dash–point	length of the displayed line segment and half length of the hidden line segments
dash–2–point	length of the line segment and a third of the hidden line segments
dash–3–point	length of the line segment and a fourth of the hidden line segments

Table C.4: Effects of the factor on the line type

Point Size

Type class: 2 / `gr2dobj`

Object type: 518 / `point_size`

Default : 0.1

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	FLOAT	*pointsize*
16	16	End of structure	

In the current attribute set, type *PointSize* objects set the size of a point. The point size is used only if the set point type is a vector point. In this case, the vertices of the point are calculated as described in table C.6, with the d variable being the point size set using *PointSize*.

Point Type

Type class: 2 / `gr2dobj`

Object type: 522 / `point_style`

Default: 0xffffffff

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	UINT32	*pointstyle*
16	16	End of structure	

Points can be differentiated into bitmap points and vector points. The orientation of bitmap points is always the same and they are always the same size irrespective of the scaling and the set point size. The size and orientation of vector points is determined by the set point size as well as the transformation of a parent compound object, if necessary.

The specification of points is done using a bit mask so that different bitmap points as well as different vector points can be combined with each other[2]. Vector points may be combined in any way; the combination of bitmap points is limited to point types of different classes.

Tables C.5 and C.6 contain the constants for the specification of bitmap and vector points. Table C.6 contains the provision for calculating the vertices of the points, with the d variable being the point size set using *PointSize*.

Font

Currently, different fonts are not supported.

[2] i.e., that bitmap and vector points cannot be combined.

Constant	diameter	constant	diameter
−1	default value		
filled circle:		cross:	
0x40000001	1 pixel	0x40000008	5 pixels
0x40000002	3 pixels	0x40000010	10 pixels
0x40000003	5 pixels	0x40000018	15 pixels
0x40000004	7 pixels	0x40000020	20 pixels
0x40000005	9 pixels	0x40000028	25 pixels
0x40000006	11 pixels	0x40000030	30 pixels
0x40000007	13 pixels	0x40000038	40 pixels
diagonal cross:		circle:	
0x40000040	5 pixels	0x40000200	5 pixels
0x40000080	10 pixels	0x40000400	10 pixels
0x400000c0	15 pixels	0x40000600	15 pixels
0x40000100	20 pixels	0x40000800	20 pixels
0x40000140	25 pixels	0x40000a00	25 pixels
0x40000180	30 pixels	0x40000c00	30 pixels
0x400001c0	40 Pixel	0x40000e00	40 Pixel
square:			
0x40001000	5 pixels	0x40002000	10 pixel
0x40003000	15 pixels	0x40004000	20 pixel
0x40005000	25 pixels	0x40006000	30 pixel
0x40007000	40 pixels		

Table C.5: Bitmap point types and their constants

Font Height

Type class: 2 / `gr2dobj`

Object type: 519 / `font_height`

Default: 0.1

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	FLOAT	$fontheight$
16	16	End of structure	

In the current attribute set, type *FontHeight* objects set the font height, measured from the baseline to the upper edge of normal capital letters. Thus, the height specification does not take into consideration ascenders and descenders.

The font height is **not** given in points.

Constant	Type
`0x00000001`	small cross: $[(-0.5 \times d, 0), (0.5 \times d, 0)], [(0, -0.5 \times d), (0, 0.5 \times d)]$
`0x00000002`	large cross: $[(-\sqrt{0.5} \times d, 0), (\sqrt{0.5} \times d, 0)], [(0, -\sqrt{0.5} \times d), (0, \sqrt{0.5} \times d)]$
`0x00000004`	small cross rotated by 45°: $[(-\sqrt{0.125} \times d, -\sqrt{0.125} \times d), (\sqrt{0.125} \times d, \sqrt{0.125} \times d)], [(-\sqrt{0.125} \times d, \sqrt{0.125} \times d), (\sqrt{0.125} \times d, -\sqrt{0.125} \times d)]$
`0x00000008`	large cross rotated by 45°: $[(-0.5 \times d, -0.5 \times d), (0.5 \times d, 0.5 \times d)], [(-0.5 \times d, 0.5 \times d), (0.5 \times d, -0.5 \times d)]$
`0x00000010`	circle, diameter is d, center is at $(0, 0)$
`0x00000020`	square: $[(0.5 \times d, 0.5 \times d), (-0.5 \times d, 0.5 \times d), (-0.5 \times d, -0.5 \times d), (0.5 \times d, -0.5 \times d)]$
`0x00000040`	square rotated by 45°: $[(0.5 \times d, 0), (0, 0.5 \times d), (-0.5 \times d, 0), (0, -0.5 \times d)]$
`0x00000080`	isosceles triangle, vertex to the right: $c = (0.5 - \sqrt{\frac{3}{4}}) \times d, \quad [(0.5 \times d, 0), (c, 0.5 \times d), (c, -0.5 \times d)]$
`0x00000100`	isosceles triangle, vertex up: $c = (0.5 - \sqrt{\frac{3}{4}}) \times d, \quad [(0, 0.5 \times d), (-0.5 \times d, c), (0.5 \times d, c)]$
`0x00000200`	isosceles triangle, vertex to the left: $c = (0.5 - \sqrt{\frac{3}{4}}) \times d, \quad [(-0.5 \times d, 0), (c, -0.5 \times d), (c, 0.5 \times d)]$
`0x00000400`	isosceles triangle, vertex down: $c = (0.5 - \sqrt{\frac{3}{4}}) \times d, \quad [(0, -0.5 \times d), (0.5 \times d, c), (-0.5 \times d, c)]$

Table C.6: Vector point types and their constants

Font Aspect Ratio

Type class: 2 / `gr2dobj`

Object type: 520 / `font_aspect_ratio`

Default: 1.0

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	FLOAT	*aspectratio*
16	16	End of structure	

In the current attribute, type *FontAspectRatio* objects set the font aspect ratio. This ratio determines whether the font should appear flattened, ($aspectratio < 1.0$), stretched ($aspectratio > 1.0$), or normal ($aspectratio = 1.0$).

Font Alignment

Type class: 2 / `gr2dobj`

Object type: 521 / `font_alignment`

Default: −1.0

Parameter:

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	FLOAT	*alignment*
16	16	End of structure	

In the current attribute set, type *FontAlignment* objects set the horizontal orientation of the text in relation to its origin.

If *width* is the width of the text, the Δx offset of the left side of the first letter on the baseline in relation to the reference point is determined as follows:

$$\Delta x = -(alignment + 1.0) \times (width/2.0)$$

Layer

Type class: 2 / `gr2dobj`

Object type: 523 / `layer`

Default: `DEFAULT`

Parameter

Offset ($S = 1$)	Offset ($S = 0$)	Type	Parameter
8	8	SYMBOL	*layer*
$\text{roundup}(11 + len, 8)$	$\text{roundup}(11 + len, 8)$	End of structure	

In the current attribute set, type *Layer* objects set the layer. The name of the layer must be made up exclusively of ASCII letters, digits, and underscore and must not start with a digit. This symbol is case sensitive.

The layer is used to control the visibility of objects. Generally it is possible to show and hide all graphic objects belonging to a layer at the same time. The attributes are not allocated to objects using the layer.

In a hierarchical structure of objects, objects belonging to the default layer [3] inherit the layer from the parent object which in turn can be the default layer. If this is the case, the respective object is always displayed[4].

[3] the name of the default layer is `DEFAULT`.

[4] The default layer cannot be hidden.

Appendix D

External data formats

OFML defines the external data formats described below. The corresponding files are located in a library directory or library archive. Global material definitions are located in a global directory with a relative path of *data/material*. The predefined fonts are located in a global directory with a relative path of *data/font*.

External data have to be qualified completely. For example, the taken text resource, *@collision*, must be qualified from the *::ofml::xoi* packet as follows when called:

"::ofml::xoi::@collision"

If a text resource is not qualified, the resource file is first looked for in the package of the immediate type of the instance for which the text resource is to be triggered[1]. If the resource file cannot be found in this package, the search continues in the supertype packages.

D.1 Geometries

- Geometry description files define polygon geometries, which can be loaded into *OFML* directly.
- **Name assignment:** The name of the geometric definition file results from the name of the geometry as it is applied in *OiImport* but without path or extension. Only ASCII characters may be used. However, spaces are not allowed. The extension depends on the individual file. Allowed extensions are:
  - *geo* – polygonal geometries (OFF format)
    In this case, polygons must be defined simple, planar, convex and clockwise.
  - *ipc* – optional polygon colors (OFF format)
    If polygon colors are defined, a material can be allocated on the *OFML* level, but not visualized.

[1] for example, the instance that outputs a message using *oiOutput()*, or the instance that is sent to the *oiGetStringResource()* function

- *vnm* – optional vertex normals (OFF format)
  If no vertex normals are defined, they are generated.
- *3ds* – polygonal geometries (3DS format)
  Only geometries and materials (including textures) are accepted. If polygon colors are defined, a material can be allocated on the *OFML* level, but not visualized.

- **Format:** The formats correspond to the individual definitions of the 3DS format and the OFF format.

D.2 Materials

- Material definition files substitute String identifiers from *OFML* with a corresponding set of material parameters.
- **Name assignment:** The name of a material definition file results from the name of the material in lower case. Only ASCII characters may be used. If a material name consists of more than one word, the words are joined together. In doing so, spaces are eliminated. The file extension is *mat*.

  Example: The "ashnature.mat" file contains the definition of the material *Ash Nature*.

- **Format:** Material definition files are constructed line-by-line and consist of the name of the material and any number of material parameter specifications. A material parameter specification overwrites the initial value of the corresponding material parameter. The following specifications are permitted:
  - *amb Red(Float) Green(Float) Blue(Float)*
    The *amb* key specifies the ambient color of the material. The components are floating-point numbers in the range of $0 \leq z \leq 1$. The initial ambient color is white (1.0 1.0 1.0).
  - *dif Red(Float) Green(Float) Blue(Float)*
    The *dif* key specifies the diffuse color of the material. The components are floating-point numbers in the range of $0 \leq z \leq 1$. The initial diffuse color is white (1.0 1.0 1.0). The ambient and diffuse colors are usually the same.
  - *spe Red(Float) Green(Float) Blue(Float)*
    The *spe* key specifies the specular color of the material. The components are floating-point numbers in the range of $0 \leq z \leq 1$. The initial specular color is black (0.0 0.0 0.0).
  - *shi Shininess(Float)*
    The *shi* key specifies the specular exponent using a positive floating-point number. The higher the exponent is, the lower the spread of the specular highlights. The initial specular exponent has the value of 30.0.
  - *tra Transparency(Float)*
    The *tra* key specifies the transparency using a nonnegative floating-point number that is less than or equal to 1. The value of 0.0 stands for complete impermeability; the value of 1 means complete transparency. The initial transparency is 0.0.

– *ref Refraction(Float)*

The *ref* key specifies the refraction using a positive floating-point number. The initial refraction has a value of 1.0 and is equivalent to the refraction in a vacuum.

– *tex image Format(String) Name(String)*

The *tex* key specifies an image map texture. Initially, no texture is applied in the scope of the material being defined. The supported formats are Targa (*tga*), BMP (*bmp*), JPEG (*jpg*) and SGI RGB (*rgb*). The *Name* parameter specifies the name of the image without path or extension.

– *scale X(Float) Y(Float) Z(Float)*

If a texture has been defined using the *tex* key, the *scale* key specifies the scaling of the texture. This is done using a positive scalar for each dimension. Each initial value is 1.0, meaning the image, regardless of its resolution, is scaled to a size of 1x1m.

– *rotate AngleX(Float) AngleY(Float) AngleZ(Float)*

If a texture has been defined using the *tex* key, the *rotate* key specifies the rotation of the texture by the angle specified in degrees to the corresponding axis. The initial value is 0.00.00.0.

– *prjx*

If an image mapping has been defined using the *tex* key, the *prjx* key specifies the projection of the image on the x-axis.

– *prjy*

If an image mapping has been defined using the *tex* key, the *prjy* key specifies the projection of the image on the y-axis.

– *prjz*

If an image mapping has been defined using the *tex* key, the *prjz* key specifies the projection of the image on the z-axis.

– *prj X(Float) Y(FLoat) Z(Float)*

If an image mapping has been defined using the *tex* key, the *prj* key specifies the projection of the image on the axis specified by *X*, *Y* and *Z*.

– *circ R(Float)*

If an image mapping has been defined using the *tex* key, the *circ* key specifies the mapping of the image on a circle with the radius of *R*

– *sph R(Float)*

If an image mapping has been defined using the *tex* key, the *sph* key specifies the mapping of the image on a sphere with the radius of *R*

– *cyl R(Float) H(Float)*

If an image mapping has been defined using the *tex* key, the *cyl* key specifies the mapping of the image on a cylinder with the radius of *R* and height of *H*.

– *cone R1(Float) R2(Float) H(Float)*

If an image mapping has been defined using the *tex* key, the *cone* key specifies the mapping of the image on a cone with radii of *R1* and *R2* and height of *H*.

- *quad X1(Float) Y1(Float) Z1(Float) X2(Float) Y2(Float) Z2(Float) X3(Float) Y3(Float) Z3(Float) X4(Float) Y4(Float) Z4(Float)*

  If an image mapping has been defined using the *tex* key, the *quad* key specifies the mapping of the image on a common quadrilateral surface with corresponding coefficients.

- *interp Mode(Int)*

  If an image mapping has been defined using the *tex* key, the *interp* key specifies whether interpolation takes place (1) or not (0). The initial value is 1.

- *once Mode(Int)*

  If an image mapping has been defined using the *tex* key, the *once* key specifies whether a repeated mapping of the image takes place (1) or not (0). The initial value is 1.

The use of the material parameter depends on the applied display method. As long as objects already define their own colors and materials, possibly concerning *OiImport*, the materials defined here are not accepted.

In special cases, materials can be specified alternatively without external files. In such cases, the parameter specifications can be entered directly in place of the material name. A material defined in this manner must begin with a '$' sign. Furthermore, semicolons are used in place of the line ends. Using the *mat* key is not permitted in this case.

Example: The "$ amb 1.0 0.0 0.0; dif 1.0 0.0 0.0" string sets the color of red as a material without the use of an external material definition file.

D.3 Fonts

- The fonts supported in *OFML* are based on the fonts created by Dr. A. V. Hershey (U.S. National Bureau of Standards). These are vector fonts, which describe continuous lines.

  The following fonts are to be prepared by an *OFML*-conforming runtime environment:

  - *default*
  - *cyrillic*
  - *cursive*
  - *timesg*
  - *timesi*
  - *timesib*
  - *timesr*
  - *timesrb*

- **Name assignment:** The name of a font results from an identifier (the name of the font), in which all letters are lower case. The font name does not have an extension.

- **Format:** The format corresponds to the definition of the Hershey font format.

D.4 External Tables

- External tables, such as product databases, are saved in a simple text format. Data records are separated by line breaks. The individual fields of a data record have fixed lengths; there are no field separators. Fields that are shorter than their corresponding lengths are filled in to achieve their fixed lengths.

  This generic table format can be read within *OFML* using the global *oiTable()* function (Chapter 6).

- **Name assignment:** The name of a product database can be chosen freely.

- **Format:** The following field types are understood:

  - Character strings. These are left-justified and, as necessary, filled with spaces, except the last field in a data record. In the later case, the character string is closed with the line end.

    If the last field is the empty string, it can be omitted completely. In this case, the field before the last is handled according to the rules above. If the field before the last is empty as well, the rules can be applied again.
  - Integers are right-justified and are filled with zeros.
  - Fixed point numbers are right-justified and are filled with zeros; the decimal point is left out.
  - Fields that serve as the first key for access must be sorted in ascending order.

D.5 Text Resources

- Text resources substitute a symbol identifier from *OFML* with a corresponding text from an external file. This might find use, for example, for property names, description texts or output texts.

- **Name assignment:** The name of a resource file is made up a link of the library names and the corresponding ISO country abbreviation, separated by an underscore. The file extension is *sr*. All letters are lower case.

  Example: The "room_de.sr" file contains the German text resources for the *Room* library.

- **Format:** The relevant lines are formatted as follows:

  ```
  @SYMBOL=<Text>
  ```

  Here, the left expression is the assignment of a valid symbol as understood in *OFML*. The right expression is a text in any 8-bit character format. For Western Europe, the ISO-Latin 1 (ISO 8859-1) character set is applied. Another valid character format, for example, is UTF 8. For use with formatted output, the text can be a format character string (Section 6.1). All other lines are ignored and can be used for structuring and comments. Based on convention, a single pound sign indicates a comment. Two pound signs leads to structuring of the resource file. By convention, the following structurings are established:

– *## messages:* – Character strings that follow stand for messages, warnings etc.
– *## properties:* – Character strings that follow stand for property titles.

D.6 Archives

- Archives represent containers, each of which usually contains all of the files belonging to a library. The archive structure corresponds to the format used by the UNIX SVR4 *ar* utility program.

- **Name assignment:** The name of an archive is lower case. The extension is *alb*. No other standards apply.

- **Format:** All archives begin with the string, `!<arch>\n`. The rest of the archive is made up of objects, each of which consists of a header and the actual content of the file.

  The header consists of six, fixed-length fields of ASCII characters. With two exceptions (see below, these fields contain the file names (16 characters), the most recent time the file was modified (12 characters), the user and group numbers of the file owner (6 characters each), the access mode (8 characters) and the size of the file in bytes. All numerical fields are decimal, except the access mode, which is specified in octal. The header is closed with the `‘\n` string.

  File names that are longer than 16 characters are treated differently. If at least one such file exists in the archive, the first object in the archive is not a file, but a table named `//`, which contains the long file names. In place of the file name in the header of each file is the character, `/`, followed by a number that indicates the offset of the file name in regard to the table.

  A line break is appended to files having an uneven number of bytes, which, however, has no effect on the size specified in the header. This ensures that every object starts on an even-numbered address.

  ¿From the *OFML* runtime environment, each archive becomes a special file called `__attrib`, which contains the attributes of the archive. Each attribute claims one line and contains a key and value pair, the elements of which are separated by a space. The following attributes are standardized:

`version`	The version of the archive, consisting of two numbers separated by a period.
`valid_span`	The validity range of the archive, consisting of two date entries separated by an underscore.
`pwdcheck`	A password for checking encryption.
`md5sum`	The MD5 checksum of the archive.

  All attributes are optional. Any number of attributes can be added.

Appendix E

Format Specifications

E.1 Format Specifications for Properties

This section describes syntax and meaning of format specifications for properties. Format specifications can be entered during the setting of properties in the *setupProperty()* function of the *Property* interface (Section 4.4).

The format specification has one of the following forms:

property-format:
"*@L*"
"*@A*"
"*%[-][width][.prec]type*"

The first two formats can be used with properties of the `"f"` base type and indicate that the property value is a length or angle measurement and that the unit of measure set by the user should be used for the presentation of entry of the value. The property editor must then perform a conversion between the user-defined unit of measure and the unit of measure used in OFML for length and angle measurements (m or rad).

The third form is used if an OFML object intends to force a special format for the presentation or entry of property values. The format character string of this form begins with a % sign. Afterwards, the following specifiers in the respective sequence are allowed:

- an optional left-align indicator – "*[-]*"
- an optional width specifier – *[width]*
- an optional precision specifier – *[.prec]*
- a required type specifier – *type*

  The following discrete value range is predefined for the type specifier:

– Decimal number (*Int*) – *d*

  The argument must be an *Int* value. The value is converted to a character string that contains the decimal places. If the format specification contains a precision specifier, the specifier indicates that the resulting character string contains at least the specified number of places. If the value features fewer places, it is filled with zeros dependent upon the optional left-align indicator. If the left-align indicator is given, zeros are filled in on the right side. Otherwise, zeros are filled in on the left side.

  If the width specifier is used, it indicates the maximum number of places that the resulting character string may possess. If width and precision specifier are used, then the following applies: $width \geq prec$.

– Floating point number (*Float*) – *f*

  The argument must be a floating point number. The value is converted to a character string of the form "-*ddd.ddd...*". The resulting character string starts with a minus sign if the number is negative. The number of places after the decimal point is indicated by the precision specifier. If no precision specifier is given, 2 is assumed as the number of decimal places after the period.

  If a width specifier is used, it indicates the exact width of the resulting character string. Here, the minus sign is counted, but the decimal point is not. If the value has fewer digits, zeros are filled in on the left side. The left-align indicator is ignored, if present. If the value has more digits, the leading places are suppressed.

– Character string (*String*) – *s*

  The argument must be a character string. It is inserted instead of the format specifier. If the precision specifier is indicated, it defines the maximum length of the resulting character string. If the length of the argument exceeds the maximum length, the character string is cut off accordingly.

  If the format specification contains a width specifier, the specifier indicates the minimum number of characters of the resulting character string. If the character string features fewer characters, the resulting character string is filled with spaces on the left side (without set left-align indicator) or on the right side (with set left-align indicator).

E.2 Definition Format for Properties

This section describes the format of a property definition description that describes all properties of an instance and is delivered as the result of the *getProperties()* function in the *Property* interface (Section 4.4).

The following rules apply to the format of the definition of properties:

- The description of all properties consists of the descriptions for each individual property separated from each other by semicolons.
- Each property definition reflects the data that are transferred to the *Property::setupProperty()* function and consists of a set of required and optional specifications that are separated by semicolons.

- A semicolon can be followed by a random number of spaces.
- The first specification of a property definition is the key specifier:
  - *k* <*str*> – key of the specified property.
- The last specifier of a property specification is the type specifier. It must have one of the following values:
  - *b* – a boolean type (0 or 1).
  - *i* – a decimal type.
  - *f* – a floating point number type.
  - *s* – a character string type.
  - *ch* <*str*>*n* – a selection list with $n, n > 0$ character strings for use with character string entry.
  - *chf* <*str*> – a selection list whose possible character strings are delivered by the listed function.
  - *u* – a user-defined type with a given editor identification.
- Additional optional specifiers between key and type specifier are:
  - *n* <*str*>*n* – the name of the property.
  - *d* <*str*>*n* – the initial value of the property.
  - *mn* <*str*> – minimum value of a decimal or floating point number or minimum number of characters in a character string property.
  - *mx* <*str*> – maximum value of a decimal or floating point number or maximum number of characters in a character string property.
  - *fmt* <*str*> – C-type format specifier (Section E.1)

Appendix F

Additional Types

The following defined types are not a direct component of OFML, but they can be used in OFML-conform libraries. The *Base* interface is implemented, but with a few specific limitations in each case.

F.1 Interactor

Description

- *Interactor* implements the base class for interactors.
- **Interface(s):** *Base* with limitations:

  The functions *isCat()*, *hide()*, *show()*, *isHidden()*, *selectable()*, *notSelectable()*, *isSelectable()*, *setCutable()*, *isCutable()*, *enableCD()*, *disableCD()*, *isEnabledCD()*, *measure()*, and *unMeasure()* are not available. The instance variable *mIsCutable* is not available.

Initialization

- *Interactor(pFather(MObject), pName(Symbol))*

  The function initializes an instance of the *Interactor* type.

Methods

- *final makeVisible(pType(Type) ...)* → *Void*

  The function generates an instance of the indicated type as element which represents the geometry of the interactor. After the type argument, additional constructor arguments may follow. If the interactor is already visible, the function is without effect. The transfer of ZERO makes the interactor visible.

- *final isVisible()* → *Int*

  The function delivers 1 if the interactor is visible, otherwise 0.

- *final getState()* → *Symbol*

  The function delivers the state of the interactor which is described by one of the symbols *@ENABLED*, *@DISABLED* or *@ACTIVE*.

- *final enable()* → *Int*

  It sets the interactor in the state "free" (*@ENABLED*) and always delivers 1 (success).

- *final disable()* → *Int*

  It sets the interactor in the state "blocked" (*@DISABLED*) and always delivers 1 (success).

- *final activate()* → *Int*

  It sets the interactor in the state "active" (*@ACTIVE*) where it must already be in the state *@ENABLED*. It delivers 1 with success and 0 if the interactor was blocked.

F.2 Light

Description

- *Light* is a globally acting active light source that is, however, integrated in an instance hierarchy. The following procedure applies for converting the light source in a local lighting model: If the light source features children, it is a directional point light source. It is located in the local origin and lights along the local negative y-axis. The aperture of the cone of light results from the arcus tangent of the relationship of the maximum z-value of the local delimiting volume of the light source to the negative minimum y-coordinate of the local delimiting volume.

  If the light source does not have any children or the minimum y-coordinate of the local delimiting volume is equal to 0.0, it is a nondirectional point light source.

  In a global lighting model, this explicit differentiation is unnecessary.

- The *Light* type may **not** be derived.

- **Interface(s):** *Base* with limitations:

  The functions *getType()*, *isCat()*, *setCutable()*, *isCutable()*, *enableCD()*, *disableCD()*, *isEnabledCD()*, *measure()*, and *unMeasure()* are not available. The instance variable *mIsCutable* is not available.

Initialization

- *Light(pFather(MObject), pName(Symbol))*

  The function initializes an instance of the *Light* type.

Methods

- *final setColor(pColor(Float[3]))* → *self*

  The function sets the color of the light source. The elements of the *pColor* vector must be real numbers in the interval from 0.0 to 1.0 where the interval boundaries are acceptable values. The vector elements are interpreted as amplitudes of the wavelengths red, green, and blue. Their linear combination results in the actual color. The initial light color is white.

- *final getColor()* → *Float[3]*

  The function furnishes the current light color of the implicit instance.

- *final on()* → *self*

  The function activates the light source.

- *final off()* → *self*

  The function deactivates the light source.

- *final isOn()* → *Int*

  The function signals the status of the light source via its return value: activated (1) or deactivated (0).

F.3 MLine

Description

- *MLine* implements an automatic dimensioning primitive that automatically dimensions the higher-order object in the hierarchy.

  The line thickness measures 1 in the smallest representation unit of the image space in each case, e.g., 1 pixel.

- The *MLine* type may **not** be derived.

- **Interface(s):** *Base* with limitations:

  The functions *getType()*, *isCat()*, *hide()*, *show()*, *isHidden()*, *selectable()*, *notSelectable()*, *isSelectable()*, *setCutable()*, *isCutable()*, *enableCD()*, *disableCD()*, *isEnabledCD()*, *measure()*, and *unMeasure()* are not available. The instance variable *mIsCutable* is not available.

Initialization

- *MLine(pFather(MObject), pName(Symbol), pDirection(Symbol))*

  The function initializes an instance of the *MLine* type. The *pDirection* parameter defines how the topologically higher-order primitive is dimensioned. Either the width, the height, or the depth of the local delimiting volume of the father is dimensioned. The following symbols are allowed:

@NX The width is dimensioned at the bottom rear. The dimensioning lies in the local x-y-plane of the father and can be read from the front.

@NXG The width is dimensioned at the bottom rear. The dimensioning lies in the local x-z-plane of the father and can be read from the front and the top.

@NXT The width is dimensioned at the bottom rear. The dimensioning lies in the local x-z-plane of the father and can be read from the rear and the top.

@PX The width is dimensioned at the top rear. The dimensioning lies in the local x-y-plane of the father and can be read from the front.

@PXT The width is dimensioned at the bottom front. The dimensioning lies in the local x-z-plane of the father and can be read from the front and the top.

@NY The height is dimensioned from the left rear. The dimensioning lies in the local x-y-plane of the father, can be read from the front, and is aligned from bottom to top.

@PY The height is dimensioned from the right rear. The dimensioning lies in the local x-y-plane of the father, can be read from the front, and is aligned from bottom to top.

@NZ The depth is dimensioned from the bottom left. The dimensioning lies in the local y-z-plane of the father and can be read from the left.

@NZT The depth is dimensioned from the bottom left. The dimensioning lies in the local x-z-plane of the father and can be read from the left and the top.

@PZ The depth is dimensioned from the bottom right. The dimensioning lies in the local y-z-plane of the father and can be read from the right.

@PZT The depth is dimensioned from the bottom right. The dimensioning lies in the local x-z-plane of the father and can be read from the right and the top.

Methods

- *final setMaterial(pMaterial(String)) → self*

  The specified material is assigned. The ambient component of the material is assigned as color during the display. The presentation should be done without considering the lighting and tint.

- *final getMaterial() → String*

  The function delivers the currently valid material of the implicit instance.

- *final setOffset(pOffset(Float)) → self*

  This function sets the offset of the dimension line with respect to the edge to be dimensioned. The initial offset measures 0.1.

- *final getOffset() → Float*

  The function furnishes the current offset of the implicit instance.

- *final setText(pText(String)) → self*

  Initially, entities of *MLine* automatically dimension the respective edge of the delimiting volume of the father object and automatically adjust to the dimensional changes. However, by using this function the text can be set explicitly. In this case, the *pText* parameter presents the text to be displayed by means of an ASCII character string.

- *final getText()* → *String*

  The function delivers the currently displayed text.

F.4 MSymbol

Description

- *MSymbol* implements a polymorphic dimensioning primitive. All variants are generated in the local x-y-plane. The z-coordinate is always 0. Coordinates with respect to this plane are represented by a vector with 2 elements (x- and y-value, in this order).

  The line thickness measures 1 in the smallest representation unit of the image space in each case, e.g., 1 pixel.

- The *MSymbol* type may **not** be derived.

- **Interface(s):** *Base* with limitations:

  The functions *getType()*, *isCat()*, *hide()*, *show()*, *isHidden()*, *selectable()*, *notSelectable()*, *isSelectable()*, *setCutable()*, *isCutable()*, *enableCD()*, *disableCD()*, *isEnabledCD()*, *measure()*, and *unMeasure()* are not available. The instance variable *mIsCutable* is not available.

Initialization

- *MSymbol(pFather(MObject), pName(Symbol), pMode(Symbol), pValues(Float[][2]))*

  The function initializes an instance of the *MSymbol* type. In this context, the *pMode* parameter together with the variable *pValues* parameter specifies the implementation of *MSymbol*. The evaluation of *pValues* is dependent upon the assignment of *pMode*. The following symbols may be used for *pMode*:

@ARCLINE No contour of a segment of a circle is generated. The origin of the corresponding circle is indicated by *pValues[0]*. *pValues[1][0]* defines the radius of the circle through a positive number. *pValues[1][1]* defines the length of the line in the radian measure through a non-negative number. If the length is positive, the line starts at

(pValues[0][0], pValues[0][1]+pValues[1][0])

in clockwise direction. If it is negative, it starts at the same point, but runs in counter-clockwise direction.

@CIRCLE A filled circle is generated in the local origin. *pValues[0][0]* defines the radius of the circle through a positive number.

@POLYLINE A continuous line is generated that connects the given points in the respective order. The last and first point are not connected. The orientation is not taken into account.

@RECTANGLE A filled rectangle is generated. *pValues[0]* describes the lower left corner. *pValues[1]* describes the upper right corner.

@X_CIRCLE A circle is generated. *pValues[0]* defines the origin of the circle with respect to the local coordinate system. *pValues[1][0]* defines the radius of the circle through a positive number. If *pValues[1][1]* equals 0.0, only the contour is shown. Otherwise, a filled circle is drawn.

Methods

- *final setMaterial(pMaterial(String)) → self*

  The specified material is assigned. The ambient component of the material is assigned as color during the display. The presentation should be done without considering the lighting and tint.

- *final getMaterial() → String*

  The function delivers the currently valid material of the implicit instance.

F.5 MText

Description

- *MText* implements a vector-text-primitive.

  The line thickness measures 1 in the smallest representation unit of the image space in each case, e.g., 1 pixel.

- The *MText* type may **not** be derived.

- **Interface(s):** *Base* with limitations:

  The functions *getType()*, *isCat()*, *hide()*, *show()*, *isHidden()*, *selectable()*, *notSelectable()*, *isSelectable()*, *setCutable()*, *isCutable()*, *enableCD()*, *disableCD()*, *isEnabledCD()*, *measure()*, and *unMeasure()* are not available. The instance variable *mIsCutable* is not available.

Initialization

- *MText(pFather(MObject), pName(Symbol), pText(String))*

  The function initializes an instance of the *MText* type. The *pText* parameter specifies the text to be displayed in form of an ASCII character string.

Methods

- *final setMaterial(pMaterial(String)) → self*

  The specified material is assigned. The ambient component of the material is assigned as color during the display. The presentation should be done without considering the lighting and tint.

- *final getMaterial() → String*

  The function delivers the currently valid material of the implicit instance.

- *final setFont(pFont(String)) → self*

  The specified font is assigned. *pFont* specifies the font through a corresponding font name without path or extension information in accordance with Chapter D.

- *final getFont()* $\rightarrow$ *String*

  The function furnishes the current font of the implicit instance.

- *final setText(pText(String))* $\rightarrow$ *self*

  The test to be displayed is set anew through the ASCII character string *pText*.

- *final getText()* $\rightarrow$ *String*

  The function furnishes the current text of the implicit instance.

- *final setScale(pScale(Float))* $\rightarrow$ *self*

  The positive *pScale* parameter sets the scaling of the text. The initial scaling measures 0.05.

- *final getScale()* $\rightarrow$ *Float*

  The function furnishes the current scaling of the implicit instance.

- *final setAlignment(pAlignment(Symbol))* $\rightarrow$ *self*

  The *pAlignment* parameter determines the horizontal alignment of the text. The following symbols can be used here:

@LEFT The text is left-aligned with respect to the local reference point.

@CENTER The text is centered with respect to the local reference point.

@RIGHT The text is right-aligned with respect to the local reference point.

  The initial alignment is *@CENTER*.

- *final getAlignment()* $\rightarrow$ *Symbol*

  The function furnishes the current alignment of the implicit instance.

- *final setMode(pMode(Symbol))* $\rightarrow$ *self*

  The *pMode* parameter sets the presentation mode of the text. The following symbols are allowed here:

@NORMAL The text is shown in normal mode.

@UNDERLINE The text is highlighted through underlining.

@BOX The text is highlighted by a box.

  The initial display mode is *@NORMAL*.

- *final getMode()* $\rightarrow$ *Symbol*

  The function furnishes the current display mode of the implicit instance.

Appendix G

Applied Notation

G.1 Class Diagrams based on Rumbaugh

The notation used in this document for class diagrams is a modified form of the notation by *Rumbaugh* [Rumb91].

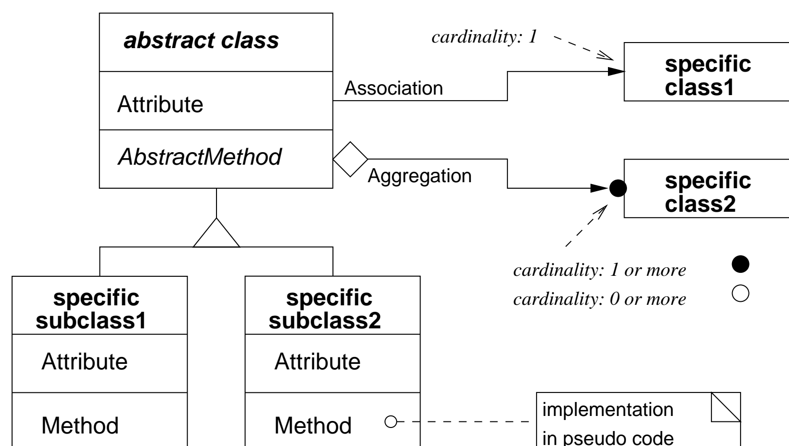

Figure G.1: Modified Rumbaugh Notation for Class Diagrams

In object-oriented software engineering, class diagrams are used to visualize the properties (attributes, methods) of classes and relationships between classes. Principally, there are three **types of relationship**:

- *Vererbung* (Inheritance).
  A subclass inherits the properties of its super-class(es).

- *Aggregation* (Consists Of).
  An instance of a class (an object) contains (consists of) one or more object(s) of another class.

- *Assoziation bzw. Bekanntschaft* (Acquaintance).
  An object of a class "knows"' an object of another class or is associated with it.

In abstract models, the attribute and/or method part of a class can be omitted.

Appendix H

Categories

The categories outlined below are predefined according to the definition. The use of these categories is optional; the applicability and readability of data acquired in OFML increases accordingly if these categories are used.

H.1 Interface Categories

For each OFML interface, a predefined corresponding category exists whose symbolic designator is formed by the prefix "'IF_"'[1] and the name of the interface, e.g., *@IF_Article*. In this way, every instance of an OFML type can be queried using the *isCat()* function whether it implements a special interface.

H.2 Material Categories

The following categories are predefined to designate the assignment of a geometric object or a complex object to a certain material category:

- *@FRONT* – The object belongs to the front of a complex object or represents it.
- *@GRIFF* – The object belongs to the handle of a complex object or represents it.
- *@KORPUS* – The object belongs to the corpus of a complex object or represents it.
- *@KRANZ* – The object belongs to the border of a complex object or represents it.
- *@RUECK* – The object belongs to the back of a complex object or represents it.
- *@SOCKEL* – The object belongs to the base of a complex object or represents it.

[1] acronym for interface

- *@S_FUSS* – The object belongs to the foot of a chair or represents it.
- *@S_LEHNE* – The object belongs to the back rest of a chair or represents it.
- *@S_SITZ* – The object belongs to the seat of a chair or represents it.
- *@T_FUSS* – The object belongs to the foot of a table or represents it.
- *@T_GESTELL* – The object belongs to the stand of a table or represents it.
- *@T_GEST_ABDECK* – The object belongs to a lateral stand cover or represents it.
- *@T_KANTE* – The object belongs to the edge of a table top or represents it.
- *@T_PLATTE* – The object belongs to a table top or represents it.

H.3 Planning Categories

The following categories are predefined to designate the ability of adding sections of an object:

- *@CEILING_ELEM* – The object (e.g., a ceiling lamp) can be planned below an object.
- *@TOP_ELEM* – The object (e.g., a desk lamp) can be planned on the surface of an object.
- *@WALL_ELEM* – The object (e.g., an electrical outlet) can be planned at the surface of an object.

Appendix I

Terms

- **Bounding box**
  - A bounding box is a rectangular volume that minimally encloses a body.
  - The definition of bounding boxes makes reference to the local coordinate system of an object or to the common coordinate system of all objects (world or global coordinate system).
- **Category**
  - A category is a classification of → types or → entities that results from a certain viewing perspective.
  - Categories represent an expansion of the concept of types.
- **Clipboard**
  - A clipboard is a buffer storage in which objects can be placed. Objects can be written to the clipboard using operations such as Cut and Copy. They can be read out again from the clipboard using the Paste operation.
- **Coordinate system**
  - A coordinate system is an orthogonal space defined by three axes (x, y, z) to which position and direction information are referenced.
  - In a specific case, the z- and x-axis span a plane on which the y-axis is located at a right angle.
- **Father object**
  - A father object is an → object from which properties are inherited, e.g., a name space, the spatial modeling, the material, etc.

- **Identity**
  - The identity of an → instance results from a → name in the hierarchical name space that exists only once and uniquely describes the position in an instance hierarchy.
- **Instance**
  - An instance is a concrete implementation of a → type. It differs from other entities through a local copy of → attributes, especially through a unique → identity.
  - Synonyms for instance are → object and entity.
- **Interface**
  - An interface is the collection of a number of methods and member variables that a → type must define or implement for interface compatibility.
- **Lighting model**
  - A lighting model uses great simplification to simulate the lighting of bodies (3D objects).
  - For a **local** lighting model, only the lighted object and the light source (distance, materials, etc.) are viewed.
  - For a **global** lighting model, other objects of the → scene that cause shadows or reflections are also included.
- **Name**
  - The (absolute) name of an → instance uniquely describes the topological position of the instance. Alternatively, an instance can also be referenced through a → symbol relating to the respective context or through a variable.
- **Object**
  - From a programmer's view, an object is a synonym for the → instance of a → type. From a user's view, an object represents a certain unit that can be generated, selected, modified, and deleted as a whole.
- **Program**
  - A set of products combined by a manufacturer for functional and/or aesthetic points of view.
  - Synonyms: collection, product line
- **Property (Feature)**
  - A property is a feature of an instance, e.g., a geometric measurement or the designation of an execution that may be changed interactively by the system user with the help of appropriate dialogs (property editors).

- **Symbol**
  - A symbol is a string-like value that is used primarily to designate constants and instance → names.
- **Root object**
  - A root object is an → object that is located at the root of an object hierarchy. Consequently, a root object has no → father object. All objects that are directly located in a → scene are root objects.
- **Scene**
  - A scene is the collection of a number of 3D objects, in the context of OFML also called → entities.
- **Type**
  - A type combines a number of homogenous → entities and defines structure and behavior for them.
  - A type implements one or several → interfaces.
  - A type features no more than one direct super type; its characteristics are inherited from the super type.
  - Class is a synonym for type.
- **Units**
  - If no other specific definition exists, units of length implicitly feature the unit *meter*.
  - If no other specific definition exists, units of angle implicitly feature the unit *radiant*.

Index

Change Status, 81
Spatial Modeling, 82
Spatial model, 95
2D Representation, 86
2D interface, 169
3DS file, 125, 201

ABAP/4, 156
Action, 168
Activation Status of a Property, 94
add()
  MObject, 78
addInfoObj()
  OiPlanning, 136
addPart()
  Complex, 97
addProductDB()
  OiPDManager, 157
  OiPlanning, 139
Archive, 205
Article
  Article, 98
  Information, 99, 100, 158, 162
  information, 14
  Information, general, 99
article2Class()
  OiPDManager, 158
  OiPlanning, 139
article2Params()
  OiPDManager, 158

Base, 79
Basic interfaces, 77
Block, 118
Bounding Box
  global, 84
  global, geometric, 85
  local, 84
  local, geometric, 84

callRules()
  Base, 85
Category, 16
changedPropList()
  Property, 93
Check String, 115
checkAdd()
  Complex, 95
  OiLevel, 165
  OiPart, 151
  OiPlanning, 137
  OiPlElement, 145
  OiProgInfo, 141
checkBorder()
  OiPlanning, 136
checkChildColl()
  Complex, 98
  OiPlanning, 137
checkConsistency()
  Article, 101
  OiPart, 152
  OiPDManager, 159
  OiPlanning, 139
  OiPlElement, 147
  OiProductDB, 162
  OiProgInfo, 141
checkElPos()
  Complex, 97
  OiPlanning, 137
checkObjConsistency()
  OiPlanning, 140
checkPosition()
  OiPlanning, 138
Child, 12, 78
  and instance variable, 12
  Creation and management, 95
  Transformation, 147
Class, 10

class2Articles()
  OiPDManager, 158
clearInfoObjs()
  OiPlanning, 136
clearMethod()
  Complex, 97
clearProductDBs()
  OiPDManager, 157
Clipboard, 80, 96, 110, 114
Collision Detection, 82
Collision detection, 111, 141
  for children, 97
Complex, 94
Condition, 168
Consistency check, 101
Constraint, 168
CREATE_ELEMENT, 103
createOdbChildren()
  OiOdbPlElement, 155
createOdbObjects()
  Base, 87
Cylinder, 119
Cuttability, 80

Database, 115
delegationDone()
  OiPlanning, 134
delInfoObj()
  OiPlanning, 136
delProductDB()
  OiPDManager, 157
Diagram, 216
Dimensioning, 82
disableCD()
  Base, 82
disableChildCD()
  Complex, 97
Dissolving text resources, 113
Distance measurement, 113
doCheckAdd()
  OiPlanning, 137
doSpecial()
  OiPlanning, 140
  OiProgInfo, 141
Dynamic Properties, 86

EasternGraphics Metafile, 180
EGM, 180
Element, 12, 78
  Transformation, 138
elemRotation()
  OiPlanning, 138
  OiPlElement, 147
elemTranslation()
  OiPlanning, 138
  OiPlElement, 147
Ellipsoid, 120
elRemoveValid()
  OiPart, 151
  OiPlElement, 146
enableCD()
  Base, 82
enableChildCD()
  Complex, 97
Environment, 135
Epsilon, eps, 79
Error log, 134
evalPropValue()
  OiPDManager, 158
Existence check, 113
external data, 200
external geometry (ODB)
  2D, 180
Extrusion body, 129

Father, 12
Father-child-relation, 12
Feature, 89
FINISH_DUMP, 107
FINISH_EVAL, 107
finishCollCheck()
  OiProgInfo, 142
Font, 203
Format specifications, 206
Frame, 121

Generating a dump representation, 112
geometric object, 117
Geometry, 117, 200
getAllMatCats()
  Material, 89
  OiPart, 150
  OiPlElement, 144
getArticleFeatures()
  Article, 101
  OiPart, 152

OiPDManager, 160
OiPlElement, 146
getArticleParams()
Article, 100
OiPart, 151
OiPlElement, 146
getArticlePrice()
Article, 100
OiPart, 151
OiPDManager, 159
OiPlElement, 146
OiProductDB, 163
getArticleSpec()
Article, 99
OiOdbPlElement, 154
OiPart, 151
OiPlElement, 146
getArticleText()
Article, 100
OiPart, 152
OiPDManager, 160
OiPlElement, 146
OiProductDB, 163
getBorder()
OiPlanning, 135
getChildren()
MObject, 78
getClass()
MObject, 77
getCMaterial()
Material, 89
OiPart, 150
OiPlanning, 136
OiPlElement, 144
OiProgInfo, 141
getCMaterials()
Material, 89
OiPart, 150
OiPlanning, 136
OiPlElement, 144
OiProgInfo, 141
getDataBasePath()
OiProductDB, 161
getDepth()
Complex, 95
OiPart, 149
OiPlElement, 143
getDistance()
Base, 85
getDynamicProps()
Base, 86
getElements()
MObject, 78
getEnvironment()
OiPlanning, 135
getErrorLog()
OiPlanning, 135
getExtPropOffset()
Property, 92
getFather()
MObject, 78
getFinalArticleSpec()
OiProductDB, 163
getHeight()
Complex, 95
OiLevel, 165
OiPart, 149
OiPlElement, 143
getID()
OiProductDB, 161
OiProgInfo, 141
getInfo()
OiPlanning, 136
getInfoIDs()
OiPlanning, 136
getLanguage()
OiPlanning, 133
getLocalBounds()
Base, 84
getLocalGeoBounds()
Base, 84
getMatCategories()
Material, 88
OiPart, 150
OiPlanning, 136
OiPlElement, 144
OiProgInfo, 141
getMatName()
Material, 89
OiPart, 150
OiPlanning, 136
OiPlElement, 144
OiProgInfo, 141
getMethod()

Complex, 96
getName()
MObject, 78
getOdbInfo()
Base, 86
OiOdbPlElement, 155
getOrderID()
Article, 99
getOrigin()
OiPart, 149
OiPlElement, 143
getPasteMode()
Complex, 96
getPDB_IDs()
OiPDManager, 157
getPDBFor()
OiPDManager, 158
getPDistance()
OiPlElement, 145
getPDManager()
OiPlanning, 139
OiProductDB, 161
getPictureInfo()
Base, 86
getPlanning()
OiPart, 149
OiPlElement, 142
OiProgInfo, 141
OiPropertyObj, 153
getPlanningMode()
OiLevel, 165
getPlanningWall()
OiLevel, 165
getPlElementUp()
OiPlanning, 135
getPosition()
Base, 83
getProductDB()
OiPDManager, 157
getProgPDB()
OiPDManager, 158
getProgram()
Article, 98
getPrograms()
OiProductDB, 161
getPropDefs()
OiProductDB, 161
getPropDescription()
OiProductDB, 163
getProperties()
Property, 92
getPropertyDef()
Property, 92
getPropertyKeys()
Property, 92
getPropertyPos()
Property, 92
getPropInfo()
Property, 94
getPropObj()
OiPlanning, 135
getPropState()
Property, 94
getPropTitle()
Property, 92
getPropValue()
Property, 93
getRegion()
OiPlanning, 134
getResolution()
Base, 81
getRoot()
MObject, 78
getRotation()
Base, 84
getRtAxis()
Base, 84
getTempArticleSpec()
Complex, 96
getTopPlElement()
OiPlanning, 135
getTrAxis()
Base, 83
getType()
MObject, 77
getVarCode()
OiProductDB, 162
getWallOffset()
OiPlElement, 145
getWallParams()
Wall, 164
getWidth()
Complex, 95
OiPart, 149

OiPlElement, 143
getWorldBounds()
Base, 84
getWorldGeoBounds()
Base, 85
getXArticleSpec()
Article, 99
OiPDManager, 159
Global planning object, 132
GO types, 8

hasProductKnowledge()
OiProductDB, 161
hasProperties()
Property, 91
hasProperty()
Property, 92
hide()
Base, 81
hierSelectable()
Base, 79
Hole, 122
Hyperlink, 114

Import of Geometries, 125
Information object, 136
Inheritance of features, 12
Initialization, 16
Instance, 10–12
Identity, 13
identity, 78
Initialization, 16
name, 13
variable, 11, 14
INTERACTOR, 108
Interactor, 17, 209
Interface, 11, 14
Interfaces
Basic interfaces, 77
Categories, 218
invalidatePicture()
Base, 87
isA()
MObject, 77
isCat()
MObject, 78
isCutable()
Base, 80
OiPropertyObj, 153
isElemCatValid()
OiPart, 150
OiPlElement, 144
isElOrderSubPos()
OiPart, 151
OiPlElement, 146
isEnabledCD()
Base, 82
isEnabledChildCD()
Complex, 97
isHidden()
Base, 81
isMatCat()
Material, 88
isSelectable()
Base, 80
isValidForCollCheck()
Complex, 97
OiPlanning, 137
OiProgInfo, 142

Language Selection, 133
Light, 210
Light source, 210
Link, 114

Material, 87
-definition, 201
Categories, 88, 218
measure()
Base, 82
Measurement line, 211
Measurement symbol, 213
Measurement text, 214
Metafile, 180
Method, 10, 14
Difference compared to rule, 15
MLine, 211
MObject, 77
Modal dialog, 111
Module, 10
moveTo()
Base, 83
MSymbol, 213
MText, 214

Name space

hierarchical, 13
Names
  for entities, 13
  of methods, 15
  predefined, 13
  reserved, 13
NEW_ELEMENT, 104
Notation, 216
notHierSelectable()
  Base, 79
notSelectable()
  Base, 79

OAM, 8
OAS, 8
Object, 11
Object model, 8
object2Article()
  OiPDManager, 158
objInLevel()
  OiLevel, 165
OCD, 8
ODB, 8
  2D Representation and ODB, 86
OEX, 8
OFF file, 200
OFML
  Concepts, 10
  Features, 7
  Overview, 8
OFML database, 8
oiApplPaste(), 110
OiBlock, 118
oiClone(), 110
oiCollision(), 111
oiCopy(), 111
oiCut(), 111
OiCylinder, 119
oiDialog(), 111
oiDump2String(), 112
OiEllipsoid, 120
oiExists(), 113
OiFrame, 121
oiGetDistance(), 113
oiGetNearestObject(), 113
oiGetRoots(), 113
oiGetStringResource(), 113
OiHole, 122
OiHPolygon, 124
OiImport, 125
OiLevel, 164
oiLink(), 114
OiOdbPlElement, 154
oiOutput(), 114
OiPart, 148
oiPaste(), 114
OiPDManager, 157
OiPlanning, 132
OiPlElement, 142
OiPolygon, 126
OiProductDB, 160
OiProgInfo, 140
OiPropertyObj, 153
oiReplace(), 115
OiRotation, 127
oiSetCheckString(), 115
OiSphere, 128
OiSurface, 131
OiSweep, 129
oiTable(), 115
OiUtility, 153
OiWall, 166
OiWallSide, 166
onCreate()
  OiPlElement, 145
onRotate()
  OiPart, 152
onTranslate()
  OiPart, 152
Open-form areas, 131

Pi, 79
PICK, 105
Planning check, 101
Planning element, 142
Planning environment, 135, 164
Planning hierarchy, 132
Planning limit, 133, 135
Planning mode, 165
Polygon, 124, 126
Price, 100
Primitive, 117
Procedure, 168
Product Data, 99

Product Data Management, 156
Product data management, 139
Product data model, 167
Product database, 204
Program Access, 98
Program Information, 140
Program information, 136
Property, 14, 89
  Definition format, 207
Property Information, 94
propsChanged()
  OiOdbPlElement, 155
  Property, 93

Quboid, 118

Reference Types, predefined
  CFunc, 31
  Func, 31
  Hash, 42
  List, 38
  String, 31
  Type, 30
  Vector, 36
Relational database, 115
remove()
  MObject, 78
REMOVE_ELEMENT, 104
removeProperty()
  Property, 91
removeValid()
  Base, 80
  OiPropertyObj, 153
Resolution, 81
Resource, 204
Restoring an instance from a dump representation, 115
Root object, 113
ROTATE, 106
rotate()
  Base, 83
rotated()
  OiPlElement, 148
rotateValid()
  OiPlElement, 148
Rotation, 83
Rotational body, 127
Rule, 10, 15, 85, 103
  Difference compared to method, 15
  explicit call, 85
  predefined, 103
  user-defined, 103

Sales region, 133
SAP/R3, 156
Scaling of geometries, 126
Scene, 12
Selectability, 79
selectable()
  Base, 79
Selection criterion, 168
SENSOR, 108
setAlignment()
  OiGeometry, 118
setArticleSpec()
  Article, 99
  OiOdbPlElement, 154
  OiPart, 151
  OiPlElement, 146
setBorder()
  OiPlanning, 135
setChanged()
  Base, 81
setCMaterial()
  Material, 89
  OiPlanning, 136
  OiProgInfo, 141
setCutable()
  Base, 80
setDataBasePath()
  OiProductDB, 161
setDefaultHeight()
  OiLevel, 165
setDepth()
  OiPart, 149
  OiPlElement, 143
setErrorLog()
  OiPlanning, 135
setExtPropOffset()
  Property, 91
setHeight()
  OiPart, 149
  OiPlElement, 143
setLanguage()
  OiPlanning, 133

setMatCat()
OiGeometry, 118
setMethod()
Complex, 96
setOdbType()
OiOdbPlElement, 154
setOrderID()
Article, 98
setOrigin()
OiPart, 149
OiPlElement, 143
setPasteMode()
Complex, 96
setPDManager()
OiPlanning, 139
setPlanningWall()
OiLevel, 165
setPlProgram()
OiPlElement, 143
setPosition()
Base, 82
setProgram()
OiPlanning, 134
setPrograms()
OiProductDB, 161
setPropPosOnly()
Property, 91
setPropState()
Property, 94
setPropValue()
OiOdbPlElement, 155
Property, 93
setRegion()
OiPlanning, 133
setResolution()
Base, 81
setRtAxis()
Base, 84
setTempArticleSpec()
Complex, 96
setTrAxis()
Base, 83
setUnchanged()
Base, 82
setupProperty()
Property, 89
setupProps()
OiPDManager, 158
setWidth()
OiPart, 149
OiPlElement, 143
setXArticleSpec()
Article, 100
OiPDManager, 159
show()
Base, 81
SPATIAL_MODELING, 106
Sphere, 128
START_DUMP, 107
START_EVAL, 107
startCollCheck()
OiProgInfo, 142
Structure of Order Lists, 98

Table, external, 204
Text output, 114
Text resource, 204
TIMER, 108
Topology
Name space, 13
Scene, 12
topological independence, 11
TRANSLATE, 105
translate()
Base, 83
translated()
OiOdbPlElement, 155
OiPlElement, 147
translateValid()
OiPlElement, 147
Translation, 83
Type, 10
abstract, 10
Uniqueness, 10
Type identity, 77

unMeasure()
Base, 82
UNPICK, 105

varCode2PValues()
OiProductDB, 162
Visibility, 81

Wall, 164