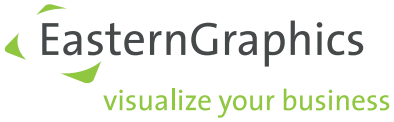

OAP
OFML Aided Planning

Version 1.4

1st revised version

Thomas Gerth, EasternGraphics GmbH (Editor)

November 3, 2022

Contents

References

[article] The OFML Interfaces Article and CompositeArticle (Spezification). EasternGraphics GmbH (`//fsegrde1/archives/docs/specs/OFML/_misc`)

[dsr] Data Structure and Registration (DSR) Spezification. EasternGraphics GmbH (`//fsegrde1/archives/docs/specs/DSR`)

[mt] OFML Metatypes (MT) Spezification. EasternGraphics GmbH (`//fsegrde1/archives/docs/specs/GO`)

[ofml] OFML – Standardized Data Description Format of the Office Furniture Industry. Version 2.0, 3rd revised edition. Industrieverband Büro und Arbeitswelt e. V. (IBA) (`//fsegrde1/archives/docs/specs/OFML/OFML_2.0-3rd_rev`)

[property] The OFML Interface Property (Spezification). EasternGraphics GmbH (`//fsegrde1/archives/docs/specs/OFML/_misc`)

1 Introduction

OFML Aided Planning describes concepts, techniques and corresponding data tables, which should enable a largely uniform data creation and implementation of planning techniques (inter–product rules) both in online and offline applications.

OAP represents an additional layer on top of the conventional OFML data [ofml]. Among other things, you can refer to the properties of article instances. With this, OAP data creation can be linked to metadata creation [mt], for example.

Certain concepts that were previously used to implement planning techniques in offline applications, such as OLAYER–based snapping methods, will be replaced by OAP in the future.

Note:
A feature described in this specification which is not yet supported in the current applications of EasternGraphics is highlighted in gray.

2 Basics

2.1 Technology

OAP integrates the following techniques:

- **Interactors**

  Interactors are two-dimensional graphical symbols that are drawn by the application over an object[1] and which are linked to one ore more → *actions* to be performed when the interactor is selected by the user[2], allowing the user to modify the planning or to get some information.

  Interactors are object-specific. When an object is selected, the interactors are displayed which are defined for the object and which are *valid* in the current planning context.

  The size of the interactor symbols does not change depending on the distance of the camera to the object.

  It can be specified in the OAP data whether an interactor may be hidden by other objects or not. In addition, *visibility areas* can be defined for the interactor symbols so that they are visible only from certain (reasonable) angles.

  There is a distinction between 2D and 3D interactor symbols:

  – Independent of the camera perspective, *2D symbols* always are parallel to the image plane, so they are not subject to any perspective distortion.
  – *3D symbols* have a defined orientation in space, thus, depending on the camera perspective, this results in a perspective distortion.

  3D symbols can and should be used in situations where, when using a 2D symbol, the meaning of the interactor would not be clearly recognizable in every camera perspective, e.g. arrows to illustrate a direction of movement.

  Note:
  (Application) Interactors were already supported in pCon.planner 8 before OAP, with the interactors to be defined directly in the OFML data, see Application Note AN-2013-001. In the context of OAP, however, the interactor concept described in the mentioned Application Note is adapted and significantly expanded[3].

[1] graphical representation of a product, more on the term *object* see next section
[2] The selection is done by clicking on the interactor symbol and is called also the *activation* of that interactor.
[3] Perspectively, the interactors described in the Application Note become obsolete.

- **Actions**

  For certain events, e.g. the activation of an → *application interactor* by the user, actions can be defined. An action specifies the functionality to be performed when the event occurs.

  In conjunction with → *attach areas*, it is also possible to implement inter-product rules, which currently are realized based on metadata[4].

- **Smart attach areas**

  Smart attach areas extend the concept of conventional OFML attach points:
  In addition to points, also lines and areas can be specified, optionally with a raster. Furthermore, the area can be linked with *actions* (see below) to be executed if the attach area was used to connect two objects or if this connection is broken up again.
  With smart attach areas, in the future snapping methods can/will be supported, which currently are implemented in pCon.planner on the basis of OLAYERs `D2SNAP` resp. `ATTACH` & `ORIGIN`.

2.2 Terms

The following terms of fundamental relevance are used in this document:

- **Article vs. article variant**

  An *article* (synonym: *product*) is a commodity that can be produced or is produced by a manufacturer or supplier and/or is offered for sale.

  If some of the properties of an article can be specified by the buyer or the user of the OFML application, this is referred to as a *configurable article*. The specific values of an article with regard to its configurable properties then is referred to as an *article variant* (synonym: *article configuration*).

  Within a manufacturer/supplier, an article uniquely is identified by an alphanumeric code, the *article number*. The values of the configurable properties are coded in the *variant code*, where different schemes can be applied.

  Since most articles are configurable, this document normally uses only the term *article* for the sake of simplicity, even though, in the case of a configurable article, actually the term *article variant* is meant. Only when a distinction is mandatory, the term *article variant* is used explicitly.

- **Article representation vs. OFML instance**

  An *article representation* (synonyms: *planning element*, *object*) is the object that graphically represents an article in a planning system or a configuration system.

  The configuration of a (configurable) → *article* is carried out based on OFML *properties*. For this purpose, an OFML article instance (short *OFML instance*) has to be created. OFML instances also are required to determine various article information (texts, prices, etc.).

  An OFML instance also includes a graphical representation of the article. However, for technical reasons, OFML instances normally are not used as → *article representations* in the planning system. In this case, in order to configure an article or to determine article information, an OFML instance (not visible to the user) has to be created temporarily.

  In the online applications (based on the EAIWS), an article has two representations: a graphical one in the client (planning element) and a commercial one in the basket, which is managed by the server. For this reason, in difference to the planning element, the representation in the basket of the server is called the *basket instance*. Also, the server is responsible for creating a (temporary) OFML instance for a basket instance, if this is necessary to fulfill a request from the client.

- **Active vs. passive planning element**

  When inserting a new element into the planning or when removing a planning element or when moving a planning element, 2 roles can distinguished that planning elements play: The element

[4] Article polymorphism and intra-product rules can and should continue to be implemented using metadata.

that is inserted, deleted or moved plays the active role and accordingly is called the *active planning element*. The other elements play a passive role and accordingly are referred to as *passive planning elements*.

In particular, this distinction is relevant with respect to the attach areas: some of the attach areas of a planning element only can/should be used if the element plays the active role, others only if it plays a passive role. Respectively, the attach areas also are referred to as active and passive, where there may be attach areas that can be used in both roles.

2.3 Other issues

- **Proximity**

  A *proximity relationship* (synonym: connection) between two planning elements exists if there is (at least) one pair of attach areas of the two elements which match one another logically and geometrically.

- **Dependancy on variants**

  Dependencies in the OAP data on an article variant are represented on the basis of OFML properties[5].

  For this purpose, the so-called *Property variant code* (abbreviated *PropVarCode*) is introduced, which encodes the current values of the OFML properties of an article variant (for details see field type *PVC* in section 3.2):

  ```
  <Property>=<Value>;<Property>=<Value>;...
  ```

  In the OAP tables (section 4), fields are provided in which can be specified either a (partially determined) PropVarCode or an expression that operates with OFML properties.

  One consequence is that, in order to evaluate the OAP data for a given article variant, its PropVarCode must be known. The PropVarCode has to be retrieved from the OFML instance of the article[6].

  In order to ensure a good performance, it is the responsibility of the applications to minimize the number of OFML instance creations by using suitable caching resp. persistence techniques.

- **OAP types**

  An *OAP type* comprises the set of all articles resp. article variants which, in the context of OAP, have the same features and should be treated equally. OAP types are defined in table `Type` (see section 4.1.1).

  Corresponding mapping tables are used to assign specific articles resp. article variants or even meta types to a specific OAP type.

[5] Thus, OAP data may be set up on top of, e.g., metatyp-based data or specially programmed OFML data.

[6] This is done by means of new method *getPropVarCode(pState(Int))* of base class *OiPlElement*, where value `0` is to be used for the status parameter, so that the invisible (and in part graphics-relevant) properties also are represented in the code.

3 General rules and definitions

3.1 Regulations regarding the format

CSV tables (comma separated values) are used as the physical exchange format between OFML conform applications. The following regulations apply for this:

1. Each of the tables described below is included in exactly one file. The file name is made of the prefix `oap_`, the specified table name and the suffix `.csv` where the table name is written completely in small letters.
2. UTF-8 is used as the character set[7]. Optionally, the byte order mark can be specified at the beginning of the file.
3. Each line of the file represents a data record[8].
   Blank lines, i.e. lines consisting of zero or several blank characters (U+0020) or tabulators (U+0009), are ignored.
   Lines starting with a number sign ('`#`'=U+0023) are interpreted as a comment and are ignored, too.
4. The representations of the individual fields of a data record are separated from each other by a semicolon ('`;`'=U+003B).
5. The value of a field consists of zero or more Unicode characters with a valid UTF-8 encoding, except for the control characters U+0000..U+001F as well as U+007F..U+009F.
6. The representation of a field is derived from the value of the field replacing each quotation mark ('`"`'=U+0022) by two quotation marks and enclosing the resulting string in quotation marks. If the value of a field does not start with a quotation mark and does not contain a semicolon ('`;`'=U+003B), the value itself (i.e. without any modifications) can be used as the field representation.

3.2 Table descriptions and field types

In the table descriptions, a field of a data record is specified by the following attributes:

- Number
- Name
- Mark, whether the field belongs to the primary key of the table[9]
- Field type (see below)
- Maximum length of the field (number of characters)[10]
- Mark, whether the field has to be filled (obligatory field)

In key fields of a table, there must not be two values that differ only in spelling[11].

[7] The normal form should be NFC (Normalization Form Canonical Composition).
[8] A line is terminated either by an LF character (U+000A) or by a sequence of CR (U+000D) and LF.
[9] For a given primary key there may be only one record in the table.
[10] While in principle there are no restrictions in CSV data records concerning individual field lengths, for certain fields of data type `Char` maximum possible resp. reasonable lengths resulting from the intended purpose are specified here. Moreover, in data creation further restrictions that are imposed by the program used in the data creation process should be observed.
[11] with respect to upper and lower case

The following **field types** are defined:

Text Text

All characters according to regulation 5 above are allowed. except the non-breaking spcae (U+00A0) and the soft hyphen (U+00AD).

Char Character string

All characters according from the ASCII character set are allowed.

PVC Property variant code

In a property variant code, properties and their values are represented in the form

`<property_key>=<property_value>`

with individual property representations being separated by a semicolon (';'=U+003B).

Property keys are specified without a preceding '`@`' character (U+0040).
Values are represented according to the rules for literal OFML constants.

A property variant code does not necessarily have to include all the properties of a given OFML instance, but, on the other hand, may contain non-visible properties.

Lang Language code

The code consists of the two-digit language code according to ISO 639-1 and the two-digit country resp. region code according to ISO 3166-1 (ALPHA-2), separated by a hyphen (U+002D)[12].

The specification of the region code is optional (see section 3.3).

Examples:
`en-US` f'ur amerikanisches English
`en-GB` f'ur britisches English

If a data element can be used for any language, then the corresponding field must be empty.

Symbol Symbol

All alphanumeric characters from the ASCII character set ('`0`'..'`9`'=0x30..0x39, '`A`'..'`Z`'=0x41..0x5A, '`a`'..'`z`'=0x61..0x7A) are allowed as well as the underscore ('`_`'=0x5F), but the first character must not be numeric.

ID Identifier

All alphanumeric characters from the ASCII character set ('`0`'..'`9`'=0x30..0x39, '`A`'..'`Z`'=0x41..0x5A, '`a`'..'`z`'=0x61..0x7A) are allowed as well as the minus sign ('`-`'=U+002D) and the underscore ('`_`'=U+005F).

An identifier everywhere must be used in the same spelling[13].

OID Objekt identifier

An object identifier references a specific object or set of objects.

An object identifier either is a simple identifier that corresponds to field type `ID` or a hierarchical name in which the individual hierarchy levels represent a simple identifier and where the levels are separated by a period ('`.`'=U+002E).

Hierarchical object names can be used if there are articles in the planning that are related to each other in a parent-child relationship. The front name segments identify the higher levels. If the parent-level identifier references a set of objects, the child-level identifier is applied to all the objects in that set (product quantity).

[12] This definition is based on the specification of the IETF for language tags. Lower case of the language code and capitalization of the country code must be observed!.
[13] with respect to upper and lower case

ID_List comma-separated list of identifiers (field type *ID*)

OID_List A comma-separated list of object identifiers (field type *OID*)

OFML A name according to OFML standard (part III) [ofml]

Possible names:

- OFML package
- OFML interface
- fully qualified OFML type (class)

Int Non-negative integer

All numeric characters from the ASCII character set (U+0030..U+0039) are allowed.

Num Numerical value

All numeric characters from the ASCII character set (U+0030..U+0039) are allowed as well as the decimal point ('-'=U+002E), where the decimal point only may occur once. Optionally, a minus sign ('-'=U+002E) can be used at the first position.

Bool Boolean value

'1' – true, '0' – false

NumExpr Numerical expression

It is expected that the result of the evaluation of the expression is a numerical value according to field type *Num*.

More on expressions see below.

BoolExpr Boolean expression

It is expected that the result of the evaluation of the expression is a boolean value:

- The result of the evaluation of the expression is considered (unambiguously) *true* if either it has a numeric type and the value is nonzero, or it is a string and its value is a non-empty string.
- The result of the evaluation of the expression is considered (unambiguously) *false* if either it has a numeric type and the value is equal to zero, or it is a string and its value is an empty string.
- In all other cases the result is *undefined*.

More on expressions see below.

Lexical structure and syntax of OAP *expressions* (field types *NumExpr* and *BoolExpr*) are described in detail in appendix A[14]. To a large extent, OAP expressions correspond to the expressions specified in part III of the OFML standard. In addition, OAP expressions (among others) offer the following special features:

- The names (keys) of the OFML properties of the active planning element (and possibly other objects) can be used as variables.
- Function

  `methodCall(<Action-ID>)`

  can be used in order to call OFML methods. The argument of the function is the ID of an action[15] of type `MethodCall`, which specifies the method call. (For details see A.4.15.)

[14] The data types described there are not identical to the field types described here.
[15] or an expression yielding an ID

If errors occur when evaluating expressions (for example, syntax errors or references to non-existent properties), the following rules apply:

- For field type *NumExpr* value `0.0` is assumed.
- For field type *BoolExpr*, the result is *undefined* (not unambiguously true or false).
  For each field of this type, the specification of the respective tables explicitly determines the behavior in the case of an undefined expression.

3.3 Language-specific data elements

The following provisions apply to the use of language and region codes in fields of type *Lang*[16]:

- For each table entry with a (non-empty) code in the field of type *Lang*, containing both the language code **and** the region code, there should also exist a table entry with a code containing only the language code in question. This entry serves as fallback for applications that do not support the region specific code.
- Which language (region) is used for the fallback entry lies within the discretion of the manufacturer resp. data creator.
- If the contents of both table entries are identical, the entry with the region specific code can (and should) be omitted.

  Example:
  If a text resource is created for both American and British English, and the fallback for `en` is the text in British English[17], the language code `en-US` is specified in the language field of the table entry with the text in American English, while for the table entry with the text in British English the language code is given only as `en`.

When selecting a table entry from the entries that match a given search key (text ID resp. image ID), the application then proceeds as follows:

1. An application in which language and region are set uses the entry with the appropriate code in the language field that contains the language **and** the region code.
2. If there is no matching entry with the region specific code, or if only the language but not the region is set in the application, then the entry will be used whose language field contains only the respective language code.
3. If there also is no matching entry with the simple code in the language field (only consisting of the language code), the entry with empty language field is used (if available)[18].

3.4 Regulations regarding storage

By default, the tables are stored region specific in the relevant OFML series:

`<data>/($manufacturer)/($program)/($region)/($version)/oap`

The tables have to be compiled into an EBase database named `oap.ebase`.

If cross-serial, manufacturer-wide logics have to be realized, the OAP data has be created and stored in a specific series of the manufacturer (e.g. *global*). Then, in the registration files of the relevant product series this series has to be referenced by means of key `oap_program`[19].

[16] currently this concerns tables `Text` and `Image`, field `Language`
[17] assuming that the OFML data is mainly used in Europe
[18] For texts, a language always should be specified, for images, however, the language field usually is empty.
[19] The syntax of this key corresponds to the syntax for the key `catalogs`, i.e. `::($manufacturer)::($program)::`

3.5 Regulations regarding persistency of articles

In order to be able to process the articles in the online applications and for high-performance processing in pCon.planner, value `STATECODES` has to be specified in the registration data for key `persistency_form`.

Accordingly, the state of all objects (articles) has to be completely described by the state codes defined in OFML interface *Article*. This also applies to any partial plannings (planning groups) that may be used[20].

[20] Thus, if a partial planning does not represent a real articles, it has to be created as a so-called *pseudo article*.

4 The Tables

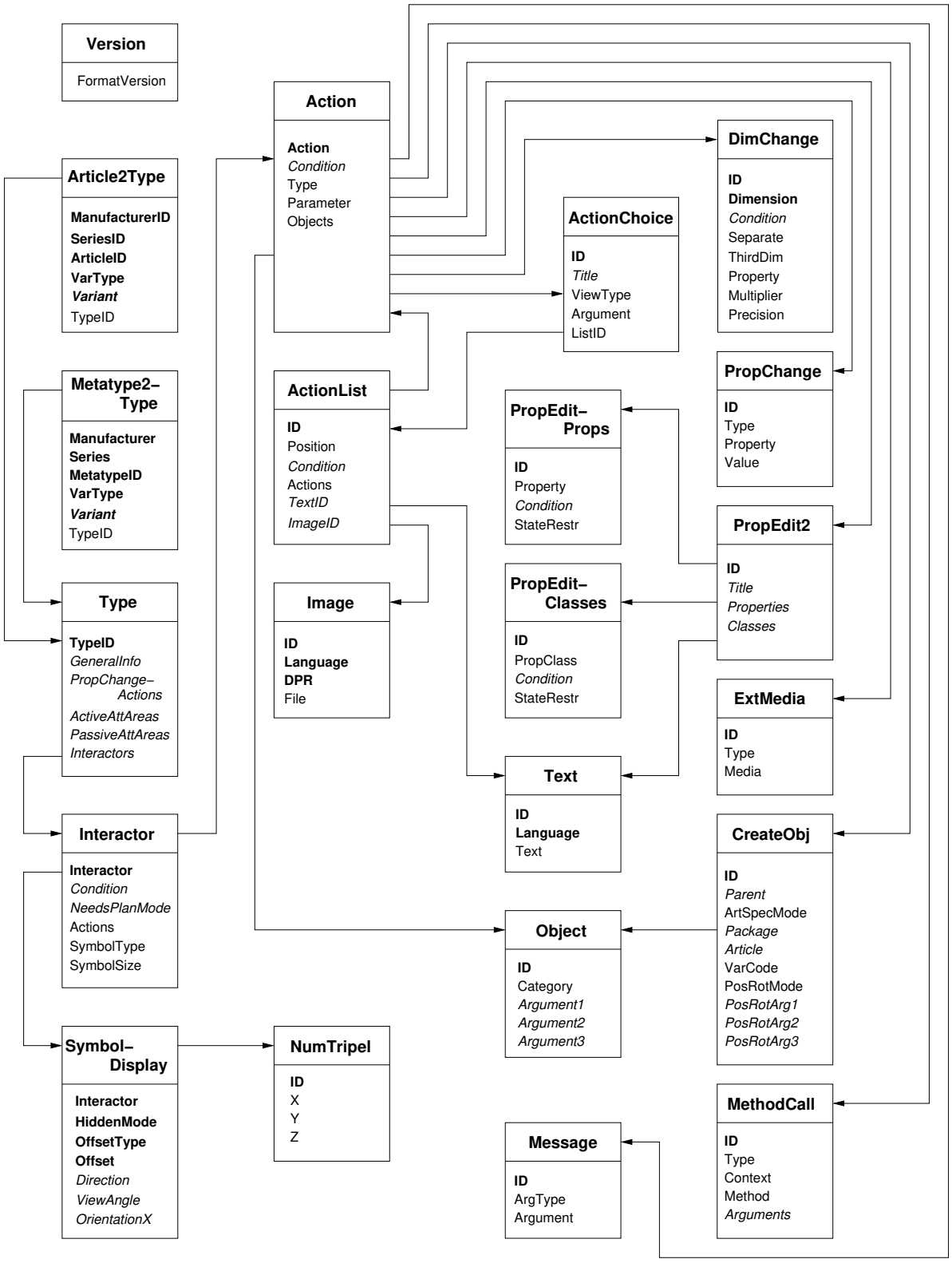

Figure 1: Table Overview

Primary key fields are highlighted in bold. Fields which are not mandatory are indicated in italics.

4.1 OAP types

4.1.1 The Type table

Table name: `Type`
Obligatory table: yes

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	TypeID	X	ID		X	ID of the OAP type
2.	GeneralInfo		ID			General information
3.	PropChangeActions		ID_List			Property change actions
4.	ActiveAttAreas		ID_List			Active attach areas
5.	PassiveAttAreas		ID_List			Passive attach areas
6.	Interactors		ID_List			Interactors

Remarks:

- The ID in field 2 refers to table `GeneralInfo`.
- The identifiers in field 3 refer to table `Action`.

  The *PropChangeActions* are executed after a property of the article has been changed in the property editor of the application. They are *not* executed if a property change took place as part of an action of types *PropValue* and *PropEdit2*.
  The actions are executed in the order of the identifiers!
- The identifiers in fields 4 and 5 refer to table `AttachArea`.

  The active attach areas are used if the planning element representing the article plays the active role. Accordingly, the passive attach areas are used in case of a passive role. (Correspondingly, an attach area that is relevant in both roles has to be referenced in both fields.)
  If no active attach areas are specified, the article can only be placed freely, i.e., the snapping mechanism of the application does not take effect. If no passive attach areas are specified, it is not possible to attach other articles to the article using the snapping mechanism of the application.
- The identifiers in field 6 refer to table `Interactor`.

4.1.2 The Mapping tables

Table name: `Article2Type`
Obligatory table: no

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ManufacturerID	X	Char	16	X	Commercial manufacturer ID
2.	SeriesID	X	Char	16	X	Commercial series ID
3.	ArticleID	X	Char		X	Base article number
4.	VarType	X	Symbol		X	Type of variant specification
5.	Variant	X	Char			Variant specification
6.	TypeID		ID		X	ID of assigned OAP type

Table name: `Metatype2Type`
Obligatory table: no

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	Manufacturer	X	Char		X	OFML manufacturer ID
2.	Series	X	Char		X	OFML series ID
3.	MetatypeID	X	Char		X	Metatype ID
4.	VarType	X	Symbol		X	Type of variant specification
5.	Variant	X	Char			Variant specification
6.	TypeID		ID		X	ID of assigned OAP type

In both mapping tables, field 5 can be used to assign OAP types to specific article variants. The type of variant specification is defined in field 4. Currently the following 3 types are supported:

None The entry is valid for all article variants.

Field 5 is empty in this case (or any existing content is ignored).

Expr The variant is defined by a Boolean expression (which uses certain properties)[21].

PVC The variant is defined by a *PropVarCode*[22].

Usually, it is sufficient to specify a partially determined PropVarCode, which only encodes the properties that are necessary to distinguish the article variants.

The mapping entries for a given article or metatype may use only either *Expr* or *PVC* to specify article variants, and may include only one entry with value *None* in field 4!

The procedures for selecting the appropriate mapping entry are described in the following section.

4.1.3 Determination of the appropriate mapping entry

The first step is to determine the set of all table entries where manufacturer, series and the article number or the metatype ID match.

Table access fails if no matching table entry is found, or if the entries in the resulting set use different types of variant specification (*Expr* and *PVC*), or if the resulting set contains more than one entry with value *None* in field 4.

Variant specification via Boolean expression

From the entries determined in step 1, all entries with a variant specification are removed for which the evaluation of the expression in field 5 does not yield definitely *true*.

Table access fails if the resulting set is empty or if it contains more than one entry with a variant specification.
If the resulting set contains an entry with a variant specification and one without (*None* in field 4), the entry with the article variant is used.

Variant specification via PropVarCodes

The (partially determined) PropVarCodes (field 5) of each of the entries with a variant specification determined in step 1 will be compared with the PropVarCode of the currently treated article instance. For each property, which is contained in both Codes, the corresponding values are compared with each other.

[21] Field 5 then is treated as a field of type *BoolExpr*.
[22] Field 5 then is treated as a field of type *PVC*.

Two property values are considered equal if

- both values are symbol literals and have the same symbolic value
- both values are string literals and both string literals represent the same character string
- both values are `NULL`
- both values are numeric literals (integer or floating-point), have a valid value[23], and represent the same numeric value[24]

(See appendix A.3.2 for details on literals.)

If not all compared property values are equal, the table entry is removed from the set of entries to be considered further[25].

Finally, the set is reduced to the table entries with the most matching property valuest[26].

Table access fails if the resulting set does not contain exactly one entry.

4.2 The NumTripel table

A triple is used to indicate the coordinates of a position, rotation axes, translation vectors and other three-dimensional parameters.
Triples are referenced from various tables of this specification.

Table name: `NumTripel`
Obligatory table: no

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the triple
2.	X		NumExpr		(X)	X value
3.	Y		NumExpr		(X)	Y value
4.	Z		NumExpr		(X)	Z value

4.3 General information

Not yet supported.

4.4 Attach areas

Not yet supported.

4.5 Matching attach areas

Not yet supported.

[23] See appendix A for details on the valid ranges of numeric literals.
A consequence of the regulation is that the results of the comparison is *false* if both values are invalid, even if they are represented by the same character string.

[24] The only limitation of the current implementation when comparing decimal integers and floating-point numbers is that the amount of any exponent may not be greater than 9999 (in which case the value is considered invalid). Apart from that, the exact decimal value is always compared without restriction of precision.

[25] Since no values can be compared for table entries without a variant specification (*None* in field 4), these entries are included in the set of table entries to be considered further.

[26] If the same property occurs several times in a PropVarCode, the property is counted only once.

4.6 Interactors

Table name: `Interactor`
Obligatory table: no

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	Interactor	X	ID		X	ID of the interactor
2.	Condition		BoolExpr			Validity condition
3.	NeedsPlanMode		BoolExpr			<deprecated>
4.	Actions		ID_List		X	Actions
5.	SymbolType		Symbol		X	Type of interactor symbol
6.	SymbolSize		Symbol		X	Size of interactor symbol

Remarks:

- The interactor is *valid* if field 2 is empty or if the evaluation of the expression specified in the field definitely yields *true*.

- The identifiers in field 4 refer to table `Action`.

  The actions are executed in the order of the identifiers when the interactor is activated.
  In doing so, an action is skipped if its execution condition is currently not met, or if it is otherwise *invalid*[27]. (The context for evaluating the execution condition(s) of the actions is updated after the execution of each single action.)

  If an error occurs when accessing the action parameters in the OAP database, or if the execution of an action fails, the processing of the action list is aborted!

  The processing of the action list is also aborted after an action of type *SelectObj* (see section 4.7).

- The symbol of an interactor is a pictogram that illustrates the (main) effect of the interactor. In the interest of a uniform design of the GUI of the applications, there is no provision in the OAP to directly specify an image file for the symbol. Instead, an abstract, predefined symbol type is specified (field 5). The application then uses an image matching the type to represent the symbol.

  The position and, if necessary, the orientation and visibility range of the symbol is defined in the table `SymbolDisplay` described below.

  Certain actions (such as *DimChange*, see section 4.7) are executed in a special application mode in which application-specific interactors (not defined in OAP) are used. The OAP interactors are not visible during these modes.
  Symbol types for interactors, the first action of which activates a special application mode, are marked with *App* in the following list. Considering the validity conditions of the interactors, there should be only a single interactor with one of these special symbol types.

  The following symbol types are defined:

Add	Adding planning elements
Attention	Output of important information (using an action of type *Message*)
ChangeDimHorizontal	Changing horizontal dimension
ChangeDim2Left	Decreasing horizontal dimension
ChangeDim2Right	Increasing horizontal dimension
ChangeDimVertical	Changing vertical dimension
ChangeDimDown	Decreasing vertical dimension
ChangeDimUp	Increasing vertical dimension
Delete	Removing planning elements
Edit	Changing properties/settings (see also *Material*)

[27] e.g., unsupported action type or false or missing action parameters

Flip	Switching the front/back orientation of a planning element (rotation about vertical axis by 90 degrees)
Material	Changing material characteristics
OnOff	Switching a functionality on/off[28]
PosHorizontal	Changing horizontal position
Pos2Left	Moving to the left
Pos2Right	Moving to the right
PosVertical	Changing vertical position
PosDown	Moving downwards
PosUp	Moving upwards
RotateNY90	Rotation about negative Y axis (vertical axis) by 90 degrees (counterclockwise)
RotatePY90	Rotation about positive Y axis (vertical axis) by 90 degrees (clockwise)
StartDimChange^{App}	Start an action of type *DimChange*
Video	Playing a video

- Field 6 specifies the desired abstract size of the interactor symbol. The exact dimensions of the symbols (pictograms) for each abstract size grade are determined by the applications.

  The following size grades are defined:

  - **small**
  - **medium**
  - **large**

  For a given symbol type, it is not necessary to have a pictogram in all sizes. The application then uses the pictogram in the next larger or next smaller version.

Table name: `SymbolDisplay`
Obligatory table: no

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	Interactor	X	ID		X	ID of the interactor
2.	HiddenMode	X	BoolExpr		X	Hidden mode
3.	OffsetType	X	Symbol		X	Type of offset specification
4.	Offset	X	Char		X	Offset of the symbol
5.	Direction		ID			Orientation of the symbol
6.	ViewAngle		NumExpr			Opening angle of the visibility range
7.	OrientationX		ID			Orientation of X axis for 3D symbols

Remarks:

- This table specifies the position, orientation and visibility range of the symbols of an Interactor.
- The table can contain several entries for an interactor ID to represent different positions with separate visibility ranges, e.g. to realize different representations for the front and rear view of an object.
  It is the responsibility of the data creator to ensure that in this case the visibility ranges do not overlap, so that, depending on the camera perspective, not two (or more) symbols will be displayed for the interactor.

[28] As the current state (on/off) cannot be seen from the symbol itself, it should only be used if the user can see from the graphic of the concerned object whether the functionality is switched on or not (e.g. light on/off, screen on/off).

- The mode in field 2 specifies whether the interactor symbol should be hidden by objects lying between the position of the symbol and the camera position (user/observer)[29]:
  If the evaluation of the expression specified in the field definitely yields *true*, the symbol will be hidden, otherwise it will not[30].

- The position of the symbol is determined by an offset relative to the position of the planning element to which the interactor is bound.

  The specification of the offset can be done in two ways. The way used is indicated in field 3:

  Tripel The 3 coordinates (x, y, z) are specified in an entry in table `NumTripel` (see section 4.2). The ID of this entry has to be specified in field 4.

  If no entry with the ID specified in field 3 is found in table `NumTripel`, the symbol is placed at the position of the planning element.

  Expr The offset is specified by an expression stored in field 4, which yields a *Sequence* of 3 *Float* values[31]. The expression may include method calls (function *methodCall()*, see A.4.15).

  If the expression in field 4 does not yield a sequence of 3 Float values, the symbol is placed at the position of the planning element.

- If at least one of the fields 5 and 6 is empty, the symbol is visible regardless of the camera perspective[32].
  In order to avoid overloaded and confusing views, especially if an object has a lot of interactors, it is recommended to restrict the visibility range of the symbols. In most cases, for example, it makes sense that a symbol is only visible if the side of the object to which the symbol is attached also is visible. The *visibility range* of a symbol is determined by a direction vector (field 5) starting from the position of the interactor (field 4) and the opening angle relative to the direction vector (field 6): If the inside of the cone defined by the position of the symbol, direction vector and opening angle, completely or partially, is captured by the camera with the current camera settings, then the symbol is visible[33].

- In order to specify a direction vector, an entry has to be created in table `NumTripel` and the ID of this entry has to be stored in field 5. The coordinates of the vector are relative to the local coordinate system of the object to which the interactor is bound.

  > For a symbol on the front of an object, e.g., the triple `[0.0, 0.0, 1.0]` would indicate that the symbol is oriented in the direction of the positive Z axis (forward-facing).

  If no entry with the ID specified in field 5 is found in table `NumTripel`, the behavior is as in the case that the field is empty, i.e., the symbol always is visible.

- The angle in field 6 has to be givem in degrees in the range of 0 to 360.

  > For a symbol on the front of an object, e.g., with a forward-faced direction vector (field 5, see example above), opening angle `180` would cause the symbol to be visible only if the front of the object is also visible[34].

- If fields 5-7 are not empty and if an entry is found in table `NumTripel` for the ID specified in field 7, the symbol is treated as a three-dimensional object (*3D symbol*). In all other cases, the symbol is treated as a two-dimensional object, which always is displayed parallel to the image plane (*2D symbol*).

[29] This includes the object to which the interactor is bound.
[30] If the field is empty, the behavior is the same as before the field was introduced: If *no* visibility range is specified (fields 5 and 6), the symbol is hidden by objects in front, otherwise not.
[31] for the data types which can be used in expressions, see appendix A.2
[32] However, it is obscured by objects that lie between the position of the symbol and the camera position if the hidden mode in field 2 has value *true*.
[33] if the hidden mode in field 2 has value *false* or if it is not obscured by objects that lie between the position of the symbol and the camera position
[34] Rather, in practice angles of less than `180` degrees make sense.

The vector specified by the entry in table `NumTripel`[35] referenced in field 7 defines the X axis of the local coordinate system of the 3D symbol whose origin is at the symbol's position (field 4). The Z axis of this coordinate system is determined by the direction vector[36] (field 5) and the Y axis results from the cross product of Z and X axis[37].

The image/pictogram associated with the type of the symbol (see field *SymbolType* in table `Interactor` above) then is placed and oriented in such a way that it resides in the X-Y plane of the 3D symbol, the origin (center) of the image matches the position of the symbol, and the X axis of the image coincides with the X axis of the coordinate system of the symbol, see figure 2.

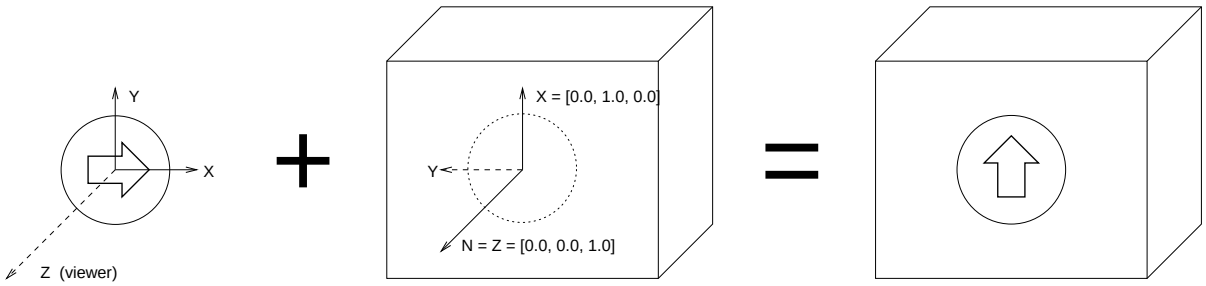

Figure 2: Mapping of pictograms (icons) for 3D symbols

The image of a 2D symbol is placed and oriented in such a way that it resides in a plane parallel to the image plane (projection plane), the origin (center) of the image matches the position of the symbol, and the X axis of the image is pointing horizontally to the right.

- The length of the vectors referenced in fields 5 and 7 must be greater than zero, otherwise an error is triggered and the behaviour of the application is undefined.

4.7 Actions

Table name: `Action`
Obligatory table: no

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	Action	X	ID		X	Identifier of the action
2.	Condition		BoolExpr			Execution condition
3.	Type		Symbol		X	Type of the action
4.	Parameter		ID		(X)	Parameter for the action
5.	Objects		OID_List		(X)	Target objects

[35] The coordinates are relative to the local coordinate system of the object to which the interactor is bound.

[36] In the case of a 3D symbol, the direction vector is also referred to as the *normal*, since it also defines the plane in which the two-dimensional image of the pictogram lies.

[37] If the vector for the X axis is not orthogonal to the normal, the application will normalize the vector. The vector for the X axis must not be parallel to the normal. If this is the case, an error is triggered and the behavior of the application is undefined.

Remarks:

- The action is executed if field 2 is empty or if the evaluation of the expression specified in field 2 definitely yields *true*.
- Supported types (field 3) are:

  ActionChoice
  This is not an action in the true sense, but rather a list of options from which the user can choose one. Selecting an option then causes the execution of one or more actions.

  The ID in field 4 refers to table `ActionChoice`.

  CreateObj
  The action creates an object.

  The ID in field 4 refers to table `CreateObj`.

  DeleteObj
  The action removes the objects specified in field 5.

  (Field 4 has no meaning for this action type.)

  DimChange
  The action allows for an interactive change of one or more dimensions of the active object.

  The ID in field 4 refers to table `DimChange`.

  Message
  The action issues a message to the user.

  The ID in field 4 refers to table `Message`.

  An action of this type is only allowed for interactors with symbol type *Attention*!

  MethodCall
  The action calls an OFML method.

  The ID in field 4 refers to table `MethodCall`.

  If the method is an instance method, it is called on the objects specified in field 5. For class methods, field 5 has no meaning.

  PropChange
  The action assigns a (new) value or status to a property.

  The ID in field 4 refers to table `PropChange`.

  PropEdit <deprecated>

  PropEdit2
  The action executes a dialog for entering or selecting property values.

  The ID in field 4 refers to table `PropEdit` in case of an action of type *PropEdit* resp. to table `PropEdit2` in case of an action of type *PropEdit2*.

  Parameter table `PropEdit2` offers more setting options. Actions of type *PropEdit* (therefore) should no longer be used. (Support for action type *PropEdit* is expected to be discontinued in the future. In case of need see version 1.1 of the OAP specification on details regarding table `PropEdit`.)

  The set of objects resulting from field 5 may contain only one object, otherwise the action has no effect.

  SelectObj
  The action selects the object specified in field 5.

  (Field 4 has no meaning for this action type.)

  The set of objects resulting from field 5 may contain only one object, otherwise the action has no effect.

Attention:
The action results in a change of the active object (see object category *Self*, section 4.9). Therefore, the processing of the action list of an interactor (see 4.6) is aborted after this action! therefore, actions of this type should always be the last action in the action list of an interactor.

ShowMedia
The action displayes an (external) media content to the user.

The ID in field 4 refers to table `ExtMedia`.

- In fields of type *ID_List* specifying a list of action identifiers, there has to be only one action of a type involving an interaction with the user[38].
  Furthermore, the following regulations also apply to these actions:
  - Actions of types *DimChange*, *Message* and *ShowMedia* have to be the single action in the list. (These actions always are executed, i.e. any specified execution condition will be ignored.)
  - Actions of type *PropEdit2* (resp. of deprecated type *PropEdit*) have to be the last action in the list.
  - For actions of types *ActionChoice*, if the interaction is aborted by the user (without a selection), possible subsequent actions are not executed.

- Field 5 specifies the objects for which the action is to be performed.

  For action types *ActionChoice*, *CreateObj*, *DimChange*, *Message* and *ShowMedia* field 5 has no meaning[39]. The following applies to the other action types:

  If an OID is a simple identifier (see field type *OID*), the identifier is used to access table `Object` in order to determine the relevant set of objects.

  For hierarchical names, the total set of the affected objects results from the product of the sets at the individual hierarchy levels.

  The action is executed for each of the specified objects in the specified order of the OIDs. If an OID references more than one object, the order of execution within that set is undefined.

4.8 The Tables for the action parameters

4.8.1 Action Choice

Two tables are required to specify action choices: Table `ActionChoice` describes the general attributes of an action choice (e.g. the appearance). Furthermore, this table refers to the list of selectable options, which is specified in table `ActionList`[40]. In this table, names and/or images are assigned to the actions/options.

Table name: `ActionChoice`
Obligatory table: no

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the parameter set
2.	Title		ID			Text ID for dialog title
3.	ViewType		Symbol		X	Type of view (appearance)
4.	Argument		(Char)		(X)	(conditional) parameter for the view type
5.	ListID		ID		X	ID of the action list

[38] concerns: *ActionChoice*, *DimChange*, *Message*, *PropEdit2* and *ShowMedia*
[39] The possible parent objects for actions of type *CreateObj* are specified in the parameter table `CreateObj`.
[40] The table better should be named `ChoiceList`. However, for a simpler migration the previous name is used, which was introduced when only one action could be linked to an option.

The ID in field 2 refers to table `Text`.
If the field is empty or no text can be determined for the ID, the choice dialog is displayed without a title.

Currently, 2 view types (field 3) are supported:

List The selection options are shown in a list below each other, with each option represented by an (optional) small image on the left and a text (name) on the right.

The size of the images should be geared to the size for the material images in property editors (see [dsr]), i.e. 50 x 18 image points (width x height). But that is not mandatory. Ideally, however, the images should be the same size for a given action choice.

If there is a image for at least one option, then the area reserved for the image remains empty for options without an assigned image, i.e. the texts of all options are left-aligned.

In this representation form, there should always be a text for an option, since the images alone usually are not meaningful due to the small size.

Field 3 is not relevant for this view type.

Tile The selection options are displayed in tiles of the same size next to and/or below each other. This view type is intended for displaying options with larger images.

A tile consists of the image and an (optional) text below or right beside it.

The desired tile size is specified in field 4. (This refers only to the size of the image. With text, the actual displayed tile correspondingly is larger.)

Currently, the following tile sizes are supported (image points, width x height):

small 50 x 50

medium 100 x 100

large 200 x 200

The image files for normal resolution displays are expected to be in exactly the size (in pixels) determined by the tile size specified in field 4.

The above mentioned image dimensions are information in (logical) image points. For a good representation on high-resolution displays, image files with a correspondingly larger resolution (pixels) are to be provided. For details see table `Image` (section 4.11).

An option without image *and* text is not displayed.

The ID in field 5 refers to the following table.

Table name: `ActionList`
Obligatory table: conditional (yes, if table `ActionChoice` exists)

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the action list
2.	Position	X	Int		X	Position of the action in the list
3.	Condition		BoolExpr			Validity condition
4.	Actions		ID_List		X	Actions
5.	TextID		ID			ID of the text
6.	ImageID		ID			ID of the image

Remarks:

- The list entry is *valid* if field 3 is empty or if the evaluation of the expression specified in the field definitely yields *true*.
- The identifiers in field 3 refer to table `Action`.

The referenced actions themselves may be of type *ActionChoice*, i.e. a nesting of actions of type *ActionChoice* is supported[41].
Besides that, no actions should be used which themselves include a user dialog.

To the execution of the actions apply the same regulations as to the actions of an interactor, specifically:

The actions are executed in the order of the identifiers. In doing so, an action is skipped if its execution condition is currently not met, or if it is otherwise *invalid*.

If an error occurs when accessing the parameters of the referenced actions in the OAP database, or if the execution of a referenced action fails, the processing of the action list is aborted and the enclosing *ActionChoice* action fails!

- The ID in field 5 refers to table `Text` (section 4.10). In this table, language-specific texts (names) for the action/option can be specified.
- The ID in field 6 refers to table `Image` ((section 4.11). In this table, an image (icon) illustrating the action/option can be specified[42].

  If required, also language-specific image files can be referenced in table `Image`.

4.8.2 `PropChange`

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the parameter set
2.	Type		Symbol		X	Type of change
3.	Property		Symbol		X	Property key
4.	Value		Char		X	Expression for value of change

Remarks:

- Following change types are defined:

  Value A (new) value is assigned to the property.

  The expression[43] in field 4 must yield a value that matches the data type of the property.

  Visibility The visibility of the property is changed.

  The expression in field 4 must yield a boolean value (see field type *Bool*).
  The value determines whether the property should be visible or not.

  Editability The editability of the property is changed.

  The expression in field 4 must yield a boolean value (see field type *Bool*).
  The value determines whether the property should be editable or not.

- Field 3 specifies the key of the property to be changed (without the preceding `@` character).

  The change is made for all objects affected by the action that possess this property.

[41] The concrete implementation in the user interface can vary from application to application.
[42] where several image files with corresponding resolutions for displays with different resolution have to be referenced for one image (for details see table `Image`)
[43] on expression see appendix A

4.8.3 PropEdit2

Table name: `PropEdit2`
Obligatory table: no

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the parameter set
2.	Title		ID			Text ID for editor title
3.	Properties		ID		(X)	ID for `PropEditProps`
4.	Classes		ID		(X)	ID for `PropEditClasses`

Remarks:

- The ID in field 2 refers to table `Text`.
  If the field is empty or no text can be determined for the ID, the editor dialog is displayed without a title[44].
- The properties to be edited are specified in table `PropEditProps` (see below). The ID for the access to this table is given in field 3.
  If all properties of one or more property classes are to be edited, these classes are specified in table `PropEditClasses` (see below). The ID for the access to this table is given in field 4.
  Only one of the fields 3 or 4 may be empty.
- Properties from classes that are referenced via table `PropEditClasses` should not be specified in table `PropEditProps`.

Table name: `PropEditProps`
Obligatory table: conditional (yes, if table `PropEdit2` exists and contains references in field *Properties*)

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the parameter set
2.	Property	X	Symbol		X	Key of the property
3.	Condition		BoolExpr			Validity condition
4.	StateRestr		Symbol		X	Restriction regarding property state

Property keys (field 2) are to be specified without the preceding `@` character.

For fields 3 and 4 see below.

Table name: `PropEditClasses`
Obligatory table: conditional (yes, if table `PropEdit2` exists and contains references in field *Classes*)

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the parameter set
2.	PropClass	X	Char		X	Name of the class
3.	Condition		BoolExpr			Validity condition
4.	StateRestr		Symbol		X	Restriction regarding property state

[44] In this case, the application should not use the name of the property as a fallback (if only one property is involved).

A property resp. class is used in the dialog only if field 3 (*Condition*) is empty or if the evaluation of the expression specified in the field definitely yields *true*.

Note:
Since the conditions are *not* (re)evaluated during the dialog after a property change, they must not refer to properties whose values can change themselves during the dialog!
In order to handle the dependency of the visibility of a property on other properties in general and in particular on properties that are used in the dialog itself, an according value (not equal to `None`) should be specified in the following field 4.

Field 4 (*StateRestr*) indicates whether the property resp. the properties of the class should be displayed in the dialog depending on the current property status. (The field is evaluated only for properties resp. classes that are valid according to field 3.)

Currently, the following values are defined for field *StateRestr*:

None	No restriction, i.e., the property resp. the properties of the class should be displayed regardless of the current status.
Visible	The property resp. the properties of the class should be displayed only if they are currently visible[45].
VisibleEditable	The property resp. the properties of the class should be displayed only if they are currently visible **and** editable.

If, due to the validity conditions, only one property is suitable for use in the dialog, field *StateRestr* has no meaning for this property, i.e., the property is displayed and can be changed by the user[46] regardless of its current status.

If, taking into account the conditions (field 3) and the property status (field 4), only one property is to be displayed, only the input field or the selection (choice) list of this property is displayed in the editor dialog. (The property name is not displayed.) The dialog ends as soon as the user has selected a value or confirmed the entry.
If several properties are edited, the application offers a suitable GUI technology allowing the user explicitly to close the dialog. However, the graphical representation of the relevant object is updated with every property change, i.e. not only at the end of the dialog.

For an optimal appearance of the dialog, the material images for the property values should also be given in the large variant (see [dsr], section 6.4.2).

4.8.4 `DimChange`

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the parameter set
2.	Dimension	X	Symbol		X	Affected dimension
3.	Condition		BoolExpr			Validity condition
4.	Separate		BoolExpr		X	To be changed only separately?
5.	ThirdDim		BoolExpr		X	Use as the third dimension?
6.	Property		Symbol		X	Property to use
7.	Multiplier		NumExpr		X	Factor for conversion in m
8.	Precision		NumExpr		X	Precision in m

Remarks:

- As already stated in section 4.7, actions of type *DimChange* always have to be the first action in the list of actions of an interactor. The symbol type of the interactor has to be *StartDimChange*[47].

[45] This restriction is useful or necessary if the specified properties have interdependencies.
[46] even if it is actually not editable
[47] Other types are ignored and will be replaced by *StartDimChange*.

- Currently, actions of type *DimChange* are supported only for objects on top planning level.
- In field 2, the axis of the concerned dimension is specified.

  The possible values are `X`, `Y`, `Z`, `PX`, `PY`, `PZ` as well as `NX`, `NY` and `NZ`:
  - With `X`, `Y` and `Z`, the change may be made on both sides of the object.
  - With `PX`, `PY` as well as `PZ` the change may only be made on the side in positive direction of the axis and with `NX`, `NY` as well as `NZ` only on the side in negative direction of the axis.
  - A change on the side in negative direction of the axis causes a corresponding repositioning of the object.
  - Only one of the 3 variants (possible change directions) may be used per dimension, or the condition in field 3 (see below) has to be used to ensure that only one of the 3 variants is valid at the time of the evaluation. (Otherwise, the concerned dimension does not apply.)
- The dimension specified in field 2 is applied if field 3 is empty or if the evaluation of the expression specified in field 3 definitely yields *true*.

  If, after evaluating the conditions of all entries for the action (field 1), more than one dimension is affected, it is up to the application whether and how it realizes the interaction. (This may be dependent on the current view of the user.) However, the behavior of the application in this situation also can be influenced to a certain extent by the content in fields 4 and 5.
- If 2 (or 3) dimensions should be changed simultaneously, the value sets and value ranges of the concerned properties must not be dependent on one another! (Otherwise, there can occur an unexpected or confusing feedback experience for the user.)
  If this requirement is not met, the evaluation of the expression given in field 4 for the concerned dimensions has to yield *true*. Then, once a dimension has been changed[48], the applications recall the value sets and value ranges of the concerned properties in order to react to possible changes due to the dependencies.
- Not all applications resp. application modes allow the simultaneous change of all three dimensions. If a simultaneous change is possible on the part of the data (see field *Separate*), field 5 (*ThirdDim*) should indicate which of the three dimensions should be changed separately if necessary. For that, the evaluation of the expression given in the field has to yield *true*.
  If not exactly one dimension is declared as the *third dimension*, the behavior of the application is undefined.
- Field 6 specifies the property to be used to implement the change of the affected dimension.

  The property key is to be specified without the preceding `@` character.

  The property has to meet the following conditions[49]:
  - The data type of the property has to be numeric (`N`) or a symbolic choice list (`Y`), see [property].
    In the case of a symbolic choice list, the class of the OFML instance must implement the following method:
    symbolicPropValue2Float(pPKey(Symbol), pPValue(Symbol)) → Float

    The method returns the numeric value in *Meter*, which corresponds to the given (symbolic) value of the specified property of the implicit instance.

    The statements below refer to the native values in the case of a numeric property resp. to the numeric values determined via *symbolicPropValue2Float()* in the case of a symbolic choice list.
  - The property values have to be positive.
  - The property either has to define a (single) closed value range (in the case of a numeric property) or a choice list of single values (value set).
    If both are defined, only those values from the choice list are used in the dialog that are within the value range.

[48] and if the action has not yet been completed
[49] If one of them is not fulfilled, the action fails.

- In field 7, the factor is specified which has to be used to convert property values to *Meter*.
  In the case of a symbolic choice list, factor `1.0` is assumed[50].
- Field 8 specifies the precision of the property values (in *Meter*).

Thus, before assigning a (new) value determined by the dialog for the affected dimension to the property specified in field 6, the application performs the following calculations (in this order):

1. Rounding of the value to the precision specified in field 8.
2. Division of the (rounded) value determined in the first step by the factor given in field 7.
3. Rounding of the value determined in the second step to the precision of the property values according to the property definition[51].
   (This step is required if the factor in field 7 is not a power of ten, e.g. in the case of property values in inches.)

In the case of a symbolic choice list, the symbolic value corresponding to the numeric value determined in this way is assigned to the property.

4.8.5 `CreateObj`

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the parameter set
2.	Parent		OID		X	Superior article
3.	ArtSpecMode		Symbol		X	Type of indication of the article to be created
4.	Package		OFML		(X)	OFML package
5.	ArticleID		Char		(X)	(Base) article number
6.	VarCode		Char			OFML variant code
7.	PosRotMode		Symbol		X	Type of indication of position/rotation
8.	PosRotArg1		(Char)		(X)	Argument 1 for position/rotation
9.	PosRotArg2		(Char)		(X)	Argument 2 for position/rotation
10.	PosRotArg3		(Char)		(X)	Argument 3 for position/rotation

Remarks:

- The object ID in field 2 may reference only a single object, otherwise no article creation takes place.
- The mode in field 3 specifies how the article to be created is indicated. The following modes are supported:

  Explicit The details are given explicitly in fields 4-6. If a specific variant is to be created, the (possibly partially determined) OFML variant code hast to be specified in field 6.

  Self An article is created with the same article number and configuration (OFML variant code) as the object to which the interactor is bound for which the action was triggered[52].

[50] since method *symbolicPropValue2Float()* returns the numeric values in *Meter*
[51] In the case of a symbolic choice list, 3 decimal places are assumed
[52] I.e., the information stored in the fields 4-6 with mode *Explicit* here are queried by the application from the active object.

- The mode in field 7 specifies how position and rotation for the new object are indicated. The following modes are supported:

  DataDefined
  Position and rotation are determined by the OFML data.

  The application determines position and rotation for the new article by calling method *checkAdd()* (OFML interface *Complex*) on the OFML instance of the superior article.

  In field 8, a reference object to be passed to method *checkAdd()* can be specified by means of an object ID. (The reference object must be an immediate sub-article of the superior article specified in field 2.)

  In addition, in field 9 the key of an OFML attachment point (without the preceding `@` character) can be specified, which preferentially should be used for placement[53].

4.8.6 `MethodCall`

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the parameter set
2.	Type		Symbol	8	X	Type of call
3.	Context		OFML		X	Call context
4.	Method		Char		X	Name of method
5.	Arguments		Char			Arguments

Two types of method calls are supported. The used type must be specified in field 2:

Instance The method is called on an OFML instance (object).

For this purpose, field 3 must specify the fully qualified identifier of the OFML type or of the OFML interface which defines the method. The name of the method has to be specified in field 4.

The method is called on each target object of the action (see field *Objects* in table `Action`, section 4.7) whose class is derived from the type specified in field 3 or implements the interface specified there[54].

Class A (static) class method is called.

For this purpose, in field 3 the fully qualified identifier of the class (OFML type) must be specified and in field 4 the name of the method.

In field 5, optional arguments for the method call can be specified. Several arguments must be separated by a comma. Each argument is specified as an OAP expression (see appendix A). Amongst others, it is possible to use *placeholders* in these expressions (see A.4.17).

[53] This refers to method *setActiveAttPt()* of OFML interface *AttachPts*

[54] If the articles are not already represented in the application by an OFML instance, the application must create a corresponding temporary OFML instance for this purpose.

4.8.7 `Message`

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the parameter set
2.	ArgType		Symbol		X	Type of argument
3.	Argument		ID		X	Argument

Remarks:

- The text for the message either can be stored in the text table or provided via a method call. (The latter is useful if the text has to be composed of boilerplates depending on the situation.)

  Field 2 indicates which of the two variants is used:

  Text The ID in field 3 refers to table `Text` (section 4.10).

  Method The ID in field 3 refers to table `Action` (section 4.7).

  The referenced action must be of type *MethodCall* and the result of the method call must be of OFML type *String*.
  If this is not the case, the ID itself is output as the message.

  The method is expected to return the text in the language that is currently to be used for product data texts of the OFML series of the active object and/or the target object (see function *getPDLanguage()* of OFML interface *Article* in [article]).

  The String returned by the method must either be read from a string resource file or be encoded in US-ASCII.

- The following options are available for formatting:
  - The character string `\n` causes a line break.
- The dialog is terminated when the user "clicks" next to the dialog window or on the OK button.

4.8.8 `ExtMedia`

No.	Name	Key	Typ	Length	Oblig.	Explanation
1.	ID	X	ID		X	ID of the parameter set
2.	Type		Symbol		X	Media type
3.	Media		String		X	Media ID

Remarks:

- In general, external media content is *not* accessed directly by specifying an URL. Instead, identifiers referencing the content are used (field 3).
- The following media types (field 2) are defined:

  PIM content provided by the PIM[55]

  In field 3, the ID has to be specified, by means of which the media content can be accessed via the PIM interface.
  (This type currently is not supported yet.)

  YouTube a video hosted by YouTube

  In field 3, the YouTube video ID has to be specified.
  This ID can be taken from the URL of the video. (For example, in case of URL `https://www.youtube.com/watch?v=k-W5A-mvphg` the video ID is `k-W5A-mvphg`.)

- It is up to the applications whether the media content is displayed in a separate window (dialog) of the application itself or via an external app (e.g. browser).

[55] Product Information Management

4.9 Object definitions

An entry in this table references a specific object or set of objects. Referencing is done primarily by specifying an object category that describes a defined set of objects. For some categories describing a set of more than one object, the set can be restricted by category-specific arguments.

Table name: `Object`
Obligatory table: no

No.	Name	Key	Typ	Length	Oblig.	Explanation
1.	ID	X	ID		X	Object ID
2.	Category		Symbol		X	Object category
3.	Argumen1		Char			Argument 1
4.	Argumen2		Char			Argument 2
5.	Argumen3		Char			Argument 3

Remarks:

- The ID in field 1 is (possibly the only) part of a hierarchical object identifier (field type *OID*). Object identifiers (OIDs) specify the target objects of actions (see field *Objects* in table `Action`, section 4.7) or arguments of specific action types.
- Following object categories (field 2) are defined:

  Self Refers to the active object in case of an *AttachAction* or a *DetachAction* of a matching attach area pair (see table `AttAreaMatch`,section 4.5), or the object to which the interactor is bound for which the action was triggered.

  The argument fields 3-5 have no meaning.

  ParentArticle
  Refers to the parent article of *Self*.
  The resulting set of objects is empty if *Self* is not a subarticle.

  The argument fields 3-5 have no meaning.

  TopArticle Refers to the superior article of *Self* at the top level of the hierarchy.
  The resulting set of objects is empty if *Self* is not a subarticle.

  The argument fields 3-5 have no meaning.

  MethodCall The objects are determined by means of a method call.

  This object category is not allowed for the target object of an action whose ID is used as the argument of a call to function *methodCall()* (see A.4.15) !

  In field 3, the ID of an action of type *MethodCall* has to be specified.
  In the case of an instance method, only one object definition with one of the categories *Self*, *ParentArticle* or *TopArticle* is permitted for determining the target object of this action.

  The method is expected to yield either a reference to an OFML instance or a (non-empty) sequence (Vector, List) of references to OFML instances, where each OFML instance has to represent an article[56].
  The method must not have any side effects, which would require an update of the information about OFML instances, stored in different parts of the application[57].

  The argument fields 4 and 5 have no meaning.

[56] for the terms see also section 2.2
[57] see also remarks regarding function *methodCall()* in A.4.15

The execution of the action in which the object definition is used fails in the following cases:

- The action specified in field 3 does not have type *MethodCall*.
- The action specified in field 3 specifies an instance method, but there is more than one target object specified or the object definition for the target object does not use one of the object categories *Self*, *ParentArticle* or *TopArticle*.
- The method returns no OFML instance or more than one OFML instance where exactly one object is expected.

4.10 Texts

In this table, language-specific texts are stored, which are required for actions of types *ActionChoice*, *Message* and *PropEdit2*.

Table name: `Text`
Obligatory table: no

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	Text ID
2.	Language	X	Lang	5		Language key
3.	Text		Text		X	Text content

For using and evaluating the language key (field 2) see 3.3.

If the table does not exist, the behavior of the application is undefined. Usually the affected action will fail.
If the table exists, but no text can be determined for an ID, the ID itself is used as text.

Unless otherwise specified for the above actions, the escape sequences for special characters known from OFML are *not* permitted in the text!

4.11 Images

Table name: `Image`
Obligatory table: no

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	ID	X	ID		X	Image ID
2.	Language	X	Lang	5	X	Language key
3.	DPR	X	Int		X	Device-Pixel-Ratio
4.	File		Char		X	Filename/path

Remarks:

- For using and evaluating the language key (field 2) see 3.3.

  If no image can be determined for an ID, the behavior of the application is undefined. Usually the affected action will fail.

- The *Device-Pixel-Ratio* (field 3) is a term from the web design: it specifies how many physical pixels of the output device – per dimension – are used to represent a (logical) image point (of the web page). With a DPR of 2, e.g., 4 pixels are used to represent one image point. Normal resolution displays have a DPR of 1, high-resolution displays of smartphones and tablets have a DPR of 2 (and more).

  In order to display an image with 100x100 image points (e. g. a tile of size *medium* with actions of type *ActionChoice* and view type *Tile*) a high-resolution display with DPR 2 expects an image file

of the size 200x200 pixels[58]. If instead an image file with only 100x100 pixels would be provided, the image would be scaled up to 200x200 pixels[59], which would lead to quality losses.

Therefore, in OAP it is required that in addition to the image file for normal resolution (DPR 1) at least another image file for DPR 2 is provided[60]. (If this file is not provided, the behavior of the application is undefined.)

- By default, the image files referenced in field 4 are stored in the same directory as the OAP data. If necessary, however, they can also be stored in a subdirectory of it. In this case, the filename must be preceded by a relative path, using the slash ('/'=U+002F) as the separator between the individual path components.

- The image format of the file specified in field 5 must be JPEG, PNG or SVG.

  Images in JPEG format have to comply with the specification of the *JPEG File Interchange Format (JFIF)*[61] and

  - have to be sequentially structured (not interlaced/progressive)
  - have to use Huffman coding (not arithmetic coding)
  - have to use the YCbCr color model (no black/white)
  - have to use 8 bit color channel (no more).

  Images in PNG format have to comply with the *PNG (Portable Network Graphics) Specification, Version 2.2*[62] and

  - have to be sequentially structured (not interlaced/progressive)
  - have to use the RGB color model
  - have to use 8 bit color channel (no more, no black/white)
  - optionally an 8 bit alpha channel can be used for transparent images.

  Images in SBG format have to comply with the specification *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*[63] where animations and interactions are not allowed.

- The image files for a given image ID may use several image formats for different DPR's (field 3). For example, it is conceivable resp. wise to use a raster graphic (JPEG, PNG) for the normal resolution representation, but SVG for higher resolutions.

4.12 Version information

Table name: `Version`
Obligatory table: yes

No.	Name	Key	Type	Length	Oblig.	Explanation
1.	FormatVersion		Char		X	Number of the used format version

Remarks:

- The table serves to give information concerning the used format.
  The table may contain only one entry.

- The format version (field 1) has to be indicated in the form `MajorNumber.MinorNumber` according to the OAP specification.

[58] which, correspondingly, can contain more image details
[59] If the image file is larger than 200x200 pixels, the image will be scaled down accordingly.
[60] since this corresponds to the DPR of currently popular smartphones and tablets
[61] http://www.jpeg.org/public/jfif.pdf
[62] http://libpng.org/pub/png/spec/1.2/png-1.2-pdg.html
[63] https://www.w3.org/TR/SVG11/

A OAP expressions

A.1 General definitions

For boolean and numeric expressions the same syntax and semantics are used.

The values of all data types supported by OAP (see A.2) can be formatted according to the OFML Script Object Notation (OSON). This is utilized, amongst others, when passing arguments to OFML methods called in the course of actions of type *MethodCall*.

The other way round, most, if not all, value representations formatted according to OSON can be parsed and stored using one of the data types supported by OAP. This is used, among other things, for values returned from OFML methods called via function *methodCall()* (see A.4.15).

Each OAP expression is evaluated in a defined context. When evaluating the validity condition of an interactor, e.g., the active planning element and the ID of the interactor are known (and relevant). For this purpose, the application core module that executes the evaluation of OAP expressions is provided with a so-called *evaluation context*. This context contains an *object table*, in which the reference to the relevant object[64] is stored for defined scopes, and a *value table*, which stores the values for defined keys (for example, identifiers of placeholders, see A.4.17).

A.2 Supported data types

Each expression or subexpression returns a value that has one of the types described in the following subsections.

For each type an information is given on OSON formatting resp. how to parse an OSON input[65].

A.2.1 Error

Values of type *Error* are returned if an error occurred during the evaluation of a (sub) expression[66]. The value consists of an error code and an error message.

OSON formatting: `NULL`
OSON parsing: not applicable

A.2.2 Null

Type *Null* has the only value `NULL`.

It is usually used when there is no value available in a given context, as, e.g., in the case of a non-existent property during property value access (see A.4.16).

OSON formatting and parsing: `NULL`

A.2.3 Int

A value of type *Int* is an integer in the range from -2^{31} to $2^{31} - 1$ (inclusive).

Arithmetic operations with an integer result, that can not be represented as an integer in the specified range, return an error.

OSON formatting: decimal integer literal
OSON parsing: decimal, octal oder hexadecimal integer literal

[64] internal term: article instance identifier (AIID)
[65] The literals mentioned therein refer to the literals specified in part III of the OFML standard, not to the OAP literals specified in A.3.2.
[66] Errors are returned as specifically typed values in order to be able, with instantaneous evaluation during the syntactic analysis, to easier ignore errors occuring in non-relevant operands of operators with conditional evaluation.

A.2.4 Float

A value of type *Float* is a 64-bit floating-point number according to IEEE 754.

`Infinity` and `NaN` are not supported. Operations with an corresponding result return an error.

OSON formatting und -Parsen: floating-point literal

A.2.5 Symbol

A value of type *Symbol* is character string from a restricted range of values.

OSON formatting: If possible as symbol literal, otherwise as symbol constructor (`Symbol()`) with a string literal as argument
OSON parsing: symbol literal or symbol constructor

A.2.6 String

A value of type *String* consists of a possibly empty string of characters.

OSON formatting and parsing: string literal

A.2.7 Sequence

A value of type *Sequence* contains an ordered, possibly empty sequence of values.

OSON formatting: Vector
OSON parsing: Vector or List

A.2.8 Name

A value of type *Name* represents a simple name such as `t`, an hierarchical name such as `c.e1` or a (partially resp. fully) qualified name such as `::ofml::oi::OiPlElement`.

OSON formatting and parsing: simple name, hierarchical name, qualified name

A.2.9 Numeric types

The two types *Int* and *Float* are called numeric types.

A.2.10 Boolean types

Numeric types as well as type *String* are called boolean types.

If an operator expects an operand of Boolean type, the value of the operand is considered to be *true* if it is either a numeric type and the value is not equal to zero, or if it is of type *String* and the value is a non-empty string. All other values of boolean types will be considered to be *false*.

A.3 Lexical structure

An expression consists of a nonempty sequence of *token*[67].

Spaces are allowed before the first, after the last and between tokens.

Tokens are divided into *operators* (A.3.1), *literals*[68] and *identifier* (A.4.19).

During lexical analysis, starting from the current position, the next token is determined to be the longest possible string that yields a valid token. The lexical analysis fails if no such token can be determined.

[67] smallest units with a distinctive meaning
[68] Literals are defined strings for the direct representation of the values of base types.

A.3.1 Operators

The following operators are defined:

```
?       :       ::      ||      |       &&      &       ^       ==      !=
!       <       <=      <<      >       >=      >>      >>>     +       -
*       /       %       ~       (       )       $       ,
```

A.3.2 Literals

NULL literal

The NULL literal consists of the four consecutive letters `NULL`. It represents the only value of type *Null*.

Symbol literals

There are two forms of symbol literals[69]:

- The normal form starts with character `@` (U+0040) immediately followed by a letter or underscore ('_'=U+005F), followed by zero or more letters, digits and/or underscore, with only ASCII characters allowed[70].
- The second form consists of character `@` immediately followed by a string literal (see below). It can and should solely be used to represent symbols that can not be represented using the normal form.

String literals

A string literal consists of a possibly empty string of characters enclosed in double quotes ('"'=U+0022). The double quote itself is not allowed in the string. To represent this and certain special characters, the following escape sequences, starting with a backslash ('\'=U+005C), have to be used:

`\a`	U+0007	bell character (BEL)
`\b`	U+0008	backspace (BS)
`\t`	U+0009	horizontal tab (HT)
`\n`	U+000A	newline (NL)
`\v`	U+000B	vertical tab (VT)
`\f`	U+000B	form feed (FF)
`\r`	U+000D	carriage return (CR)
`\"`	U+0022	double quote
`\'`	U+0027	single quote
`\\`	U+005C	backslash
`\`[0-7]	octal escape sequence	
`\x`[0-9A-Fa-f]	hexadecimal escape sequence	

An octal escape sequence consists of the backslash followed by up to three octal digits ('`0`'..'`7`'=U+0030..U+0037), which is converted to an integer value.

A hexadecimal escape sequence consists of the character `\x`, followed by one or more hexadecimal digits ('`0`'..'`9`'=U+0030..U+0039, '`A`'..'`F`'=U+0041..U+0046, '`a`'..'`f`'=U+0061..U+0066), with the longest possible digit sequence being consumed and converted to an integer value.

The value determined by octal and hexadecimal escape sequences modulo 256 is the code of the represented character.

If the backslash is not followed by any of the characters listed above, or if `\x` is not followed by a hexadecimal digit, the backslash is ignored and the character following it is taken unchanged.

[69] The character `@` is not part of the value of the symbol.
[70] see also field type *Symbol* in section 3.2

Integer literals

Integer literals are divided into decimal, octal and hexadecimal ones:

- *Decimale integer literals* consist of a sequence of decimal digits ('0'..'9'=U+0030..U+0039), where the first digit can not be null unless it is the only digit.
- *Octal integer literals* consist of a leading null ('0'=U+0030), followed by one or more octal digits ('0'..'7'=U+0030..U+0037).
- *Hexadecimal integer literals* consist of a leading null ('0'=U+0030), followed by 'x' (U+0078) or 'X' (U+0058), followed by one or more hexadecimal digits ('0'..'9'=U+0030..U+0039, 'A'..'F'=U+0041..U+0046, 'a'..'f'=U+0061..U+0066).

The unsigned number U represented by an integer literal must not be greater than $2^{32} - 1$. The signed value of the literal I is equal to U if $0 \leq U < -2^{31}$, otherwise $I = U - 2^{32}$ is valid.

Floiting-point literals

Floating-point literals consist of the mantissa and an optional exponent.

The mantissa consists of a non-empty sequence of decimal digits ('0'..'9'=U+0030..U+0039) and a maximum of one decimal point ('.'=U+002E) at any position[71].

The exponent consists of the letter 'e' (U+0065) or 'E' (U+0045), followed by an optional sign '+' (U+002B) or '-' (U+002D), followed by a nonempty sequence of decimal digits (see above).

Mantissa and exponent are generally interpreted as decimal numbers.

Decimal point or exponent can be missing, but not both.

A floating-point literal whose absolute value after convertion to the internal binary representation and rounding to 53 (binary) digits is greater than or equal to 2^{1024}, leads to an error.

A.4 Syntax of expressions

A.4.1 Precedence and associativity of operators

The *precedence* (priority, rank) of operators controls the order in which the corresponding operations in expressions are executed, unless a different order has been explicitly given by bracketing.

For operators of equal precedence, *associativity* controls the order of evaluation of the operations. Most operators are left-associative, so *A op B op C* is evaluated as *(A op B) op C*.

The following table gives an overview of precedence and associativity of operators in OAP[72].

[71] including the first or the last digit

[72] The table is for information only. The binding definition of precedence and associativity is given by the syntax rules in the following sections.

Precedence	Operator	Description	Associativity
1	`::`	scope for access to property values	
	`$`	placeholder	
	`()`	funktion call	
2	`+ -`	unary Plus and Minus	right
	`~ !`	bitwise and logical negation	
3	`* / %`	multiplication, division and modulo	left
4	`+ -`	addition and subtraction	links
5	`<< >> >>>`	shift operations	left
6	`< > <= >=`	relational operations	left
7	`== !=`	comparisons	left
8	`&`	bitwise AND	left
9	`^`	bitwise exclusive OR	left
10	`|`	bitwise OR	left
11	`&&`	logical AND	left
12	`||`	logical OR	left
13	`? :`	conditional evaluation	right
14	`,`	separator of expressions in argument lists	links

Subexpressions are generally evaluated from left to right[73].

A.4.2 Expressions

Expression:
UnaryExpression
ConditionalExpression

The evaluation of an expression or subexpression yields a value of one of types described in section A.2.

If one or more subexpressions of an expression return an error, unless otherwise specified, the error that first occurred in the evaluation order is returned. Whether further subexpressions are evaluated or not after the occurrence of an error in a subexpression is not specified.

If it is required in the following sections that a condition must be met, and the condition is not satisfied, then the corresponding expression returns an error.

A.4.3 Conditional evaluation

ConditionalExpression:
LogicalOrExpression
LogicalOrExpression `?` *Expression* `:` *ConditionalExpression*

The left expression must yield a boolean value. If the value is *true*, the result of the middle expression is returned, otherwise the result of the right expression.

The implementation behaves as if the respectively other expression would not be evaluated[74].

[73] With the functionality currently supported, the order of evaluation is of lesser importance, since all operators have no visible side effects, and OFML functions called via *methodCall()* are also required to have no side effect (see A.4.15).

[74] Implementations may speculatively evaluate the other subexpression as long as it is ensured that any errors that may occur are ignored and no side effects occur, or any side effects that may occur will be withdrawn if the evaluation of the other subexpression turns out to be unnecessary to determine the result of the expression

A.4.4 Logical OR operator

LogicalOrExpression:
LogicalAndExpression
LogicalOrExpression `||` *LogicalAndExpression*

The left expression must yield a boolean value. If the value is *true*, the result is `1`. Otherwise, the right expression is evaluated and must also yield a boolean value. If this is *true*, the result is `1`, otherwise `0`.

In case the left expression yields *true*, the implementation behaves as if the right expression would not be evaluated[75].

A.4.5 Logical AND operator

LogicalAndExpression:
BitwiseOrExpression
LogicalAndExpression `&&` *BitwiseOrExpression*

The left expression must yield a boolean value. If the value is *false*, the result is `0`. Otherwise, the right expression is evaluated and must also yield a boolean value. If this is *false*, the result is `0`, otherwise `1`.

In case the left expression yields *false*, it is unspecified, whether the right expression is evaluated. If it is evaluated and an error occurs during the evaluation, then the error is ignored.

In case the left expression yields *false*, the implementation behaves as if the right expression would not be evaluated[76].

A.4.6 Bitwise combinations

BitwiseOrExpression:
BitwiseExclusiveOrExpression
BitwiseOrExpression `|` *BitwiseExclusiveOrExpression*
BitwiseExclusiveOrExpression:
BitwiseAndExpression
BitwiseExclusiveOrExpression `^` *BitwiseAndExpression*
BitwiseAndExpression:
EqualityExpression
BitwiseAndExpression `&` *EqualityExpression*

The left and right operands must be of type *Int*. Both operands are combined bitwise with each other according to the operator. The result again is of type *Int*.

A.4.7 Operators to test for equality

EqualityExpression:
RelationalExpression
EqualityExpression `==` *RelationalExpression*
EqualityExpression `!=` *RelationalExpression*

When testing for equality, the following rules are applied in the given order:

- If one of the two operands is of type *Null*, both operands are considered equal if both operands are of type *Null*. Otherwise they are not equal.

[75] see footnote above for conditional evaluation
[76] see footnote above for conditional evaluation

- If both operands are of type *String*, *Symbol* or *Name*, then they are considered equal if their values match characters for characters.
- If both operands have a numeric type, they are considered equal if both have the same numeric value.
- If both operands are of the type *Sequence*, the comparison returns an error if one of the two sequences contains a value of type *Error*. Otherwise, the sequences are considered not equal if their length does not match. Otherwise, a pairwise comparison of the elements of both sequences occurs. If an error occurs, the comparison of the two sequences also yields an error. Otherwise, the sequences are considered equal if all elements are equal in pairs.
- Otherwise, the comparison returns an error.

The result of the operator `==` is `1` if both operands are equal, and `0` if they are not equal.

The result of the operator `!=` is `1` if both operands are not equal, and `0` if they are equal.

A.4.8 Relational operators

RelationalExpression:
ShiftExpression
RelationalExpression `<` *ShiftExpression*
RelationalExpression `<=` *ShiftExpression*
RelationalExpression `>=` *ShiftExpression*
RelationalExpression `>` *ShiftExpression*

Relative comparison of two values applies the following rules in the given order:

- If both operands are of type *String*, then both strings are compared character by character until a difference is found or the end of the shorter string is reached. In the case of a difference, the result of the comparison of the character strings is equal to the result of the comparison of the different characters, the character with the smaller character code being considered smaller. Otherwise, the shorter string is considered smaller.
- If both operands have a numeric type, the operand is smaller, which has the smaller numeric value.
- Otherwise, the comparison returns an error.

The result of operator `<` is `1` if the left operand is smaller than the right operand.

The result of operator `<=` is `1` if the left operand is less than or equal to the right operand.

The result of operator `>=` is `1` if the left operand is greater than or equal to the right operand.

The result of operator `>` is `1` if the left operand is greater than the right operand.

Otherwise, the result of the operators is `0`.

A.4.9 Shift operators

ShiftExpression:
AdditiveExpression
ShiftExpression `<<` *AdditiveExpression*
ShiftExpression `>>` *AdditiveExpression*
ShiftExpression `>>>` *AdditiveExpression*

The left and right operands must be of type *Int*. The least significant 5 bits of the right operand are considered as an unsigned integer and provide the number of bits, hereafter n, by which the left operand should be shifted.

Operator `<<` shifts the left operand n bits to the left, discarding the most significant n bits of the operand and setting the least significant n bits of the result to `0`.

Operator `>>` shifts the left operand n bits to the right, copying the most significant bit of the operand into the most significant n bits of the result and discarding the least significant n bits of the operand.

Operator `>>>` shifts the left operand n bits to the right, discarding the least significant n bits of the operand and setting most significant n bits of the result to `0`.

A.4.10 Binary arithmetic operators

AdditiveExpression:
    MultiplicativeExpression
    AdditiveExpression `+` *MultiplicativeExpression*
    AdditiveExpression `-` *MultiplicativeExpression*
MultiplicativeExpression:
    UnaryExpression
    MultiplicativeExpression `*` *UnaryExpression*
    MultiplicativeExpression `/` *UnaryExpression*
    MultiplicativeExpression `%` *UnaryExpression*

Both operands must have a numeric type. Operator `+` also can be used with two operands of type *String*.

If both operands have a numeric type and at least one operand has type *Float*, the other operand, if necessary, is converted to *Float* and the calculation is performed in *Float*. The result then also has type *Float*. Otherwise, the result has the same type like both operands.

If operator `+` is applied to operands of type *String*, the result is the concatenation of the left operand and the right operand, in this order.

The result of dividing by zero or modulo zero is always an error even if the left side of the operator is zero.

The operation $a \ \% \ b$ calculates the result of $a - n * b$, where n is the result of a/b rounded in direction to zero. The result of $-2^{31} \ \% -1$ is `0`.

A.4.11 Unary arithmetic operators

UnaryExpression:
    `+` *UnaryExpression*
    `-` *UnaryExpression*

If the operand of the unary operators `+` and `-` does not have a numeric type, then these operators return an error. Otherwise, the result has the same type and the same absolute value as the operand. In the case of operator `+`, the result has the same sign as the operand. In the case of operator `-`, the result has the opposite sign of the operand[77].

A.4.12 Operators for bitwise and logical negation

UnaryExpression:
    `~` *UnaryExpression*
    `!` *UnaryExpression*

The result of both operators has type *Int*.

In the case of operator `~`, the operand must be of type *Int*. The result is the bitwise negation of the operand.

In the case of operator `!`, the operand must have a boolean type. The result is `1` if the operand is considered *true*, otherwise `0`.

[77] The only exception is the negation of `0`, since there is no distinction between `+0` and `-0` in the two's complement representation.

A.4.13 Primary expressions

UnaryExpression:
    PrimaryExpression
PrimaryExpression:
    `(` *Expression* `)`
    FunctionCall
    PropertyReference
    Placeholder
    Literal
    `[` $ExpressionList_{opt}$ `]`
ExpressionList:
    Expression
    ExpressionList `,` *Expression*

A.4.14 Funktion call

FunctionCall:
    Identifier `(` $ExpressionList_{opt}$ `)`

A function call consists of the name of a predefined function followed by a possibly empty list of arguments enclosed in parentheses, where arguments are separated by commas.

Function calls can be nested, that is, the argument list on its part also can contain calls of arbitrary functions.

If an error occurred while evaluating the arguments of a function[78] or if there is no function with the specified identifier, the function call fails.

Currently predefined functions are the functions described in appendix B and the function *methodCall()* described in the next section.

A.4.15 Execution of MethodCall actions

Function *methodCall(*actionId*)* can be used in order to perform function calls defined by MethodCall actions. The argument *actionId* is any expression which must yield a string interpreted as the ID of the corresponding MethodCall action.

The function returns an error if any of the following conditions is satisfied:

- The function does not have exactly one argument of type *String*.
- The maximum number[79] of nested calls of MethodCall actions has been reached[80]. When counting the depth of nesting, the name of the action does not matter.
- In the evaluation context of the expression that contains the call of *methodCall()*, object `SELF` is not defined.
- An error occurred while querying the data of the MethodCall action, the type of action is not *MethodCall*, or the action's data is incorrect.
- The evaluation of the execution condition of the action (if specified) definitely yields *true*.

[78] i.e., one of the arguments has type *Error*
[79] In the current implementation (October 2017) the value is set to 10.
[80] Nested MethodCall actions may occur if the execution condition of a MethodCall action itself uses function *methodCall()*.

- In case of an instance method:
  - An error occured during the evaluation of an object definition or during the generation of the corresponding OFML instance.
  - After evaluating all object definitions, not exactly one object was determined.
  - The verification of the context (class, interface) defined for the MethodCall action failed for the determined object.
- An error occurred while evaluating the arguments defined by the MethodCall action using the current evaluation context.
- The call of the OFML method failed.
- The result returned by the OFML method could not be converted to an OAP data type.

OFML methods which are called via function *methodCall()* must not have any visible side effects, because for performance reasons it is not possible to update the information about OFML instances[81], stored in different parts of the application, after every call of *methodCall()*.

A.4.16 Access to property values

PropertyReference:
    PropertyName
    Scope `::` *PropertyName*
PropertyName:
    Identifier
Scope:
    Identifier

A property value access without specified scope corresponds to an access with scope `SELF` (see also below).

During a property value access, the identifier for the scope is looked up in the object table of the evaluation context (see A.1) to determine the corresponding *article instance identifier* (AIID). This then is used to query the application for the value of the specified property of the referenced article instance[82].

If no entry is found in the object table for the specified scope, or if the referenced object has no property with the specified name, or if the value of the property can not be converted to an OAP value, the result of the property value access is `NULL`. Supported types of property values are *Symbol*, *String*, *Null*, *Int* and *Float*.

Currently, the following *scopes* are defined:

SELF References the active object (corresponds to object category *Self*, see 4.9).

PARENT References the immediate parent article of the active object (corresponds to object category *ParentArticle*, see 4.9).

TOP References the parent article of the active object that is highest in the hierarchy (corresponds to object category *TopArticle*, see 4.9).

[81] such as article numbers, variant codes, article and variant texts

[82] In fact, in the current implementation (October 2017), the entire PropVarCode is queried by the application and stored in the evaluation context for possible reuse, or a PropVarCode stored there is used directly. The property value is extracted from the PropVarCode.

A.4.17 Placeholder

Placeholder:
`$` *Identifier*

In placeholder substitution, the identifier specified as a placeholder is looked up in the value table of the evaluation context (see A.1). If an entry is found, the corresponding value is returned.

Otherwise, the identifier specified as a placeholder is looked up in the object table of the evaluation context. If no entry is found there, the result of the placeholder substitution is `NULL`. Otherwise, for the *article instance identifier* (AIID) that was found in the object table, the corresponding OFML instance is determined or recreated (if it currently does not exist). The placeholder substitution result then is a value of type *Name* representing the name of the OFML instance.

Currently, the following *placeholder* are defined:

INTERACTOR
: This placeholder is replaced by the identifier of the interactor for which currently information is being determined or to which the currently executed action is bound.

SELF
: This placeholder is replaced by the name of the OFML instance of the active object (see also scope `SELF` in A.4.16).

A.4.18 Literals

Literal:
SymbolLiteral
StringLiteral
IntegerLiteral
FloatingPointLiteral
NullLiteral

The syntax of literals is described in A.3.2.

A.4.19 Identifier

An identifier consists of a non-empty sequence of letters, numbers and/or underscore, where the first character may not be a number and only ASCII characters may be used[83].

[83] corresponds to field type *Symbol* in the OAP tables (see 3.2)

B Functions

B.1 Mathematical functions

All arguments must have a numeric type (see A.2.9). Arguments of type *Int* are converted to type *float* before the function is calculated. The value of the arguments must be in the specified range[84]. If no error is returned, the return value of all functions has type *Float*.

Trigonometric functions

acos(x) Calculates the inverse cosine of x for $-1 \leq x \leq 1$.

acosh(x) Calculates the inverse hyperbolic cosine of x for $1 \leq x < +\infty$.

asin(x) Calculates the inverse sine of x for $-1 \leq x \leq 1$.

asinh(x) Calculates the inverse hyperbolic sine of x for $-\infty < x < +\infty$.

atan(x) Calculates the inverse tangent of x for $-\infty < x < +\infty$. The result is in the range $[-\pi/2, +\pi/2]$.

atan2(y,x) Calculates the inverse tangent of y/x for $-\infty < x < +\infty$ and $-\infty < y < +\infty$ using the signs of both arguments to determine the quadrant of the return value. The result is in the range $[-\pi, +\pi]$.

In general, the result has the sign of y.

For positive x the absolute value of the result is less than $\pi/2$.

For negative x the absolute value of the result is greater than $\pi/2$.

If x is zero, the following rules apply:

- If y is not equal to zero, then the result is $\pm\pi$.
- If x is +0.0 and y is $\pm$0.0, then the result is $\pm$0.0.
- If x is -0.0 and y is $\pm$0.0, then the result is $\pm\pi$.

atanh(x) Calculates the inverse hyperbolic tangent of x for $-1 < x < 1$.

cos(x) Calculates the cosine of x for $-\infty < x < +\infty$.

cosh(x) Calculates the hyperbolic cosine of x for $-\infty < x < +\infty$.

sin(x) Calculates the sine of x for $-\infty < x < +\infty$.

sinh(x) Calculates the hyperbolic sine of x for $-\infty < x < +\infty$.

tan(x) Calculates the tangent of x for $-\infty < x < +\infty$[85].

tanh(x) Calculates the hyperbolic tangent of x for $-\infty < x < +\infty$.

[84] Even if all arguments are in the specified range, an error can be returned if the result exceeds the value range of type *Float*. Underruns of the value range (values with very small amount) do not lead to an error. Instead, zero is returned.

[85] Theoretically, the tangent function is not defined for $\pi/2 + n * \pi$ with integer n. However, since π can not be accurately represented, that should not be a problem in practice. For example, the result of $tan(atan2(1, 0))$ is `1.63312393531954e+16`.

Power, exponential and logarithmic functions

exp(x)	Calculates the value of exponential function e^x for $-\infty < x < +\infty$.
log(x)	Calculates the natural logarithm of x for $0 < x < +\infty$.
log10(x)	Calculates the common logarithm of x for $0 < x < +\infty$.
logb(x)	Calculates the binary logarithm of x for $\infty < x < +\infty$ rounded towards negative infinity[86].
pow(x,y)	Calculates the value of x^y for $-\infty < x < +\infty$ and $-\infty < y < +\infty$. If x is negative, y must be an integer value. If x is `0`, y must not be negative. The result is `1.0` if both x and y are equal to `0`.
scalb(x,y)	Calculates $x * 2^n$, where n is y rounded towards zero, for $-\infty < x < +\infty$ and $-\infty < y < +\infty$.
sqrt(x)	Calculates the square root of x for $0 \leq x < +\infty$.

Rounding, absolute value and remainder

ceil(x)	For $-\infty < x < +\infty$ calculates the smallest integer value that is not less than x.
fabs(x)	Calculates the absolute value of x for $-\infty < x < +\infty$.
floor(x)	For $-\infty < x < +\infty$ calculates the largest integer value that is not greater than x.
fmod(x,y)	Calculates the remainder of the floating-point division x/y for $-\infty < x < +\infty$ and $-\infty < y < +\infty$ with $y \neq 0.0$. This corresponds to the operation $x\%y$ with the difference that the result always has type *Float*.
remainder(x,y)	Calculates the remainder of the division x/y for $-\infty < x < +\infty$ and $-\infty < y < +\infty$ with $y \neq 0.0$. The result is $x - n * y$, where n is the result of x/y rounded toward the next integer. If the absolute value of $x - n * y$ is `0.5` then n is chosen to be even.

[86] The result is equal to the value of the exponent of the internal floating point representation, converted to *Float*.

B.2 Type conversion functions

All conversion functions exist in two variants.

The first variant has an argument specifying the value to be converted.
This variant returns an error (a value of type *Error*) if the conversion was not successful.

The second variant has two arguments. The first argument specifies the value to be converted, the second a default value, which is used as the return value of the conversion function if the conversion was not successful. The type of the default value is not restricted.
It is not defined whether the expression in the argument for the default value is evaluated or not.

B.2.1 Conversion to *Int*

int(value)
int(value, default)

The function converts the value of the expression passed in argument *value* into a value of type *Int*.

The following conversions are supported:

- from *Int* to *Int*:

  The return value of the conversion function is equal to the value of argument *value*.

  This conversion always is successful.

- from *Float* to *Int*:

  In the first step of the conversion, the fractional part of the value of argument *value* is truncated (rounding toward zero).

  The conversion function returns the value determined in step 1 as *Int* if it can be represented as such ($-2^{31} \leq I < 2^{31}$). Otherwise, the conversion is not successful.

- from *String* to *Int*:

  The value of argument *value* has to be a string composed of the following components: optional leading spaces, an optional negative sign ('-'=U+002D), an integer literal as described in appendix A.3.2, optional trailing spaces. Otherwise, the conversion is not successful.

  Without sign, the return value of the conversion function is equal to the value of the literal, otherwise equal to the negated value of the literal (see A.4.11).

B.2.2 Conversion to *Float*

float(value)
float(value, default)

The function converts the value of the expression passed in argument *value* into a value of type *Float*.

The following conversions are supported:

- from *Int* to *Float*:

  The return value of the conversion function is a floating-point number with the same value as argument *value*.

  This conversion always is successful.

- from *Float* to *Float*:

  The return value of the conversion function is equal to the value of argument *value*.

  This conversion always is successful.

- from *String* to *Float*:

  The value of argument *value* has to be a string composed of the following components: optional leading spaces, an optional negative sign ('-'=U+002D), a floating-point literal as described in appendix A.3.2, optional trailing spaces. Otherwise, the conversion is not successful.

  Without sign, the return value of the conversion function is equal to the value of the literal, otherwise equal to the negated value of the literal (see A.4.11).

B.2.3 Conversion to *Symbol*

symbol(value)
symbol(value, default)

The function converts the value of the expression passed in argument *value* into a value of type *Symbol*.

The following conversions are supported:

- from *Symbol* to *Symbol*:

  The return value of the conversion function is equal to the value of argument *value*.

  This conversion always is successful.

- from *String* to *Symbol*:

  The conversion function returns a value of type *Symbol*, representing the same string as the string to be converted.

  This conversion always is successful.

  > Note:
  > There is no special handling for the character '`@`' (U+0040) at the beginning of the string. Therefore, expression `symbol("@FOO")` returns a symbol yielding string `"Symbol(\"@FOO\")"` after formatting according to the OFML Script Object Notation (OSON) (instead of `"@FOO"`).

B.2.4 Conversion to *String*

string(value)
string(value, default)

The function converts the value of the expression passed in argument *value* into a value of type *String*.

The following conversions are supported:

- from *Int* to *String*:

  The return value of the conversion function is a string that conforms to the rules for decimal integer literals (see appendix A.3.2) and represents the same value as the argument *value*.

  This conversion always is successful.

- from *Float* to *String*:

  The return value of the conversion function is a string that conforms to the rules for floating-point literals (see appendix A.3.2) and Though, the exact representation is unspecified apart from the following exceptions:

  - The string contains a decimal point or an exponent, or both.
  - The conversion is done with an precision of at least fifteen decimal digits in the mantissa.
  - The conversion does not produce any significant zeros after the first decimal digit of the mantissa.

  This conversion always is successful.

Note:
The conversion is not exact. It is not guaranteed that the expression `float(string(x)) == x` is true.

- from *Symbol* to *String*:

  The conversion function returns a value of type *String*, representing the same string as the symbol to be converted.

  This conversion always is successful.

  Note:
  The value of the expression `string(@FOO)` is `"FOO"` (not `"@FOO"`).

- from *String* to *String*:

  The return value of the conversion function is equal to the value of argument *value*.

  This conversion always is successful.

C Modification history

C.1 OAP 1.4, 1st revised version

- Correction and clarification regarding the usage of object category *MethodCall* (section 4.9).

C.2 OAP 1.4 vs. OAP 1.3

- New interactor symbol type *Attention* (section 4.6).
- New action type *Message* (section 4.7).
- Clarification on the use of conditions for properties and classes with actions of type *PropEdit2* (section 4.8.3).
- Actions of type *DimChange* now can also use properties representing a symbolic choice list if the mapping method *symbolicPropValue2Float()* is implemented (section 4.8.4).
- New section describing the parameter table for new action type *Message* (section 4.8.7).
- Clarification on the usage of escape sequences in texts (section 4.10).

C.3 OAP 1.3 vs. OAP 1.2

- Correction regarding possible follow-up actions after actions of type *PropEdit2* (section 4.7).
- Clarification on the handling of the status in the case of an action of type *PropEdit2* with only one valid property (section 4.8.3).
- Scopes PARENT and TOP now are supported for property value access (appendix A.4.16).
- Placeholder $INTERACTOR now is generally supported[87] (appendix A.4.17).

C.4 OAP 1.2 vs. OAP 1.1

- The descriptions of tables and features that are not yet supported have been removed or grayed out.
- Corrections and more detailed definition regarding the interactor concept (section 2.1).
- Field *NeedsPlanMode* in table `Interactor` is marked as *deprecated* (section 4.6).
- New interactor symbol types *StartDimChange* and *Video* (section 4.6).
- Action type *PropEdit* is *deprecated*. Instead, new action type *PropEdit2* should be used (section 4.7).
- New section describing the parameter tables for new action type *PropEdit2* (section 4.8.3).
- More detailed definition regarding material images for property values in `PropEdit2` dialogs (section 4.8.3).
- In field *Dimension* in table `DimChange` (section 4.8.4) now it is possible to specify the change direction for a given dimension.
  Furthermore, table `DimChange` now gets additional fields *Separate* and *ThirdDim*.
  Note: If an existing project is ported to OAP 1.2, any existing table `DimChange` must be adapted!
- In table `CreateObj` PosRotMode *AttachAreas* was removed (section 4.8.5).
- New section describing parameter table `ExtMedia` (section 4.8.8).

C.5 OAP 1.1 vs. OAP 1.0

- New interactor symbol type *OnOff* (section 4.6).

[87] not only in validity conditions of interactors