X-definition 4.2

Language Description

Author: **V.Trojan**
Version: **4.2.0.0**
Date: **2022-06-15**

Contents

Tables

Changes in this document

Questions, remarks, and bug reports please send to: xdef@syntea.cz.

The actual version of X-definition can be downloaded from https://github.com/Syntea/xdef

or https://www.xdefince.cz/en.

Version 4.2

(1) X-definition namespace version 4.1 is: http://www.xdef.org/xdef/4.2

(2) X-definition version 2.0 is no more supported. However, versions 3.1, 3.2, 4.0, and 4.1 are supported.

(3) Utilities for conversion XML schema to/from X-definition (classes `org.xdef.util.XdefToXsd`, `org.xdef.util.XsdToXdef`) were removed from this project. (Those utilities are now available in a separate project on https://github.com/Syntea/xdef-transform.)

(4) Implemented XON, the new format of object notation (see Chapter 6).

(5) Implemented models of JSON, YAML, Properties, and INI data.

(6) Implemented new validation methods `gps` and `price` (see table

1 Annotation

This document describes the programming language **"X-definition 4.2**". X-definition is designed for the description, processing, and construction of data in the form of XML.

X-definition is an open-source tool that describes both the structure and properties of data values in an XML document. Moreover, X-definition allows describing the processing and construction of XML objects. Thus X-definition may replace existing technologies commonly used for XML validation - namely the DTD (Data Type Definition) or the XML schemas, Schematron, and also in some cases XSLT.

X-definition enables us to merge both validations of XML documents and the processing of data (i.e. actions assigned to events when XML objects are processed). Compared to technologies based on DTD and XML schemas, the advantage of X-definitions is (not only) higher readability and easier maintenance. X-definition has been designed for processing XML data files of unlimited size, up to many gigabytes.

The principal property of X-definition is the maximum respect for the structure of the described data. The form of X-definition is an XML document with a structure similar to described XML data. This makes it possible to quickly and intuitively describe given XML data and its processing.

Properties of XML items (and events that may happen in the process) are described by the language **X-script**. In many cases, in the X-definition you just replace values of described XML data with the *X-s*cript language. You can also gradually add to your *X-*script the required actions for data processing. You can take a step-by-step approach to your work.

It is assumed that the reader already knows the elementary principles of XML. To get the most out of this document, you should have at least basic knowledge of the Java programming language.

X-definition technology also enables us to generate a source code of Java classes representing XML elements described by X-definition. Such a class is called the **X-component**. You can use the instances of XML data in the form of X-components (similar way as in the JAXB technology).

In X-definition 4.2, it is possible to validate and process also JSON data. The models of the structure of JSON data are described by elements xd:xon.

X-definition 4.2 enables language localization of tag names of described XML data. Different language versions may be described by elements xd:lexicon.

For the term "X-definition" we use it in two different meanings: either as the name of the programming language or as an XML element containing the code of X-definition language.

The initial information about the X-definition is available at:

https://xdef.syntea.cz/tutorial/en/index.html

2 Essential concept

The basic concept of X-definition is the "**model**" of an XML element. A simple example of the model you can get is if all data values (i.e. attributes and/or text nodes) in an XML element are replaced by a description of its data type properties.

Let's have the following example of XML data:

```
<Employee FirstName   = "Andrew"
          LastName    = "Aardvark"
          EnterDate   = "1996-3-12"
          Salary      = "21700" />
  <Address Street = "Broadway"
           Number = "255"
           Town   = "Beverly Hills"
           State   = "CA"
           Zip    = "90210" />
  <Competence> electrician </Competence>
  <Competence> carpenter </Competence>
</Employee>
```

You can see that this record contains different data types (names, dates, salary, description of qualification, etc.). We can describe the contents of a such record when we replace the data values with their description. Let's say the attribute "Salary" is optional, and the other attributes are required. The expected form of the value of text can be checked with special parsing methods. Since the child element "Competence" may repeat more times, the range of occurrences is described in a special attribute "xd:script". The model of the element above would look as follows:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2">
  <Employee FirstName = "required string()"
            LastName  = "required string()"
            EnterDate = "required date()"
            Salary    = "optional decimal()" />
     <Address Street = "required string()"
              Number = "required int()"
              Town   = "required string()"
              State  = "required string()"
              Zip    = "required int()" />
     <Competence xd:script = "occurs 1..5">
        required string()
     </Competence>
  </Employee>
</xd:def>
```

The names of elements and attributes of X-definition are from the namespace of X-definitions:

```
http://www.xdef.org/xdef/4.2
```

This way it is possible to distinguish between the objects of the model and the auxiliary objects of the X-definition. The qualified names with the prefix "xd" assigned to the namespace of X-definitions are used for the auxiliary attributes and elements. Inside the model, there are models of the attributes, models of text nodes, and models of child elements.

The language used for description in the values of attributes and text nodes is called the X-s**cript of X-definition**, or hereafter shortly X-s**cript**. The *X-*script language can be written in different parts of the X-definition source code and it is divided into several parts according to the purpose of what is described.

Usually, you need to design a set of X-definitions to describe a variety of processed data. The set of X-definitions is compiled into the binary code. The compiler creates the binary code saved as an instance of Java class XDPool. The set of X-definitions used in the compilation we call the **project**.

The processor of X-definition runs in two different modes: **validation** and **construction mode**. In the validation mode, the input XML data are validated and processed according to the X-definition and the X-script. The result will be a valid XML document (representative of the model). In the construction mode, the processor creates new XML data constructed according to the model from X-definition. The result is a constructed XML document. In both modes, the process generates a log file containing information about error messages, warnings, etc.

Normally, the processor of X-definition returns data in the form of org.w3c.dom.Element. However, if required, the result of the processing may be returned in the form of an instance of the Java class (an **X-component)**. The generation of Java code of X-component classes is described in Chapter 7 X-components.

2.1 Quantifier and Validation Method

Validate an XML document requires checking the occurrences of data items as well as checking the formal correctness of actual data values (a data type). It is possible to specify attributes and/or text nodes of elements and describe properties that they must fulfill. This part of the X-script we call the **validation section** of the X-script.

The description of the occurrence of an item in the model we call the **quantifier**. Quantifiers of text nodes and attributes holding the information about the item can be required or optional. However, the explicit form of Quantifier is:

```
occurs min..max
```

"min" here is an integer expressing the minimum limit of occurrences of the item and "max" is the maximum limit of occurrences. To make the X-script clearer you can write "required" instead of "occurs 1..1" and "optional" instead of "occurs 0..1". The unlimited number of occurrences is specified by the asterisk ("*") instead of the max parameter. E.g.:

```
occurs 1..*
```

The validation process consists of two steps:

a) Checking of occurrence (elements, attributes, text nodes) according to the Quantifiers

b) Checking the correctness of a text value (only attributes or text nodes) according to preset validation methods

2.2 Model of Element

The element model is declared as the direct child element of the X-definition. From the model of an element, you can make references to other models in different parts of the X-definition. An element model can be either the description of the root element of the data or the description of child elements. The occurrence is taken from the place where the model is referred to. However, the occurrence of the model of the root element is set to "occurs 1..1" when it is processed. The element models are units from which the X-definitions are composed. Within one X-definition XML source, many element models may be described.

2.3 Model of the attribute or text node (model of data value)

The description of attributes and text values requires parsing their text values. To check if the value is correct, a **validation method** is performed. The result of the validation method is an object ParseResult which contains the original text value and the parsed object. If a validation method recognizes an error, it adds to the ParseResult object an error message. For example, if we want the data to be a whole number, we can specify the validation method "integer()" (in fact, this represents invoking of a method that parses the text value and it returns the ParseResult object containing parsed integer number or the error message why parsing failed). If the parsing method didn't recognize an error, the result is true. If an error was found, the result is false.

The validation method may also have a list of parameters – if, for instance, the value of a number must be greater than or equal to 100 and less than or equal to 999, you can write "integer(100, 999)". Or, if the length of a character string must be between 2 and 30 characters, we can write "string(2,30)". All validation methods are described in paragraph 4.1.13 Implemented validation methods.

So, the **model of a data value** is described by a **quantifier** ("occurs 0..1" or "required" or "occurs 1..1" or "optional") and by a **validation method**.

2.4 Arbitrary attribute(s) (xd:attr)

If you also want to process attributes that were not specified in a model, you can write a special attribute "xd:attr" and further describe what should happen in this case. In the following example. In the element <a> may occur any attribute and it must be an integer number:

```
<a xd:attr = "occurs 0..*; int();" />
```

The selection of those attributes may be described by the "match" action specified in the X-script:

```
<a b="float()" xd:attr="match int(); occurs 1..*">
```

Only the attribute "b" is a float number. Any value of other attributes must be an integer (at least one such attribute is required).

2.5 Arbitrary element (xd:any)

In X-definition, you can specify the model of an element with an arbitrary name. The declaration of such an element can be written as <xd:any>. The attributes and the child nodes of this element are described the same way as in models of an element. An example of “any" element:

```
<xd:any xd:script = "occurs *" />
```

The element from our example can have an arbitrary XML name. Attributes and child nodes are not allowed.

If the element has attributes and child nodes it can be expressed by the options "moreAttributes", "moreElements" and "moreText":

```
<xd:any xd:script = "options moreAttributes, moreElements, moreText"/>
```

If the <xd:any> is declared as a direct child of X-Definition, you must specify the attribute "xd: name". Then it is possible to refer to such a model by the "ref" command with the X-position pointing to this name:

```
<xd:def xd:root="foo" ... >
  <xd:any xd:name = "foo" ...>
  ...
  <x>
    <xd:any xd:script="ref foo"/>
 </x>
  ...
```

2.6 Concatenated text of element (the attributes xd:text and xd:text content)

X-definition enables one to describe individual child text nodes of a given model of the element. However, it is possible to describe the text values that are not described by explicit models. There are two possibilities for such a description:

a) using the attribute **xd:text** - specification of this attribute processes **all** text nodes **not** processed by any model of a text node. The text value of all such text nodes is validated by the validation section of the attribute "xd:text".

   X-definition:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2" xd:root="A">
  <A xd:text= "string(5) finally outln(getText());">
    <B/>
  </A>
</xd:def>
```

   Data:

```
<A>text1<B/>text2</A>
```

   Prints:

```
text1
text2
```

b) by the attribute **xd:textcontent** - before the event "finally" is created a string which is constructed as the concatenation of all text nodes of an element and then validated.

   X-definition:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2" xd:root="A">
  <A xd:textcontent= "string(10) finally outln(getText());">
    <B/>
  </A>
</xd:def>
```

   Data:

```
<A>text1<B/>text2</A>
```

   Prints:

```
text1text2
```

2.7 Reference

Let us consider an XML data structure describing a family:

```
<Family>
  <Father    GivenName  = "John"
             FamilyName = "Smith"
             PersonalID = "7107130345"
             Salary     = "18800" />
  <Mother    GivenName  = "Jane"
             FamilyName = "Smith"
             PersonalID = "7653220029"
             Salary     = "19400" />
  <Son       GivenName  = "John"
             FamilyName = "Smith"
             PersonalID = "9211090121" />
  <Daughter  GivenName  = "Jane"
             FamilyName = "Smith"
             PersonalID = "9655270067" />
  <Residence Street     = "Small"
             Number     = "5"
             Town       = "Big"
             Zip        = "12300" />
</Family>
```

In the above example, let's consider that the father may not exist, while the occurrence of a mother must be one and only one. Sons and daughters may occur in an unlimited quantity, or may not occur at all.

If we want to utilize a defined structure separately, you can use a **reference**. In the X-script, the reference is introduced by the keyword "ref" and followed by a specification of the X-position of the referred model. In our example, it is provided by the specification of the name of the referred model. However, the position may link to any object of X-definition. The description of the position of an object in the X-definition we call **X-position**.

Parts of the X-script of the referred object may be overloaded. So, for example, the X-script containing a reference may contain the specification of its occurrence. Use the delimiting character ";" (semicolon) to separate individual sections of the X-script. The occurrence of an element is written as the keyword "occurs" followed by a specification of the minimal and the maximum number of occurrences separated by ".." (two dots). If the minimum value is equal to the maximum, then the occurrence may be represented by only one number.

The X-script which belongs to the element description is specified with the auxiliary attribute "xd:script":

```
<Son  xd:script = "occurs 0..1; ref Person" />
```

This way we can create aggregated X-definitions. Consider the model of the family where you have defined the unlimited number of sons and daughters. Our data with family may or may not have a father, but must have a mother. The model of family data may then appear as:

```
<Family>
  <Father    xd:script = "occurs 0..1; ref Person" />
  <Mother    xd:script = "occurs 1..1; ref Person" />
  <Son       xd:script = "occurs 0..*; ref Person" />
  <Daughter  xd:script = "occurs 0..*; ref Person " />
  <Residence xd:script = "occurs 1; ref Address" />
</Family>

<Person GivenName  = "string()"
        FamilyName = "string()"
        PersonalID = "int()" />

<Address Street = "string()"
        Number = "int()"
        Town   = "string()"
        Zip    = "int()" />
```

With the reference to the model Person, the structure expression of all family members is equal (and it makes the source simpler and clearer).

2.8 X-position

In the previous paragraph was described the possibility to specify a reference to an object in the X-definition was. The reference may link to the model in the actual X-definition or to another X-definition. The X-position must then begin with the name of the X-definition followed by the character "#":

```
<xd:def name="A">
  <a>
    <x/>
    <y z="string()"/>
    <x/>
  </a>
</xd:def>

<xd:def name="B">
  <b xd:script="ref A#a"/>
</xd:def>
```

It is also possible to link to a child part of a model. The X-position then contains the path to the referred child node (to the second occurrence of the node "x" in the model "a" of the X-definition "A"):

```
<b xd:script="ref A#a/x[2]">
```

The first occurrence of "x" is not required to be specified as "[1]":

```
<b xd:script="ref A#a/x">
```

has the same meaning as:

```
<b xd:script="ref A#a/x[1]">
```

The X-position of an attribute is recorded with the character "@":

```
<aa bb="ref A#a/y/@z">
```

So, the X-position starts with the optional specification of the X-definition name followed by the character "#". The name of the referred model is required. After the specification of a model, the path to a child object may be specified. Note that the specification of the model can be the name of the group, "xd:text" model, "xd:any" model, etc.:

```
<p xd:script="ref b/y"/>
<q xd:script="ref c"/>
<r>ref d; <r>
<s att="ref d">
...
<xd:mixed xd:name="b">
    <x/>
    <y/>
<xd:mixed>
<xd:any xd:name="c"> int() </xd:any>
<xd:text xd:name="d"> date() </xd:text>
```

2.9 Extension and modification of the referred model

In the model, which includes a reference to the model, it is possible to add the specification of attributes and child nodes and modify the X-script. In the example of a family above was written a quantifier of each member of the family. You can change or add also the other parts of the X-script. E.g.:

```
<Father xd:script = "occurs 0..1; ref Person; finally outln('Father: ' + @firstName)" />
```

when an element "Father" is processed, his first name will be printed

You can also add an attribute or a child node:

```
<Father xd:script = "occurs 0..1; ref Person;"
        Salary = "decimal" >
  <Profession> string </Profession>
</Father>
```

The model of "Father" is extended from the model "Person" and contains the additional attribute "Salary" and the child element "Profession".

2.10 Event

In the X-script language of X-definitions, you can describe what should happen in different events that may occur during the processing of data. For instance, you can describe what to do when the validation returns the value of “True” or “False”. For each event, you can assign an **event name --** “onTrue” or “onFalse”, for example. The resulting action is described in the X-script by the keyword with the event name, followed by the action command. An example of validation of the item “Salary“ might look like this:

```
Salary="optional int(1000,50000); onTrue outln('ok'); onFalse error('Salary error');"
```

In the above example, there are two events connected with the validation: The event where the validation section passed without error is called **onTrue**; if the validation is negative the event is called **onFalse**. If the validation is positive the "outln" method is invoked and if validation is negative then it is invoked the method "error" with the parameter "Incorrect salary". If the event is not described, the system invokes the relevant standard action (copy data to the output document, in the event "onTrue", or write an error message in the log file in the event "onFalse").

2.11 Group

When we are describing the data structures it is sometimes necessary to describe groups of models. The description of a group is placed either in the model element or as the direct child element of an X-definition (with the attribute "xs:name" which enables referencing to this group from models).

In X-definitions we can define three kinds of groups: mixed groups, choice groups, and sequences. any group may have specified the optional attribute "xd:script". On the level of X-definition, the attribute "xd:name" is required.

2.11.1 A mixed group of nodes (xd:mixed)

Mixed groups describe lists of models that can occur as an arbitrary sequence of nodes. To describe a mixed group, use the auxiliary element "xd:mixed". Two nodes with the same name cannot be in the list of elements in the mixed group (if not specified an action "matches"). Example:

```
<Family>
     <Father xd:script = "ref Person; occurs 0..1 "/>
     <Mother xd:script = "ref Person; occurs 1..1" />
     <xd:mixed>
       <Son      xd:script = "ref Person; occurs 0..12 "/>
       <Daughter xd:script = "ref Person; occurs 0..12 "/>
     </xd:mixed>
     <Residence xd:script = "ref Address; occurs 1..1" />
</Family>
```

***Note**: In the example above, all nodes are declared as optional. However, at least one of the nodes must be present in the data. If you want to accept also an empty group specify "optional" in the X-script:*

```
   <xd:mixed xd:script="optional">
      ...
   </xd:mixed>
```

2.11.2 Selection of models (xd:choice)

The choice groups allow you to describe a selection of a node from the list. The names of the nodes in the list must be unambiguous, as with mixed groups. The auxiliary element "xd:choice" is used to define a choice group.

Example:

```
<Subject>
   <xd:choice>
       <Person xd:script = "ref Person; occurs 1..1"/>
       <Company xd:script = "occurs 1"
                Name        = "required string ()"
                CompanyID   = "required num(8)" />
   </xd:choice>
   <Residence xd:script = "ref Address; occurs 1"/>
</Subject>
```

Note on filters in Choice Sequences

Filters are designed for special cases. You can use filters in choice sequences as described below, but they are not generally useful for other functions.

Some applications need to distinguish between several models of elements, not by the name of an element, but according to the value of an attribute. For such cases, you may describe an auxiliary action "**match**". This action contains a command which returns "true" or "false" and filters the choice items in the same way as the selection by the element name:

```
<Subject>
   <xd:choice>
       <Object xd:script  = "match @Type EQ 'Person'; occurs 1"
```

```
               Type         = "fixed 'Person'"
               GivenName    = "required string(1,30)"
               FamilyName   = "required string(1,30)"
               BirthDate    = "required date" />

      <Object xd:script  = "match @Type EQ 'Company'; occurs 1"
               Type         = "fixed 'Company'"
               Name         = "required string"
               CompanyID    = "required num(8)" />
   </xd:choice>
   <Residence xd:script = "ref Address; occurs 1..1"/>
</Subject>
```

Note the result of the action "match" here is a Boolean value that evaluates the attributes of the current element. If there is a specification of the attribute which is not followed by a relational operator then the value is true if the attribute with the specified attribute exists (see 4.1.2.7 boolean (Boolean values)*). So, we could also write the previous example as:*

```
<Subject>
   <xd:choice>
      <Object xd:script  = "match @BirthDate; occurs 1..1"
               GivenName    = "required string (1,30)"
               FamilyName   = "required string (1,30)"
               BirthDate    = "required date"/>

      <Object xd:script  = "match @CompanyID; occurs 1..1"
               Name         = "required string ()"
               CompanyID    = "required num(8)" />
   </xd:choice>
   <Residence xd:script = "ref Address; occurs 1..1"/>
</Subject>
```

2.11.3 A sequence of nodes (xd:sequence)

The sequence group describes a group of elements that must occur in the data in the given order. The sequence group behaves the same way as the specification of child nodes of an element. The nodes can be specified repeatedly, and the names are not required to be unique. The sequence group is specified by the auxiliary element "xd:sequence":

```
<Object>
   <xd:sequence>
     <Person  xd:script = "occurs 1" Name  = "required string()" />
     <Company xd:script = "occurs 1" Title = "required string()" />
   </xd:sequence>
</Object>
```

2.11.4 Group declared as a model and the reference to the group

A group may be declared on the level of X-definition (as the direct child of "xd:def"). Then it is possible to use a reference to a group from a model. In this case, the attribute "xd:name" must be specified. The name is used then as the reference to the model of a group in the attribute X-script.

Example:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2">

  <Family>
     <xd:mixed xd:script="ref FamilyGroup" />
  </Family>

  <xd:mixed xd:name = "FamilyGroup">
     <Father ...
     <Mother ...
     <Son ...
     <Daughter ...
  </xd:mixed>
</xd:def>
```

2.12 BNF grammar in X-definition

The BNF grammar is described with extended Backus-Naur form (EBNF). The EBNF describes the formal syntax of a string by the set of production rules.

2.12.1 BNF production rule

Each production rule (hereafter "rule") has a name. The name of a rule must start with a letter or the character '_' (underscore character). After the first character may follow a sequence of letters, underscores, and decimal digits. The name of a rule is on the left side of "::=". On the right side of "::=" follows a formula describing the rule. Each rule describes one symbol of the grammar in the form:

```
ruleName ::= BNF expression
```

2.12.2 BNF terminal symbols

Terminal symbols (character sequences) are described by the following formulas:

#xN	is the character expressed as a numeric UTF-16 (the code) N. The N is expressed as a hexadecimal number. Leading zeroes are ignored
"string" or **'string'**	is the sequence of characters in quotation marks or apostrophes

2.12.3 Case insensitive symbols

Case-insensitive symbols (character sequences) are described by the following formulas:

#xN%	character with numeric UTF-16 (the code) N. The N is expressed as a hexadecimal number. Leading zeroes are ignored
"string"% or **'string'%**	sequence of characters in quotation marks or apostrophes

2.12.4 BNF set of characters

[a-zA-Z] or **[#xN-#xN]**	the record a-b represents the set of characters from the closed interval <a,b>
[abc] or **[#xN#xN#xN]**	list of characters
[^a-z] or **[^#xN-#xN]**	all characters out of the specified interval
[^abc] or **[^#xN#xN#xN]**	all out of the list

Note that "[" you must specify as #135, and also the characters below the numeric value of space you must declare as a numeric value (e.g. CR as #10, LF as #13, etc.), and the "-" must be specified as the first character of the list.

2.12.5 BNF quantifier (repetition of a rule)

The quantifiers allow you to describe the allowed number of consecutive string occurrences corresponding to the rule to which the quantifier relates:

A?	rule A is optional
A+	rule A may occur once or more times
A*	rule A may not occur or occur more times
A{n}	rule A must occur exactly n-times
A{m, n}	rule A may occur at minimum m-times and maximum n-times
A{m,}	rule A may occur minimum m-times or more times

2.12.6 BNF expression

The above constructs can be presented in compound rules describing non-terminal symbols. Expressions on the right side may contain the elements or links to another rule using the rule name and can be composed of the following components. Any part of the entry may be in brackets:

A – B	restriction, a character string that meets rule A but also doesn't meet rule B. The restriction operation has a higher priority than the concatenation operation or selection operation. So: **A - B C - D** is equivalent to **(A - B) (C – D)** or **A - B | C - D** is equivalent to **(A - B) | (C - D)**
A B	concatenation. The character sequence meeting rule A followed by characters that meet rule B. The sequence has higher priority than all lists. So:

	A B , C D is equivalent to **(A B) , (C D)**
A | B	selection. The sequence of characters meets rule A or rule B.

2.12.7 Comments and whitespaces

Anywhere between terminal symbols and rule names may be any sequence of spaces, new lines and tabulators, and comments.

The comment is a text between "/*" and "*/". Nesting of comments is not allowed.

2.12.8 Implemented predefined rules

The following implemented methods provide parsing of the actual source

$anyChar	parses any character (returns true if a character exists and false if the parser reached the end of the parsed string).
$base64	parses base64 format. Parsed text is put into the internal stack as an array of bytes.
$boolean	parses "true" or "false". Parsed text is put to the internal stack as a Boolean value.
$date	parses date according to ISO specification. Parsed text is put to the internal stack as an org.xdef.sys.SDatetime value.
$datetime	parses date and time according to ISO specification (argument may be a mask). Parsed text is put to the internal stack as an org.xdef.sys.SDatetime value.
$datetime(mask)	parses the date and time according to the mask in the argument. Parsed text is put to the internal stack as an org.xdef.sys.SDatetime value.
$day	parses day according to ISO specification. Parsed text is put to the internal stack as an org.xdef.sys.SDatetime value.
$digit	parses decimal digit.
$duration	parses duration according to ISO specification. Parsed text is put to the internal stack as an org.xdef.sys.SDuration value.
$error(s)	writes the error message with the parameter to the reporter and returns that the rule fails.
$eos	checks if the end of the source was reached.
$find(s)	skips characters from the actual source position until it reaches a string from the argument s (it fails if the string from the argument was not found)
$findOneOfChars(s)	skips the position from the actual position to a character from the string from the argument (returns true if the character was found and false if not)
$float	parses float number without sign (with a decimal point and/or exponent). Parsed text is put to the internal stack as a java.lang.Double value.
$hexData	parses hexadecimal format. Parsed text is put into the internal stack as an array of bytes.
$integer	parses integer number without sign (sequence of digits). Parsed text is put to the internal stack as a java.lang.Long value.
$JavaName	parses Java name. Parsed text is put to the internal stack as a java.lang.String value.
$JavaQName	parses Java qualified name (may contain dots). Parsed text is put to the internal stack as a java.lang.String value.
$letter	parses letter
$letterOrDigit	parses letter or digit
$lowercaseLetter	parses lowercase letter
$month	parses month according to ISO specification. Parsed text is put to the internal stack as an org.xdef.sys.SDatetime value.
$monthDay	parses month and day according to ISO specifications. Parsed text is put to the internal stack as an org.xdef.sys.SDatetime value.
$ncName	parses NCName according to W3C specification. Parsed text is put to the internal stack as a java.lang.String value.

$nmToken	parses NMTOKEN according to W3C specification. Parsed text is put to the internal stack as a java.lang.String value.
$time	parses time according to ISO specifications. Parsed text is put to the internal stack as an org.xdef.sys.SDatetime value.
$skipToNextLine	skips to the next line of parsed data or the end of data.
$stop	parsing is stopped at the position of the rule.
$stop(s)	parsing is stopped at the position of the rule and the argument is stored in the internal stack.
$UTFChar	parses any legal UTF character
$uppercaseLetter	parses a capital letter.
$xmlChar	parses an XML character according to W3C specification
$xmlName	parses an XML name according to W3C specifications. Parsed text is put to the internal stack as a java.lang.String value.
$xmlNameExtchar	parses the following characters of XML name according to W3C specification
$xmlNamestartchar	parses the first character of XML name according to W3C specification
$year	parses a year according to ISO specification. Parsed text is put to the internal stack as an org.xdef.sys.SDatetime value.
$yearMonth	parses a year and month according to ISO specifications. Parsed text is put to the internal stack as an org.xdef.sys.SDatetime value.
$whitespace	parses whitespace according to W3C specification

Note the "rule" $error doesn't parse any actual text. However, it forces the parsing process to fail in the actual position.

2.12.9 Implemented methods for handling the internal stack

The following methods (nothing is parsed) are implemented to handle the internal stack:

$clear	clears the internal stack
$info	puts to the internal stack the information containing the name of the actual rule and source position (line, column, etc.)
$info(params)	puts to the internal stack the information containing the name of the actual rule and parameter list in parenthesis and source position (line, column, etc.)
$pop	removes the item from the top of the internal stack
$push	puts to the internal stack the text parsed by the actual rule.
$push(arg)	puts a value from the argument to the top of the internal stack (the parameter can be specified in the declaration section). If no argument is specified it is pushed to the internal stack of the parsed text.
$rule	puts to the internal stack the name of the actual rule. After the rule name are positions where parsing of the rule started parsing and where parsing ended (separated by the space).

2.12.10 Declaration of externally defined method

In the BNF grammar of X-definition, it is possible to apply an external rule, which is implemented in a Java class. The name of the external rule in BNF starts with the character "$" (dollar) after which follows the specification of an external Java method.

The external rules are declared in the command starting with "%define". This keyword must be followed by the name of the external rule, the colon character (":"), and the specification of a Java method. If the external method has parameters, the list of values separated by a comma is specified in brackets. The values of parameters may be only an integer, float, String, Datetime, or Duration.

The specification of all external or implemented methods must be described at the beginning (before the description of BNF rules).

Example:

```
%define $rule1: $myproject.BNFPravidla.pravidlo1
%define $rule2: $myproject.BNFPravidla.pravidlo(123, "abc")
%define $rule3: $myproject.BNFPravidla.pravidlo(-1)
%define $operator: $push("op")
%define $date: $datetime("dd.MM.yyyy")

Rule1 ::= $rule1 | $rule2 | $rule3
...
```

2.12.11 BNF declaration

The simplest way to create an object with a BNF grammar is to create it as an X-script variable:

```
    BNFGrammar x = new BNFGrammar(string with BNF grammar...);
```

Since the BNF grammar may be quite big, it can be also recorded in the auxiliary element (it creates a declared variable similar way as xd:declaration):

```
<xd:BNFGrammar name = "x" scope = "global" >
   Text with BNF grammar ...
</xd:BNFGrammar>
```

From the given grammar you can create a new grammar extended by other rules by using the attribute "extends". E.g.:

```
<xd:BNFGrammar name = "y" extends = "x" , scope="local">
   Rules extending grammar x ...
</xd:BNFGrammar>
```

It is also possible to create extended grammar with a constructor:

```
  BNFGrammar g = new BNFGrammar("numbers ::= [0-9]+ ( ',' [0-9]+ )*");
    …
  /* g1 is g extended with the rule „hexa“ */
  BNFGrammar g1 = new BNFGrammar("hexa ::= ('X' | 'x') [0-9A-F]+", g);
```

The rule from a grammar can be obtained by the method "rule" on a BNFGrammar. E.g.:

```
   BNFRule x = g1.rule("hexa");
```

You can use a BNF rule to parse the text value of an attribute or a text node:

```
   <elem a="optional x">
     required g1.rule("numbers");
   </elem>
```

The attribute "a" of the element <elem> must meet the rule "hexa" from the grammar g1 and the value of the text node of this element must meet the rule "numbers" from g.

To check if a string meets a BNF rule you can use the method "check". The result is true if the string meets the rule and false if not:

```
   BNFGramar g = new BNFGramar("A ::= [0-9]+(',' [0-9]+)*");
   BNFRule r = g.rule("A");
   String s = "123,4,5";
   boolean x = r.check(s);
```

the variable "x" will be true.

You can also declare the validation method based on the BNFrule as a validation method:

```
   <xd:declaration>
     BNFGramar g = new BNFGramar("A ::= [0-9]+(',' [0-9]+)*");
     type numbers g.rule("A");
   </xd:declaration>
   ...
   <A a="required numbers" />
```

You can try BNF declaration on the web at: https://xdef.syntea.cz/tutorial/examples/BNF.html

2.13 Macro (xd:macro)

Macros are used to simplify and help clarify the X-script for the easier overall maintenance of the source of the X-script in X-definitions. Macro specifies a value (a character string), and has an assigned name in the attribute "name".

The macro is declared using the element <xd:macro>, which must be placed on the level of the direct descendants of the X-definition. The name of the macro is written to the attribute "name" (or "xd: name"), which must be the unique name within an X-definition (i.e., there cannot be two macros with the same name). The value of the macro is recorded as a text value of the element xd:macro. A reference to a macro ("call" of a macro) is written in the X-script ax a "${name}". All references to the macro where ever they occur in an X-script are replaced with the value of the referred macro.

Macros can also have parameters (each parameter has a name). The value of the declared parameter must be set in the macro declaration using attributes that have the name of the parameter. A reference to the parameter of a macro in the body of the macro is written as "#{paramName}". In the macro reference, the parameter is specified in parentheses, and each parameter is referred to with the parameter name. If a parameter is not specified in the macro reference the default value from the macro declaration will be set. Example of macro reference with parameters:

```
${macroName(parName='value')}
```

Note: Single or double quotes inside a parameter value must be escaped using ' \ '

All macros are processed before the X-script is compiled (this is executed by the macro preprocessor). All macro references are replaced with the expansions of a macro. This is done until a macro reference exists in the expanded text, i.e. macro references may be nested. Also, the macro declaration may be a reference to another macro. The number of nested macro calls is limited to a fixed value of 100 (this avoids an endless loop of nested macros and if this limit is exceeded, an error is reported). Macro reference can be recorded anywhere in the X-script, even inside the values of constants, keywords, or identifiers. For this reason, you should be aware of the possibility of inadvertent call macros e.g. inside the declaration of character strings. If you do not want an entry to be interpreted as a macro reference, it is necessary to replace the character "$" by writing "\u0024". Example:

```
outln("This is not a macro reference: \u0024{name}");
```

If the X-script contains a reference to the macro from another X-definition, the name must be introduced by the name of the referred X-definition followed by the "#" symbol (normally the scope of macro validity is limited to the X-definition where it was declared):

```
outln("However, this is the macro reference: ${name#name}");
```

The result of the macro processing is in the X-script copied including spaces and newlines. So, with a macro, it is possible to insert new rows into the models of attributes.

Examples of a macro declaration:

```
<xd:macro name = "name">string(2,30)</xd:macro>
<xd:macro name="colors" p1="white" p2="black">enum('#{p1}', '#{p2}')</xd:macro>
<xd:macro name="familyName">required ${name}</xd:macro>
<xd:macro name="greeting" p="'Hello'">outln(#{p});<xd:macro>
<xd:macro name="text">outln('Macro call has the form: \u0024{text} ');"
```

Examples of macro reference:

```
<Person firstName="optional ${name}" lastName="${familyName}"/>
<Cover title="required ${colors(p2='red')}"/>
<Description print="optional ${colours)" />
<Greeting xd:script="finally ${greeting}">
<Greeting xd:script="finally ${ greeting p='\'Hi'\'}" />
<t xd:script="finally ${text}"></t>
```

The results after macro expansion:

```
<Person firstName = "optional string(2,30)" lastName = "required string(2,30)" />
<Cover print="required enum('white','red')"/>
<Description print="optional enum('white','black')"/>
<Greeting xd:script="finally outln('Hello');"/>
<Greeting xd:script="finally outln('Hi');"/>
<t xd:script = "finally outln('Macro call has the form: ${text}'); "/>
```

2.14 X-definition Header

X-definition is an XML element named "xd:def" where "xd" is the prefix of the namespace "`http://www.xdef.org/xdef/4.2`" (of course you can use another prefix; however, in this text, we use the prefix "xd"). The list of attributes of the X-definition element we call the X-definition **header**. There are some obligatory attributes of the header and some optional attributes. The header contains the information needed for the processing of an X-definition.

2.14.1 xd:name

One X-definition in the project may be unnamed, and the other X-definitions compiled in a project (XDPool) must have an unambiguous name. The name of the X-definition is specified in the **X-definition header** in the attribute "`xd:name`" or just "`name`" and it must be in the form of a valid XML name.

2.14.2 xd:root

Because a single X-definition may include several element models, it is necessary to specify which model might represent the description of the root element of the processed data. Therefore, it is necessary to specify the names of those models used as root elements in the attribute "xd:root" or just "root". If other elements might be root elements, the names of the corresponding acceptable models are separated by the character "|". You can describe several models that can be accepted as the root.

In version 4.1 and above it is possible to refer from the xd:root attribute not only to a model of an element but also to an xd:choice group. Example:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/4.2" xd:root = "x">
  <xd:choice name = "x">
    <A/>
    <B/>
  </xd:choice>
</xd:def>
```

Note that it is possible to use also “match” section in the script of the element:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/4.2" xd:root = "x">
  <xd:choice name = "x">
    <A xd:script= "match @a == '1'" a="int()"> <B/> </A>
    <A xd:script= "match @a == '2'" a="int()"> <C/> </A>
  </xd:choice>
</xd:def>
```

2.14.3 X-script options in X-definition header

The X-script in the header of the X-definition may contain the actions "init", "onIllegalRoot", "onXmlError" and the specification of options (see 4.1.22 Options). Here are the options you can specify:

noSetAttrCase	**setAttrLowerCase**	**setAttrUpperCase**
noSetTextCase	**setTextLowerCase**	**setTextUpperCase**
ignoreAttrWhiteSpaces	**preserveAttrWhiteSpaces**	**acceptEmptyAttributes**
ignoreTextWhiteSpaces	**preserveTextWhiteSpaces**	
noTrimAttr	**trimAttr**	
noTrimText	**trimText**	
ignoreComments	**preserveComments**	
ignoreEmptyAttributes	**preserveEmptyAttributes**	

If an option is not specified the default values are:

noSetAttrCase
noSetTextCase
preserveAttrWhiteSpaces
preserveTextWhiteSpaces
trimAttr
trimText
ignoreComments
preserveEmptyAttributes

2.14.4 Specification of meta namespace

For the X-definition itself to be described, it is possible to specify the attribute "xd:metaNamespace" from the X-definition namespace, which specifies the namespace, which will be interpreted as the namespace for X-definition objects. Also, it is possible to refer to these objects.

Example:

```
<meta:def xmlns:meta       ="mynamespace"
          xmlns:xd         ="http://www.xdef.org/xdef/4.2"
          name             ="dummy"
          xd:metaNamespace ="mynamespace" >

  <xd:def xd:name ="required string()">
    <meta:any meta:script="optional; options moreAttributes, moreElements, moreText" />
  </xd:def>
</meta:def>
```

2.14.5 xd:include

The "xd:include" attribute contains the list of URL items or pathnames of files with the collections of X-definitions that are imported to the project. The separator of the entries in the list is the comma (","). The item may be expressed as the relative path from the actual position of the X-definition. In the item, it is possible to use wildcards "*" or "?" (i.e. "*.xdef" meets all files with the extension "xdef". If an item does not refer to a local filesystem the wildcard is not allowed.

Example:

```
<xd:def xmlns:xd   = "http://www.xdef.org/xdef/4.2"
        xd:name    = "foo"
        xd:include = "http://www.syntea.cz/project/*.xdef">
```

2.14.6 xd:importLocal

This attribute contains a comma-separated list of names of X-definitions from which are “visible” for the local declarations from those X-definitions. The X-definition without a name is written as “#”. The order defines how the X-declarations are searched. Example:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2" xd:name="A" xd:root="X" xd:importLocal="B, #">
  <X a="mytype()" b="_type()"/>
</xd:def>
<xd:def xmlns:xd = "http://www.xdef.org/xdef/4.2" xd:name = "B">
  <xd:declaration scope="local"> type mytype string(); </xd:declaration>
</xd:def>
<xd:def xmlns:xd = "http://www.xdef.org/xdef/4.2">
  <xd:declaration scope="local"> type _type int(); </xd:declaration>
</xd:def>
```

The type of attribute “a” in the element “X” will be “string” and the type of attribute “b” will be “int”.

2.14.7 Implementation information

If a name of an attribute in the X-definition header starts with the prefix "impl-", then the value of such is stored in the compiled pool and it is available in the X-script by the method getImplProperty(name) where the name is part of the name of the attribute which follows the prefix "impl-".

Example:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/4.2" root = "A" impl-version = "001.002">
  <A xd:script="init outln('The version of this project is: ' + getImplProperty('version'))"/>
</xd:def>
```

The processor writes to standard output:

```
The version of this project is: 001.002
```

2.15 Declaration of variables, methods, and data types (xd:declaration)

The variables, methods, and data types may be declared in the element "<xd:declaration>" that is a direct descendant of an X-definition. To specify the scope of accessibility (or say, " visibility") of declared objects is possible using the attribute "xd:scope". If the value is "global", the declared variables, methods, and data types can be accessed from any point of an X-script in an X-definition. However, if the given attribute is xd:scope=“local”, the declared variables, methods, and data types are "visible" only from the X-definition in which they were declared or from X-definitions where is written the attribute “xd:importLocal” with the name of this X-definition. The attribute "scope" is optional and the default value is "local". For a detailed description of the declaration section, see paragraphs 4.1.6.1 Declared Variables, 4.1.6.2 Declared methods, and 4.1.6.3 Declared data type.

2.16 Structure of X-definition

X-definition is an element that starts with the X-definition header and contains child elements, which are a mixture of:

- macros
- element models
- group models
- attribute models
- arbitrary element models
- text models
- declarations of variables, methods, and types
- BNF grammar specifications
- X-component descriptions (not described yet, will be described later)
- JSON model

Any above elements are optional, any number of them. All of them have a unique name within the X-definition, and their sequence is arbitrary.

2.17 Collection of X-definitions

Larger projects may use many X-definitions. In such a case, we can create a **collection of X-definitions.** The collection of definitions – "xd:collection" -- is the parent node into which you can insert X-definitions as child nodes, or you can specify the path to the source where the X-definitions are located.

In the following example, we separated the description of the model "Person" and "Address" into two different X-definitions named "CommonObjects" and "Family". Both X-definitions have been recorded in a single XML document as a collection:

```
<xd:collection xmlns:xd = "http://www.xdef.org/xdef/4.2">

 <!-- X-definition with model "Family" -->
 <xd:def xd:name = "Family" xd:root = "Family" >
   <Family>
     <Father  xd:script = "ref CommonObjects#Person; occurs 0..1" />
     <Mother  xd:script = "ref CommonObjects#Person; occurs 1" />
     <xd:mixed>
        <Son      xd:script = "ref CommonObjects#Person; occurs 0..15" />
        <Daughter xd:script = "ref CommonObjects#Person; occurs 0..15" />
     </xd:mixed>
     <Residence xd:script = "ref CommonObjects#Address; occurs 1" />
   </Family>
 </xd:def>

 <!-- X-definition with models Person and Address -->
 <xd:def xd:name = "CommonObjects" >
   <Person GivenName  = "required string (2,30)"
           FamilyName = "required string (2,30)"
           BirthDate  = "required date"
           Salary     = "optional int(1000,9999); onFalse error('Incorrect salary')" />
   <Address Street = "optional string (2,36)"
            Number = "required string (1,6)"
            Town   = "required string (2,36)"
            Zip    = "required int(10000,99999)" />
 </xd:def>
</xd:collection>
```

The collection may only have the optional attribute xd:include. E.g.:

```
<!-- the source codes of X-definitions are imported from files specified in "xd:include" -->
<xd:collection xmlns:xd = "http://www.xdef.org/xdef/4.2"
   xd:include = "C:/data/xdefA.xdef, C:/common/*.xdef"/>
```

3 Language localization of XML (xd:lexicon)

You can describe the names of elements and names for different languages in the element xd:lexicon. Each line in the text of this element describes an X-position if the node and separated by equal sing (“=”) follows the name in a language. The language name is specified by the attribute xd:language of the element xd:lexicon. Example:

Let’s have the following XML document:

```
<Contract ID = "12345">
   <Date>2019-05-12</Date>
   <Client name = "John Smith" PersonalId = "0987654321"/>
</Contract>
```

X-definition:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/4.2" xd:root = "Contract" xd:name = “Example”>
   <Contract ID="int()">
      <Date>date()</Date>
      <Client name="string()" PersonalId="num()"/>
   </Contract>
</xd:def>
```

Lexicon for the Russian language:

```
<xd:lexicon xmlns:xd="http://www.xdef.org/xdef/4.2" language="rus" >
  Example#Contract                    =  контракт
  Example#Contract/@ID                =  ид
  Example#Contract/Date               =  Дата
  Example#Contract/Client             =  клиент
  Example#Contract/Client/@Name       =  имя
  Example#Contract/Client/@PersonalID =  Персонномер
</xd:lexicon>
```

Then the version in the Russian language will be:

```
<контракт ид = "12345">
  <Дата>2019-05-12</Дата >
  <клиент имя = "John Smith" Персонномер = "098765432" />
</контракт>
```

You can define a language where the names are the same as in the X-definition by the attribute “xd:default”. E.g. English:

```
<xd:lexicon xmlns:xd = "http://www.xdef.org/xdef/4.2" language = "eng" xd:default = "true" >
```

4 X-script of X-definition

The X-script of X-definition (hereafter the **X-script**) is the language used for the description of properties of data objects (both models and their child nodes). The X-script is written either as a value of the auxiliary attribute ″xd:script″ as a value of the attributes of models or as a description of a text node. The attribute xd:script may also be specified in the X-definition header, where it may describe several properties of the X-definition. With the X-script we describe the properties of data and the actions which are invoked in different events when an X-definition is processed.

The X-script has a free format (the syntax tokens of the X-script may be separated by an unlimited number of white spaces). There are several parts of the X-script (we are speaking about the **sections of the X-script**). The parts of the X-script may be specified in an arbitrary order. The separator of different sections of the X-script is the character "`;`" (semicolon). In the case where it is not necessary concerning the syntax of the X-script, the semicolon may be omitted (at the end of the X-script, after a compound statement in curly brackets). Generally, there are the following sections of the X-script:

1. **Validation section:** The validation formula of the X-script is different for the description of text values and elements.
    a. The text values are described by the specification of the occurrence (required, optional, etc.) followed by the specification of the validation of the text value. The second part here may be omitted (then it is considered as any kind of string, including an empty one).
    b. The occurrence of an element is specified by two numbers the interval of the minimal number of occurrences and the maximal number.

    Note that X-definition does not use the concept "type of element" which is used in XML schema. The type of text value of an element is described in the X-definition by the element text value.
2. **Sections describing actions associated with events:** Specification of action defines what to do in different events (states) during the processing of objects. Each specification of action starts with the keyword, which is the name of an event, followed by the command (it may be also a compound command in curly brackets) which should be invoked in the relevant event. The names of events are found in Table 5 - Events. You can specify more actions in a single X-script.
3. **Options:** The specification of options starts with the keyword "options" followed by the list of option names separated by a comma. See Table 38 - Options. The X-script can only specify one option list.
4. **References:** With reference, we specify the link to the model of an element. This model must exist in the X-definition. Within one X-script you can only refer to one model. With references, you can also simplify and maintain designed X-definitions more easily. If you refer to the model "address" in the example below, you describe it once, and if you make changes to your model "address" it is changed automatically in objects where there is a reference.
5. **Declaration of variables of an element**: this section declares variables connected with the instance of an element. The section starts with the keyword "var". This section must be specified before the other sections.

Within the X-script you can insert comments, (similar to C or Java), between the character sequence "/*" and "*/".

Warning: *Line comments used in C or Java that start with the sequence "//" is not allowed because the new line characters (when used in attributes) are replaced by the XML processor with single-space characters.*

The sequence of the above-mentioned parts of the X-script is arbitrary except for the declaration of variables, which must be specified before other sections.

The X-script of elements is recorded in the auxiliary attribute xd:script.

```
<Family>
     <Father  xd:script = "occurs 0..1"/>
     <Mother  xd:script = "occurs 1"/>
     <Child   xd:script = "occurs 0.."/>
</Family>
```

The X-script describing a text value is written as the text in the appropriate place of a given element:

```
<Text>
   required string ()
</Text>
```

Example of an X-script with a reference:

```
<Stay xd:script = "occurs 1; ref Address"/>
```

4.1.1 Identifiers in X-script

In the X-script identifiers are used for names of X-definitions, methods, variables, events, keywords, etc. The identifier in the X-script corresponds to the form of the XML QName. Moreover, for names of methods, variables, or constants it is also possible to use the character “$" (dollar) in the identifier. Capital and small letters are distinguished. Letters of the national alphabet are permitted. The names of XML elements and attributes, of course, comply with the rules for XML objects (e.g. they may contain the characters ".", "-", ":", IE. dot, dash, and colon). We do not recommend the use of identifiers with periods, colons, and the character "-" (even when it is not disabled, ambiguous entries may occur).

4.1.2 Types of values of variables and expressions in X-script

In the X-script commands and variables with values of several types can occur. Value types that can occur in the X-script are as follows:

4.1.2.1 int (integer numbers)

Values are in the range:

```
-9223372036854775808 <= n <= 9223372036854775807
```

Note that the int type in the X-script corresponds to the long type in Java, C #, C, etc. (it is represented as a 64-bit integer). Therefore, the long type is not implemented in the X-script.

Whole numbers in X-script commands can be written either as a decimal number or as a hexadecimal number. Hexadecimal numbers must begin with the characters "0x" or "0x" followed by a sequence of hexadecimal digits (i.e., the letters 'a' to 'f' or 'A' to 'F' or digits '0' to '9').

The special predefined constants ($MAXINT, $MININT) are described in Table 3 - Built-in variables and constants.

To make an entry easy to read it is possible to insert between the digits the characters "_" (underscore), which does not affect the value of a number. For example, 123_456_789 is equivalent to 123456789.

To convert a number to a character string, it is possible to use the method "toString(mask)" where the argument is the output format mask, which represents a string of characters that contains control characters, which have the following meaning:

- `0` digits, leading zeros are replaced by a space
- `#` digits, leading zeros are appended to the output
- `.` creates an output with a decimal point (period)
- `'` prefix and suffix of the string that contains control characters, which are to be interpreted as characters fill (i.e., the character string is enclosed in single quotation marks).

Other characters make up the padding that is copied to the output string.

Example:

```
"012" matches the pattern "# #0".
```

4.1.2.2 float (floating-point numbers)

Values are in the range:

```
-1.7976931348623157E308 <= x <= -4.9E-324
```

or `0.0` or

```
from 4.9 E-324 to 1.7976931348623157E308
```

The specification of numbers with a floating-point corresponds to the commonly used format of floating-point numbers, including the exponent. The decimal point is always a dot (regardless of local or national conventions). The exponent can be written as the capital or the small letter "e". To convert a number to a character string, it is possible to use the method "toString (mask)", where the argument is the string with an output format mask, i.e., a string with control characters, which have the following meaning:

0	digits, leading zeros are replaced by a space

#	digits, leading zeros are appended to the output
E	separates mantissa and exponent. Creates output with a decimal point (period)

or the prefix and suffix of the string that contains control characters but is to be interpreted as padding (i.e., the generic character string enclosed in single quotation marks or apostrophes)

Other characters make up the padding that is copied to the output string.

The special predefined constants ($MINFLOAT, $MAXFLOAT, $PI, $E, $POSITIVEINFINITY, $NEGATIVEINFINITY, and $NaN) are described in Table 3 - Built-in variables and constants.

Examples:

```
"012.00" matches the pattern "##0.00".
"654.32" matches the pattern "##0.00".
"012.00" matches the pattern "##0.00".
"4" matches the pattern "# #0".
```

4.1.2.3 Decimal (decimal numbers)

The decimal numbers in the X-script are implemented as java objects java.math.BigDecimal. This number type starts with the character "0d" followed by an integer number or number with a floating point. In writing it is possible to use an underscore, e.g. "0d123__456_890_999_0.00_333". Values of the type Decimal are only possible to compare in expressions. Other operations must be carried out using the appropriate methods.

4.1.2.4 BigInteger (any integer number)

The BigInteger numbers in the X-script are implemented as java objects java.math.BigInteger. This number type starts with the character "0i" followed by an integer number. In writing it is possible to use an underscore, e.g. "0i123__456_890_999_000_333". Values of the type BigInteger are only possible to compare in expressions. Other operations must be carried out using the appropriate methods.

4.1.2.5 String (character strings)

Character strings can contain any characters that are acceptable in XML documents. Strings literals are written with apostrophes, or quotation marks around them (because the values of XML attributes may also be inside quotation marks or apostrophes, you should use another character inside an attribute, that is, if the attribute value is enclosed in quotation marks, inserting character values between apostrophes, and vice versa). If a character occurs within the string, which is a string (i.e. apostrophe or quotation marks), enter the backslash character \ before the apostrophe '. The occurrence of the character '\' is written doubled as '\\'. Using the character '\' can also describe any Unicode character 16 writing "\uxxxx" where x is a hexadecimal digit. You can also write some special characters by using the following escape characters:

\n	end of line (linefeed, LF, \u000a)
\r	return to the beginning of the lines-carriage return (CR, \u000d)
\t	horizontal tab (HT, \u0009)
\f	form feed (FF, \u000c)
\b	backspace (BS, \u0008)
\\	backslash ("\", \u005c)

Warning: If the text of X-script is specified as the attribute value, the XML processor replaces all occurrences of the new line with space. Therefore, you must write into the X-script in attributes to explicitly specify character strings new lines such as "\n". Additionally, you should avoid accidentally calling a macro. The occurrence of the pair of characters "$ {" anywhere in the X-script is interpreted as the beginning of a macro reference, and therefore it should be inside the character strings. In this case, write the initial character "$" for this pair of characters by the escape sequence such as "\u0024".

4.1.2.6 Datetime (date and time values)

The value represents a date and time. Contain year, month, and day. It can be written to the constructor as a string of characters according to ISO8601 or can be converted to the internal shape by using the implemented method "parseDate": e.g. parse date ('2004-08-10T13:59:05'). The recommended format is according to ISO8601, otherwise, the function "parseDate" must be given a string as the second parameter with a mask, specifying the format of the registration. Similarly, you can use the mask as a parameter for the method "toString", e.g. date.toString("d. m. yyyy"). The mask is a string of characters that contains control characters used for the processing of input data or creating a printable string from the data object (formatting). Other characters in the

mask are understood as a character constant (literal), IE. a copy is required at the input or output will be copied. If the literal contains letters, you must write it between apostrophes into the mask (if the apostrophe should be part of the literal, then that is doubled). If there is an escaped character, then when the formatting is completed the number of leading zeros and when parsing the processor reads the specified number of digits. For some control characters, there is a different meaning (see control characters ' a ', ' E ', ' G ', 'M ', ' y ', ' Z ', ' z '). Also, the format may contain the following sections:

a. Initialization section. The initialization section is enclosed in curly braces "{" and "}".

   Describes the country or language-dependent conventions (location) and may set the default values of a date or time. Description of the default values requires that each value was preceded by an escape character. In the initialization sections, only the following escape characters: d, M, y, H, m, s, S, z, Z are allowed. The zone name is specified after the character name "z" in parentheses. For example: "{d1M1y2005H16m0s0z(CET)}" sets the default date and time values to 1-1-2005T16:00:00 CET. In the parentheses, it is possible to write the full name of the place, e.g. "Europe/Prague".

   The description of the language-dependent or local conventions is given by the letter "L" followed by a language ID in parentheses and, if appropriate, a country may also be specified after the comma. After the next comma, the variant of local conventions may also be specified. L (*) sets the location according to the running operating system. The language and country identifiers are two-letter and must conform to the standards ISO639 and ISO3166 (the language in lowercase letters and the country in uppercase letters). E.g "L(en)" defines English. The default value is set to L(en,US). E.g. the "L(es,ES,Traditional_WIN) sets Spanish, Spain, traditional conventions.

b. Variant section. For parsing, it is advantageous to allow more variations of file formats. The different variants are separated by '|'. Each variant has its initialization part.

   Example: Mask d/M/yyyy|yyyy-M-d|{L'en'd MMM yyyy} allows you to read the data in the following formats: 1/3/1999 or 1999-0-1 or 1 Mar 1999. The variant has significance for parsing. In the process of output formatting, only the first option is used.

c. Optional section. A description of the optional section is enclosed in square brackets "[" and "]". The section specified as optional has meaning only when parsing and relevant data of the input data may be missing. Example: Mask for HH: mm [: ss] corresponds to the 13:31 or 13:31:05 data. Optional sections can be nested (for example. HH: mm [: ss [from]]). The optional section has meaning when parsing. When creating the string it is ignored.

d. Variant character. If the character sequence enclosed in apostrophes follows the character "?" in the mask then the parsing engine accepts a character equal to a character from the enclosed sequence (e.g. d?/.'m?/.'yyyy allows both forms of date, either "1/3/1999" or "1.3.19990").

e. Control characters of the mask. The parser and formatter of date values according to the mask are listed in the following table:

Table 1 - Control characters in the date mask

Character	Type	Description	Example:
a (and more)	Text	information about the part of the day (AM, PM; localized)	AM
D	Number	day of the year without leading zeros	4
DD (and more)	Number	day of the year with leading zeros	09
d	Number	day of the month (starts with 1)	5
dd (and more)	Number	day of the month with leading zeros (starts with 1)	05
E, EE, EEE	Text	abbreviated day of the week (localized)	Mon
EEEE (and more)	Text	full weekday name (localized)	Monday
e	Number	day of the week as a number (1 = Mon, 7 = Sun) without leading zeros	3
ee (and more)	Number	day of the week as a number (1 = Mon, 7 = Sun) with leading zero	03
G (and more)	Text	designation of the era (AD, BC; localized)	AD
H	Number	hours in the range of 0-23 without leading zeros	8

HH (and more)	Number	hours in the range of 0-23 with leading zeros	08
h	Number	hours in the range of 1-12 without leading zeros	9
hh (and more)	Number	hours in the range of 1-12 with leading zeros	09
k	Number	hours in the range of 0-11 without leading zeros	9
kk (and more)	Number	hours in the range of 0-11 with leading zeros	09
K	Number	hours in the range of 1-24 without leading zeros	9
KK (and more)	Number	hours in the range of 1-24 with leading zeros	09
M	Number	day of the year without leading zeros	6
MM	Number	day of the year with leading zeros	06
MMM	Text	abbreviated month name (localized)	Jan
MMMM (and more)	Text	full month name (localized)	January
m	number	number of minutes (without leading zeros)	1
mm (and more)	number	number of minutes (with leading zeros)	1

RR	number	year of (two digits e.g. in Oracle database). Century shall be supplemented by the following rules:
		If a RR is in the range of 00 - 49, then
		a) if the last two digits of the year are 00 - 49. then the first digits will be completed from the current century.
		b) if the last two digits of the year are 49 - 99. then the first digits will be completed from the current century decreased by one.
		If a RR is in the range of 50 - 99, then
		c) if the last two digits of the year are 00 - 99. then the first digits will be completed from the current century increased by one.
		d) are the last two digits of the year 49 - 99, then the first digit will be completed from the current century.
			1945, 2011

S (and more)	number	number of milliseconds	123
s	number	number of seconds (without leading zeros)	5
ss (and more)	number	number of seconds (with leading zeros)	05
YY	number	(**deprecated**) two digits, century part from the current date (can be used only in formatting mode)	20
y	number	year from a date	1848
yy	number	year in two digits form, which is interpreted so that the values of "01" to "99" are assigned the values and value of 1901 to 1999 and the value "00" is assigned the value 2000	
yyyy (and more)	number	year in four-digit (or more digits) form	1989
z	text	abbreviated name of the zone	CEST
zz (and more)	text	full name of the zone	Central European Sumer Time
Z	zone	zone in the form of ”+” or ”-” followed by HH: mm	+01:00
ZZ	zone	zone in the shape of ”+” or ”-” followed by HH: mm	+1:0
ZZZZZ	zone	zone in the form of ”+” or “-” followed by an HHmm	+ 0100
ZZZZZZ (and more)	zone	zone in the form of ”+” or ”-” followed by HH: mm	+01:00

4.1.2.7 boolean (Boolean values)

The boolean values may be used in the expressions, parameters of methods and the X-script commands similarly to in the Java language. The possible values are "true" and "false." Boolean values may be a result of the expression, comparing, etc. If in a Boolean expression occurs a reference to the attribute of the current element (recorded as "@" followed by a name of the attribute), then it automatically is converted to "true" if the attribute exists, and "false" if it does not exist. If in the Boolean expression occurs a ParseResult value, then it is "true" if the value was parsed without errors and "false" if an error was detected.

4.1.2.8 char (character)

The char value represents an utF-16 character. The char value has no constructor. It can be obtained from the string by the "charAt" method, or the result of the "char" validation method. The char value may be also the result of casting an int value. When adding to a string, a character is added to the end.

Examples:

```
char c1 = (char) 33; /* c1 has the value '! ' */
char c2 = "abc". charAt(1); /* c2 has the value of 'b' */
String s = "abc" + c2; /* s value is "abcb" */
```

The char validation method reads one character. If it is a backslash, it must be doubled, otherwise, it is used as an escape character ("\n", "\t", "\f" and possibly followed by a hexadecimal representation of the UTF-16 character in the form "\uxxxx").

4.1.2.9 Locale (information about the region)

This type contains information about language, country, and geographical, political, or cultural region. It may be used when the printable information is created from data values (number format, currency, date time format, etc). Value of this type can be created by the following constructors:

```
new Locale(language) or
new Locale(language, country) or
new Locale(language, country, variant)
```

where language is lowercase two-letter ISO-639 code, the country is uppercase two-letter ISO-3166 code, and the variant is vendor and browser-specific code.

4.1.2.10 Regex (Regular expression)

Objects of this type can be created with the constructor "new Regex(s)" where s is the string to the source of the shape of a regular expression. The regular expression matches the specification based on the XML schema.

4.1.2.11 RegexResult (a result of the regular expression)

Objects of this type are created as a result of the method "r.getMatcher(s)" where s is a string to be processed with the regular expression r.

4.1.2.12 Input/Output (stream)

The objects of this type are used to work with files and streams. Two variables with the Output value are automatically created: the "$stdOut" (writes to the java.lang.System.out) and "$stdErr" (writes to the java.lang.System.err) and one variable "$stdIn" of the type "InputStream" (reads from java.lang.System.in). The value of the variable "$stdOut" is automatically set to the methods of "out" and "outln" as the default parameter. "Similarly, the "$stdErr" value is used as the default output of the method "putReport".

4.1.2.13 Element (XML element)

The objects of this type are the X-script instances of “org.w3c.dom.Element”. They may be, for example, produced as the result of the method “getElement”.

4.1.2.14 Bytes (array of bytes)

This object can be the result of the “parseBase64” or "parseHex" methods. The constructor for an empty array of bytes is:

```
Bytes bb = new Bytes (10); /* Array of 10 bytes Assigned. */
```

For methods to work with arrays of bytes see 4.1.19 Implemented X-script methods.

4.1.2.15 NamedValue (named value)

The named value object is a pair consisting of the name and the assigned X-script value (any type of X-script). The name must match the XML name. You can create a named value by writing the beginning character "%" followed by the name, followed by an equal sign ("="), and then the specification of a value. For example:

```
NamedValue nv = %x:y, named-value = "ABC";
```

Note "x:y" is here the name of the named value "nv" and "ABC" is its value.

4.1.2.16 Container (sequence and/or map of values)

The objects of this type can be the result of certain methods (XPath, XQuery, etc.). The object Container contains two parts:

1. the part with named values (the mapped part, the entry is accessible by a name)
2. the part with a list of values (the sequential part, the entry is accessible by an index)

The empty Container can be created using the constructor Container c = new Container();

The value of type Container can also be specified in square brackets "[" and "]", where the list of values is written. The items are separated by a comma. The named values are stored in the mapped part and the not-named values are stored in the sequential part of the created container. For example:

```
Container c = [%a=1, %b=x, p, [y,z], "abc"];
```

The mapped part contains the named values "a" and "b". The sequential part is the list of the value of p, the next object is a Container, and the string "abc".

To work with the object "Container" you can use a variety of methods listed below (e.g. "toElement", see 4.1.19, Implemented X-script methods).

The container can occur in Boolean expressions (i.e., it may be in the "match" section or the "if" command, etc.). The value of a Container object is converted to the Boolean value according to the following rules:

1. When an object contains exactly one sequence item of type Boolean, then the result is the same as the value for this item.
2. If it contains exactly one sequence item of the type "int", "float" or "BigDecimal", then the result is true if the value of this entry is different from zero.
3. In all other cases, it is true, if the object is part of a nonempty sequence, otherwise, the result is false.

Note: The type of Container is also the result of expressions XPath or XQuery. If the XML node on which the expression is null, the return should be an empty Container.

4.1.2.17 Exception (program exception)

This object is passed when you capture an exception of the executed program (error) in the construction "try {...} catch (Exception ex) {...} ". The exception can be caused in the X-script with the "throw" command. An object of type “exception” is possible to create in the X-script using the constructor "new Exception(error message)".

4.1.2.18 Parser (the tool used to parse string value)

Objects of this type are mostly created when the X-definition is compiled. A parser is an object on which it is possible to invoke a validation method. The result of this method is a ParseResult object. The parser object is constructed when a validation method is specified in the X-script.

4.1.2.19 Parseresult (a result of parsing/validation)

Objects of this type are the results of a parser. If a ParseResult instance occurs in a boolean expression, it is converted to a boolean value, and it is true if errors are not reported, otherwise, the value is false (i.e., an automatic call of the method "matches ()").

4.1.2.20 Report (message)

This object represents a parameterized type and language-customizable message. We can create the message:

```
Report r = new Report ("MYREP001", "this is an error");
```

Alternatively, we can use, for example, the method "getLastError".

4.1.2.21 BNFGrammar (BNF grammar)

Objects of the type BNFGrammar are declared in the element "xd:BNFGrammar" (model) or it is possible to create them using the constructor. See 2.12 BNF grammar in X-definition.

4.1.2.22 BNFRule (BNF grammar rule)

The reference to a rule of BNF grammar. You can use the grammar rule, for example, to validate the text values of attributes or text nodes. The rule from the BNF grammar can be obtained by using the method “rule(ruleName)”.

4.1.2.23 EmailAddr (Email address)

The email address contains an email address, including a user name and an internet address.

Constructor:

```
EmailAddr(String email)
```

The string with the e-mail address must be according to the rfc822 specification.

Example:

```
EmailAddr addr = new  EmailAddr("(John Smith) j.smith@some-company.org"); // string with email address
```

The validation method is "emailAddr".

In XComponents, the type value is stored as a class instance: org.xdef.XDEmailAddr

4.1.2.24 GPSPosition (value containing GPS position)

The GPS position contains latitude, longitude, and altitude. Optionally contains the name of a location.

Constructors:

```
GPSPosition(float latitude, float longitude)                 // latitude, longitude
GPSPosition(float latitude, float longitude, String name)    // latitude, longitude, name of place
GPSPosition(float latitude, float longitude, float altitude) // latitude, longitude, altitude
GPSPosition(float latitude, float longitude, float altitude, String name) // latitude, longitude,
altitude, name
```

Where:

latitude the latitude	in degrees; the range of values is (-90.0 .. 90.0).
longitude	in degrees; the range of values is (-180.0 .. +180.0).
altitude	in meters; the range of values is (-6376500.0 .. unlimited). The value 6376500.0 is used in the calculations as the Earth's radius instead of space; any character string or null.

Methods over GPSPosition type:

latitude()	returns latitude
longitude()	returns the longitude
altitude()	returns altitude, if not specified it returns -6376499.9 (minimum value - the center of the planet Earth)
name()	returns the name of a location or null
distanceTo(GPSPosition pos)	returns the distance to the "pos" location in meters. The Earth radius used in the HaverSine formula is 6376500.0 m. The altitude is ignored in the calculation.

A type validation method is "gps()". The string format in text values is:

```
"latitude, longitude[, altitude[, name]]"
```

Where the values between brackets are: latitude (float number), longitude (float number), and altitude (float number). Name, if specified, must be either a string of letters or arbitrary characters in quotes, where the occurrence of quotes must be preceded by a backslash.

Examples of "gps" values:

```
48.2,  16.37  48.2, 16.37, Wien
48.97, 14.47, 381 48.97,
14.47, 381, "České Budějovice"
```

In the X-components, the type value is a class instance: org.xdef.sys.GPSPosition.

The longitude and optionally an altitude and/or name of a place. A GPSPosition can be obtained as a result of the validation method “gps” or by the constructor. Examples.:

```
GPSPosition x = new GPSPosition (50.08, 14.42); // latitude, longitude
GPSPosition x = new GPSPosition (50.08, 14.42, 399.0); // latitude, longitude, altitude
GPSPosition x = new GPSPosition (50.08, 14.42, “Prague”); // latitude, longitude, name of a place
GPSPosition x = new GPSPosition (50.08, 14.42, 399.0, “Prague”); // latitude, longitude, altitude, name
of a place
```

4.1.2.25 Price (price value containing a monetary amount in a given currency)

Constructor:

```
Price(BigDecimal amount, String code)
```

Example:

```
Price x = new Price(new BigDecimal(“3.25”), "USD");
```

Where the amount is a monetary value in the currency specified in the code parameter.

The value can be either float, integer, or Decimal. The parameter code is a string with three characters corresponding to the ISO 4217 specification (e.g. "USD").

Methods over Price type:

amount()	returns the value of the monetary amount as a decimal
currencyCode()	returns a string with the ISO 4217 currency code
fractionDigits()	returns an integer with the recommended number of decimal places for the currency according to ISO 4217, or -1 if not specified.
display()	returns a string with a value in ISO 4217 format.

The validation method is "price". Examples of values of type "price":

```
12 USD
0.45678 XAU
```

In X-components, the type value is stored as a class instance: en.syntea.xdef.sys.

4.1.2.26 Currency (value of Currency)

The Currency contains currency information.

Constructor:

```
new Currency(string with ISO 4217 currency code)
```

4.1.2.27 InetAddr (value containing IP address)

The InetAddr contains an IP address (IPv4 or IPv6).

Constructor:

```
new InetAddr(string with IP address)
```

4.1.2.28 URI (URI – universal relocation identifier)

The value URI contains an object with.URI identifier.

Constructor:

```
URI(String uri)
```

The validation method is "anyURI".

In X-components, the type value is stored as a class instance: java.net.URI

4.1.2.29 uniqueSet (set of unique items – table of rows)

This type is used to ensure the uniqueness of a **key** that defines items in the uniqueSet. The key may be composed of more **key parts** that together represent the key. The uniqueSet may be considered as a table of rows where each row contains key values (we can call it the key of a row). To this key may be also assigned specified values (i.e. the uniqueSet is a table with rows similar to a database table where the row contains a key and assigned

values). It is used in the conjunction with validation of the text values of attributes or values of text nodes. A row from the uniqueSet table contains a key part, which is unique within a table, and values are assigned to a key. Each key part and each value have a name and are declared in the uniqueSet specification. Values to the actual item may be set by assignment to the value. The key of the last row inserted into the table is possible to store to the uniqueSetKey object by the method getActualKey(). (see 4.1.14, Set of unique values (uniqueSet)).

4.1.2.30 uniqueSetKey (the key of a row from the uniqueSet table)

This type contains a key part from an actual valid row in a given uniqueSet table (note it may be obtained after a key was inserted (or found) in the uniqueSet table). It is possible to set the value to the actual key of the uniqueSet table by the method resetKey().

4.1.2.31 Service (database service; access to a database)

This object allows you to access services of different databases. Mostly it is passed to the X-definition from an external program. However, you can also create the Service object in the X-script:

```
Service connection = new Service (s1, s2, s3, s4);
```

The s1 parameter is the type of database (e.g. "jdbc"), s2 is the database URLs, s3 is the user name and finally, the password is s4.

4.1.2.32 Statement (database command)

The Statement object contains a prepared database command. It is possible to create it from the Service e.g. by the "prepareStatement(s)" method, where "s" is a string with a database command:

```
Statement stmt = connection.prepareStatement(s);
```

4.1.2.33 ResultSet (a result of a database command)

This object contains the result of the database command. In the case of a relational database, it is a table whose rows have named columns. It is possible to access the individual rows with the "next()" method. In the case of an XML database, the result depends on the command, e.g., it can be an object Container.

4.1.2.34 XmlOutStream (data channels used for continuous writing of XML objects to a stream)

This object type allows you to write large XML data, whose range could exceed the size of the computer's memory. This way of writing is often used in conjunction with the command "forget". The object can be created by the constructor "new XmlOutputStream(p1, p2, p3)". The parameter p1 is mandatory. It must match the path and the name of the file to which the writing is provided. The p2 parameter is the name of the character encoding table. The parameter p3 indicates whether to create the header of the XML document. Example of typical use in the X-script of X-definitions:

```
  XmlOutStream xstream = new XmlOutStream ("c:/data/file.xml", "UTF-8", true);
...
  XStream.writeElementStart();
  XStream.writeElement (); // write the whole child
...
  xstream.writeElement();
  xstream.writeElementEnd(); // write end of started element
...
  XStream.close();
```

4.1.3 Access to values from the processed document

In the X-script it is also possible to use values of attributes or text nodes obtained from a processed XML document, e.g. by using the method 'getText', 'getElement', 'getElementText' (see 4.1.19 Implemented X-script methods).

It is possible to write "@attributeName". If this entry appears in the expression of the type Boolean, then the value is true if the attribute with that name exists, otherwise, it is false. If the entry is listed in an expression, the result is the string value of the attribute or an empty string.

```
<A xd:script = "match (@a AND @b OR @c)" ...
```

The result of the section "match" will be true if in the element "A" are both attributes "a" and "b" or "c" attribute.

4.1.4 A local variable in X-script

In the X-script of X-definition, it is possible to declare local variables in a command block. The local variable is represented by a name (identifier). The local variable must have a specified value type it represents. The validity of a local variable is within the statement (i.e., "for") of the compound statement (i.e., between the curly brackets) in which it was declared.

```
for (int i=0, j=5; i LT j; i++ {
  int k; /* local variable k*/
  ...
}
```

4.1.5 Variables of the element model

In the X-script of an element, you can declare the variables that are valid (and therefore accessible) only at the time of the processing of a current element. To declare such variables, write them in the X-script section "var", which must be written at the beginning of the X-script:

```
<A xd:script="var int b=0, c=0; occurs *; finally outln('B=' + b + ', C=' + c)">
  <B xd:script="*; finally b++"/>
  <C xd:script="*; finally c++"/>
</A>
```

At the end of the processing of element A, the number of occurrences of elements (B) and (C) is displayed.

If you need to specify more declarative statements write them between curly brackets:

```
<A xd:script="var { int i=1; String s; }; *">
```

4.1.6 Declared objects

Declaration of variables, methods, and types is written in the element <xd:declaration> as the direct child descendant of an X-definition. The scope of accessibility of the declared objects may be specified in the optional attribute "xd:scope". Possible values are either "global" (this is the default value) and then all declared objects in this declaration are accessible from any X-definition from the project, or "local" and then the declared objects are accessible only from the X-definition where the specification is written.

4.1.6.1 Declared Variables

The declaration of a variable can be preceded by the qualifier "final" and "external".

The qualifier "external" indicates the value of a variable can be set externally before the process of X-definition was started (so the value of the variable is not initialized by the X-definition processor). Some values of the declared variables are released from the memory at the end of the process (database objects, streams, etc.). However, if a variable was declared as external then even those values are not released.

The qualifier "final" sets a variable to be constant and thus it is prevented from any further modification. To such objects, there must be assigned a value in the declaration statement (only if it was not declared as external - in such case, the assignment is done externally).

Example:

```
<xd:declaration xd:scope="global">
    external int globalVariable;
    final String const = "bla bla";
    external final extConst;
    int id, start = 0, end = 50;
     ...
</xd:declaration>
```

The variables not declared as final, (as well as Java class objects) are initialized by the default initial value. That is, the uninitialized global variables are set to the default values (zero for the numbers, false for the Boolean values, and null for the other objects).

All objects that are created in the X-script are, if necessary, automatically closed (by the method "`close`" immediately after completion of the X-definition process: i.e. that after returning control code to the Java program from which the process was called. However, if a variable was declared as external, then its closing is left to the programmer, even if the appropriate variable was set by an X-script command. E.g in the case of an object of the "Service" type the "close" method is generated only if the corresponding variable is not marked as "`external`".

The X-definition compiler reports an error for any attempt to assign a value to a variable marked as "final". However, if the variable is marked as "external" the initialization value can only be assigned externally. That is the variable can't be initialized in the declaration statement (nevertheless, the external variable can be also marked as "final" and it can't be changed in the X-script command).

4.1.6.2 Declared methods

Before a method, the name of the type of the result must be specified, followed by the name of the method, and followed by the declaration of a parameter list. The command block of the method that is recorded is in curly braces "{" and "}". The result value is passed to the method with the command "return". Methods for the validation of text values must return the ParseResult or the Boolean value true or false. The method specified for the "create" event in the elements must return e.g. a value of type "Container", for the "create" event in the attributes and text nodes it should return e.g. a value of type "String".

The formal parameter list of the declared methods is written in parentheses. The individual parameters are separated by a comma. Each parameter is written as a pair, consisting of the type of the value of a parameter and the name of the parameter. The parameter list can be empty, then it is written as "()". The executive commands are recorded as the command block in curly brackets "{" and "}". Parameters can be used in the command of the method in a similar way as the variables. The initial value is determined when you name the method. The names of the types of values that can be used as parameters of a declared method are listed in the following table:

Table 2 - Names of types of parameter values

Name of type	Description of value
`boolean`	Boolean value
`Datetime`	date and time
`float`	floating-point number
`int`	integer number
`String`	character string
`Regex`	regular expression
`RegexResult`	result of a regular expression
`Output`	output stream
`Input`	input stream
`Bytes`	array of bytes
`Container`	an array of objects of other types
`Exception`	exception (the parameter of the "catch" statement)
`Message`	Message
`BNFGramar`	BNF Grammar
`BNFRule`	BNF grammar rule
`XMLOutStream`	the stream used to write XML data

Note: Do not confuse the names of types with similarly named validation methods of the data types whose names may be different in capitalization.

In the following example, validation methods are declared. These methods must return a value of the ParseResult value or a Boolean value (see the command "return"). Note that if the body of a validation method returns the result, an appropriate error report can be also set (see the method "color"). E.g. the command "return error (' ... ')" returns false because the method "error" returns the value "false". The variable "today" contains a date and time of the start of the process. The method "getToday" returns the date in a given format. It is advisable to write the X-script declaration into the CDATA section (then you can write characters "<", ">", "&" without the XML entities):

```
<xd:declaration xd:scope="global">
<![CDATA[
Datetime today = now(); /*date and time of processing start */

/* would return date and time of processing start */
String GetToday() {
  return today.toString("dd. MM. yyyy HH: mm: ss");
}
```

```
/* Check the value with a name */
boolean name() {
   If (string (5,30)) {
      outln ('Name: ' + getText ());
      return true;
   }
   return error ('Error of name length: ' + length (getText ()));
}

/* Check the value with a color */
boolean color() {
   /* report error provides the validation method "enum", which also returns the string with value */
   if (enum('red', 'green', 'blue')) return true;
   return error("Incorrect color");
}

/* Value can be -1 or a number in the given interval */
boolean value(int min, int max) {
   if (!int()) {
      return error('Not numeric value');
   }
   i = parseInt(getText ());
   if (i == -1)
      return true;
   if (i< min)
      return error('Value is too small');
   else if (i > max)
      return error('Value is too big');
   return true;
}
]]>
</xd:declaration>
```

4.1.6.3 Declared data type

In some cases, it is appropriate to declare the data type. In the declaration section, the data type declaration starts with the keyword "type". It points to a validation method. Example:

```
<xd:declaration>
   type myType int(10, 20);
</xd:declaration>

<elem attr="required myType">
```

The value of the attribute "attr" shall comply in the same way as with the validation method int(10, 20).

If the user is not with those implemented types, he can define the method for a custom type. The result can be of type boolean or ParseResult. If the result is the value of type ParseResult it is possible to use the parsed value. Example with ParseResult:

```
<xd:declaration>
   ParseResult oddNumber() {
      ParseResult p = int ();
      If (p.intValue () % 2 == 0) {
         p.error("The number must be odd!");
      }
      return p;
   }
</xd:declaration>
<a a="oddNumber(); onTrue outln(getParsedValue() == 1)" />
```

Prints “true" if the attribute has a value of 1.

An example with Boolean is simpler. However, in the "onTrue" section you cannot work with the parsed value result (it will be, in this case, the same as the parsed string):

```
<xd:declaration>
   boolean oddNumber {
      return parseInt(getText ()) % 2! = 0? true: error("Number must be the odd!");
   }
</xd:declaration>
<a a="oddNumber(); onTrue outln(getParsedValue() == 1)" />
```

Here it always prints "false", because the value of the validation method is in this case a string, and it is never equal to the integer value 1.

4.1.7 Built-in variables and constants

In the X-script it is possible to use some predefined variables and constants:

Table 3 - Built-in variables and constants

Name	Type	Description
`$stdIn`	`Input`	standard input stream
`$stdErr`	`Output`	standard error protocol stream
`$stdOut`	`Output`	standard output stream
`$PI`	`Float`	constant π, the ratio of the circumference of a circle to its diameter (3.141592653589793) 3.14159265 ….)
`$E`	`Float`	constant with the number of the nearest e (the base of natural logarithms: 2.71828182….)
`$MAXINT`	`Int`	constant with the highest whole number (9223372036854775807, ie. 2^{63}-1)
`$MININT`	`Int`	constant with the highest whole number (-9223372036854775807, ie. -2^{63})
`$MAXFLOAT`	`Float`	constant with the largest floating-point number, applicable in the X-script
`$MINFLOAT`	`Float`	constant with the smallest floating-point number applicable in the X-script
`$NEGATIVEINFINITY`	`Float`	constant corresponding to negative infinity in floating-point operations
`$POSITIVEINFINITY`	`Float`	constant corresponding to positive infinity in floating-point operations

4.1.8 Expressions

In the X-script it is possible to obtain values as the result of expressions that are similar to the ones in other programming languages (Java, C, etc.). The detailed description of this goes beyond this text and the reader can familiarize themselves with it e.g. in the description of Java programming language. The result of the expression is always the value of any of the above types. An example of the use of the expression in the method parameter (regarding the value of the declared variable "max"):

```
<Product count = "required int(0, max + 100)" />
```

Because of the text of XML attributes and text nodes the characters "&", "<", ">" must be expressed using the predefined entities "&", "<", ">", there are defined keywords used as an alternative notation of the operators. That allows you to write the X-script so it is easier to read:

Table 4 - Alias keywords used as the alternative notation of operators

Operator	Alias	Meaning	Data types
		Binary operators:	
&	AND	Logical AND.	boolean, int
&&	AAND	Conditional logical AND.	boolean
|	OR	Logical OR.	boolean, int
||	OOR	Conditional logical OR.	boolean
<	LT	The relation is less than.	int, float, String, Datetime, Duration
>	GT	The relation is greater than.	int, float, String, Datetime, Duration
<=	LE	The relation is less or equal than.	int, float, String, Datetime, Duration
>=	GE	The relation is greater or equal than.	int, float, String, Datetime, Duration
==	EQ	The relation is “equals”.	*Any type*
!=	NE	The relation is “not equals".	*Any type*
<<	LSH	Left shift of integer.	Int

>>	RSH	The right shift of integer.	Int
>>>	RRSH	Binary zero-filled the right shift.	Int
%	MOD	Arithmetic modulus.	int, float
^	XOR	Logical or bitwise XOR“.	boolean, int
+	*Not exists*	Addition of numbers or the concatenation of strings.	int, float, String
-	*Not exists*	Subtraction.	int, float
*	*Not exists*	Multiplication.	int, float
/	*Not exists*	Division.	int, float
Unary operators:			
!	NOT	Logical NOT.	Boolean
~	NEG	Bitwise negation (of a number).	int
++	*Not exists*	Increment by 1.	int
--	*Not exists*	Decrement by 1.	int
Assignment operators:			
=	*Not exists*	Simple assignment. The left operand is set to the value of the right operand.	*Any type*
+=	*Not exists*	Add to the left operand the right operand.	int, float, String
-=	*Not exists*	Subtract from the left operand to the right operand	int, float
*=	*Not exists*	Multiply the left operand by the right operand.	int, float
/=	*Not exists*	Divide the left operand by the right operand.	int, float
%=	*MODEQ*	The left operand is the modulus of the left and the right operand.	int, float
<<=	*LSHEQ*	The left operand is bitwise shifted left by the right operand.	int
>>=	*RSHEQ*	The left operand is bitwise shifted right by the right operand.	int
>>>=	*RRSHEQ*	The left operand is bitwise right shift zero-filled by the right operand.	int
&=	*ANDEQ*	The left operand is bitwise or logical AND with the right operand.	int, boolean
^=	*XOREQ*	The left operand is bitwise or logical XOR with the right operand.	int, boolean
|=	*OREQ*	The left operand is bitwise or logical OR with the right operand.	int, boolean

Example:

```
x = p GE 125 AAND q LT 3;
```

is equivalent to:

```
x = p >= 125 && q < 3;
```

4.1.9 Events and actions

The specification of action always starts with the name of the event, followed by the command that performs the appropriate action.

Table 5 - Events

Event name	Description
`Create`	The action assigned to this event is performed only in the construction mode when the processor launches the new object from the X-definition (even before the event "init"). This action returns the value that is used for the construction of the corresponding object (an element, attribute, or text value). For the attributes and text nodes, it is expected a value from which is possible to create a text string. For elements, it is expected an object from which is possible to create it. E.g. it can be an XML element (name and ancestors of this element are insignificant for further processing). If no action is specified, then the current context is used (see **Chyba! Nenalezen zdroj odkazů. Chyba! Nenalezen zdroj odkazů.**). The result type of the expression must be one of the following: 1. null - then the item is not created 2. org.w3c.dom.Element or org.w3c.dom.NodeList if the action is specified in the X-script of Element. The elements are created according to the nodes from the list. 2. Container. If the action is specified in the X-script of Element. The elements are created according to the items from the sequential part of the Container. In the case of an attribute, the value of the named value of the Container with the same name is used. 3.StatementResult. If the action is specified in the X-script of Element, the elements are created from rows of the table. In the case of an attribute, it uses the value of the column with the same name (case insensitive) from the actual row. 4. String, if the action is part of the X-script of an attribute or a text node. In the case of an element, it is created if the string is not null. 5. integer number in the case of the X-script of an Element 6. boolean value in the case of the X-script of an Element 7. The other values are converted to a String value
`Default`	This keyword may be defined only in the X-script of models of attributes or text nodes. If the attribute or text value of the specified model does not exist, the string created from the associated action will be set. The event occurs after handling events onFalse, and onAbsence.
`Match`	The event "match" occurs before the further processing of an element or attribute. This action must return the Boolean value "true" or "false". If the value is "true", then processing continues if it is "false", then the current element or attribute or text node is not handled according to the model in which the action "match" is specified. An action in this event has available only the actual data of an XML document, i.e. the actual process of the element and its attributes (however, not yet processed by X-definition).
`Finally`	The event "finally" occurs at the end of the processing of an element according to the model. After that, delete from the memory happens only in the case of the action "forget".
`Forget`	The event of the action "forget" occurs at the end of the processing of the element (even after the event "finally"). This action is important in particular when processing large XML data that cannot be fully placed into computer memory. The "forget" action causes removal from the memory of the appropriate element after processing and after all actions (even after the event "finally") are completed. However, the symptoms of examinations and the occurrence of an element remain set. WARNING: the action "forget" (unlike the other events) is not inherited from referred objects, it is always necessary to specify in the respective element!
`Init`	The “init” action will be performed before further processing of an object. Automatic execution of certain functions can be set using the "options" (see ignoreAttrWhiteSpaces, trimAttr, ignoreTextWhiteSpaces, setAttrLowerCase, setAttrUpperCase, trimText, setTextLowerCase, setTextUpperCase). In the code of the "init" action of elements, you can access only attributes of the processed element since child nodes might not be available yet.
`onAbsence`	This event occurs when the minimum of the specified occurrence is not met (e.g. if the required object is missing). If no action is specified and a minimum condition is not met, an error message is recorded in a log file.

onExcess	The action of this event is performed when an element exceeds the upper limit of the specified maximum number of occurrences. If the action is not specified, an error message is recorded in a log file.
onFalse	The action of this event is performed if the result of the validation method returns an error. If the action is not specified, an error message is recorded in a log file.
onIllegalAttr	The action of this event is performed when an undefined or unauthorized attribute occurs. If the action is not specified, an error message is recorded in a log file.
onIllegalElement	The action of this event is performed when an undefined or unauthorized element occurs. If the action is not specified, an error message is recorded in a log file.
onIllegalText	The action of this event is performed when an undefined or unauthorized text node occurs. If the action is not specified, an error message is recorded in a log file.
onIllegalRoot	This event description is allowed only in the X-definition X-script. The action is performed when the element is not found in the list from the attribute "xd:root" in X-definition. If the action is not specified, an error message is recorded in a log file.
onStartElement	This event is performed after processing all attributes, but before the processing of the child nodes of the element.
onTrue	The action of this event is performed if the result of the validation method is not an error (parsing was OK). If no action is specified, the string from the parsed object is stored as a text value of the parsed attribute or text node.
onXmlError	The action of this event is allowed only in the X-definition X-script. The action is performed when the parser detects an error in the format of the source XML document. If the event is not specified, an error message is recorded in a log file or, if it is a serious error, further processing does not continue, and the program ends with an exception.

4.1.10 Quantifier (Specification of occurrence)

The description of objects in a model requires the specification of the limits of the occurrence of an object (the **quantifier**). Specifications of the quantifier of an element can be written in one of the following forms:

- occurs ? - the element may not occur or may occur once (same as "optional" or occurs 0 .. 1)
- occurs * - the element may not occur or the number of occurrences is not limited (same as 0 .. *)
- occurs + - the element must occur once or more times (the same thing as occurs 1 .. *)
- occurs m - the element must occur exactly m times (the same as occurs m..m)
- occurs m..n - the element must occur minimum m-times and may occur maximum n times
- occur n..* - the element must occur minimum n-times and may occur unlimited times
- required - the element must occur exactly once (same as occurs 1 or occurs 1..1)
- optional - the element may occur once or may miss (same occurs 0..1)

To specify the action of events related to the occurrence, there may be specified in the X-script sections "onExcess" and "onAbsence". The action "onExcess" is performed if the occurrence of the given object exceeds the maximum limit of occurrences. The action "onAbsence" is performed if the minimum number of occurrences has not been reached.

Note: for compatibility reasons, it is possible to skip the quantifier keyword "occurs". The specification "required" is the default, and it can be omitted.

The following specifications are equal:

```
a = 'occurs 1..1 string()'
a = '1 string()'
a = 'required string()'
a = 'string()'

b = 'occurs 0..1 string()'
b = 'optional string()'
b = '0..1 string()'
b = '? string()'

<c xd:string="occurs 1..*">
<c xd:string="1..*">
<c xd:string="occurs +">
<c xd:string="+">
...
```

4.1.11 Special quantifiers (ignore, illegal, fixed)

ignore - The node can occur unlimited times, but its incidence in processing is ignored and not set to the result data.

illegal - The element may not occur; the error is reported and its incidence in processing is ignored and not set to the result data.

fixed - The validation section of attributes and text nodes consists of a quantifier and a validation method. In the case the value is fixed it is possible to specify the keyword "fixed" and a value that must occur in the validated data, then the text node or attribute must have this value. If it is missing, the value is inserted into validated data. The specification of the quantifier is not allowed here.

Example

Example:

```
fixed '2.0'
```

is identical to:

```
required eq('2.0'); onAbsence setText('2.0')
```

Note as a value of "fixed" a variable can also be written:

```
<xd:declaration xd:scope="local"> String today = now().toString(); ... </xd:declaration>
...
fixed today
```

In some cases, it is appropriate to use a fixed value to also specify the data type of value. For example:

```
required float(); fixed '2.0'
```

4.1.12 Check the data type

The specification of a quantifier can be followed by a description of the validation method used to check the value of the attribute or text node. The result of the validation of the event is either a ParseResult object or a Boolean value. If the validation method is not described, the "string()" is used as default. If the option "ignoreEmptyAttributes" is specified, the attributes with empty strings are completely ignored.

A set of in-line functions to check the format of values commonly encountered in the X-script is implemented. In addition to the implemented validation methods, you can declare the custom functions or use the external functions.

The result of a validation method returns the information if a data type is valid or not. If there is no action "onFalse" specified and if the result is false, an error message is recorded into a log file. If the "onFalse" action is specified, then errors recognized by the validation method are cleared, and you can report your error message.

Examples of the validation of data type and the associated actions:

```
int(100,999); onTrue out (getText()); onFalse error('This is my error message');
string(10.20);
xdatetime(' yyyyMMddHHmmss ');
```

If an appropriately implemented validation method is not available, you can declare your custom validation method specified in the declaration section and you can refer to it by its name.

4.1.13 Implemented validation methods

The data types implemented in the X-definition correspond to the types of the XML schema. Table 4a is a list of implemented methods. These methods may include named parameters, where the name corresponds to a facet of the respective type of XML schema.

Allowed named parameters are listed in the following table, and the corresponding letter sequences are described in the last column.

Table 6 - Named parameters corresponding to facets in XML schema

Named parameter corresponding to the facet in XML schema	Value	Letter
%base	string with the name of a base type	b

%enumeration	list of allowed values of a type "[" ... "]"	e
%fractionDigits	number of digits in the fractional part of a number	f
%item	reference to the validation method	I
%length	the length of a string, array, etc.	L
%maxExclusive	parsed value of the data must be less than the parameter.	M
%maxInclusive	parsed value of the data type must be less than or equal to the parameter.	M
%maxLength	the length of a string, array, etc.	L
%minExclusive	parsed value of the data must be greater than the parameter.	M
%minInclusive	parsed value of the data must be greater than or equal to the parameter.	M
%minLength	minimal length of a string, array, etc.	L
%pattern	list (Container) of strings with regular expressions, which must be met when processing the data	P
%totalDigits	number of digits of the whole part of the validated number	T
%whiteSpace	specification of how to process white spaces in the validated data. Possible values are: "collapse", "replace" or "preserve"	W

For example:

```
string(5, 10) corresponds to the string(%minLength=5, %maxLength=10)
```

or

```
decimal(3, 5) corresponds to the decimal(%totalDigits=5, %fractionDigits=3)
```

After the sequence parameters, the named parameters can be listed:

```
string(5, 10, %whitespace="preserve", %pattern=["a*", "*.b"])
```

or

```
decimal(3, 5, %minExclusive=-10, %maxExclusive=10)
```

Note the methods that handle the date check if the year value from a given date is in the interval <actual year-200, actual year+200>. This check can be disabled using the property "xdef_checkdate" to "false" (the default value is "true"). Therefore, the date of 1620-08-11 is evaluated as an error if you do not set properties.setProperty("xdef_checkdate", "false").

A list of the implemented validation methods compatible with XML schema is described in the following table.

A detailed description of the data types of XML schema can be found at http://www.w3.org/TR/xmlschema11-2#datatype.

The penultimate column of the following table describes the result type of a validated string. The last column describes the named parameter and sequence parameters. The capital letter M describes the possibility of specification sequential parameters representing the "minInclusive" and the "maxInclusive" values. The capital letter L describes the possibility of specification sequential parameters representing the "minLength" and "maxLength" values.

Table 7 - Validation methods of XML schema data types

Method name	Description	Result type	Parameters
anyURI	URI	URI	L belp
base64Binary	an array of bytes in base64-encoded format	Bytes	L belp
boolean	Boolean value ("true", "false")	Boolean	p
Byte	8-bit integer number	Int	M bempt
Date	Date	Datetime	M bempt
dateTime	date and time	Datetime	M bempt
decimal	decimal number	Decimal	T befmpt

double	floating-point numbers	double	M befmpt
duration	XML duration.	Duration	M bempt
ENTITY	name of the XML entity	String	L epl
ENTITIES	list of the XML entity names separated by a space	Container	L epl
Float	floating-point numbers	float	M bempt
gDate	date	Datetime	M bempt
gDay	the day of the date	Datetime	M bempt
gMonth	the month of the date	Datetime	M bempt
gMonthDay	month and day of the date	Datetime	M bempt
gYear	day of the date	Datetime	M bempt
gYearMonth	year and month of the date	Datetime	M bempt
hexBinary	the array of bytes, in the hexadecimal format	Bytes	L belp
ID	the unique value of NCName in the XML document	String	L belp
IDREF	reference to a unique value in the XML document	String	L belp
IDREFS	list of the references to unique values in the XML document	Container	L belp
int	32-bit integer number	Int	M bempt
integer	integer number	decimal	M bempt
language	XML schema language specification (RFC 3066 or IETF BCP 47)	String	L belp
list	array of values.	Container	L beilp
long	64-bit integer number	int	M bempt
Name	name (according to the XML name specification).	String	L
NCName	XML NCName value	String	L
negativeInteger	negative integer number	decimal	M bempt
NMTOKEN	XML NMTOKEN (i.e. letters, digits, "_", "-", ".", ":")	String	L
NMTOKENS	list of NMTOKEN, separated by a space	Container	L
nonNegativeInteger	the positive integer number and zero	Decimal	M bempt
nonPositiveInteger	negative integer number and zero	Decimal	M belpw
normalizedString	character string	String	L bempt
positiveInteger	the positive integer number	Decimal	M bempt
QName	XML QName	String	L belp
short	16-bit integer number	int	M bempt
string	Character string. The named parameter %whiteSpace can only have here a value of "replace", "collapse", or "preserve". The default value of the %whiteSpace parameter is "preserve".	String	L belp
time	Time	Datetime	M bempt
token	XML token (according to XML specification).	String	L
union	union of more data types	Any	eip
unsignedByte	unsigned 8-bit integer	int	M bempt
unsignedLong	unsigned 64-bit integer	Decimal	M bempt
unsignedInt	unsigned 32-bit integer	int	M bempt
unsignedShort	unsigned 16-bit integer	int	M bempt

Some other data types implemented in the X-definition are listed in the following table:

Table 8 - Other validation methods of data types implemented in X-definition (and not in XML schema)

Method name	Description	Result type	Parameters
An	alphanumeric string (only letters or numbers)	String	L
BNF(g, s)	the value must match the rule name from the BNF Grammar g	String	-
contains(s)	any string that contains s	String	-
containsi(s)	any string that contains s regardless of upper/lower case.	String	-

country(s)	String with country code (ISO 3166 alpha-2 or alpha-3)	String	L belp
countries(s)	String with the list of country codes (ISO 3166 alpha-2 or alpha-3)	Container	elp
currency	currency code (three characters according to ISO 4217 code)	Currency	ep
CHKID	reference to a unique value – similar to IDREF in Table 4a, but the occurrence of the referred value must already exist at this time	String	-
CHKIDS	list of values according to CHKID separated by white spaces.	String	-
dateYMDhms	date and time corresponding to the mask "yyyyMMddHHmmss".	Datetime	M
domainAddr	Internet domain address	String	elp
dec	the decimal number corresponding to XML schema “decimal" data type. However, the decimal point can also be recorded as a comma). This method is **deprecated**, use “decimal” instead.	Decimal	T efmpt
emailAddr	email address according to RFC822 (the deprecated name is “email”)	EmailAddr	L
emailDate	date in the format form email (see RFC822).	Datetime	M
emailAddrList	list of email addresses separated by commas or semicolons	Container	L
ends(s)	the value must end with the string value in the parameter s.	String	-
endsi(s)	the value must end with the string value s regardless of upper/lower case.	String	-
enum(s, s1, ...)	the value must match one parameter from the list. Parameters s, s1, ... must be strings.	String	-
enumi(s, s1, ...)	the value must match with one parameter from the list regardless of the upper/lower case. Parameters s, s1, ... must be strings.	String	-
eq(s)	the value must be equal to the string s.	String	-
eqi(s)	the value must equal the string s regardless of upper/lower case.	String	-
ipAddr	Internet IPv4 or IPv6 address (see RFC 2818 and RFC 6125)	IPAddr	-
file	the value must be a correct file path	String	L
gps	GPS position (latitude, longitude[, altitude], [name])	GPSPosition	-
ipAddr	Internet address (IPv4, IPv6)	IPAddr	-
languages	list of values separated by a space which are equal to an item from the list of language codes 0according to XML schema language specification (RFC 3066 or IETF BCP 47)	Container	elp
list(s1, s2, ...)	the value must be equal to a parameter from the parameter list.	String	-
listi(s1, s2, ...)	the value must be equal to a parameter from the parameters list, regardless of the upper/lower case.	String	-
MD5	An MD5 checksum (32 hexadecimal digits)	Bytes	elp
NCNameList	list of NCName values according to the specification of the XML schema NCName. A separator is a white space.	String	elp
NCNameList(s)	list of NCName values according to the specification of the XML schema NCName. A list of characters that is used as a separator is in the parameter s.	String	elp
num	value is any sequence of digits.	String	L
pic(s)	the value must match the structure of the string s, where '9' means any digit, 'a' means any alphabetic ASCII character, 'X' is any alphanumeric (ASCII) character, and other characters must match. **Deprecated**, use the regex method or the string method with the parameter %pattern.	String	elp
price	amount (decimal number) and currency name (3 capital letters)	Price	-
printableDate	date in the usual “printable” format (e.g.: "Mon May 11 23:39:07 CEST 2020")	Datetime	L
QNameList	the value must be a list of QName values according to the XML specification. A separator is a white space.	Container	elp

QNameList(s)	the value must be a list of QName values according to the XML specification. A list of characters that is used as a separator is in the parameter s.	Container	elp
QNameList(s)	the value is the list of qualified names according to the XML specification, and for each name, the namespace in the context of the current element must be defined.	Container	elp
QNameURI	the value must be a QName according to the XML specification, and the namespace must be defined in the context of the current element	String	elp
QNameURI(s)	checks whether if in the context of the current element there exists the namespace URI corresponding to the value in the argument s.	String	elp
regex (s)	the value must match the regular expression s. The s must be a regular expression according to the XML schema.	RegexResult	-
sequence	allows you to describe a sequence of different values. Parameter %item = [type1, type2, ...], describes the sequence of validation methods.	Container	L ielmp
SET	stores the value of a table of unique values similar to the ID schema type. However, it does not report an error if the value already exists.	String	-
SHA1	SHA1 checksum (40 hexadecimal digits)	Bytes	e
starts(s)	the value must begin with the value of the string s.	String	-
startsi(s)	the value must begin with the value of the string s regardless of upper/lower case.	String	-
uri	the value must be a formally correct URI, as implemented in Java.	String	-
uriList	a formally correct list of URIS, as implemented in Java. The delimiter is a comma or white space.	String	-
url	the value must be a formally correct URI, as implemented in Java.	String	-
urlList	the value must be formally correct URL list as it is implemented in Java, the delimiter is a comma or whitespace	String	-
xdatetime	date and/or time of the corresponding ISO 8601 format (parses also the variants, which do not support date in XML schema).	Datetime	-
xdatetime(s)	date and/or time corresponding to the mask s (see Table 1 - Control characters in the date mask).	Datetime	-
xdatetime (s, t)	date and/or time corresponding to the mask s (see Table 1 - Control characters in the date mask). The resulting value will be formatted according to the mask t.	Datetime	-
xdtype	Checks if the value is a valid declaration of the implemented type validation method (i.e. from the table Table 7 - Validation methods of XML schema data types or this table)	Parser	-

Table 9 - JSON validation methods

Method name	Description	Result type	Parameters
Jboolean	Boolean value ("true", "false")	Boolean	P
Jnumber	an array of JSON number format	Decimal	M bempt
Jnull	JSON null value ("null")	Boolean	P
Jstring	any JSON string	String	L belp
Jvalue	any of the values above	Any	-

4.1.14 Set of unique values (uniqueSet)

To ensure a value is unique within a part of the XML document the uniqueSet table which contains a set of these unique values can be declared. The rows of a table are the unique values (i.e. they cannot be present more times in the table). The implicit part of the table is an object that we call the “key". It describes the structure of values of

the rows of the table. The key may have one or more entries, which are set with the results of the validation methods. The values of a key may be set step by step. For any part of the key, a validation method is defined. If a validation method finds an error when it parses a value it is normally reported. Other errors are indicated by the methods of uniqueSet tables. After the entries of a key are set with values (an entry can also be a null value) it can be saved to a table with the method ID or SET. It is checked if the key already exists in the table, and if not, it is stored as the next row of the table. If there is already a value of the key in the table, the method ID reports the error that the key already exists in the table. However, the method SET doesn't report this error. On the contrary, the method IDREF checks whether the key exists in the table. If not, it reports an error that the key does not exist in the table. Note that the key may be stored in the table later than when this method was invoked - the unresolved references to a key are reported after the scope of validity of the table expires. In some cases, we require that the key in the table already exists at the time the method was invoked. It allows the method CHKID, which checks whether a key is stored already in the table at the time of the method call. The SET method is similar to the ID method. It stores the key to the table. However, it does not report an error if the key already exists in the table (i.e. the key may be stored more times, therefore the value must be unique). The methods ID, IDREF, IDREFS, CHKID, SET, and CLEAR can be called on uniqueSet, object, or any of the key items. In both cases, it uses the current value of the key.

The table of unique values is declared by writing "uniqueSet" followed by the table name, and a description of the key. The key may have one or more items. Each entry of the key has a name and must be given the appropriate validation method. For example:

```
uniqueSet tab { first: int; second: string(5, 10); }
```

Sometimes we may request that an item has not been set, i.e. the key entry was null (this is, of course, also a value). In this case, before the description of the type of entry you write the character "?":

```
uniqueSet tab { first: int; second: ? string(5, 10); }
```

A question mark before the type has yet another meaning. Normally, the methods ID, IDREF, and CHKID use the key value and leave the key value unchanged. However, if an entry was declared with a question mark, after usage of the key it sets the key entry value to null (i.e. after you call one of the method ID, IDREF, or CHKID the entry value will be null).

The check process is invoked if you write the name of the table, then the character ".", and the name of an entry. See the following example:

```
<xd:declaration> uniqueSet house{number: int; person: ? string; company: ? string} </xd:declaration>

<Street>
  <House xd:script="occurs +" number="house.number.ID() /* the key is stored into the table “house";
                                                          entries "person" and “company" are still null */">
    <Person xd:script="occurs *" name="house.person.ID()  /* save the key with entries “number” and
                                                          "person", and "company" is still null */" />
    <Company xd:script="occurs *" name="house.company.ID()  /* save the key with entries “number” and
                                                          "company", and “person” is still null */" />
  </House>

  </Occupant xd:script="occurs *; finally house.IDREF() /* number and person or company */">
    House = "house.number"
    Person = "optional house.person()"
    Company = "optional house.company()"/>
</Street>
```

At any time, we can call the method CLEAR above a table. This method first checks whether in the table there are unresolved references to a key and if so, the errors are reported. Then all rows in the table are deleted. If you do not call this method, it will be invoked automatically before the expiry of the object validity (if the table was declared in the section xd:declaration, then this method is invoked at the end of the X-definition process).

If at some point we want to reset all key entries, we can call the method NEWKEY. This does not affect the contents of the table, but it sets all key entries to null.

4.1.15 Set of unique values (“table”) without named entries

For the validation method, you can write just ID and IDREF, IDREFS, so as an XML schema type. Therefore, the processor of X-definition automatically generates one internal variable with the global table that has rows of NCName data types. However, the user can declare his table with one not-named key entry and specify the data type of values in the table. Such a table might be declared as follows:

```
<xd:declaration>
  uniqueSet number: int;
<xd:declaration>

<Houses>
  /* store into the table the key with the house number */
  <House  xd:script="occurs +" Number="number.ID()/>

  /* The number must already be in the table */
  <House  xd:script="occurs +" Number="number.IDREF()/>
</Houses>
```

4.1.16 Linking tables of unique values

Sometimes we need to link more tables. This is done so that we specify the parameter with the validation method of a key entry from another table. Note that the reference to the entry from the other table means only taking over the appropriate validation method. Its use is clear from the following example:

```
<xd:declaration>
  uniqueSet house{number: int; apartment: ?int; person: ? string; company: ? string}
  uniqueSet street {name: string; number: house. number;}
</xd:declaration>

<Town name="string()"></Town>
  <Street xd:script="occurs +" name="street.name()"></Street>
    <House xd:script="occurs +" number="house.number(street.number.ID())"/>
      <Person xd:script="occurs *" name="house.person.ID()"></Person>
      <Company xd:script="occurs *" name="house.company.ID()"></Company>
    </House>
  </Street>
</Town>
```

Processing the attribute "name" in the element "Street" sets the key entry "street.name". Processing the attribute "number" in the element "house" sets the key entry "street.number". The key (i.e., the pair name, number) is stored in the table "street", checks its uniqueness, and sets the key entry "house.number".

4.1.17 Template element

Sometimes it is useful to describe the element model as "one to one", in other words, all attribute values, text values, or children including their occurrence are constant. In the construction mode, such an element is copied to the result. This can be achieved by writing the word "template" into the X-script of the model of the element. However, if at some point the text value starts with "$$$script:" the value is processed as the X-script. Example:

```
<elem xd:script = "template"
      attr1 = "abcd"
      attr2 = "$$$script: optional datetime ('yyyy/M/d'); create now (). toString ('yyyy/M/d') ">
   <child1/>
   <child2>Text1</child2>
   <child2>Text2</child2>
</elem>
```

This is the same as writing:

```
<elem xd:script = "required; create newElement()"
    attr1 = 'fixed 'abcd'; create newElement()"
    attr2 = "optional xdatetime('yyyy/M/d'); create.now().toString ('yyyy/M/d')">
  <child1 xd:script = "required; create newElement()" />
  <child2 xd:script = "required; create newElement()" />
     fixed 'text1'; create newElement()
   </child2>
  <child2 xd:script = "required; create newElement()" >
     fixed 'text2'; create newElement()
   </child2>
</elem>
```

Note: For the keyword "template" it is possible after the semicolon to add the "options trimText" or "options noTrimText". If "trimText" is not set, all spaces, new lines, and tabs between elements are interpreted as literals (i.e. string constants)!

4.1.18 X-script commands

The X-script command can either be the method call terminated with a semicolon or a statement block {similar to the statement block of the declared method). The executive commands are placed between the curly brackets "{" and "}". The return can be done with the command "return" the same as in a method declaration. The expressions and executive commands are written similarly as in the language "Java" or "C" (including the compound statement).

The syntax of statements is almost the same as in the language "Java". Implemented statements are:

the variable declaration statement (value types, however, must match the types in the X-script)
assignment statement
the method call statement
the statement "break"
the statement "continue"
the statement “do"
the statement “do"
the statement “for"
the statement "return"
the statement “switch"
the statement "throw"
the statement block "try" and "catch"
the statement "while"

A detailed description is beyond the scope of this text. The reader can find a description, for example, in a description of Java programming language.

4.1.19 Implemented X-script methods

In the X-script a variety of implemented methods can be called. Some of them can only be used in some parts of the X-script. The following tables list methods and result types. The parameter types are described in the following way:

AnyValue	**v, v1, v2, …**
Datetime	**d**
Element	**e, e1, e2, …**
Container	**c, c1, c2, …**
int	**m, n, n1, n2, …**
float	**f, f1, f2, …**
Object	**o**
String	**s, s1, s2, …**

Table 10 are listed all implemented general methods in X-script.

Note many methods are implemented on different objects. These methods are described in tables *Table 12* to *Table 33 – Methods of objects of the type uniqueSetKey*

Method name	Description	Result type
resetKey()	Sets the value of the actual key of the uniqueSet table to the value from this object.	

Table *Mathematical methods are listed in* Table 35 *and* Table 36*.*

Note many methods are implemented on different objects. These methods are described below.

Table 10 - General methods implemented in X-script

Method name	Description	Result type	Where use
addComment(s)	adds an XML comment with a value of s at the end of the		element

	child list of the current element.		
addComment(n, s)	adds an XML comment node with a value of s after node n.		anywhere
addPI(s1, s2)	adds the processing instruction at the end of the child list of the current element. The target name is s1, data is s2.		element
addPI(n, s1, s2)	adds the processing instruction after node n. The target name is s1, data is s2.		anywhere
addText(s)	adds a text node with a value of s to the current element.		attribute, text node, element
addText(e, s)	adds a text node with the value of s to the child nodes of element e.		anywhere
bindSet(u[,u1…])	this method can be specified only in the “init” section of the X-script of the model of Element. At the end of processing the element where it was invoked, it sets to all specified uniqueSet tables the value of the actual key which was at init time (after the “finally” section).		element
clearReports()	clears all current (temporal) error reports generated by the preceding validation method (used e.g. in onFalse action).		anywhere
cancel()	forced end of the processing of X-definition		anywhere
cancel(s)	forced end of the processing of X-definitions and sets the error message with text s.		anywhere
defaultError()	writes a default error message into a temporary report log and returns the boolean value false. The error is XDEF515 Value differs from expected.	boolean	attribute, text node
easterMonday(n)	returns the date with Easter Monday for year n	Datetime	Anywhere
error(r)	writes an error message with report r into a temporary report log. The result is the Boolean value false.	boolean always false	Anywhere
error(s)	writes an error message with the text s into a temporary report log. The result of the function is the Boolean value false.	boolean always false	Anywhere
error(s1, s2)	writes an error message report created from s1 and s2 into a temporary report log. The result is the Boolean value false. The s1 parameter is the identifier of the error, the s2 is the error text.	boolean always false	Anywhere
error(s1, s2, s3)	writes an error message report created from s1 and s2 into a temporary report log. The result of the function is the Boolean value false. The s1 parameter is the identifier of the error, the error text is s2 and s3 is a modifier with parameters of text.	boolean always false	Anywhere
errors()	returns the current number of errors reported during processing.	int	attribute, text node
errorWarnings()	returns the current number of errors reported during processing.	int	attribute, text node
format(s, v1, ...)	returns string created from values of parameters v1, ... according to the mask s (see method format in java.lang.String).	String	Anywhere
format(l, s, v1, ...)	returns string created from values of parameters v1, ... according to the region specified by Locale in parameter l and mask s (see method format in java.lang.String).	String	Anywhere
from()	returns the Container corresponding to the current context. The method can only be used in the "create" action in the X-script element, the text value, or the	Container	create element

	attribute. If the result is null, returns an empty Container.		
from(s)	returns the Container created after the execution of the XPath expression s in the current context. The method can only be used in the "create" action in the X-script element, the text value, or the attribute. If the result is null, returns an empty Container.	Container	create attribute create element create text node
from(e, s)	returns the Container after the execution of the XPath expression s in the element e. The method can only be used in the "create" action in the X-script of an element, the text value, or an attribute. If e is null, the result is an empty Container	Container	create attribute create element create text node
getAttr(s)	returns the value of the attribute with the name s (from the current element). If this attribute does not exist, returns an empty string.	string	element
getAttr(s1, s2)	returns the value of an attribute with the local name s and namespace s2 (from the current element). If this attribute does not exist, returns an empty string.	string	element
getAttrName()	returns a string with the name of the current attribute.	String	all actions in the X-script attributes
getElement()	the result is the current element.	Element	element, text node attribute
getElementName()	name of the current element.	String	Anywhere
getElementLocalName()	name of the current element.	String	attribute, text node
getElementText()	returns a string with the concatenated text content of nodes that are direct descendants of the current element.	String	attribute, text node
getImplProperty(s)	returns the value of the property from the current X-definition that is named s. If the item does not exist, return an empty string.	String	attribute, text node
getImplProperty(s1, s2)	returns the value of the property s1 of X-definition, whose name is s2. If the appropriate X-definition or item does not exist will return an empty string.	String	attribute, text node
getItem (s)	returns the value of the items from the current context.	String	element, text node attribute
getLastError()	returns the last reported error report.	Message	element, text node attribute
getMaxYear()	returns the maximum allowed value of the year when parsing the date.	int	Anywhere
getMinYear()	returns the minimum allowed value of the year when parsing the date.	int	Anywhere
getNamespaceURI()	returns a string whose value is the namespace URI of the current element. If the namespace URI does not exist, returns an empty string.	String	attribute, text node
getNamespaceURI(n)	returns a string whose value is the namespace URI of the node n (it can be either an element or attribute). If the namespace URI does not exist, returns an empty string.	String	attribute, text node
getNamespaceURI(s)	returns a string whose value is a namespace URI matching prefix s in the context of the current element. If the namespace URI does not exist, returns an empty string.	String	attribute, text node
getNamespaceURI(s, e)	returns a string whose value is a namespace URI matching prefix s in the context of the element e. If the namespace	String	attribute, text node

	URI does not exist, returns an empty string.		
getNSUri(s, e)	Returns namespace URI of given prefix s from the context of element e	String	Anywhere
getOccurrence()	returns the current number of instances of the object.	int	Element
getParentContextElement()	returns the element of the context of the parent of the current element	Element	Element
getParentContextElement(n)	returns the element of the context of n - parent of the current element	Element	Element
getParsedBoolean()	returns the Boolean value of the ParseResult (if the previous X-script was read by a validation method). Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.	boolean	only after the method type checking
getParsedBytes()	returns the value of a byte array of ParseResult (if the previous X-script was a validation method). Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.	Bytes	only after the method type checking
getParsedDatetime()	returns the value of Datetime from ParseResult (if the previous X-script was a validation method). Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.	Datetime	only after the method type checking
getParsedDecimal()	returns the value of a decimal number of ParseResult (if the previous X-script was a validation method). Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.	Decimal	only after the method type checking
getParsedDuration()	returns the value of Duration of ParseResult (if the previous X-script was a validation method). Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.	Duration	only after the method type checking
getParsedFloat()	returns the float value of ParseResult (if the previous X-script was read by a validation method). Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.	float	only after the method type checking
getParsedInt()	returns the int value of ParseResult (if the previous X-script was read by a validation method). Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.	int	only after the method type checking
getParsedValue()	returns an object with the parsed value of ParseResult (if the previous X-script was read by a validation method). Warning: this method must be called immediately after a	AnyValue	only after the method type checking

	call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.		
getQnameLocalpart(s)	returns a string with a local name of the argument that is a qname.	String	attribute, text node
getQnamePrefix(s)	returns a string with the prefix from s that is a qname. Returns an empty string, if the argument is a qname or does not have a prefix.	String	attribute, text node
getQnameURI(s)	returns a string whose value is a namespace URI matching the qname s the in the context of the current element. If the namespace URI does not exist, returns an empty string.	String	attribute, text node
getQnameURI(s, e)	returns a string whose value is a namespace URI matching the qname s the in the context of the element e. If the namespace URI does not exist, returns an empty string.	String	attribute, text node
getRootElement()	returns the root element of the current element.	Element	attribute, text node, element
getSourceColumn()	returns column of source position of processed node (if it is not available returns 0)	int	attribute, text node, element
getSourceLine()	returns a line of source position of the processed node (if it is not available returns 0)	int	attribute, text node, element
getSourcePosition()	returns the printable form of the source position of the processed node (if it is not available returns an empty string)	String	attribute, text node, element
getSpecialDates()	returns the Container with the date values that are permitted even if the year date is not in the allowed range.	Container	attribute, text node
getSysId	returns system ID of source position of processed node (if it is not available returns an empty string)	String	attribute, text node, element
getText()	returns a string with the value of a current attribute or a text node.	String	attribute text text of element
getTextContent()	returns a string with the text content of the current element and its descendants. Returns an empty string if no text is available.	String	text of element and descendants
getTextContent(e)	returns a string with the text content of the element e and its descendants. Returns an empty string if no text is available.	String	Anywhere
getXDPosition()	returns a string with the current X-position.	String	attribute, text node, element
getXpos()	returns the current position of the processed XML document in XPath format.	String	attribute, text node, element
getUserObject()	returns an external user object.	Object	attribute, text node
getVersionInfo()	returns information about the version of the current X-definition.	String	attribute, text node
hasAttr(s)	returns true if the current element has an attribute with the name s.	boolean	element
hasAttr(s1, s2)	returns true if the current element has the attribute with local name s1 and the namespace s2.	boolean	element
IsCreateMode()	returns true, if the current processing mode is the construction mode.	boolean	attribute, text node
isDatetime(s)	the result is true when the date from the string s matches the format according to ISO 8601 (i.e., the mask of "y-M-d	boolean	attribute, text node

	[TH: m: s[.S] [Z]] ").		
isDatetime(s1, s2)	the result is true when the date in the string s1 matches the mask s2.	boolean	attribute, text node
isLeapYear(n)	returns true if the year n is a leap year.	boolean	anywhere
isNumeric(s)	returns true when the string s contains only digits.	boolean	anywhere
insertComment(s)	Insert the comment s before the actual node.		element
insertComment(n, s)	Insert comment s before node n.		anywhere
insertPI(s1, s2)	insert processing instruction before the actual element. The target name is s1, data is s2.		element
insertPI(n, s1, s2)	Insert processing instruction before node n. The target name is s1, data is s2.		anywhere
insertText(s)	Insert the text node s before the actual node. Note the parent of the actual node must be an element!		element
insertText(n, s)	insert text node s before node n. Note the parent of the node n must be an element!		anywhere
newElement()	creates a new element (the name is derived according to the location, where the method was specified).	Element	create
newElement(s)	creates a new element named according to argument s.	Element	attribute, text node
newElement(s1,s2)	creates a new element named according to argument s1 and the namespace s2.	Element	attribute, text node
newElements(n)	creates a Container with n new elements (the name is derived according to the location, where the method was specified).	Container	create
newElements(n, s)	creates a Container with the n new elements named by the argument s.	Container	attribute, text node
newElements(n,s1,s2)	creates a Container with the n new elements named by the argument s1 and the namespace s2.	Container	attribute, text node
now()	returns current date and time	Datetime	anywhere
occurrence()	returns a number corresponding to the current number of the occurrence of the element.	int	element, text node
out(v)	value v is converted to a string and is written on the standard output		anywhere
outln ()	the output of the new line to the standard output		anywhere
parseBase64 (s)	converts a string to an array of bytes If the string is not Base64, then the method returns null.	Bytes	anywhere
parseDate(s)	converts the string s with a date in the ISO 8601 format to Datetime value (i.e. according to the mask "yyyy-M-dTH: m[: s][.S] [Z]"). If the string s does not date according to ISO, then the method returns null.	Datetime	attribute, text node
parseDate (s1, s2)	converts the string s1 to Datetime according to the mask in the s2 parameter. If the string is not a DateTime according to ISO the method returns null.	Datetime	attribute, text node
parseEmailDate(s)	converts the string s with a date in the format of RFC822 to the Datetime value. If the date is not in the specified format then the method returns null.	Datetime	attribute, text node
parseFloat (s)	converts the string to a float value If the string s is not a float number, then the method returns null.	float	attribute, text node
parseInt(s)	converts the string s to an integer value If the string s is	int	attribute, text node

	not an integer number, then the method returns null.		
parseDuration(s)	converts the string s with a time interval in the format ISO 8601 to the Duration value. If the string s is not duration according to ISO, then the method returns a value null.	Duration	attribute, text node
parseHex(s)	converts the string s to an array of bytes If the string is not Base64, then the method returns null.	Bytes	anywhere
pause()	in the debug mode, write the information about actual processing to the standard output. The program stops and waits for a response from the standard input. If the answer is "go", the program continues. Instead of "go", you can specify other commands. The list of possible commands will be printed by typing "?". If the debug mode is not set this method is ignored.		attribute, text node
pause(s)	in the debug mode, print the information line and the string s to the standard output. The program waits for a response from the standard input. If the answer is "go", the program continues. Instead of "go", you can specify other commands. The list of possible commands will be printed by typing "?".		attribute, text node
printf(s, v1, ...)	prints to the standard output stream the string created from values of parameters v1, ..., according to the mask s (see method printf in java.io.PrintStream).		anywhere
printf(l, s, v1, ...)	prints to the standard output stream a string created from values of parameters v1, ... according to the region specified by Locale in parameter l and to the mask s (see method printf in java.io.PrintStream).		anywhere
removeAttr(s)	removes an attribute with the name s from the current element.		element, attribute, text node
removeAttr(s1, s2)	removes an attribute with the local name s1 and the namespace s2 from the current element.		element, attribute, text node
removeText()	deletes the current (being processed) node with a text value (i.e. a text or attribute node)		attribute, text node
removeWhiteSpaces(s)	all occurrences of the white spaces in the string are replaced by a single space.	String	anywhere
replace(s1, s2, s3)	all occurrences of the string s2 in the string s1 are replaced with the string s3.	String	anywhere
replaceFirst(s1, s2, s3)	the first occurrence of the string s2 in the string s1 is replaced with the string s3.	String	anywhere
returnElement(o)	The result created from the argument o is an Element set as the result of the X-definition process. The process of X-definition will be finished and the method returns, as a result, the created value.		
setAttr(s1, s2)	sets the value of the attribute named s1 in the current element to s2.		element, attribute, text node
setAttr(s1, s2, s3)	sets the value of an attribute with the local name s1 and namespace s2 in the current element to s3.		element, attribute, text node
setElement(e)	insert the element e at the current location of the processed XML element (e.g. you may use it to add an element in the action onAbsence).		attribute, text node, element
setMaxYear(n)	sets the maximum allowed value of the year of the validated DateTime.		anywhere

setMinYear(n)	sets the minimum allowed value of the year of the validated DateTime		anywhere
setParsedValue(v)	stores the value v in the current parsed result. Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue or onTrue. If this condition is not met, then the result of the method is not defined.	ParsedValue	only after the method type checking
setSpecialDates(c)	sets the Container with the date values in the list of permitted dates (even if the year date is not in the allowed range).		attribute, text node
setText(v)	the string to which is converted the argument v replaces the value of the current attribute or a text node.		the text attribute
setUserObject(o)	sets the external user object (Java object).		attribute, text node
tail(s, n)	returns the last n characters in the string	String	anywhere
toString(v)	converts the value v in the standard manner to a character string. The value v can be of type String, Integer, Float, Date, Element, and List	String	anywhere
toString(v, s)	converts the value v according to mask s to a character string. The value v can be of type int, float, or Datetime. The value of s is a format mask. The mask (see 4.1.2) Types of values of variables and expressions in X-script.		anywhere
trace()	in the debug mode it writes the information about actual processing to the standard output. If the debug mode is not set this method is ignored.		attribute, text node
trace(s)	in the debug mode it writes the information line and text s to the standard output. If the debug mode is not set this method is ignored.		attribute, text node
translate(s1, s2, s3)	replaces all occurrences of characters in the string s1 which matches one character from s2 with the character at the corresponding position in the string s3. For example: translate("bcr", "abc", "ABa") returns "Bar". If in the appropriate position in the string s2, there is not a character then this character is skipped: translate("-abc-", "ab", "BA") Returns the "BAc"	String	anywhere
xcreate(c)	a result is an object constructed by the model of element c (used in the construction mode).	Element	attribute, text node
xparse(s)	parses an XML document from a string with the current X-definition and returns the parsed value of the root element. If the s is a string, describing an URL or a path to a file, the parser uses a stream created from s. However, if the string s begins with the character "<", then the parser uses the value of s converted to a UTF-8 byte stream.	Element	attribute, text node
xparse(s1, s2)	parses an XML document from a string s1 with the X-definition named s2 and returns the parsed value of the root element. If the s1 is a string, describing an URL or a path to a file, the parser uses a stream created from s1. However, if the string s1 begins with the character "<", then the parser uses the value of s converted to a UTF-8 byte stream.	Element	attribute, text node
xparse(s, null)	parses an XML document from a string s without an X-definition. Returns the root element. If the s is a string, describing an URL or a path to a file, the parser uses a	Element	attribute, text node

	stream created from s. However, if the string s begins with the character "<", then the parser uses the value of s converted to a UTF-8 byte stream.		
xpath(s)	returns a Container created after the execution of the XPath expression s on the current element. If the actual element is null, it returns an empty Container.	Container	element, attribute, text node
xquery(s)	returns the Container created after the execution of the xquery expression s on the current context. The method can only be used in the "create" action in the X-script of an element, the text node, or an attribute. If the actual element is null it returns an empty Container. This method is implemented only if the Saxon library is available.	Container	element, attribute, text node
xquery(s, e)	returns the Container created after the execution of the XQuery expression s on the element e. If the actual element is null, it returns an empty Container. This method is implemented only if the Saxon library is available.	Container	attribute, text node

The following tables describe methods implemented by the individual object types. The type of an object is expressed by the name of a type in the X-script (e.g. "int", "String", etc. – see 4.1.2 Types of values of variables and expressions in X-script). A number is also assigned to each type (called “Type-ID”). The names of the types and the corresponding identifiers in the X-script and the Java code by enumeration org.xdef.XDValueType is in the following table).

Table 11 - Names of the X-script types and the corresponding type ID

X-script name	Type ID	Name of the item in Java enumeration XDValueType
boolean	$BOOLEAN	BOOLEAN
BNFGrammar	$BNFGrammar	BNFGRAMMAR
BNFRule	$BNFRULE	BNFRULE
Bytes	$BYTES	BYTES
char	$CHAR	CHAR
Container	$CONTAINER	CONTAINER
Datetime	$DATETIME	DATETIME
Decimal	$DECIMAL	DECIMAL
Duration	$DURATION	DURATION
Element	$ELEMENT	ELEMENT
EmailAddr	$EMAILADDR	EMAILADDR
Exception	$EXCPTION	EXCEPTOIN
float	$FLOAT	FLOAT
GPS	$GPS	GPS
Input	$INSTREAM	INPUT
int	$INT	INT
NamedValue	$NAMEDVALUE	NAMEDVALUE
Object	$OBJECT	OBJECT
Output	$OUTSTREAM	OUTPUT
Parser	$PARSER	PARSER
ParseResult	$PARSERESULT	PARSERESULT
Price	$PRICE	PRICE

Regex	$REGEX	REGEX
RegexResult	$REGEXRESULT	REGEXRESULT
Report	$REPORT	REPORT
ResultSet	$RESULTSET	RESULTSET
Service	$SERVICE	SERVICE
Statement	STATEMENT	STATEMENT
String	$STRING	STRING
URI	$URI	URI
XmlOutStream	$XMLWRITER	XMLWRITER
XPathExpr	$XPATH	XPATH
XQueryExpr	$XQUERY	XQUERY

For each type listed in the previous tables, the following methods are implemented:

Table 12 - Methods of objects of all types

Method name	Description	Result type
x.toString()	returns a string in the "viewable" shape of the value x.	String
typeName(v)	returns the name of the type of v.	String
valueType(v)	returns Type-ID of v.	Int

Table 13 - Methods of objects of the type BNFGrammar

Method name	Description	Result type
BNFGrammar x	construction of BNF grammar object x is recorded in the element <xd:BNFGrammar name="x"> Text of BNF grammar specification see 2.12 BNF grammar in X-definition	BNFGrammar
BNFGrammar x	construction of BNF grammar object x, which is an extension of the grammar g, is recorded in the element <xd:BNFGrammar name="x" extends="g"> see 2.12 BNF grammar in X-definition	BNFGrammar
x.parse(s)	returns a parsed value of the attribute of the text node following the rule with the grammar of x	ParseResult
x.parse(s1, s2)	returns a value of the parsed string s2 according to the rule s1 from the grammar x.	ParseResult
x.rule(s)	returns the rule from the grammar x.	BNFRule

Table 14 - Methods of objects of the type BNFRule

Method name	Description	Result type
x.parse ()	returns a parsed value of the current attribute of the text node by grammar rule x	ParseResult
x.validate(s)	returns true if string s fits rule x.	Boolean

Table 15 - Methods of objects of the type Bytes

Method name	Description	Result type
x = new Bytes (n)	returns an array of bytes of x of size n. All bytes are set to 0.	Bytes
x.add(n)	adds the value n after the last item in the array of bytes x.	
x.clear()	clears the array of bytes x.	
x.getAt(n)	Returns n-th item of the array x (the index of the first item is 0).	Int
x.insert(n1, n2)	inserts a byte n2 before the n1-th item of x (the index of the first item	

	is 0).	
x.remove(n)	removes the n-th item from the array of bytes x and returns the original value (the index of the first element is 0).	Int
x.setAt (n1, n2)	sets the value of the n2 to the n1-th item of the byte array x (the index of the first item is 0).	
x.toBase64()	returns a string with the value of the byte arrays in Base64-encoded format.	String
x.toHex()	returns a string with the value of the array of bytes in the hexadecimal format.	String

Table 16 - Methods of objects of the type Container

Method name	Description	Result type
x = new Container()	creates an empty Container x.	Container
x.addItem(o)	adds the object o to the end of the sequence part of the Container x.	
x.getElement()	returns the first XML element found in the sequence part of the Container x (or returns null if does not element exist).	Element
x.getElement(n)	returns the n-th XML element found in the Container x (or returns null if such element not exists).	Element
x.getElements ()	returns the new Container with the XML elements found in Container x.	Container
x.getElements (s)	returns the new Container with the XML elements with the name s found in Container x.	Container
x.getItemType(n)	returns the type-ID of the n-th item in the Container x.	int
x.getLength()	returns the number of the items in the sequential part of Container x.	int
x.getNamedItem(s)	returns the value of a named item in the mapped part of Container x.	AnyValue
x.getNamedString(s)	returns the value of the named item s in the mapped part of Container x as a string.	String
x.getText()	returns a string concatenated from the string items in the sequential part of Container x.	String
x.getText(n)	returns a string with the n-th item of type string in the sequential part of Container x.	String
x.hasNamedItem(s)	returns true if Container x has a named item with the name s.	boolean
x.isEmpty()	returns true if Container x has no items.	boolean
x.item(n)	returns the n-th item in the Container x.	AnyValue
x.removeItem(n)	deletes the n-th element of the Container x.	
x.removeNamedItem(s)	deletes a named item with the name s in Container x.	
x.setNamedItem(v)	stores named item in the Container x.	
x.setNamedItem(s, v)	creates a named item with the name s and the value v in Container x.	
x.sort()	returns the ascending sorted sequential part of Container x (according to the compareTo method on the items).	Container
x.sort(s)	returns the ascending sorted sequential part of Container x. The result of the XPath expression s is applied as a key for the XML elements.	Container
x.sort(s, b)	returns the sorted sequential part of Container x. the direction of the sort is according to the Boolean argument b (true for ascending and false for the descending sort). The result of the XPath expression s is applied as a key for the XML elements.	Container
x.toElement()	creates an element from Container x.	Element
x.toElement(s)	creates an element with the name s from Container x.	Element
x.toElement(s1, s2)	creates an element with the name s2 and namespace s1 from Container x.	Element

Table 17 - Methods of objects of the type Datetime

Method name	Description	Result type
x = new Datetime (s)	creates an object x from the string s, which must be in the form of ISO8601.	Datetime
x.addDay(i)	adds to the date x the number of days i (i can even be negative, then the days will subtract from x) and returns a new value	Datetime
x.addHour(i)	adds to the date x the number of hours i (i can even be negative, then the hours will subtract from x) and returns a new value	Datetime
x.addMillisecond(i)	adds to the date x the number of milliseconds i (i can even be negative, then the milliseconds will subtract from x) and returns a new value	Datetime
x.addMinute(i)	adds to the date x the number of minutes i (i can even be negative, then the minutes will subtract from x) and returns a new value	Datetime
x.addMonth(i)	adds to the date x the number of months i (i can even be negative, then the months will subtract from x) and returns a new value	Datetime
x.addNanosecond(n)	adds to the date x the number of nanoseconds i (i can even be negative, then the nanoseconds will subtract from x) and returns the new value	Datetime
x.addSecond(i)	adds to the date x the number of seconds i (i can even be negative, then the seconds will subtract from x) and returns a new value	Datetime
x.addYear(i)	adds to the date x the number of years i (i can even be negative, then the years will subtract from x) and returns a new value	Datetime
x.easterMonday(i)	returns the date with Easter Monday of the year i from a date x.	Datetime
x.getDay()	returns the day of the date	Int
x.getFractionalSecond()	returns the value of seconds of date x including the fractional part of seconds	Float
x.getHour()	returns the hour of a date x	Int
x.getMillisecond()	returns the number of milliseconds of the date x since the beginning of the day	Int
x.getMinute()	returns the minutes from the date x since the beginning of the day	Int
x.getMonth()	returns the month from a date x (January is 1)	Int
x.getNanosecond()	returns the number of nanoseconds of the date x since the beginning of the day	int
x.getSecond()	returns the seconds of the date x since the beginning of the day	int
x.getWeekDay()	returns the day of the week from date x (1 is Sunday, 7 is Saturday)	int
x.getYear()	returns the year from a date x	int
x.getZoneName()	returns the name of the time zone of the date x.	String
x.getZoneOffset()	returns the offset of the time zone of the date x to the Prime Meridian, in milliseconds	int
x.isLeapYear()	returns true if the year x is a leap year.	boolean
x.lastDayOfMonth()	returns the last day of the month of a date x.	int
x.setDay(i)	sets the day i to the date x and return a new value	Datetime
x.setDaytimeMillis(i)	sets the time to the date x according to the number of milliseconds i in the argument i and returns a new value	Datetime
x.setHour(i)	sets the hour i to the date x and returns a new value	Datetime
x.setMillisecond(i)	sets the millisecond i to the date x and returns a new value	Datetime
x.setMinute(i)	sets the minute i to the date x and returns a new value	Datetime
x.setMonth(i)	sets the month i to the date x and returns a new value (January is 1)	Datetime
x.setSecond(i)	sets the second i to the date x and returns a new value	Datetime
x.setYear(i)	sets the year i to the date x and return a new value	Datetime
x.setZoneName(s)	sets the name of the time zone in the date x on s and returns a new value	Datetime
x.setZoneOffset(i)	sets the offset for the time zone in the date (i are milliseconds) and returns	Datetime

	a new value	
x.toMillis()	returns an integer value that corresponds to the number of milliseconds since January 1. January 1970	int
x.toString(s)	returns the character string with a date according to the mask s	String

Table 18- Methods of objects of type Duration (time interval)

Method name	Description	Result type
x = new Duration(s)	the constructor of the Duration object. Creates an object based on the string, which must be in ISO8601 format	Duration
x.getDays()	returns the number of days in the interval from x.	int
x.getEnd()	returns the date of the end of the interval from x.	Datetime
x.getFractionalSecond()	returns the number of seconds including the fractional part of the interval of x.	float
x.getHours()	returns the number of hours from the interval of x	int
x.getMinutes()	returns the number of minutes from the interval of x	int
x getMonths()	returns the number of months from the interval of x	int
x.getNextDate()	returns the next date and time from the interval of x	Datetime
x.getRecurrence()	returns the number of times of the interval from x	int
x.getSeconds()	returns the number of seconds from the interval of x	int
x.getStart()	returns the starting date and time from the interval of x	Datetime
x.getYears()	returns the number of years of the interval of x.	float

Table 19 - Methods of objects of the type Element

Method name	Description	Result type
x = new Element (s)	the constructor of Element. Creates new Element with the name s.	Element
x = new Element(s1, s2)	the constructor of Element. Creates a new element with the namespace s1 and the name s2.	Element
x.addElement(e)	adds the element e at the end of the list of child nodes of the element x. If x is the root element, it will produce an exception	
x.addText(s)	adds a text node with the value s at the end of the list of child nodes of the element x. If x is the root element of the XML document it will produce an exception	
x.getAttribute(s)	returns the value of the attribute s in the element x. If the attribute does not exist, it returns an empty string.	String
x.getAttribute(s1, s2)	returns a value of the attribute with the local name s2 and the namespace s2 from the element x. If the attribute does not exist, it returns an empty string.	String
x.getChildNodes()	returns the Container with a list of the child nodes of the element x.	Container
x.getNamespaceURI()	returns a string with the namespace URI of the element x.	String
x.getTagName()	returns the qualified name of the element x.	String
x.getText()	returns the string with the concatenated text of the element x.	String
x.hasAttribute(s)	returns true if the element x has an attribute with the name s	boolean
x.hasAttributeNS(s1, s2)	returns true if the element x has an attribute with the name s2 and the namespace s1.	boolean
x.isEmpty()	returns true if the element x has no child nodes and attributes.	boolean
x.setAttribute(s1, s2)	sets the attribute with the name of s1 and s2 value in the element x	
x.setAttribute(s1, s2, s3)	sets the attribute with the namespace of s1 and the name s2 and the value s3 in the element x	
x.toContainer()	returns the Container that was created from the element x.	Container
x.toString(b)	returns the string that was created from the element x. If b is true, then	String

	the string is an indented form of the element x.	

Table 20 - Methods of objects of the type Exception

Method name	Description	Result type
x = new Exception (s)	creates an Exception with the message s	Exception
x=new Exception(s1,s2)	creates an Exception with the report ID s1 and message text s2.	Exception
x=new Exception(s1,s2, s3)	creates an Exception with the report ID s1 and message text s2 and modification parameters in the string s3.	Exception
x.getReport()	returns a report message from the Exception x	Report
x.getMessage()	returns a message string from the Exception x	String

Table 21 - Methods of objects of the type Input

Method name	Description	Result type
x = new Input (s)	creates an input stream according to the argument s.	Input
x = new Input(s, b)	creates an input stream according to the argument s. If b is true, it will be read in XML format.	Input
x = new Input(s1, s2)	creates an input stream by the argument s1. Argument s2 specifies the name of the encoding table.	Input
x=new Input(s1, s2, b)	creates an input stream by the argument s1. Argument s2 specifies the name of the encoding. If argument b is true, it will be read in the XML format	Input
x.eof()	returns true if the Input x is at the end	Boolean
x.readln()	reads a line of Input.	String

Table 22 - Methods of NamedValue objects

Method name	Description	Result type
x = new NamedValue (s, v)	creates a named value x with the name s and the value v.	NamedValue
x.getName()	returns the name of a named value x.	String
x.getValue()	returns the value of a named value x.	AnyValue
x.setName(s)	sets the name s to a named value x.	

Table 23 - Methods of objects of the type Output

Method name	Description	Result type
x = new Output (s)	creates an output stream according to the argument s.	Output
x = new Output(s, b)	creates an output stream according to the argument s. If b is true, it will be written in XML format.	Output
x = new Output(s1, s2)	creates an output stream by the name of s1 and the code page s2.	Output
x = new Output(s1, s2, b)	creates an output stream by the name of s1 and the code page s2. If b is true, it will be written in XML format	Output
x.error(s)	writes an error record with message s.	
x.error(s1, s2)	writes an error record with message ID s1 and the message string s2.	
x.error(s1, s2, s3)	writes an error record with message ID s1, the message string s2, and modification s3.	
x.getLastError()	returns the last written error record.	Report
x.out(s)	writes the text s to the x.	
x.outln()	writes a new line to the x.	
x.outln(s)	writes a new line with the text s to x.	
x.printf(s, v1, ...)	prints to the x a string created from values of parameters v1, ... according to the mask s (see method printf in java.io.PrintStream).	

x.printf(l, s, v1, ...)	prints to the x a string created from values of parameters v1, ... according to the region specified by Locale in parameter l and the mask s (see method printf in java.io.PrintStream).	
x.putReport(r)	writes the Report r to x.	

Table 24 - Methods of objects of the type ParseResult

Method name	Description	Result type
x = new ParseResult(s)	creates a ParseResult value x from the string.	ParseResult
x.booleanValue()	returns the boolean value from x.	Boolean
x.bytesValue()	returns an array from x.	Bytes
x.matches()	returns true if the x does not contain errors. Otherwise, it returns false	boolean
x.datetimeValue()	returns a Datetime value from x.	Datetime
x.durationValue()	returns a Duration value from x.	Duration
x.decimalValue()	returns a Decimal value from x.	Decimal
x.error(s)	sets the error message s to x.	
x.error(s1, s2)	sets the error Id s1 in the message s2 to x,	
x.error(s1, s2, s3)	sets the error Id s1 and the message s2 modified by s3 to x.	
x.floatValue()	returns a float value from x.	Float
x.getError()	returns an error message from x.	Report
x.getParsedString()	returns the parsed string from x.	String
x.getValue()	returns the parsed value from x.	AnyValue
x.setParsedString(s)	sets s as the parsed value to x.	
x.setValue (v)	sets the parsed value v in x.	

Table 25 - Methods of objects of the type Regex

Method name	Description	Result type
x = new Regex(s)	creates a regular expression x from s.	Regex
x.getMatcher(s)	returns the RegexResult created by the regular expression x from the string s.	RegexResult
x.matches(s)	returns true if the regular expression x has met the string s.	Boolean

Table 26 - Methods of objects of the type RegexResult

Method name	Description	Result type
x.end(n)	returns the end index of the group n from x.	Int
x.group(n)	returns a string from the group n from x.	String
x.groupCount()	returns the number of groups in x.	Int
x.matches()	returns true if the result of regular expression x has been met.	Boolean
x.start(n)	returns the initial index of the group n in x.	Int

Table 27 - Methods of objects of the type Report

Method name	Description	Result type
x = new Report (s)	creates a report with the message s.	Report
x = new Report(s1, s2)	creates a report with the ID s1 and the message s2.	Report
x = new Report(s1, s2, s3)	creates a report x with the ID s1, message s2 modified with s3.	Report
x.getParameter(s)	returns a string with the value of modification parameter s from the report x.	String
x.setParameter(s1, s2)	returns the new Report created from x where the modification parameter s1 is set to s2.	Report

x.setType(s)	returns a new Report where the type of report is set to the value s. The value of s must be one of: "E" … error "W" … warning "F" … fatal error "I" … information "M"... message "T" … text	Report

Table 28 - Methods of objects of the type ResultSet

Method name	Description	Result type
x.close()	closes the ResultSet x.	
x.closeStatement()	closes the statement associated with the ResultSet x.	
x.getCount()	returns the number of entries in the actual position of x.	Int
x.getItem()	returns the current entry in the ResultSet x as a string.	String
x.getItem(s)	returns the entry named s from the current position of x.	String
x.hasItem(s)	returns true if the named entry s exists in the current position of x.	boolean
x.hasNext()	returns true if there is another row in the ResultSet x.	boolean
x.isClosed()	returns true if the ResultSet x is closed.	boolean
x.next ()	sets the next row in the ResultSet x and returns true, if there is one.	boolean

Table 29 - Methods of objects of the type Service

Method name	Description	Result type
x=new Service(s1,s2,s3,s4)	creates object x providing access to a database. The s1 parameter is the string defining the type of database interface (e.g., "JDBC"), s2 is the database URL, s3 is a user name and s4 is a password.	Service
x.close()	close the database x.	
x.commit()	performs commit operation on the database x	
x.execute (s1, ...)	performs the command s1 with parameters s2, s3, ... Returns true if the command was performed.	boolean
x.hasItem(s1, ...)	returns true when the item defined by parameters exists.	boolean
x.isClosed()	returns true if the database x is closed.	Boolean
x.prepareStatement()	prepares and returns a statement on database x.	Statement
x.query(s1, s2)	executes a query in a database x and returns the ResultSet object.	ResultSet
x.queryItem(s1, s2, s3)	executes a query in a database x and returns a string with the item s3.	String
x.rollback()	executes a rollback in the database x.	
x.setProperty(s1, s2)	sets the property s1 to the value of s2 in a database x. Returns true, if the setting has taken place.	Boolean

Table 30 - Methods of objects of the type Statement

Method name	Description	Result type
x.close()	closes the statement x.	
x.execute (s1, ...)	executes the statement s1, ... and returns true if it has been executed.	Boolean
x.hasItem(s1, ...)	returns true if there exists an item according to parameters s1, ...	Boolean
x.isClosed()	returns true when statement x has been closed.	boolean
x.query(s1, ...)	executes a query with parameters s1, ..., and returns a ResultSet with the result.	ResultSet
x.queryItem (s1, s2, ...)	executes a query on item s1, with parameters s2, and returns a ResultSet	ResultSet

	with the result.	

Table 31 - Methods of the type String

Method name	Description	Result type
x.charAt(n)	Get character from the string x at position n	char
x.contains(s)	returns true if the string x contains a string s.	Boolean
x.containsi(s)	returns true if the string x contains a string regardless of upper/lower case.	Boolean
x.cut(n)	truncates the string x to the maximum length n.	String
x.endsWith(s)	returns true if the string x ends with a string s.	Boolean
x.endsWithi(s)	returns true if the string x ends with a string s regardless of upper/lower case.	Boolean
x.equals(s)	returns true if the string x has the same value as s.	Boolean
x.equalsIgnoreCase(s)	returns true if the string x has the same value as s ignoring the case.	Boolean
x.getBytes()	Returns an array of bytes that is created from the string x (uses the current system encoding)	Bytes
x.getBytes(s)	returns an array of bytes that is created from the string s (according to the code page that is named s)	Bytes
x.indexOf()	returns the position of the occurrence of the string s in the string x. The position starts from 0, and if the s string does not exist in the string x, it returns-1.	Int
x.indexOf(s, n)	returns the position of the occurrence of the character string s in the string x starting with position n. The position starts from 0, and if the string s does not exist in the string x after position n, it will return-1	Int
x.isEmpty()	returns true if string s is empty.	boolean
x.lastIndexOf(s)	returns the position of the last occurrence of the string s in the string x. If the string is not found, it returns-1.	Int
x.lastIndexOf(s, n)	returns the position of the last occurrence of a string s in the string x starting from the position n. If the string is not found, it returns-1.	int
x.length()	returns the number of characters in the string x.	Int
x.startsWith(s)	returns true if the string x starts with the string s.	boolean
x.startsWithi(s)	returns true if the string x starts with the string s without respect to upper/lower case.	boolean
x.substring(n)	returns part of the string x beginning from position n to the end.	String
x.substring(n1, n2)	returns part of the string x starting from position n1 to position n2.	String
x.toLower()	returns the string created from x where all uppercase letters in string x are replaced with lowercase letters.	String
x.toUpper()	returns the string created from x where all lowercase letters in string x are replaced with uppercase letters.	String
x.trim()	returns a string in which all-white spaces at the beginning and end of the string x are removed.	String

Table 32 – Methods of objects of the type uniqueSet

Method name	Description	Result type
	the instance of an uniqueSet object is created by the declaration statement. See 0.	uniqueSet
x.CHKID()	checks if the parsed value already exists as an entry in table x. If not, an error is reported.	ParseResult
x.CHKIDS()	checks if all parsed values from the list (separator is whitespace) already exist in an entry in table x. If not, an error is reported.	ParseResult

x.ID()	sets the parsed value to table x. If the value already exists an error is reported	ParseResult
x.IDREF()	checks if the parsed value exists a value in an entry in table x. If not, an error is reported either when the scope of x ends or when the method x.CLEAR() has been invoked (so the occurrence of parsed value may be set after this method was invoked).	ParseResult
x.IDREFS()	Checks if all parsed values from the list (separator is whitespace) already exist as an entry in table x. I. If not, an error is reported either when the scope of x ends or when the method x.CLEAR() has been invoked (so the occurrence of parsed value may be set after this method is invoked).	ParseResult
x.SET()	sets the parsed value to table x. If the value already exists in the table an error is NOT reported! (i.e. the value may be set more times)	ParseResult
x.CLEAR()	reports error messages if in table x are unresolved references (by methods IDREF and IDREFS). After errors are reported all entries of table x are cleared.	
x.checkUnref)	reports error messages if in table x are items that are not referred to in the scope where the method is invoked.	
x.getActualKey()	returns the value of the key of the last saved item.	uniqueSetKey
x size()	returns the number of items in the x.	Int
x.toContainer()	returns Container with items created from items in x.	Container

Table 33 – Methods of objects of the type uniqueSetKey

Method name	Description	Result type
resetKey()	Sets the value of the actual key of the uniqueSet table to the value from this object.	

Table 34 – Methods of objects of the type XmlOutStream

Method name	Description	Result type
new XmlOutStream(s)	creates an instance of the XmlOutStream object according to argument s. The s may describe a file.	XmlOutStream
x.setIndenting(b)	if b is true, the writing is done with indentation.	
x.writeElementStart(e)	writes the start of an element e (name, attributes).	
x.writeElementEnd()	writes the end of the actual element.	
x.writeElement(e)	writes the element e.	
x.writeText(s)	writes the text s.	
x.close()	Closes the writer.	

4.1.20 Mathematical methods

In the X-script it is possible to use the mathematical methods from the library's "java.lang.Math". These methods are implemented both for the X-script type "float" and "int", which, if necessary, converts it to "float". The result is either a "float" or "int" depending on the type of method. Note that the type "int" in the X-script is always implemented as the Java "long" and the "float" type is always implemented as the Java "double".

Table 35 - Methods of mathematical functions (taken from the class java.lang.Math)

Method name	Description	Result type
abs(x)	see method java.lang.Math.abs	int or float
acos(x)	see method java.lang.Math.acos	Float
asin(x)	see method java.lang.Math.asin	Float

atan(x)	see method java.lang.Math.atan	Float
atan2(x, y)	see method java.lang.Math.atan2	Float
cbrt(x)	see method java.lang.Math.cbrt	Float
ceil(x)	see method java.lang.Math.ceil	Float
cos(x)	see method java.lang.Math.cos	Float
cosh(x)	see method java.lang.Math.cosh	Float
exp(x)	see method java.lang.Math.exp	Float
expm1(x)	see method java. lang.Math.expm1	Float
floor(x)	see method java. lang.Math.floor	Float
hypot(x, y)	see` method java. lang.Math.hypot	Float
IEEERemainder(x, y)	see method java.lang.Math.IEEERemainder	Float
log(x)	see method java.lang.Math.log	Float
log10 (x)	see method java.lang.Math.log10	Float
log1p(x)	see method java.lang.Math.log1p	Float
max(x, y)	see method java.lang.Math.max	float or int
min(x, y)	see method java.lang.Math.min	float or int
pow(x, y)	see method java.lang.Math.pow	Float
rint(x)	see method java.lang.Math.rint	Float
round(x)	see method java.lang.Math.round	Int
signum(x)	see method java.lang.Math.signum	Float
sin(x)	see method java.lang.Math.sin	Float
sinh(x)	see method java.lang.Math.sinh	Float
sqrt(x)	see the method java.lang.Math.sqrt	Float
tan(x)	see method java.lang.Math.tan	Float
tanh(x)	see method java.lang.Math.tanh	Float
toDegrees(x)	see method java.lang.Math.toDegrees	Float
toRadians(x)	see method java.lang.Math.toRadians	Float
ulp(x)	see method java.lang.Math.ulp	Float

For working with the type Decimal (the value is internally implemented as java.math.BigDecimal) the available methods in the X-script are:

Table 36 - Methods of mathematical functions (taken from java.math.BigDecimal)

Method name	Description	Result type
x = decimalValue(v)	constructor; v can be int, double, String, or Decimal.	Decimal
abs(x)	see java.math.BigDecimal.abs	Decimal
add(x, y)	see java math.BigDecimal.add	Decimal
compare(x, y)	see java.math.BigDecimal.compare	Int
divide(x, y)	see java.math.BigDecimal.divide	Decimal
equals(x, y)	see java.math.BigDecimal.equals	Boolean
intValue(x)	see java.math.BigDecimal.intValue	Int
floatValue(x)	see java.math.BigDecimal.floatValue	Float
max(x, y)	see java.math.BigDecimal.max	Decimal
min(x, y)	see java.math.BigDecimal.min	Decimal
movePointLeft(x, n)	see java.math.BigDecimal.movePointLeft	Decimal
movePointRight(x, n)	see java.math.BigDecimal.movePointRight	Decimal
multiply(x, y)	see java.math.BigDecimal.multiply	Decimal

negate(x)	see java.math.BigDecimal.negate	Decimal
plus(xy)	see java.math.BigDecimal.plus	Decimal
pow(x, n)	see java.math.BigDecimal.pow	Decimal
remainder(x)	see java.math.BigDecimal.remainder	Decimal
round(s)	see java.math.BigDecimal.round	Decimal
scaleByPowerOfTen(x, n)	see java.math.BigDecimal.scaleByPowerOfTen	Decimal
setScale(x, n)	see java.math.BigDecimal.setScale	Decimal
stripTrailingZeros(x)	see java.math.BigDecimal.stripTrailingZeros	Decimal
subtract(x, y)	see java.math.BigDecimal subtract	Decimal
ulp(x)	see java.math.BigDecimal.ulp	Decimal

4.1.21 External methods

In addition to the implemented methods and declared methods, it is possible to call Java external methods. The external method must be declared as public and static. These methods must meet certain conventions. There are three ways of passing parameters to external methods:

a) External methods with parameters that correspond to the list of parameters of the calling method in the X-script take the form of a usual static method. The types of values of parameters are determined by the table below:

Table 37 - Value types passed to external Java methods

Name of type in the X-script	The type passed to the external method
Boolean	logical value (java.lang.boolean)
Datetime	date and time (java.util.Calendar)
Float	floating point number (java.lang.double)
Int	integer (java.lang.long)
Regex	compiled regular expression (java.util.regex.Pattern)
RegexResult	result of the regular expression (java.util.regex.Matcher)
String	character string (java.lang.String)
Element	element (org.w3c.dom.Element)
Bytes	byte arrays (java.lang.byte[])
Container	(org.xdef.XDContainer)

An example of external methods in Java:

```
public static boolean tab(String tabName, String colName, String value) {
...
}

public static void error (int code) {
...
}
```

In the X-script, these methods can be called, for example:

```
<elem value="required tab('table','column', getText()) onFalse taberr(123);" />
```

b) external methods with an array of parameters. These methods have a single parameter with an array of items of the type "org.xdef.XDValue[]". The number of items (i.e. number of actual parameters) in the array depends on the specific method call in the X-script. Note: this type of method is selected by the compiler of X-definition when the method with the corresponding list of parameters is not found. An example of an external Java method declaration:

```
public static boolean MyMethod(XDValue [] params) {
  int numParams = params.length;
  for (int i = 0; i < params.length; i++) {
        ...
  }
...
```

```
}
```

c) The external methods with the first parameter of the type XXNode, XXElement, or XXData. These methods allow you to use the methods implemented on the value of this parameter. Since the parameter is the control object of the X-definition engine connected with the processed XML data you can access this way to the internal values of the X-definition process. In the X-script, you do not need to specify this parameter. The compiler substitutes it automatically. Example:

```
public static XDParseResult MyDataType(XDData data) {
  String s = data.getText();
  XDParseResult result = XDFactory.createParseResult (s);
  If (error detected in s) {
    result.error ("Error message ...");
  }
  return result;
}
```

In the X-script, this method is called without the first parameter, which is set automatically: `x = "required MyDataType()"`

External methods must be declared in xd:declaration with the command "external method", followed by a definition of the external method, which consists of the following parts:

- The result value type is a Java name of the result - e.g. "String", "long", "XDService" etc.
- The qualified name of the method, e.g. "com.myproject.MyClass.myMethod".
- In parenthesis, the list of parameter types is separated by commas, e.g.

  "(cz.syntea.xd.proc.XXElement, java.lang.String, java.lang.long)". The list may be empty. Note the names of some packages such as “java.lang", and "org.xdef” is default and can be omitted. So, you can write: "(XXElement, String, long)".
- maybe followed by an optional key name "as" followed by the alias name of the method (though this name of the method will be accessed in the X-script). If the alias name is not specified, then just the name of a method is used in the X-script, as declared in the Java class (not the qualified name). So, to declare different external methods from different classes but with the same names and same parameters, you MUST use the alias name.
- the description is terminated by a semicolon.

Example:

```
<xd:declaration>
  external method void com.project.MyClass.myMethod (XDValue[]);
  external method XDParseResult cz.project.classes.MyType(XDData);
  ...
</xd:declaration>
```

If there are more methods, you can write them in the description of the block {between the curly brackets}:

```
<xd:declaration>
  external method {
    void com.project.MyClass.myMethod (XDValue[]);
    XDParseResult cz.project.classes.MyType(XDData);
    ...
  }
</xd:declaration>
```

4.1.22 Options

The part of the X-script in which are written options is optional. If it is omitted, the default values are used or the values of options that are defined in the X-definition that was used at the start of processing (the root element. If it is written within the model of an element, attribute, or text node, it sets the appropriate values according to this specification.

The specification of options is introduced with the keyword "options" which is followed by a list of names of options, separated by a comma. The names of the options are in the following table:

Table 38 - Options

Option name	Description
acceptEmptyAttributes	the empty attributes are copied from the input (regardless of their declared type).
preserveAttrWhiteSpaces	superfluous spaces in attributes are left. The default value.
preserveComments	comments are copied into the resulting document (in the validation mode). This option is only allowed in the header X-definition.
preserveEmptyAttributes	the empty attributes are left (only if the attribute is not declared as optional).
preserveTextWhiteSpaces	superfluous spaces in text nodes are left.
ignoreAttrWhiteSpaces	not significant extra spaces in attributes are removed before further processing.
ignoreComments	the comments are ignored. This option is only allowed in the header of the X-definition. This is the default value.
ignoreEmptyAttributes	the empty attributes (where the length of the value is zero) are ignored (before the operations are made to remove the white spaces). This is the default value.
ignoreEntities	the unresolved external entities in DOCTYPE in parsed XML data are ignored. This option is possible to declare only in the x-script of the X-definition header.
ignoreTextWhiteSpaces	superfluous spaces in text nodes are removed before further processing.
preserveAttrCase	the low/capital letters in the attribute remain unchanged. This is the default value.
preserveTextCase	the low/capital letters in the text node value remain unchanged. This is the default value.
setAttrUpperCase	letters in attributes are set to uppercase before further processing.
setTextLowerCase	letters of the value of a text node before further processing are set to lowercase.
setTextUpperCase	letters of the value of a text node before further processing are set to uppercase.
trimAttr	whitespaces at the beginning and end of the attribute value are removed before further processing. This is the default value.
noTrimAttr	whitespaces at the beginning and end of the attribute value are left.
trimText	whitespaces at the beginning and end of the text node values are removed before further processing. This is the default value.
noTrimText	whitespaces at the beginning and end of the text node value are left.
moreAttributes	in the element are allowed even undeclared attributes. These attributes are copied without change to the current element.
moreElements	even undeclared elements are allowed in the element. These elements are copied without change to the current element.
moreText	even undeclared text nodes are allowed in the element. These nodes are copied without change to the current element.
clearAdoptedForgets	if this option is specified in the X-script of an element, all actions "forget" are ignored for all nested elements and their descendants.
ignoreEntities	the option can be declared only in the X-script of the X-definition header and it causes files with external entities (in the DTD specification) to be ignored. This option is taken from the X-definition which was used for processing the root element.
resolveEntities	the option can be declared only in the X-script of the X-definition header and it causes files with external entities (in the DTD specification) to be processed. This option is taken from the X-definition which was used for processing the root element. This is the default value.
resolveIncludes	the option can be declared only in the X-script of the X-definition header and it causes links to external data with the elements (http://www.w3.org/2001/XInclude) to be processed. This is the default value.
ignoreIncludes	the option can be declared only in the X-script of the X-definition header and it causes links to external data with the elements (http://www.w3.org/2001/XInclude) to be ignored.
acceptQualifiedAttr	the attributes which are declared without the namespace URI are also accepted with the namespace (and with the prefix) of the parent element. This is the default value.
notAcceptQualifiedAttr	the qualified attribute is not allowed

nillable	the element can be empty if it has a qualified attribute "nill " specified with the value "true". The namespace of the attribute must be: "http://www.w3.org/2001/XMLSchema-instance". This option allows compatibility with the "nillable" property in the XML schema.
noNillable	element is not "nillable". This is the default value.
acceptOther	when validating, XML instances of undeclared objects in the model of an element are inserted into the result. By default, this option is not set.
ignoreOther	when validating, XML instances of undeclared objects in the model of an element are ignored. By default, this option is not set.
cdata	this option causes a text node to be generated as a CDATA section. This option is only permitted in the X-script of text nodes. By default, this option is not set.

Example:

```
xd:script = "options ignoreAttrWhiteSpaces, ignoreTextWhiteSpaces, preserveEmptyAttributes"
```

Note: Option "noTrimText" should be used carefully. For example. for the following X-definition:

```
<xd:def xmlns:xd="http://org.xdef/xdef/4.2" xd:name="a" xd:root="E">
 <E xd:script="options noTrimText">
        <O xd:script="*" />
        optional string();
 </E>
</xd:def>
```

the following input data will be validated incorrectly:

```
<E>
   <O>
</E>
```

The reason for this is the fact that after the initial element <E> is an empty line (before the process of validation the empty lines are not removed due to the option "noTrimText"). The empty line, therefore, is seen as a value of a text node and the engine of X-definition expects the model of a text node that does not exist and therefore reports an error. X-definition, therefore, understands the above input data if you select the "noTrimText" option as follows:

```
<E>
   optional text
   <O>
   optional text
</E>
```

4.1.23 The references to the object of X-definitions

In the X-script of a model, it is possible to insert a link to other models, groups, etc. Links make the source code of the X-definition clearer and easier to maintain in large projects.

4.1.23.1 Reference to a model of an element

If we describe a reference of a model, we can extend it with the X-script sections, and we can also add the specification of attributes that are not in the model. The unspecified attributes and sections of the X-script are taken from the referenced model. The reference starts with the keyword “ref" and follows the specification of the location of the target. Note that if the target is in the same X-definition as the reference itself, you can omit the name of the X-definition in the specification of the target location.

```
"ref"  X-definition name “#" model name
```

Example:

```
<Person name = "required string" >
  <Address xd:script = "ref Address" />
</Person>

<Company title = "required string" >
  <Address xd:script = "ref Address" />
</Company>

<Address street = "required string"
        number = "required int()"
        town = "required string"
        ZIP = "required num(5)"
```

```
        />
```

If the inner elements or text values are declared in an element with a link, the referenced models are inserted:

```
<xd:def xmlns:xd = "http:/org.xdef/xdef/4.2"
        xd:name = "Subject"
        xd:root = "Company | Person | Address ">

  <Company xd:script="ref Address ">
    <Name>required string (1.30)</Name>
  </Company>

  <Person xd:script="ref Address"></Person>
    <FirstName>required string (1.30)</FirstName>
    <LastName>required string (1.30)</LastName>
  </Person>

  <Address Street = "required string(1,30)"
          Number = "optional int ()"
          City = "required string (1.30)"
          ZIP = "optional num (5)"
  />
</xd:def>
```

Note that the element <Name> is added to the element <Company>, and the list of attributes is taken from the element <Address>.

4.1.23.2 R Reference to a sequence of descendants of the model of an element

In addition to links to the models of the elements, it is possible to refer to a sequence consisting of the descendants of an element. Take, for example:

```
<Marriage date="required date">
   <xd:includeChildNodes ref="Couple"></xd:includeChildNodes>
</Marriage>

<Couple>
  <xd:mixed>
   <Husband xd:script="ref Person" />
   <Wife xd:script="ref Person" />
   </xd:mixed>
</Couple>
```

A description of the child element <Marriage> is taken from the model element <Couple>.

4.1.24 Comparison of the structure of models

If you want to compare the structure of a model with another model, you write the keyword "implements" or "uses" in the X-script of the model and add a reference to the model, with which it should be compared. It only compares quantifiers and data types of the model in the link, the other sections are ignored.

In the case of "implements", the name of the compared model must match:

```
<xd:def xd:name="A" ...>
  <Company xd:script = "implements B#Company"
      ... attributes >
          <Child nodes ...
  </Company>
</xd:def>

<xd:def xd:name="B" ...>
  <Company xd:script="ref Address">
          <Child nodes ...
  </Company>

</xd:def>
```

In the case of "uses", the name of the model and the compared model can be different. However, the structure must be the same. In addition, if "uses" is specified in the model, then if a validation method is not declared for a value then this method copies from the referenced model.

```
<xd:def xd:name="A" ...>
  <Company xd:script = "uses B#Object"
     Street = "required"
```

```
     Number = "optional"
     Town = "required"
     ZIP = "optional" >

        <Name>required</Name>
  </Company>
</xd:def>

<xd:def xd:name="B" ...>
   <Object Street = "required string(1,30)"
           Number  = "optional int()"
           Town = "required string(1,30)"
           ZIP = "optional num(5)" >
          <Name> required string(1,30) </Name>
  </Object>
</xd:def>
```

If the validation section in the X-script of an attribute or a text node is missing, it is taken from the referred model. Example:

```
<A xd:script= "uses B"
   Attr1 = ""
   Attr2 = "string(10,20)"/>

<B Attr1 = "optional int(1,2)"
   Attr2 = "string(10,20)"/>
```

The validation section of the attribute Attr1 from model B is copied into model A. Therefore, it will be

5 “optional int(1,2)”. X-definition processing modes

There are two different modes of functionality for the X-definition process. The first mode is called "v**alidation mode**". This mode parses the input data and checks occurrences of objects according to the selected X-definition and parses string data with the validation methods. In this mode, the processor is controlled by the input data and it finds appropriate parts of models in the pool of X-definition.

The second mode is called "c**onstruction mode**". This is not controlled by the input data, but by the X-definition itself. The processor in this case processes the X-definition, which acts as the "cookbook" according to which the resulting object is constructed.

The validation mode is useful for the validation and processing of input data. The construction mode serves for the construction of resulting data or the transformation of some input data.

5.1 Validation mode

In this mode, the process is controlled by the input data. If the input data are well-formed XML documents, the process sequentially parses the input data and invokes actions in different events or stages of processing. The input data may also be passed to the processor in the form of org.w3c.dom.Element or org.w3c.dom.Document. In this case, the actions are invoked in the sequence given by recursive processing of the document or element tree. The algorithm of processing is as follows:

1) **Starting the processing of the Root Element.** At the beginning of processing, the processor only knows the name of the root element of the XML document and its attributes. The X-definition processor looks for the model element for the root element. Events that may occur are: “init“ (the action is invoked before the following processing) and “onIllegalRoot“ (if the element model is not found in the xd:root list). You can describe these actions in the header of the X-definition in the attribute xd:script. According to the model element, the next process is continued.

2) **Processing of Elements**. At the moment an element is recognized, a new element is created with the specified name, and then the element is searched for within the X-definition. When the description is found, the processor checks if the occurrence of the element exceeds the limit of the occurrence. If yes, the event “onExcess“ is activated. If a description of the element is not found, or if the element is described as illegal, the event “onIllegalElement” occurs.

3) **Processing of the list of attributes**. In this step, the list of attributes in the element is compared to the list of attributes in X-definition. Among the events that may result is "onIllegalAttr" (if the attribute is not described in X-definition or if it is described as “illegal“.) If a description of the attribute is found, then the validation code is invoked to check the content of the attribute value. If the result of validation is "true" the event "onTrue" occurs and the resulting attribute value is written to the resulting element. If the validation result is "false" then the event "onFalse" occurs. If an attribute is described in X-definition but is missing in the data, the event “onAbsence” occurs.

4) **Processing of the element contents (child nodes)**. After the attribute list is processed, the event “onStartElement“ occurs. In this event, the name of the element and all attributes are processed. After the event “onStartElement“ is complete, the processor continues to step 5 or step 6.

5) **Processing of child elements.** When an element of the data contains child elements, the process continues to step 2.

6) **Occurrence of text in a child node**. If a text child node occurs, the processor continues to proceed in the manner of attributes. If the text node is described as illegal, the event “onIllegalText“ occurs (instead of “onIllegalAttr“). Other events are the same. **Note** that text node values are available after all entity references are resolved and after all the adjacent text values and CDATA sections are connected to one text value.

7) **End of processing of an Element.** After all child nodes of the element are processed, the processor checks the minimum occurrence limit specified by X-definition. If the minimum limit for an element is not reached, the event “onAbsense“ occurs. After the minimum limits are checked, the event "finally" occurs. After the event “finally“, if described, the method “forget“ is invoked, which removes the contents of the processed element

from the XML tree. (The counter of the number of occurrences of nodes remains unchanged, even if it is "forgotten").

***Note**: If the XML data are processed in the source format, the event "onXMLError" may occur if the processor finds that the input XML data are not well-formed. In X-definition is possible to invoke the action associated with this event and the user can decide whether the processing will continue or terminate. If no XML error action is described, then the processor continues when the parser finds light errors (as described in the XML specification).*

You can try validation mode on the web at: https://xdef.syntea.cz/tutorial/examples/validate.html

5.2 Construction mode

In the construction mode, the process is controlled by X-definition (not by the input data as the validation mode). In the beginning, a model according to which the construction will start must be specified. The child models are processed recursively. For each object described in the model, the construction is provided according to the internal value we call "context". This happens as the event "create". The value of the context may be set by the action of the event "create" (however, if the specification of the action is missing, a default value may be used). In the "create" action it is possible to set data for the creation of each object from the model. Introduction to how to use the construction mode of the X-definition is available on:

http://xdef.syntea.cz/tutorial/en/userdoc/xdef-4.2_construction_mode.pdf.

The construction mode can be used e.g. for the transformation of input data to another structure. In the construction mode, the events "onAbsence", "onExcess", "onIllegalElement" "onIllegalAttr", or "onIllegalText" etc. should not happen (if yes, then it is an error of the program). The processor proceeds by the following steps:

1) **Selection of the root element model.** The model for the construction of the root element is specified in the method "xcreate" which starts the process of construction. If the model is not found then the event "onIllegalRoot" occurs. Otherwise, the processor continues to step 2.

2) For all objects specified within X-definition, the action "create" is performed at the moment the processor in its recursive processing encounters a new object. The action "create" returns a value of the context used for the construction of the object. If no action "create" is specified the default value of the context is used. After the object is constructed, all the events as described in the validation mode may occur. The sequence of actions is given by the processing of the tree described in the model, starting with the root element model.

In walking in the tree of the model, the element with attributes will first be constructed and then the child nodes will be constructed.

The construction of the objects described in the model depends on the context.

The context may be different types of values:

1. null
2. boolean
3. int
4. String
5. Element
6. Container
7. ResultSet
8. any other value will be converted to a string (or null value if it is null)

From the value types "Container" and "ResultSet" an iterator is internally created, which automatically returns the entries from the value. Note that if the result of XPath is a NodeList, it is converted to the Container. In the same way, the sequence result of XQuery is converted to the Container object. The ResultSet returns rows from a table.

You can try construction mode on the web at: https://xdef.syntea.cz/tutorial/examples/compose.html

5.2.1 Construction of element

The constructed element is given its name according to the name of the model. So, the only thing necessary to know is whether to construct it or not. If yes, then an empty element with the given name and namespace is

constructed. The element is constructed if, and only if, the maximum of the quantifier has not been reached and the value of the context type is:

1. boolean and it is not false
2. integer and the maximum number of instances of this element has not yet been created (if 0 => do not create)
3. the iterator created above the sequential part of the Container has another next item
4. the Resultset has another next row
5. any other value is not null

After the element has been constructed, the context remains available for the construction of child objects.

5.2.2 Construction of attributes

Attributes are created from models of attributes of the model of the element. The name and namespace of an attribute are created according to the model. The value is created from the string created from the value from the context. An attribute is created if the context is not null and If the context is

1. boolean and it is not false, then it is created only if a default value is specified in the X-script
2. String, the value is set to the attribute
3. Element, which has an attribute with the same name and namespace
4. Container, whose named value is the same as the name of the attribute model (the value is converted to String)
5. ResultSet, the actual row has an entry where the column name is equal to the name of the attribute (case insensitive)
6. any other value is converted to the String

5.2.3 Construction of text nodes

The text node is created if the context is not null and If the context is

1. boolean and it is not false, then it is created only if a default value is specified in the X-script
2. String, the value is set as the value of the text node (only if the string is not empty)
3. Element, which has a text node (the value of the text is used)
4. Container, if a string value exists in the sequential part (the first occurrence is used)
5. ResultSet, the concatenated value of entries is taken from the actual row
6. any other value is converted to the String and then used for the construction of the text node

6 JSON, XON, YAML, Windows INI, Properties

6.1 Models of JSON

In version 4.2 it is possible to describe and process JSON (and XON) data. A document is described in the text content of the element “xd:xon” which is specified as a child member of X-definition. Each JSON model must have a unique name which is specified in the attribute ”xd:name”. The properties of values of a JSON document are described similar way as in models of XML elements. The result of a JSON model is the XON object.

6.2 JSON simple values

JSON Simple values are strings, numbers, booleans, or nulls. The simple values parsers are

- **jstring** name of the parser of JSON string values
- **jnumber** name of the parser of JSON number values
- **jboolean** name of the parser of JSON Boolean values
- **jnull** name of the parser of JSON null values

Except for the parsers above you can use any of the XML value parsers, such as date, dateTime, base64Binary, hex64Binary, duration, long, decimal, etc. The result is an XON object (it can be converted to JSON and those parsed values are then converted to strings). The result of the null value is XON null. The results of XML numeric types (integer, float, decimal, etc.) are JSON numbers.

6.3 What is XON?

XON is an extension of JSON format so that it supports all data types described in X-definitions. The JSON format is fully supported also in XON. Moreover, in addition to JSON, it enables the description in detail of the JSON type Number as types in X-definitions. So as JSON, the XON format describes arrays and maps (called in JSON “objects”). Java types created from parsed JSON source data are:

- **java.util.Map<java.lang.String, java.lang.Object>** **named pairs of the JSON objects (maps).**
- **java.util.List<java.lang.Object>** **items of the JSON arrays.**
- **java.lang.String** **JSON string items**
- **java.lang.Boolean** **JSON Boolean items**
- **java.lang.Number** **JSON numeric items**

XON extends JSON data types. It allows the following types which are converted to the following Java objects:

Java type	Format	Examples
org.xdef.sys.SDatetime	**d ISO datetime**	**d2021-03-10T23:55Z, D13:55+02:00, d—05**
org.xdef.sys.SDuration	**ISO duration**	**P1Y1M1DT1H1M1.12S, -P2Y**
org.xdef.sys.EmailAddr	**e"email address"**	**e"John Brown <john@brown.com>"**
org.xdef.sys.GPSPosition	**g(gps notation)**	**g(51.52, -0.09, 0, London)**
java.util.Currency	**C(ISO 4217 code)**	**c(USD)**
org.xdef.sys.Price	**p(price)**	**p(12.50 CZK)**
java.lang.byte[]	**b(base64)**	**b(HbRBHbRBHQw=)**
java.lang.Character	**c"character notation"**	**c"x", c"\n", c"\u0007"**
java.lang.URI	**u"uri"**	**u"https://org.xdef/ver1"**
java.net.InetAddress	**/inetAddr**	**/129.144.52.38), /1080:0:0:0:8:800:200C:417A**

Note that GPS position must contain latitude and longitude and there may follow an optional item altitude and a name of the place. If the name contains other characters it must be recorded between quotation characters.

The Number value distinguishes more types and it can be written in the following way:

- **java.lang.Byte** **number "b"** **123b**
- **java.lang.Short** **number "s"** **123s**

-	**java.lang.Integer**	**number "i"**	**123i**
-	**java.lang.Long**	**number**	**123 (Long.MININT .. Long.MAXINT)**
-	**java.lang.Float**	**number "f"**	**3.14f, 123f**
-	**java.lang.Float**	**Not a Number**	**NaNf**
-	**java.lang.Float**	**positive infinity**	**INFf**
-	**java.lang.Float**	**negative infinity**	**-INFf**
-	**java.lang.Double**	**number "d"**	**(any number and "d")**
-	**java.lang.Double**	**Not a Number**	**NaN**
-	**java.lang.Double**	**positive infinity**	**INF**
-	**java.lang.Double**	**negative infinity**	**-INF**
-	**java.math.BigInteger**	**number "N"**	**123N, -1N**
-	**java.math.BigDecimal**	**number "D"**	**0D, -0.00314D, -3.14e-3D**

If there is no letter after an integer number and if it does not exceed the range of the Java Long type, then the type is "Long", otherwise it is "BigInteger". If there is no letter after a float point number, then the type is "Double. If the value name in the map is an XML NXName (specification of XML node name without colons), then it is possible to include this name (unlike JSON) without quotes. In XON may be written also comments as a text between "/*" and "*/" or after the "#" character up to the end of the line.

The first line of XON data may start with the "%" character and the specification of the encoding of data (default value is UTF-8).

Example of XON data:

```
%encoding="windows-1250" # the character set specification of this XON data
[ #### example of XON ####
  {
      a : "\u0007",  b : c"\n",  c : c"\"", d : c"\\",  e: c"x",  # Characters
      " name (with space)" : "xy"  # the name is not an XML name, it contains spaces
      Towns : [ # array with GPS locations of towns
          g(50.08, 14.42, 399, "Prague Old town") # the position name contains a space, so it must  be quoted
          g(48.2, 16.37, 151, Wien),
      ]
  },
  0, /* Long */
  1i, /* Integer */
  11s,  /* Short */
  -2b,  /* Byte */
  0.0,  /* Double */
  1d,  /* Double */
  INF,  /* Double (Positive Infinity) */
  NaN,  /* Double (Not a Number) */
  -3.14e-3f,  /* Float */
  -INFf,  /* Float (Negative Infinity) */
  -3.25D,  /* BigDecimal */
  2N,  /* BigInteger */
  C(USD),  /* Currency */
  p(12 USD),  /* Price */
  b(HbRBHbRBHQw=),  /* byte array (base64) */
  P1Y1M1DT1H1M1.12S, # Duration
  d2021-01-11,  /* date */
  d--11,  /* month */
  d2000Z,  /* year with zone */
  d2021-01-12T01:10:11.54012-00:01,  /* date and time (nanoseconds, zone) */
  99999999999999999999999999999999, /* BigInteger (the number is bigger than MAXINT) */
  /129.144.52.38,  /* inetAddr (IPv4) */
  /1080:0:0:0:8:800:200C:417A,  /* inetAddr (IPv6) */ "
] # End of XON example
```

6.4 Models of XON/JSON objects

Models of JSON objects are described directly as XON objects (because XON is an extension of JSON and because in JSON models all values are strings, you can describe JSON models either in JSON or in XON). The values of members (name-value pairs) are described in a similar way to the values of XML attributes or XML text nodes. Models are valid for data in both formats: JSON or XON.

Example:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/4.2" xd:root = "Person">
  <xd:xon name = "Person" >
    { Person : {
```

```
        ID :   "optional jnumber()",
        Name : "jstring()",
        Address : {
             Town: "jstring()",
             Street:  "jstring()",
             Number:  "jnumber()"
        }
      }
    }
  </xd:xon>
</xd:def>
```

Example of JSON data described by the X-definition above:

```
{ "Person": {
      "Name" : "Boris Jonson",
      "Address": {
          "Town":    "London",
          "Street":  "Downing street",
          "Number":  10
     }
   }
}
```

Note the member “ID” is missing since it is described as optional.

Note that the result of the validation process of JSON models is always an XON object. However, you can always convert it to JSON by the method “org.xdef.xon.XonUtils. xonToJson(Object x)”. All XON types, which are not supported by JSON (i.e. date, time, duration, email address, IP address, GPS, price, currency, byte[], char, or URI) are converted to the JSON string value.

6.5 Directives

It is possible to add directives in XON models in X-definitions. A directive starts with a "%" followed by the directive name. The following directives can be used to describe an XON model:

%script - adding an X-script to the model description

%oneOf - specification of model element variants

%anyName - the name of the named item in the map can be any

%anyObj - the element can be any XON object.

6.6 Specification of properties with %script directive

The properties of objects you can describe in the item designated by the directive “%script” followed by the “=” and a string value with the X-script. This item must be the first one before the description of other items. The syntax of value is the same as in xd:script in XML element models. In the following example the object containing the item “ID” is described as an optional one:

```
<xd:xon name = "Contract" >
  { Contract = {
    {%script = "optional; finally outln('ID=' + getValue());)",
      ID : "int()"
    },
    { Name : "string()", PersonalId : "int()" }
  }
</xd:xon>
```

Note that after the item “ID” is processed, the value of this item will be printed to the standard output.

6.7 XON/JSON arrays

You can specify the occurrence of values of JSON array items similar way as in the value description of XML models. The occurrence of items is specified in the value description. The following example of a JSON model of an array is the first item string followed by a minimum of 2 and a maximum of 3 integers. After the first string follows any number of objects with coordinates of points.

```
<xd:xon name = "array">
```

```
  [ "jstring()",
    "occurs 2..3 jnumber();",
    {%script = "occurs *", X: "jnumber", Y: "jnumber"}
  ]
</xd:xon>
```

The example of parsed data with the model above:

```
[ "Shape",
    123, 456, 789,
    {"X": 5,     "Y": 6},
    {"X": -15.5, "Y": 6e2}
]
```

6.8 Specification of properties of arrays with %script directive

Similarly, as in objects, the properties of arrays can be described using the "%script" directive. The following example is described the model of matrix 3 x 3 of floating-point numbers:

```
<xd:xon name = "matrix">
  [
    [%script = "occurs 3", "occurs 3 float()" ]
  ]
</xd:xon>
```

Note that the %script directive describes the occurrence of an array and the occurrence of the value describes the number of values.

The example of parsed data with the model above:

```
[
   [ 123.4, -456,  789.01 ],
   [ 1.4,   4.56,  7.8901 ],
   [ 1E2,   -6e-2, 0 ]
]
```

6.9 %oneOf directive

If the case an item has more variants you specify them as an array where the first item is %oneOf directive. Then the model chooses an item from the items following the first one. Example:

```
<xd:xon name = "genre">
   { "Genre":
      [%oneOf,
         "jstring()",
         ["occurs 2..* jstring()"]
       ]
   }
</xd:xon>
```

The data of "Genre" can be either:

```
{ "Genre": "classic" }
```

or:

```
{ "Genre": ["jazz", "pop"] }
```

6.10 %anyName directive of named items in the map

If the name of an item in a JSON/XON map is not known you can use the directive %anyName. Example:

```
<xd:xon name='Cities'>
{ "cities": [
    { %script = "occurs 1..*; ", /* specification of number of map items */
      %anyName: [%script = "occurs 1..*;", /* specification of number of named items in map */
        { %script = "occurs 1..*;",
          "to": "jstring();",
          "distance": "int();"
        }
      ]
    }
  ]
}
```

Valid data of “Cities” can be:

```
{ "cities": [
    { "Brussels": [
        {"to": "London", "distance": 322}, {"to": "Paris", "distance": 265}
      ]
    },
    { "London": [
        {"to": "Brussels", "distance": 322}, {"to": "Paris", "distance": 344}
      ]
    }
  ]
}
```

6.11 %anyObj directive

f there is a %anyObj directive at a location in the model, any object can be at that location. For example:

```
<xd:xon name = "avyarray">
   [ %anyObj ]
</xd:xon>
```

The entry describes an array whose items can be any XON objects.

6.12 Reference to the JSON model

From a JSON model it is possible to refer to other JSON models similar way as in XML models from the X-script:

```
<xd:xon name = "genres">
  [ {%script = "occurs 1..*; ref genre"} ]
</xd:xon>

<xd:xon name = "genre">
   { "Genre":
      [%oneOf,
         "jstring()",
         ["occurs 2..* jstring()"]
       ]
   }
</xd:xon>
```

6.13 YAML

YAML format is used often in configuration files. YAML objects are possible to convert to XON. Therefore, the description of YAML objects is possible to write as a JSON model. So, with the JSON model in X-definition, you can process input data in the YAML format and you can convert any XON object to JSON or YAML. Note that processing of YAML data requires adding the package “org.yaml.snakeyaml” to the classpath.

Let’s have the YAML data:

```
cities:
- from:
  - Brussels
  - {to: London, distance: 322}
  - {to: Paris, distance: 265}
- from:
  - London
  - {to: Brussels, distance: 322}
  - {to: Paris, distance: 344}
```

However, the model describing this data in X-definition is specified as JSON:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2" name="yamlExample" root="distances">
  <xd:xon xd:name="distances">
    { cities: [
        { %script = "occurs +",
          from: [
            "string()",
            { %script = "occurs +", to: "jstring()",distance: "int(0, 9999)" }
          ]
        }
      ]
```

```
    }
  </xd:xon>
</xd:def>
```

6.14 Properties and Windows INI

In X-definition you can describe models of data in the Properties and the Windows INI format. The Properties format is the sequence of named values and the Windows INI format starts with properties and follows named sections of properties (the named section starts with “[” name “]”). The value of a named item is a string and in the model, the value is described by X-script.

Example of INI data:

```
TRSUserName = John Smith
[User]
Home = D:/TRS_Client/usr/Smith
Authority=CLIENT
ItemSize=4000000
[Server]
RemoteServerURL=http://localhost:8080/TRS/TRSServer
SeverIP = 123.45.67.8
Signature = 12afe0c1d246895a990ab2dd13ce684f012b339c
```

X-definition of the Windows INI data:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2" name="iniExample" root="TRSconfig">
  <xd:ini xd:name="TRSconfig">
    TRSUserName = string()
    [User]
      Home = file()
      Authority = enum("SECURITY", "SOFTWARE", "CLIENT", "UNREGISTERED")
      ItemSize = int(10000, 15000000)
    [Server]
      RemoteServerURL = url()
      SeverIP = ipAddr()
      Signature = SHA1()
  </xd:ini>
</xd:def>
```

The result of processed data is again in the XON object which can be converted to the string in Windows INI format by the method “org.xdef.xon.XonUtils. xonToIni(Object xon)”

7 X-components

X-component is a source code of Java class generated according to a model of the element (similar way as JAXB). The description of generation X-components is written in the text of the element "xd:component". In X-component the values of attributes, text nodes, and child elements are accessible by getters and setters (the child elements are also represented by an X-component). User manual for X-components is available on:

http://xdef.syntea.cz/tutorial/en/userdoc/xdef4.2_X-component_eng.pdf

7.1 Values in X-component of getters in XON objects

Text values in the X-component are converted according to the data type as described in the table:

Table 39 - Conversion of X definition data types in X-component getters and XON values

Data type in X-definition	X-component
anyURI	java.net.URI
base64Binary, hexBinary, base64, hex	byte[]
boolean	`java.lang.Boolean`
byte, short, int, long, integer	`java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.math.BigInteger,`
char	`Java.lang.Character`
currency	`java.util.Currency`
datetime, xdatetime, gDate, gTime, gYear, gMoth, gYearMonth, gMonthDay	`org.xdef.sys.SDatetime` `(also available as java.util.Calendar, java.util.Date, java.sql.Timestamp)`
decimal, dec	`java.math.BigDecimal`
duration	`org.xdef.sys.SDuration`
Enum	`java.lang.String or user-declared Java enum`
float, double	`java.lang.Float, java.lang.Double`
emailAddr	`org.xdef.XDEmailAddr`
gps	`org.xdef.XDGPSPosition`
ipAddr	`java.net.InetAddress`
price	`org.xdef.XDPrice`
telephone	`org.xdef.XDTelephone`
other data types	`java.lang.String`

The values of child elements are represented as X-component objects. If the maximum number of the quantifier is higher than one then it is represented by java.util.List<correspondent X-component>. Also if there occurs more than one, the text value at one position will be represented as java.util.List<correspondent datatype>.

7.2 Access to values of X-component

The values of attributes, elements, and text values in the X-component, are accessible by the methods get***NAME*** and set***NAME***, where "***NAME***" is the name of the attribute or element in the X-definition. If the element or a value may have more occurrences, it creates an array of values (implemented using java.util.List), and instead of a setter, it generates the method add***NAME***. Because there are more possibilities to work with data types DateTime getters are generated in more variants: timestampOf***NAME***, calendarOf***NAME***, and dateOf***NAME***.

7.3 X-component commands

The commands are written as the text of the element "<xd:component>". This element can be specified as the direct child node of any X-definition, Each command is ended with the semicolon (";").

7.3.1 %class

This command specifies the generation of an X-component according to the model of an element. The "%class" keyword is followed by a fully qualified class name and by the keyword "%link" which specifies the X-position of a model in the set of X-definitions, from which the X-component is generated. If the X-component extends a Java class or it implements a Java interface then you can specify “extends“ SuperClassName and “implements” InterfaceName after the name of the Java class, where SuperClassName and InterfaceName must be a fully-qualified name. The syntax is the same as in the declaration of a class in the Java language.

7.3.2 %bind

With the "%bind" command can be set a new name of an item in an X-definition (the attribute model, element model, or text node model and the corresponding names of the getters and setters). The keyword "%bind" must be followed with the name that will be applied instead of the automatically generated one. After the specification of the name, the command continues with the keyword "%from" which is followed by the list of X-positions (separated by a comma) to which the statement relates. The same name can be used in more models. The getters and setters will automatically be adjusted to match the newly assigned name. If the generated Java class has an ancestor, it can be used in more models. The getters and setters will automatically be adjusted to match the newly assigned name. If the generated Java class has an ancestor, you can use the "%bind" command to bind the getter and setter defined in the ancestor. In this case, a given variable, including getters and setters will not be generated. It uses the implementation of those methods in the ancestor.

7.3.3 %interface

The command "%interface" is used when the (final) model takes the structure of another (referenced) model and, where appropriate, it also adds additional attributes, text values, or elements. To make the X-components generated from the models behave like the X-component created from the referenced ones, you can create an interface from the given model. This interface can then be added for the generating of final models. The interface command starts with the keyword %interface, which is followed by the fully qualified name of the interface and by the keyword %link followed by the X-position of the model in the project.

7.3.4 %ref

It often happens that the project (XDPool) is generated from more X-definitions. In this case, the X-component is generated from the given X-definition but XDPool is different (for example, there is an X-definition extra, missing, etc.) it can be used for the already created X-component. It is necessary to prevent its new generation (for example, if the X-component is located in another Jar file). Reference to the already generated X-component is provided by the command %ref with the fully qualified name of the already generated X-component and with the keyword %link with the X-position of the model in the XDPool.

7.3.5 %enum

If the X-definition is a data type enum is specified, its value in the X-component is represented by default as a String. However, in case we want to have a choice in the code only from the allowed values, the data value of the enum is possible to generate as a Java enum type. The data type must be defined in the X-script section <xd:declaration>. The enum will be generated by using the command %enum followed by the fully qualified name of the enum Java class and the name of the data type.

8 Invoking X-definitions from Java

The X-definition processor is implemented in the Java programming language. The executable code is distributed in the file:

"xdef.jar"

The Java documentation is distributed in the file:

"xdef-javadoc.jar"

First, you must compile the X-definition sources and create an instance of the class "org.xdef.XDPool" by invoking the static method "compileXD" of the class "org.xdef.XDFactory". The parameter list of this method allows you to compile a set of X-definitions (the project) from the files or URLs or input streams. Note that the file names may also contain the wildcards "*" and "?". The created object XDPool contains a set of compiled X-definitions.

You can modify parameters of the compilation of X-definitions and the processing by setting different properties to the parameter "props" as in the following example (it is also possible to set it via e.g. java.lang.System.setProperty() – for values see org.xdef.XDConstants).

Example of creating of XDPool:

```
String[] xdFiles; // the array of file names
Properties props = System.getProperties();
Class[] classes = null;
// create the pool of X-definitions
org.xdef.XDPool xp = org.xdef.XDFactory.compileXD(props, xdFiles);
```

Before running the validation mode or construction mode, you should also prepare the reporter that provides the recording of error messages to a log file:

```
org.xdef.sys.ArrayReporter reporter =
    new org.xdef.sys.ArrayReporter();
```

However, you must first create the object XDDocument from X-definition where the model of your data is described:

```
org.xdef.XDDocument xdoc = xpool.createXDDocument(xdName);
```

8.1 Execution of Validation Mode

The process of validation is started with the method "xparse" of the XDDocument object. This method requires you to specify the input XML data in the first parameter. A second parameter is a ReportWriter object, where the errors detected during the validation process are recorded. If this parameter is set to null, then if an error was reported the RuntimeException is thrown. Example:

```
import org.xdef.sys.ArrayReporter;
import org.xdef.XDDocument;
import org.xdef.XDPool;
import org.xdef.XDFactory;
import org.w3c.dom.Element;
...
String xdef1;
File xdef2;
URL xdef3;
InputStream xdef4;

...
XDefPool xpool = compileXD(props, xdef1, xdf2, xdf3, xdef4, ...);
....
// name of the X-definition with root model
String xdName;
// pathname of the XML data to be validated
String sourceFileName;
// reporter where error messages will be written (here to ArrayList in the memory)
ArrayReporter reporter = new ArrayReporter();
...
// prepare XDDocument
XDDocument xdoc = xpool.createXDDocument(xdName);
// validate input data
org.w3c.dom.Element el = xdoc. xparse(sourceFileName, reporter);
```

```
// check if the errors were reported
if (!reporter.errors()) {
   System.out.println("OK");
   ....
} else {
   System.err.println(reporter);
}
```

***Note**: You may use the library "org.xdef.sys.*", which contains a variety of methods for reporters, processing of reports, etc. A detailed description is available in the programming documentation. See also some useful programs providing different operations with X-definitions which are available in the library "org.xdef.util.*".*

8.2 Construction Mode

To invoke the construction mode, use the method "`xcreate`" of the `XDDocument` object. The typical usage requires the input data either in the source XML format or the object "`org.w3c.dom.Element`". The name of the element model according to which the result should be composed, may be passed to the method as a parameter as a string, or as the QName object. If the name is missing, the name of the root element of the input data is used. If you need to set the default context you can set it to the XDDocument by the method setXDContext before starting the process of construction. Example:

```
import org.xdef.sys.ArrayReporter;
import org.xdef.XDDocument;
import org.xdef.XDPool;
import org.w3c.dom.Element;
...
XDPool xpool ...
String xdName ...
String resultModelName;
String contextData;
...
ArrayReporter reporter = new ArrayReporter();
XDDocument xdoc = xpool.createXDDocument(xdName);
xdoc.setXDContext(contextData);
Element el = xdoc.xcreate(resultModelName, reporter);
if (!reporter.errors()) {
   System.out.println("OK");
} else {
   System.err.println(reporter);
}
```

To illustrate, here is an example of the X-definition, input data, and the result. X-definition:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2" xd:name="GenContract">
<xd:declaration>
<![CDATA[
   ParseResult pid() {
      String s = getText();
      ParseResult result = new ParseResult(s);
      if (!string(10,11))
         result.error('Incorrect length of PID');
      if (s.substring(6,7) != '/')
         result.error('Missing slash character');
      if (!isNumeric(cut(s,5)))
           result.error('Second part is not numeric');
      if (!isNumeric(s.substring(7)))
           result.error('First part is not numeric');
      return result;
   }
]]>
</xd:declaration>

<Contract cId = "required num(10)" >

  <Owner        Title       = "required string(1,30); create from('@title')"
                IC          = "required num(8); create from('@ic')"
                xd:script = "occurs 1; create from('Client[@role=\'1\']')"/>

  <Holder       Name        = "required string(1,30); create from('@name')"
                FamilyName  = "required string(1,30); create from('@familyname')"
                PersonalId  = "required $checkId(); create from('@pid')"
                xd:script = "occurs 1; create from('Client[@role=\'2\']')"/>
```

```
  <Policyholder Title = "required string(1,30);
                         create toString(from('@name')) + ' ' + from('@familyname')"
                IC    = "required num(8); create from('@ic')"
                xd:script = "occurs 1; create from('Client[@role=\'3\']')"/>

</Contract>

</xd:def>
```

Input data:

```
<Contract
  cId = "0123456789">
  <Client role  = "1"
          typ   = "P"
          title = "Company X Ltd"
          ic = "12345678" />
  <Client role  = "2"
          typ   = "O"
          typid = "1"
          name  = "Frantisek"
          familyname = "Novak"
          pid   = "311270/1234" />
  <Client role  = "3"
          typ   = "O"
          typid = "2"
          name  = "Frantisek"
          familyname = "Novak"
          pid   = "311270/1234"
          ic    = "87654321" />
</Contract>
```

Result:

```
<Contract cId = "0123456789">
  <Owner Title = "Company X Ltd"
         IC   = "12345678"/>
  <Holder Name       = "Frantisek"
          FamilyName = "Novak"
          PersonalId = "311270/1234"/>
  <Policyholder Title = "Frantisek Novak"
                IC    = "87654321"/>
</Contract>
```

8.3 Using XQuery

If the classpath does not have an accessible library with an XQuery implementation, the "xquery" method cannot be used. To use this method, you need to ensure that a library with an XQuery implementation is available in the classpath. The implementation might not be open source. Therefore, the implementation of XQuery in X-definition is based on the *saxonica* library. The free version of the necessary jar file from the *saxonica* are e.g.:

saxon-he-10.5.jar

and library for the XQuery/XPath3.1 interface:

saxon-xqj-10.5.jar

Both those jar files must be on the classpath!

The result of the "xquery" method is then an object of the "org.xdef.XDContainer" type in the X-definition (which is internally created from the result of the executed XQuery program, i.e. from "javax.xml.xquery.XQResultSequence").

Example X-definition:

```
<xd:def  xmlns:xd="http://www.xdef.org/xdef/4.2" xd:root="a" xd:name="TestXQuery">
  <xd:declaration>
    Container c = xquery("let $b := 'abcd'
             return (0 to string-length($b)) ! (substring($b,1,string-length($b) - .))");
  </xd:declaration>
  <a xd:script = "init {
         for (int i = 0; i LT c.getLength(); i++) {
           outln(c.item(i) + '.');
         };
       }" />
```

```
</xd:def>
```

The printed output from the given XML input data "<a./>“ will be:

```
abcd.
abc.
ab.
a.
.
```

8.4 Incremental writing of large XML files

If you need to write a large XML file, you can set the XDXmlWriter in the X-definition to write it. In the element models, invoke the write data method writeElementStart() from the XDXmlWriter object in the onStartElement section of the X-Script. In the section finally, you finish the element writing by calling the writeElementEnd() method. The “forget” statement ensures that the processed element is released from memory. If the element has no internal elements, call the writeElement() method in the ”finally” section.

Example of X-definition:

```
<xd:def xmlns:xd='http://www.xdef.org/xdef/4.2' root='a'>
  <xd:declaration>
    external XmlOutStream largeXml;
  </xd:declaration>
  <a x='?' xd:script='onStartElement largeXml.writeElementStart();
                      finally { largeXml.writeElementEnd();largeXml.close();}'>
    <b x='' xd:script='*; finally largeXml.writeElement(); forget'>
      <c xd:script='*;'>string();</c>
    </b>
    <d y='?' xd:script='*; finally largeXml.writeElement(); forget'/>
  </a>
</xd:def>
```

The external variable `largexml` you can set from the Java program:

```
  OutputStream outStr ... // output stream where to write XML document
  XDPool xpool ...
  XDDocument xdoc = xpool. createXDDocument();
  xd.setVariable("largeXml", XDTools.createXDXmlOutStream(outStr, "UTF-8", true));
...
```

8.5 JSON data

The processing of JSON data is similar to the processing of XML data. First, it is necessary to compile X-definitions and to create the XDDocument object. From the XDDocument it is possible in a Java code to invoke the method “jparse” (similar way as the xparse for XML data). See the example below.

X-definition:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2" xd:name="jsonTest" xd:root="distances">
  <xd:xon xd:name = "distances" >
     { "cities": [
        { :script= "occurs 1..*",  /* set occurrence of the item "cities" */
          "from": [
             "string()",
             { :script= "occurs 1..*",
               "to": "jstring()",
               "distance": "int()"
             }
       ]
        }
      ]
    }
  </xd:xon>
</xd:def>
```

JSON data:

```
{ "cities" : [
    { "from": [ "Brussels",
        {"to": "Paris",     "distance": 265},
        {"to": "Amsterdam", "distance": 173}
```

```
      ]
    },
    { "from": [ "London",
        {"to": "Brussels",  "distance": 322},
        {"to": "Paris",     "distance": 344},
      ]
    },
    { "from": [ "Paris",
        {"to": "Brussels",  "distance": 265},
      ]
    }
  ]
}
```

Java program:

```
import org.xdef.sys.ArrayReporter;
import org.xdef.XDDocument;
import org.xdef.XDPool;
...
XDPool xpool ...
File jsonData ...
...
XDDocument xdoc = xpool.createXDDocument("distances");
ArrayReporter reporter = new ArrayReporter();
Object json = xdoc.jparse(jsonData, reporter); // Result is the validated JSON object
if (reporter.errors()) { // reporter contains recognized errors
   System.err.println(reporter); // print errors
} else { // no errors found
   System.out.println("OK");
   ...
}
```

Note you can try to run JSON data on https://xdef.syntea.cz/tutorial/examples/json.html

8.6 YAML data

With X-definition it is also possible to work with data in YAML format if you use e.g. the snakeyaml library. With this library, you can convert the input data to JSON format and convert the output data back to JSON format. In the following example, we will use the data from the previous paragraph:

YAML data:

```
cities:
- from:
  - Brussels
  - {to: Paris, distance: 265}
  - {to: Amsterdam, distance: 173}
- from:
  - London
  - {to: Brussels, distance: 322}
  - {to: Paris, distance: 344}
- from:
  - Paris
  - {to: Brussels, distance: 265}
```

In the X-definition, the data will be described as JSON, as in the previous paragraph. The result will be in JSON format and you can convert it back to YAML again using the snakeyaml library. Example of Java code:

```
import java.io.Reader;
import java.io.StringWriter;
import org.xdef.sys.ArrayReporter;
import org.xdef.XDDocument;
import org.xdef.XDPool;
import org.xdef.json.JsonUtil;
import org.yaml.snakeyaml.Yaml;
...
XDPool xpool ...
Reader jsonData ...
...
XDDocument xdoc = xpool.createXDDocument("distances");
Object o = new Yaml().load(sourceYAML); // parse YAML source to object
String jsonData = JsonUtil.toJsonString(o, true); // convert parsed object to JSON string
ArrayReporter reporter = new ArrayReporter();
Object json = xdoc.jparse(jsonData, reporter); // Result is the validated JSON object
if (reporter.errors()) { // repoter contains recognized errors
   System.err.println(reporter); // print errors
```

```
} else { // no errors found
  StringWriter wr = new StringWriter();
  new Yaml().dump(json, wr); // convert JSON to YAML source  wr.close();
  String resultYaml = swr.toString();
  System.out.println(resultYaml);
}
```

Note you can try to run YAML data on https://xdef.syntea.cz/tutorial/examples/json.html

Appendix A: X-definition of X-definition 4.2

X-definition language enables the description of X-definition using X-definition. Note that macros mustn't be specified there and must be processed previously. Here is an X-definition of X-definition:

```
<!--
 The description of the XML document which must fit the X-definition 4.1
 specification.
 The meta namespace of X-definition ("METAXD" is http://www.xdef.org/xdef/4.1).
 The prefix of objects describing the X-definition 4.1 is XD4.1.
-->

<xd:def xmlns:xd  = "METAXD"
  name            = "Ver4.1"
  root            = "XD4.1:def | XD4.1:collection | XD4.1:lexicon
                     | XD4.1:declaration | XD4.1:BNFGrammar | XD4.1:component"
  xmlns:XD4.1     = "http://www.xdef.org/xdef/4.1"
  xmlns:w         = "http://www.xdef.org/xdef/4.1"
  w:metaNamespace = "METAXD" >

/*************************************************************************/
/*X-definition of X-definitions ver 4.1, metanamespace: METAXD, prefix: XD4.1 */
/*************************************************************************/

    <XD4.1:collection xd:script = "init NS = @metaNamespace
      ? (String) @metaNamespace : 'http://www.xdef.org/xdef/4.1';
      options moreAttributes"
    include       = "optional uriList; options acceptQualifiedAttr"
    metaNamespace = "optional uri; options acceptQualifiedAttr"
    xd:attr       = "occurs * getAttrName().startsWith('impl-');">
    <xd:mixed>
      <!-- Here may be objects from all versions of X-definition -->
      <XD4.1:def xd:script = "occurs *; ref XD4.1:def" />
      <XD4.1:declaration xd:script = "occurs *; ref XD4.1:declaration" />
      <XD4.1:BNFGrammar xd:script = "occurs *; ref XD4.1:BNFGrammar" />
      <XD4.1:component xd:script = "occurs *; ref XD4.1:component"/>
      <XD4.1:lexicon xd:script = "occurs *; ref XD4.1:lexicon" />
    </xd:mixed>
  </XD4.1:collection>

  <XD4.1:def xd:script = "init NS = @metaNamespace
      ? (String) @metaNamespace : 'http://www.xdef.org/xdef/4.1';
      options moreAttributes"
    name          = "optional QName; options acceptQualifiedAttr"
    metaNamespace = "optional uri; options acceptQualifiedAttr"
    root          = "optional rootList; options acceptQualifiedAttr"
    include       = "optional uriList; options acceptQualifiedAttr"
    script        = "optional xdefScript; options acceptQualifiedAttr"
    importLocal   = "optional importLocal; options acceptQualifiedAttr"
    xd:attr       = "occurs * getAttrName().startsWith('impl-');">
    <!-- Names of other attributes (see xd:attr) must start with "impl-" -->

    <xd:mixed>
      <XD4.1:macro xd:script = "occurs *; ref XD4.1:macro" />
      <XD4.1:declaration xd:script = "occurs *; ref XD4.1:declaration"
        scope = "optional enum('global','local'); options acceptQualifiedAttr"/>
      <XD4.1:lexicon xd:script = "occurs *; ref XD4.1:lexicon" />
      <XD4.1:BNFGrammar xd:script = "occurs *; ref XD4.1:BNFGrammar"/>
      <XD4.1:component xd:script = "occurs *; ref XD4.1:component"/>
      <xd:choice occurs = "*">
          <XD4.1:choice xd:script = "occurs *;
                          match @name || @XD4.1:name; ref XD4.1:choiceDef"
              name = "required QName; options acceptQualifiedAttr" />
          <XD4.1:mixed xd:script = "occurs *;
                          match @name || @XD4.1:name; ref XD4.1:mixedDef"
              name = "required QName; options acceptQualifiedAttr" />
          <XD4.1:sequence xd:script = "occurs *;
                          match @name || @XD4.1:name; ref XD4.1:sequenceDef"
              name = "required QName; options acceptQualifiedAttr" />
          <XD4.1:list xd:script = "occurs *;
                          match @name || @XD4.1:name; ref XD4.1:listDef"
              name = "required QName; options acceptQualifiedAttr" />
```

```
        <XD4.1:xon  xd:script = "occurs *;"
            name = "required QName; options acceptQualifiedAttr" >
            required xon;
        </XD4.1:xon>
        <XD4.1:any xd:script = "occurs *;
                        match @XD4.1:name; ref XD4.1:anyDef;
                        options moreAttributes, moreElements"
            XD4.1:name = "required QName;" />
        <xd:any xd:script = "occurs *; ref xelement" />
        optional valueScript;
    </xd:choice>
  </xd:mixed>

</XD4.1:def>

<XD4.1:macro xd:script = "occurs *; options moreAttributes"
  name    = "required QName; options acceptQualifiedAttr"
  xd:attr = "occurs * string()" >
  optional string();
</XD4.1:macro>

<XD4.1:declaration>
   <xd:mixed>
     <XD4.1:macro xd:script = "occurs *; ref XD4.1:macro" />
     * declarationScript;
   </xd:mixed>
</XD4.1:declaration>

<XD4.1:BNFGrammar extends="optional xdIdentifier; options acceptQualifiedAttr"
   name = "xdIdentifier; options acceptQualifiedAttr"
   scope = "optional enum('global','local'); options acceptQualifiedAttr">
     optional bnfGrammar; /* may be nothing */
</XD4.1:BNFGrammar>

<XD4.1:component>
    required xcomponent;
</XD4.1:component>

<XD4.1:lexicon language = "javaIdentifier" default = "optional yesNo">
  optional thesaurus();
</XD4.1:lexicon>

<!-- model of element -->
<xelement xd:script = "match getNamespaceURI() NE NS; options moreAttributes"
  xd:attr   = "occurs * attributeScript"
  xd:text   = "occurs * valueScript"
  XD4.1:script = "optional elementScript" >
  <xd:choice occurs = "*" ref = "xcontent" />
</xelement>

<xd:choice name = "xcontent">
  <XD4.1:choice xd:script = "occurs *;
                  match @ref || @XD4.1:ref; ref XD4.1:choiceRef" />
  <XD4.1:choice xd:script = "occurs *; ref XD4.1:choiceDef" />
  <XD4.1:mixed xd:script  = "occurs *;
                  match @ref || @XD4.1:ref; ref XD4.1:mixedRef" />
  <XD4.1:mixed xd:script  = "occurs *; ref XD4.1:mixedDef" />
  <XD4.1:sequence xd:script = "occurs *;
                  match @ref || @XD4.1:ref; ref XD4.1:sequenceRef" />
  <XD4.1:sequence xd:script = "occurs *; ref XD4.1:sequenceDef" />
  <XD4.1:list xd:script = "occurs *;
                  match @ref || @XD4.1:ref; ref XD4.1:listRef"/>
  <XD4.1:any xd:script = "occurs *;
                  match @XD4.1:ref; ref XD4.1:anyRef"/>
  <XD4.1:any xd:script = "occurs *; match !@XD4.1:ref; ref xelement"/>
  <xd:any xd:script = "occurs *; ref xelement" />
  <XD4.1:text> optional valueScript; </XD4.1:text>
  optional valueScript;
</xd:choice>

<XD4.1:choiceRef occurs = "optional Occurrence; options acceptQualifiedAttr"
  create = "optional elementCreateCode; options acceptQualifiedAttr"
  ref    = "required xposition; options acceptQualifiedAttr"
  script = "optional string; options acceptQualifiedAttr" />
```

```
  <XD4.1:choiceDef occurs = "optional Occurrence; options acceptQualifiedAttr"
    create = "optional elementCreateCode; options acceptQualifiedAttr"
    ref    = "illegal; options acceptQualifiedAttr"
    script = "optional groupScript; options acceptQualifiedAttr" >
    <xd:choice ref = "xcontent" occurs = "*" />
  </XD4.1:choiceDef>

  <XD4.1:sequenceRef occurs = "optional Occurrence; options acceptQualifiedAttr"
    create = "optional elementCreateCode; options acceptQualifiedAttr"
    ref    = "required xposition; options acceptQualifiedAttr"
    script = "optional groupScript; options acceptQualifiedAttr" />

  <XD4.1:sequenceDef occurs = "optional Occurrence; options acceptQualifiedAttr"
    create = "optional elementCreateCode; options acceptQualifiedAttr"
    ref    = "illegal; options acceptQualifiedAttr"
    script = "optional groupScript; options acceptQualifiedAttr" >
    <xd:choice ref = "xcontent" occurs = "*" />
  </XD4.1:sequenceDef>

  <XD4.1:mixedRef ref = "required xposition; options acceptQualifiedAttr"
    empty  = "optional booleanLiteral; options acceptQualifiedAttr"
    create = "optional elementCreateCode; options acceptQualifiedAttr"
    script = "optional groupScript; options acceptQualifiedAttr" />

  <XD4.1:mixedDef ref = "optional xposition; options acceptQualifiedAttr"
    empty  = "optional booleanLiteral; options acceptQualifiedAttr"
    create = "optional elementCreateCode; options acceptQualifiedAttr"
    script = "optional groupScript; options acceptQualifiedAttr" >
    <xd:choice xd:script = "*; ref xcontent;" />
  </XD4.1:mixedDef>

  <XD4.1:listRef ref = "required xposition; options acceptQualifiedAttr" />

  <XD4.1:listDef ref = "illegal; options acceptQualifiedAttr">
    <xd:choice xd:script = "*; ref xcontent;" />
  </XD4.1:listDef>

  <XD4.1:anyDef XD4.1:name = "required QName"
    XD4.1:script = "optional groupScript;
                    options moreAttributes, moreElements" />

  <XD4.1:anyRef XD4.1:script = "optional groupScript;
                    options moreAttributes, moreElements" />

/**************************************************************************/
/*    Declaration of value types.                                         */
/**************************************************************************/
<xd:declaration>
  /* Variable NS is used as the namespace of the checked source.*/
  String NS;

/**************************************************************************/
/* Types of values see BNF grammar below                                  */
/**************************************************************************/
  type rootList XDScript.rule('RootList');
  type xdefScript XDScript.rule('XdefScript');
  type declarationScript XDScript.rule('DeclarationScript');
  type valueScript XDScript.rule('ValueScript');
  type attributeScript XDScript.rule('AttributeScript');
  type elementScript XDScript.rule('ElementScript');
  type groupScript XDScript.rule('ElementScript');
  type groupModelScript XDScript.rule('ElementScript');
  type Occurrence XDScript.rule('Occurrence');
  type elementCreateSection XDScript.rule('ElementCreateSection');
  type elementCreateCode XDScript.rule('ElementCreateCode');
  type xdIdentifier XDScript.rule('Identifier');
  type xposition XDScript.rule('XPosition');
  type booleanLiteral XDScript.rule('BooleanLiteral');
  type bnfGrammar XDScript.rule('BNFGrammar');
  type xcomponent XDScript.rule('XCComponent');
  type languageId XDScript.rule('LanguageId');
  type javaIdentifier XDScript.rule('JavaIdentifier');
  type yesNo XDScript.rule('YesNo');
  type thesaurus XDScript.rule('Lexicon');
  type importLocal XDScript.rule('importLocal');
```

```
  type xon XDXon.rule('xon');

  /** Check element name and namespace URI (used in match section) */
  boolean xdName(String name) {
    return getElementLocalName() EQ name AND getNamespaceURI() EQ NS;
  }
</xd:declaration>

/*****************************************************************************/
/* Declaration of BNF grammar for X-Script                                   */
/*****************************************************************************/

  <xd:BNFGrammar name = "XDScript">

<![CDATA[

/*****************************************************************************/
/* x-Script BNF grammar rules                                                */
/*                                                                           */
/* Note the inline BNF method "$rule" is used to generate code for the       */
/* compiler of X-definitions. To understand the syntax you can simply        */
/* ignore them.                                                              */
/*****************************************************************************/

Letter ::= $letter /* any letter.*/

Char ::= $xmlChar /* any XML character.*/

WhiteSpace ::= [#9#10#13 ]

Comment ::= "/*" ([^*]+ | "*" - "*/")* "*/"

S ::= (WhiteSpace | Comment)+ /* Sequence of whitespaces or comments */

Digit ::= [0-9]

HexaDigit ::= Digit | [a-fA-F]

SemicolonOrSourceEnd ::= S? (";" S? | $eos)
/* At the end of the parsed source text the semicolon is not required. */

/* Keywords of the X-script */
Keyword ::= "if" | "else" | "do" | "while" | "continue" | "break" | "switch"
  | "case" | "for" | "return" | "def" | "try" | "catch" | "throw" | "finally"
  /* | "fixed" */ | "external" | "new" | "required" | "optional" | "ignore"
  | "illegal" | "occurs" | "onTrue" | "onError" | "onAbsence" | "default"
  | "onExcess" | "onStartElement" | "onIllegalAttr" | "onIllegalText" | "var"
  | "onIllegalElement" | "onIllegalRoot" | "create" | "init" | "options"
  | "ref" | "match" | "final" | "forget" | "template" | "type" | "uniqueSet"
  | "EQ" | "NE" | "LT" | "LE" | "GT" | "GE" | "LSH" | "RSH" | "RRSH" | "AND"
  | "OR" | "XOR" | "MOD" | "NOT" | "NEG" | "OOR" | "AAND" | "true" | "false"
  | "implements" | "extends" | "onFalse" | "$$$script"
  /* Note "fixed" is not the keyword since it is used also as the name of
   validation method. */

/* Predefined constants */
PredefinedConstant ::= ("$MAXFLOAT" | "$MAXINT" | "$MININT" | "$MINFLOAT" |
  "$PI" |"$E" | "$NEGATIVEINFINITY" | "$POSITIVEINFINITY" | "null") $rule

Identifier ::= JavaIdentifier - Keyword

LanguageId ::= [a-z] {2,3}

YesNo ::= "yes" | "no"

JavaIdentifier ::= ((Letter | "_" | "$")  (Letter | Digit | "_" | "$")* )

RawIdentifier ::= (Letter | "_")  (Letter | Digit | "_")*

QualifiedIdentifier ::= JavaIdentifier ("." JavaIdentifier)+ | JavaIdentifier

BooleanLiteral ::= ("true" | "false") $rule

DecimalInteger ::= ("_"* Digit+)+ ("_")*
```

```
IntegerLiteral ::= (("0" [Dd])? DecimalInteger
  | "0" [Xx] ("_"* HexaDigit+)+ ("_")*) $rule
  /* Inside a number specification it is possible to insert the character "_".
    This character does not influence the value of the number, it just makes
    a number more readable. E.g. the number 123456789 you can be written
    as 123_456_789 (or 0x0f123456 as 0x0f_12_34_56). */

FloatLiteral ::= (("0" [Dd]) DecimalInteger "." DecimalInteger? )
  | ((DecimalInteger ("." DecimalInteger Exponent? | Exponent))) $rule

Exponent ::= [Ee] [-+]? [0-9]+

NumberLiteral ::= FloatLiteral | IntegerLiteral

SpecChar ::= "\" ("\" | '"' | "'" | "n" | "r" | "t") | UnicodeCharSpecification

UnicodeCharSpecification ::= "\u" HexaDigit{4}

StringLiteral ::= ("'" ("''" | [^'\] | SpecChar)* "'" |
  '"' ('""' | [^"\] | SpecChar)* '"') $rule
  /* The opening and closing delimiter must be either """ or "'". The occurrence
     of this delimiter inside of literal can be recorded as a double delimiter or
     in the form of SpecChar. */

Literal ::= BooleanLiteral | NumberLiteral | StringLiteral

XMLName ::= $xmlName /* XMLName see XML specification */

KeyName ::= "%" XMLName

AttributeName ::= "@" XMLName

Reference ::= "ref" S XPosition

XDefName ::= XMLName

XModelName ::= XMLName

importLocal ::= S? ( (XMLName | "#") S? ("," S? (XMLName | "#") S?)*)?

XPosition ::= (XDefName? "#")? XModelName
  ("/" (XMLName | XGroupRererence | XAnyReference) XPositionIndex?)*
  ("/" (XAttrReference | XTextReference))?

XPositionIndex ::= "[" Digit+ "]"

XGroupRererence ::= "$mixed" | "$choice" | "$sequence"

XAnyReference ::= "$any"

XAttrReference ::= "@" XMLName

XTextReference ::= "$text" XPositionIndex?

RootList ::= S? RootSpecification (S? "|" S? RootSpecification)* S?
  /* all rootspecification in the list must be unique */

RootSpecification ::= XPosition | "*"

ExternalMethodType ::= QualifiedIdentifier (S?"[" S? "]")?

ExternalMethod ::= ExternalMethodType S? QualifiedIdentifier S?
  "(" S? ExternalMethodParamList? S? ")" S? ("as" S? Identifier)?

ExternalMethodParam ::= ExternalMethodType (S JavaIdentifier)?
  /* the parameter name (JavaIdentifier) is optional and ignored */

ExternalMethodParamList ::= ExternalMethodParam (S? "," S? ExternalMethodParam)*

MacroReference ::= "$" "{" S? XMLName S? MacroParams? S? "}"

MacroParams ::= "(" S? Identifier (S? "," S? Identifier)* S? ")"

/**************************************************************************/
/* Script expression                                                      */
```

```
/*****************************************************************************/

Expression ::= Expr1 (S? "?" S? Expression S? ":" S? Expression)?

OperatorLevel_1 ::= ("AND" | "AAND" | "&&" | "&") $rule

Expr1 ::= Expr2 (S? OperatorLevel_1 S? Expr2)*

OperatorLevel_2 ::= ("OR" | "OOR" | "XOR" | "||" | "|" | "^") $rule

Expr2 ::= Expr3 (S?  OperatorLevel_2 S? Expr3)*

OperatorLevel_3 ::= ("LT" | "<" | "GT" | ">" | "==" | "EQ" | "LE"
  | "<=" | "GE" | ">=" | "!=" | "NE" | "<<" | "LSH" | ">>" | "RSH" | ">>>"
  | "RRSH") $rule

Expr3 ::= Expr4 (S? OperatorLevel_3 S? Expr4)*

OperatorLevel_4 ::= ("*" | "/" | "%") $rule

Expr4 ::= Expr5 (S? OperatorLevel_4 S? Expr5)*

OperatorLevel_5 ::= ("+" | "-") $rule

Expr5 ::= Expr (S? OperatorLevel_5 S? Expr)*

Expr ::= (UnaryOperator S? | CastRequest S?)*
  (Value | Literal | "(" S? Expression S? ")") (S? "." S? Method)?

ConstantExpression ::= Literal /* ConstantExpression must be a Literal. */

CastRequest ::= S? "(" S? $rule TypeIdentifier S? ")" S?

UnaryOperator ::= "+" | ("-" | "!" | "NOT" | "~") $rule

TypeIdentifier ::= ("int" | "String" | "float" | "boolean" | "char" | "Datetime"
  | "Decimal" | "Duration" | "Exception" | "Container" | "Element" | "Message"
  | "Bytes" | "XmlOutStream" | "BNFGrammar" | "BNFRule" | "Parser" | "Service"
  | "ResultSet" | "Statement" | "ParseResult" | "Locale" | "uniqueSetKey"
  | "AnyValue" | "Output" | "Input" | "NamedValue" | "Regex" | "GPSPosition"
  | "Price" | "URI" | "EmailAddr") $rule

Value ::= (Constructor | PredefinedConstant | Increment
  | Method | VariableReference | KeyParameterReference | Literal | AttributeName
  | $rule AssignmentStatement) (S? "." S? Method)?

NewValue ::= "new" S $rule TypeIdentifier S? ParameterList

Constructor ::= NewValue | NamedValue | ContainerValue

NamedValue ::= KeyName S? "=" S? Expression

ContainerValueStart ::= (NamedValue (S? "," S? NamedValue)* ) | Expression

ContainerValue ::= "[" S? (ContainerValueStart (S? "," S? Expression)* )? S? "]"

KeyParameterReference ::= KeyName

Method ::= (SchemaTypeName | (QualifiedIdentifier - Keyword)) $rule
  S? ParameterList?

SchemaTypeName ::= "xs:" Identifier /* prefix “xs:” is deprecated */

incAfter ::= ("++" | "--") S? $rule VariableReference
incBefore ::= VariableReference S? ("++" | "--") S? $rule

Increment ::= incAfter | incBefore
  /* The type VariableReference must be an integer or a float */

Parameter ::= Expression $rule

ParameterList ::= "(" S? (Parameter
   (S? "," S? Parameter)* (S? "," S? "*")? S?)? ")" $rule

VariableReference ::= (Identifier - TypeIdentifier) $rule
```

```
MethodDeclaration ::= ("void" | TypeIdentifier) $rule
  S DeclaredMethodName S? ParameterListDeclaration S? Block

DeclaredMethodName ::= Identifier $rule

ParameterListDeclaration ::=
  "(" S? (SeqParameter (S? "," S? SeqParameter)* )? ")" $rule

SeqParameter ::= TypeIdentifier S ParameterName

KeyParameter ::= KeyName S? "=" S? (TypeIdentifier | ConstantExpression)

ParameterName ::= Identifier

ParentalExpression ::= S? "(" S? Expression S? ")" S?
  /* Result of ParentalExpression must be boolean. */

StatementExpression ::= Expression

/*************************************************************************/
/* Script statement                                                      */
/*************************************************************************/

Statement ::= S? (Statement1 | Statement2)

Statement1 ::= (Block | SwitchStatement | TryStatement)

Statement2 ::= (IfStatement | ForStatement | WhileStatement | DoStatement
  | ReturnStatement | ThrowStatement | BreakStatement | ContinueStatement
  | Method | Increment | AssignmentStatement)
  $info | EmptyStatement

EmptyStatement ::= ";"

StatementSequence ::= (S? VariableDeclaration S? ";" S? | Statement)*

Block ::= "{" StatementSequence S? "}"

SimpleStatement ::= S? (Statement1 | (Statement2? S? ";" S?)) | EmptyStatement

IfStatement ::= "if" ParentalExpression SimpleStatement
  (S? "else" S? SimpleStatement)?

ForStatement ::= "for" $rule S? "(" S? ForInit? S? ";" S?
  ForBooleanExpression? S? ";" S? ForStep? S? ")" S? SimpleStatement

ForBooleanExpression ::= $rule Expression

ForInit ::= $rule (AssignmentStatement | VariableDeclaration)

ForStepStatement ::= $rule (Method | Increment | AssignmentStatement)

ForStep ::= ForStepStatement (S? "," S? ForStepStatement)*

WhileStatement ::= "while" $rule ParentalExpression SimpleStatement

DoStatement ::= "do" $rule (S? Block | S  Statement) WhileCondition

WhileCondition ::= S? "while" $rule ParentalExpression

SwitchStatement ::= "switch" $rule S? "(" S? Expression S? ")" S?
   "{" SwitchBlockStatementVariant* S? "}"
  /* Result of Expression must be integer or string. Each variant may occur
     in the switch statement only once. */

SwitchBlockStatementVariant ::=
   S? (DefaultVariant | CaseVariant) S? ":" S? StatementSequence?
  /* Type of ConstantExpression must be integer or string. */

DefaultVariant ::= "default" S? $rule

CaseVariant ::= "case" S?  $rule ConstantExpression

ThrowStatement ::= "throw" S? $rule ExceptionValue
```

```
ExceptionValue ::= NewException | Identifier

NewException ::= "new" S? $rule "Exception" S? "(" S? (Expression S?)? ")"

TryStatement ::= "try" S? $rule "{" S? StatementSequence S? "}"
  S? CatchStatement

CatchStatement ::=
  "catch" S? $rule "(" S? "Exception" S?  Identifier S? ")" S?
  "{" S? StatementSequence S? "}"

ReturnStatement ::= "return" $rule (S? Expression)?

BreakStatement ::= "break" $rule (S? Identifier)?

ContinueStatement ::= "continue" $rule (S? Identifier)?

AssignmentStatement ::= (Identifier S? ((AssignmentOperator S? Expression) |
 (("=" S? Identifier))+ S? (AssignmentOperator S? Expression)) |
  S? AssignmentOperator S? Expression)

AssignmentOperator ::= (("|" | "OR" | "^" | "+" | "-" | "*" | "/" | "&" | "AND"
  | "%" | "<<" | "LSH" | ">>>"| "RRSH"  | ">>" | "RSH" )? "=") $rule

VariableModifier ::= ("final" | "external") $rule S

VariableDeclaration ::= VariableModifier*
  TypeIdentifier S VariableDeclarator (S? "," VariableDeclarator)*

VariableDeclarator ::= S? (AssignmentStatement | Identifier)

Occurrence ::= ("occurs" S)? ("required" | "optional" | "ignore" | "illegal"
  | "*" | "+" | "?" | ("*" | "+" | "?" |
  (IntegerLiteral (S? ".." (S? ("*" | IntegerLiteral))? )? ))) $rule
  /* The value of the second IntegerLiteral (after "..")  must be greater or
     equal to the first one. */

ExplicitCode ::= Block
  /* If the result value is required it must be returned by the command "return" */

/**************************************************************************/
/* Script of X-definition header                                          */
/**************************************************************************/

XdefScript ::=
  (S? (XdefInitSection | XdefOnIllegalRoot | XdefOnXmlError | XdefOptions)* S?)?
  /* Each item can be specified only once. */

XdefInitSection ::= "init" S Statement

XdefOnIllegalRoot ::= "onIllegalRoot" S Statement

XdefOnXmlError ::= "onXmlError" S Statement

XdefOptions ::= "options" S XdefOptionsList

XdefOptionsList ::= XdefOption (S? "," S? XdefOption)*
  /* Each option can be specified only once. */

XdefOption ::= "moreAttributes" | "moreElements" | "moreText"
  | "forget" | "notForget" | "clearAdoptedForgets"
  | "resolveEntities" | "ignoreEntities" | "resolveIncludes" | "ignoreIncludes"
  | "preserveComments" | "ignoreComments" | "acceptEmptyAttributes"
  | "preserveEmptyAttributes" | "ignoreEmptyAttributes"
  | "preserveAttrWhiteSpaces" | "ignoreAttrWhiteSpaces"
  | "preserveTextWhiteSpaces" | "ignoreTextWhiteSpaces"
  | "setAttrUpperCase" | "setAttrLowerCase"
  | "setTextUpperCase" | "setTextLowerCase"
  | "acceptQualifiedAttr" | "notAcceptQualifiedAttr"
  | "trimAttr" | "noTrimAttr" | "trimText" | "noTrimText"
  | "resolveEntities" | "ignoreEntities"
  | "resolveIncludes" | "ignoreIncludes"
  | "preserveComments" | "ignoreComments"
```

```
/*****************************************************************************/
/* Script of text nodes and attributes                                       */
/*****************************************************************************/

AttributeScript ::= ValueScript

ValueScript ::= ("$$$script:"? S?) ((ValueValidationSection | ValueInitSection
  | ValueOnTrueSection | ValueOnErrorSection | ValueOnAbsenceSection
  | ValueDefaultSection | ValueCreateSection | ValueFinallySection
  | ValueMatchSection | AttributeOptions | Reference | AttributeOnStartSection
  | ";") S?)*
  /* The keyword "$$$script" can be specified only in the template mode.
     Each section can be specified only once.*/

AttributeOnStartSection ::= "onStartElement" S? Statement?

ValueValidationSection ::= "fixed" S $rule (Expression | Block)
  | (Occurrence S? CheckValueSpecification?) | CheckValueSpecification

CheckValueSpecification ::= ExplicitCode | ValidationExpression | TypeMethodName
  /* ExplicitCode must return a value of boolean type or ParseResult. */

ValidationExpression ::= ValidationMethod | Expression
  /* Result of ValidationExpression must be a boolean or ParseResult type. */

ValidationMethod ::= SchemaValidationMethod | XDValidationMethod

SchemaValidationMethod ::= ParseMethod

XDValidationMethod ::= ParseMethod

ParseMethod ::= Method

TypeMethodName ::= Identifier

ValueMatchSection ::= "match" S? (Expression | ExplicitCode)
  /* Expression or ExplicitCode must here return a value of boolean type. */

ValueInitSection ::= "init" S? (ExplicitCode | Statement)

ValueOnTrueSection ::= "onTrue" S? (ExplicitCode | Statement)
  /* If Method or ExplicitCode returns a value, it will be ignored. */

ValueOnErrorSection ::= ("onError" | "onFalse") S? (ExplicitCode | Statement)

ValueOnAbsenceSection ::= "onAbsence" S? (ExplicitCode | Statement)

ValueCreateSection ::= "create" S? (ExplicitCode | ValueCreateExpression)
  /* ExplicitCode must return a value of String type. */

ValueCreateExpression ::= Expression
  /* ValueCreateExpression must return a value of String type. */

ValueDefaultSection ::= "default" S? (ExplicitCode | Expression)
  /* Expression or ExplicitCode must return the value of the String. */

ValueFinallySection ::= "finally" S? Statement

AttributeOptions ::= ValueOptions

ValueOptions ::= "options" S? ValueOptionsList

ValueOptionsList ::= ValueOption (S? "," S? ValueOption)*
  /* Each option can be specified only once. */

ValueOption ::= "preserveTextWhiteSpaces" | "ignoreTextWhiteSpaces"
  | "setTextUpperCase" | "setTextLowerCase" | "trimText" | "noTrimText"
  | "preserveAttrWhiteSpaces" | "ignoreAttrWhiteSpaces" | "cdata"
  | "setAttrUpperCase" | "setAttrLowerCase" | "trimAttr" | "noTrimAttr"
  | "ignoreEmptyAttributes" | "acceptEmptyAttributes"
  | "acceptQualifiedAttr" | "notAcceptQualifiedAttr" | "preserveTextCase"

/*****************************************************************************/
/* Script of elements                                                        */
/*****************************************************************************/
```

```
ElementScript ::= $info ElementExecutivePart* S?

ElementExecutivePart ::= "$$$script:"? S? (TemplateSection | Occurrence
  | ElementVarSection | ElementMatchSection | ElementInitSection
  | ElementOnAbsenceSection | ElementOnExcessSection | ElementCreateSection
  | ElementFinallySection | ElementOptions | Reference | ElementForgetSection
  | ElementOnStartSection | ElementStructureCompare | ";")
  /* Each item can be specified only once. If the occurrence is not specified,
     the implicit value is "required". The keyword "$$$script" can be specified
     only in the template model. */

TemplateSection ::= "template" $rule

ElementVarSection ::= "var" S?
  (("{"(S? ElementVarSectionItem S?)* "}") | ElementVarSectionItem S?)

ElementVarSectionItem ::= TypeDeclaration | VariableDeclaration S? ";"  | S? ";"

ElementInitSection ::= "init" S? Statement?

ElementMatchSection ::= "match" S? (Expression | ExplicitCode)
  /* Expression or ExplicitCode must here return a value of boolean type. */

ElementOnStartSection ::= "onStartElement" S? Statement?

ElementOnExcessSection ::= "onExcess" S? Statement?

ElementOnAbsenceSection ::= "onAbsence" S? Statement?

ElementCreateSection ::= "create" ElementCreateCode

ElementCreateCode ::= S? (Expression | ExplicitCode) S?
  /* Expression or ExplicitCode must return a value of Container or Element. */

ElementFinallySection ::= "finally" S? Statement?

ElementForgetSection ::= "forget"

ElementStructureCompare ::= ("implements" | "uses") S XPosition

ElementOptions ::= "options" S? ElementOptionsList

ElementOptionsList ::=  ElementOption (S?  "," S? ElementOption)*
  /* Each option can be specified only once. */

ElementOption ::= "moreAttributes" | "moreElements" | "moreText"
  | "forget" | "notForget" | "acceptEmptyAttributes" | "clearAdoptedForgets"
  | "preserveEmptyAttributes" | "ignoreEmptyAttributes"
  | "preserveAttrWhiteSpaces" | "ignoreAttrWhiteSpaces"
  | "preserveTextWhiteSpaces" | "ignoreTextWhiteSpaces" | "setAttrUpperCase"
  | "setAttrLowerCase" | "setTextUpperCase" | "setTextLowerCase"
  | "acceptQualifiedAttr" | "notAcceptQualifiedAttr" | "trimAttr" | "noTrimAttr"
  | "trimText" | "noTrimText" | "resolveEntities" | "ignoreEntities"
  | "resolveIncludes" | "ignoreIncludes" | "preserveComments" | "ignoreComments"
  | "preserveTextCase" | "acceptOther" | "ignoreOther"
  | "nillable" | "noNillable"

/*************************************************************************/
/* Script of the declaration part                                        */
/*************************************************************************/

DeclarationScript ::= (S? (TypeDeclaration | ExternalMethodDeclaration
  | VariableDeclaration | MethodDeclaration | ";"))* S?

TypeDeclaration ::= ("type" S Identifier S?
  ((Identifier (S? "," S? Identifier)* S? ";" S? )
  | TypeDeclarationBody)) | UniqueSetDeclaration

TypeDeclarationBody ::= TypeExplicitCode | Expression
  /* Expression or TypeExplicitCode must return either
     a boolean or a ParseResult value.*/

TypeExplicitCode ::= /* only X-defintion version 2.0 */
  ("{" S? "parse" S? ":" S? (ExplicitCode | Statement ";"?) S? "}")
```

```
  | /* X-definition version 3.1 and higher */ ExplicitCode

ExternalMethodDeclaration ::= "external" S "method"
  ( S? "{" S? (ExternalMethod S? ";" S? )* ExternalMethod? S? ";"? S? "}"
  | (S ExternalMethod) ) S? ";"?

UniqueSetDeclaration ::= "uniqueSet" S Identifier S? UniqueSetDeclarationBody

UniqueSetDeclarationBody ::=
  ("{" UniqueSetItem (S?";" UniqueSetItem)* (S?";"S?)? S? "}") | Method
  /* The method must be a parser. */

UniqueSetItem ::= S? (UniqueSetVar | UniqueSetKey)

UniqueSetKey ::= S? Identifier (S?":"S? (("?" | "optional") S? )? Method )?
  /* The method must be a parser. */

UniqueSetVar ::= S? "var" S TypeIdentifier S Identifier
  (S? "," S? TypeIdentifier S Identifier)*

/*****************************************************************************/
/* BNF grammar                                                               */
/*****************************************************************************/

BNFGrammar ::= S? BNFMethodDeclarationSection? BNFRules S?

BNFMethodDeclarationSection ::= BNFMethodDeclaration (S? BNFMethodDeclaration)*

BNFRuleName ::= RawIdentifier

BNFDefinedMethodName ::= "$" (BNFRuleName | Digit+)

BNFRuleDecl ::= S? BNFRuleName S? "::=" S?

BNFRuleReference ::= BNFDefinedMethodName | (BNFRuleName - BNFRuleDecl)

BNFMethodDeclaration ::= "%define" S BNFDefinedMethodName S? ":"
  S ("$" QualifiedIdentifier | BNFDefinedMethodName) S? BNFMethodparameters?

BNFMethodparameters ::= "(" S?
  (BNFMethodparameter (S? "," S? BNFMethodparameter)* ) ? S? ")"

BNFMethodparameter ::= Digit+ | BNFTerminalSymbol

BNFTerminalSymbol ::= "'" [^']* "'" | '"' [^"]* '"'
  | (BNFHexaCharacter (S? BNFHexaCharacter)* )

BNFHexaCharacter ::= "#" HexaDigit+

BNFRules ::= BNFRule (BNFRule)*

BNFRule ::= BNFRuleDecl BNFExpression

BNFQuantifier ::= S? ("+" | "*" | "?" | BNFExplicitQuantifier)

BNFExplicitQuantifier ::= "{" S? Digit+ (S? "," S? Digit+)? S? "}"

BNFSet ::= "[" ("^"? Char - "]")* "]" BNFQuantifier?

BNFTerm ::= (BNFTerminalSymbol | BNFSet | BNFRuleReference) BNFQuantifier?
  | BNFParentalExpr

BNFParentalExpr ::= "(" S? BNFExpr S? ")" S? BNFQuantifier?

BNFSequence ::= BNFTerm (S? BNFTerm)*

BNFRestriction ::= BNFSequence (S? "-" S? BNFTerm)?

BNFExpr ::= BNFRestriction (S? "|" S? BNFRestriction)*

BNFExpression ::= (BNFExpr S?)+

/*****************************************************************************/
/* XComponent                                                                */
/*****************************************************************************/
```

```
JavaTypeName ::= '<' S? $JavaQName (S? JavaTypeName)?
  (S? "," S?  JavaTypeName)* S? '>'

JavaTypedQName ::= $JavaQName (S? JavaTypeName)?

XCComponent ::= S? (XCComponentCommand
  (S? ";" S? XCComponentCommand?)* )

XCComponentCommand ::= (XCBind | XCClass | XCEnum | XCInterface | XCRef)

XCBind ::= "%bind" S XMLName (S "%with" S $JavaQName)? S XCLink

XCClass ::= "%class" S JavaTypedQName (S "extends" S JavaTypedQName)?
  (S "implements" S JavaTypedQName (S? "," S? JavaTypedQName)* )?
  (S "%interface" S JavaTypedQName)? S XCLink

XCEnum ::= "%enum" S JavaTypedQName S (Identifier? "#")? XMLName

XCInterface ::= "%interface" S $JavaQName S XCLink

XCRef ::= "%ref" S ((JavaTypedQName S XCLink)
  | ("%enum" S JavaTypedQName S (Identifier? "#")? XMLName))

XCLink ::= "%link" S ("*" | XPosition)

Lexicon ::= (S? $rule XPosition S? "=" S? XMLName)* S?

]]>

  </xd:BNFGrammar>

/*****************************************************************************/
/* Declaration of BNF grammar for XON                                        */
/*****************************************************************************/
  <xd:BNFGrammar name = "XDXon">

<![CDATA[
/*****************************************************************************/
/* JSON BNF grammar rules                                                    */
/*****************************************************************************/

WhiteSpace ::= [#9#10#13 ]

Comment ::= "/*" ([^*]+ | "*" - "*/")* "*/"

S ::= (WhiteSpace | Comment)+ /* sequence of whitespaces or comments */

boolean ::= "true" | "false"

number ::= int frac exp | int frac | int exp | int

int ::= "-"? digits

digits ::= [0-9]+

frac ::= "." digits

exp ::= [eE] ("+" | "-")? digits

controlcharacter ::= '\"' | '\\' | '\/' | '\b' | '\f' | '\n' | '\r' | '\t' |
  ('\u' [0-9a-fA-F]{4})

string ::= ('"' ([^\"] | controlcharacter)* '"' |
  "'" ([^\'] | controlcharacter)* "'")

jscript ::= S? ( ("%script" S? "=" S? string)
  | ( "%oneOf" (S? "=" S? string)? ) ) (S? ",")?

object ::= "{" jscript? (members)* S? "}"

members ::= S? pair (S? "," S? pair)*

pair ::= (string | $ncName) S? ":" S? value
```

```
array ::= "[" jscript? S? list? S? "]"

list ::= value (S? "," S? value )*

value ::= S? ("null" | boolean | number | string | array | object) S?

xon ::= S? (array | object | string)

]]>
  </xd:BNFGrammar>

</xd:def>
```

References

[1] Extensible Markup Language (XML) 1.0, W3C Recommendation, http://www.w3c.org/TR/REC-xml

[2] W3C Date and Time Formats, http://www.w3c.org/TR/NOTE-datetime

[3] W3C XML Schema, http://www.w3c.org/TR/xmlschema-1 , http://www.w3c.org/TR/xmlschema-2

[4] XML Path Language (XPath) 3.1 https://www.w3.org/TR/xpath-31/

[5] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: The Java Language Specification, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

[6] The JSON Data Interchange Syntax http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[7] XSL Transformations (XSLT) Version 3.0 https://www.w3.org/TR/xslt-30/#json

[8] Michael Key, Transforming JSON using XSLT 3.0, XML Prague 2016, *https://www.saxonica.com/papers/xmlprague-2016mhk.pdf*

[9] Curt Selak, Sourceforge 2021, X-definition For Beginner, https://sourceforge.net/p/x-definition-beginner-xml/wiki/Home/

[10] Curt Selak, DZone Java 2022, Extracting Data From Very Large XML Files With X-definition, https://dzone.com/articles/extracting-data-from-very-large-xml-files-with-x-d