X-definition 3.1

The language description (preliminary draft)

Author: **V.Trojan**
Version: **3.1.003.001**
Date: **7. 6. 2018**

Contents

1 Note about this draft 5

2 Annotation 6

3 Essential concepts 7

3.1 Quantifiers and Validation Methods 8
3.2 Model of Element 8
3.3 Model of the attribute or text node (model of data value) 8
3.4 References and XDPositions 8
3.5 Extension and modification of referred models 9
3.6 Events 9
3.7 Groups 10
3.7.1 Mixed group of nodes (xd:mixed) 10
3.7.2 Selection of models (xd:choice) 10
3.7.3 Sequence of nodes (xd:sequence) 11
3.7.4 Group declared as a model and the reference to the group 11
3.8 X-definition Header 11
3.8.1 xd:name 12
3.8.2 xd:root 12
3.8.3 xd:script 12
3.8.4 xd:metanamespace 12
3.8.5 xd:include 13
3.8.6 Implementation information 13
3.9 Declaration of variables, methods and datatypes (xd:declaration) 13
3.10 Macros (xd:macro) 13
3.11 BNF grammar in X-definition 14
3.11.1 Production rule 14
3.11.2 Terminal symbol 14
3.11.3 Sets of characters 14
3.11.4 Quantifier (repetition of a rule) 15
3.11.5 BNF expression 15
3.11.6 External rule 17
3.11.7 Declaration of external rules 17
3.11.8 Comments and whitespaces 15
3.11.9 External rules implemented in X-definition 15
3.11.10 BNF declaration 16
3.12 Arbitrary attribute(s) (xd:attr) 18
3.13 Arbitrary element (xd:any) 18
3.14 Concatenated text of element (attributes xd:text and xd:textcontent) 18
3.15 Structure of X-definition 19
3.16 Collection of X-definitions 19
3.17 XDPosition 20

4 Script of X-definition 20

4.1.1 Identifiers in the Script 21
4.1.2 Types of values of variables and expressions in the Script 22
4.1.2.1 int (the integer numbers) 22
4.1.2.2 float (the floating-point numbers) 22
4.1.2.3 Decimal (the decimal numbers) 23
4.1.2.4 String (the character strings) 23
4.1.2.5 Datetime (the date and time values) 23
4.1.2.6 boolean (Boolean values) 25
4.1.2.7 Locale (information about region) 26
4.1.2.8 Regex (Regular expressions) 26
4.1.2.9 RegexResult (results of regular expressions) 26
4.1.2.10 Input/Output (streams) 26

4.1.2.11 Element (XML elements) 26
4.1.2.12 Bytes (array of bytes) 26
4.1.2.13 NamedValue (named values) 26
4.1.2.14 Container (sequence and/or map of values) 27
4.1.2.15 Exception (program exceptions) 27
4.1.2.16 Parseresult (results of parsing/validation) 27
4.1.2.17 Report (messages) 27
4.1.2.18 BNFGrammar (BNF grammars) 27
4.1.2.19 BNFRule (BNF grammar rules) 27
4.1.2.20 uniqueSet (sets of unique values) 28
4.1.2.21 Service (database service; access to a database) 28
4.1.2.22 Statement (database commands) 28
4.1.2.23 ResultSet (results of the database commands) 28
4.1.2.24 XmlOutStream (data channels used for continuous writing of XML objects to a stream) 28
4.1.3 Access to values in the processed document 28
4.1.4 Local variable of the Script 28
4.1.5 The variables of the element model 29
4.1.6 Declared objects 29
4.1.6.1 Declared Variables 29
4.1.6.2 Declared methods 30
4.1.6.3 Declared datatype 31
4.1.7 Built-in variables and constants 32
4.1.8 Expressions 32
4.1.9 Events and actions 33
4.1.10 Quantifiers (Specification of occurrence) 35
4.1.11 Special quantifiers (ignore, illegal, fixed) 36
4.1.12 Check data type 36
4.1.13 Implemented validation methods 36
4.1.14 Table of unique values and the type of method, SET the ID, IDREF, and CHKID 40
4.1.15 Table of unique values without named entries 41
4.1.16 Linking tables of unique values 42
4.1.17 Template element 42
4.1.18 Script commands 42
4.1.19 Implemented script methods 43
4.1.20 Mathematical methods 59
4.1.21 External methods 61
4.1.22 Options 62
4.1.23 The references to the object of X-definitions 64
4.1.23.1 Reference to model of element 64
4.1.23.2 Reference to sequence of descendants of model of element 65
4.1.24 Comparison of the structure of models 65
5 X-definition modes 66
5.1 Validation mode 66
5.2 Construction mode 67
5.2.1 Construction of element 68
5.2.2 Construction of attributes 68
5.2.3 Construction of text nodes 68
6 X-components 70
6.1 Values in X-component 70
6.2 Access to values of X-component 70
6.3 X-component commands 70
6.3.1 %class 70
6.3.2 %bind 71
6.3.3 %interface 71
6.3.4 %ref 71
6.3.5 %enum 71
7 Invoking X-definitions from Java 72
Appendix A: X-definition of X-definition 75
Examples 91

Tables

Table 1 - Control characters in the date mask 24
Table 2 - Names of types of parameter values 30
Table 3 - Built-in variables and constants 32
Table 4 - Alias keywords used as the alternative notation of operators 32
Table 5 - Events 34
Table 6 - Named parameters corresponding to facets in XML schema 36
Table 7 - Validation methods of XML schema datatypes 37
Table 8 - Other validation methods of datatypes implemented in X-definition (and not in XML schema) 38
Table 9 - Main methods 43
Table 10 - Names of the Script types and the corresponding type ID 50
Table 11 - Methods of objects of all types 51
Table 12 - Methods of objects of the type BNFGrammar 51
Table 13 - Methods of objects of the type BNFRule 51
Table 14 - Methods of objects of the type Bytes 51
Table 15 - Methods of objects of the type Container 52
Table 16 - Methods of objects of the type Datetime 53
Table 17- Methods of objects of type Duration (time interval) 54
Table 18 - Methods of objects of the type Element 54
Table 19 - Methods of objects of the type Exception 55
Table 20 - Methods of objects of the type Input 55
Table 21 - Methods of NamedValue objects 55
Table 22 - Methods of objects of the type Output 55
Table 23 - Methods of objects of the type ParseResult 56
Table 24 - Methods of objects of the type Regex 56
Table 25 - Methods of objects of the type RegexResult 56
Table 26 - Methods of objects of the type Report 56
Table 27 - Methods of objects of the type ResultSet 57
Table 28 - Methods of objects of the type Service 57
Table 29 - Methods of objects of the type Statement 57
Table 30 - Methods of the type String 58
Table 31 - Methods of objects of the type uniqueSet 58
Table 32 - Methods of objects of the type XmlOutStream 59
Table 33 - Methods of mathematical functions (taken from the class java.lang.Math 59
Table 34 - Methods of mathematical functions (taken from java.math.BigDecimal) 60
Table 35 - Value types passed to external Java methods 61
Table 36 - Options 62
Table 37 - Conversion of X definition datatypes to X-component types 70

Changes in this document

Version 3.1.003.007 (26.01.2018)

(1) the Script method "getCounter()" is deprecated. please use the method "getOccurrence()".

Version 3.1.003.005 (08.01.2018)

(1) Implemented new type of value "Locale" (see 4.1.2.7 *Locale (information about region)*).

(2) Added Script methods "format" and "printf" (see *Table 9 - Main methods and Table 22 - Methods of objects of the type Output).*

Version 3.1.003.003 (17.12.2017)

(1) Added missing Script method "equalsIgnorCase(s)" on String values.

Version 3.1.002.004 (25.09.2017)

(1) The declaration if external methods moved to <xd:deklaration> with the command "external method ...".

Version 3.1.002.003 (18.08.2017)

(1) To the declaration section is added the attribute xd:scope = ["lolcal", "global"]. The default value is "global".

Version 3.1.002.001

(1) Default version of this document.

1 Note about this draft

This version of this preliminary draft of language description of X-definition is not complete. We are preparing the translation from the Czech language and the complete version will be available soon. In this document there are still missing some parts as the description of JSON tools, Thesaurus facilities, set of utilities, etc.

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

The actual version of X-definition you can download from: http://www.xdef.cz

2 Annotation

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

X-definition is a tool that provides the description of both the structure and the properties of data values in an XML document. Moreover, the X-definition allows the description of the processing of specified XML objects. Thus X-definitions may replace existing technologies commonly used for XML validation - namely the DTD (Data Type Definition) or the XML schemas and Schematron. With X-definition it is also possible to describe the construction of XML documents (or the transformation of XML data).

X-definition enables the merging in one source of both the validation of XML documents and processing of data (i.e. using actions assigned to events when XML objects are processed). Compared to the "classical" 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 of XML data files of unlimited size, up to many gigabytes.

A principal property of X-definition is maximum respect for the structure of the described data. The form of X-definition is an XML document with a structure similar to the described XML data. This makes possible quickly and intuitively describe given XML data and its processing. In many cases it requires just to replace the values in the XML data by the description written in the X-definition Script language. You can also gradually add to your script required actions providing 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 also have at least basic knowledge of the Java programming language.

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

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

The initial information about X-definition you can get from tutorial on:

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

3 Essential concepts

The basic concept of X-definition is the "**model**" of XML element. The simple example of the model you can get if all data values (attributes and/or text nodes) in the XML element are replaced by the description of its 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, date, salary, description of qualification etc.). We can describe the contents of such record when we replace the values of data with their description. Let's say the attribute "Salary" is optional, the other attributes are required. The expected form of value of text can be checked with special methods which are parsing the text value. Since the child element "Competence" may repeat more times, the range of occurrences is described in the special attribute "xd:script". The model of the element above would appear as follows:

```
<xd:def xmlns:xd="http://www.syntea.cz/xdef/3.1">
  <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.syntea.cz/xdef/3.1
```

This way it is possible to distinguish between the objects of the model and the auxiliary objects of 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 **Script of X-definitions**, or hereafter shortly **Script**. The Script language can be written in different parts of X-definition source code and it is divided to several parts according to the purpose what is described.

Usually you need to design a set of X-definitions to describe variety of processed data. The set of X-definition must be compiled to the binary code. The compiler creates the binary code saved as the 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: the **validation mode** and the **construction mode**. In the validation mode the input XML data are validated and processed according to X-definition and the Script. The result will be the valid XML document (convenient to the model). In the construction mode the processor creates new XML data constructed according to the model in X-definition. The result is the constructed XML document. In both modes there is generated a log file containing information with error messages, warnings etc.

Normally, processor of X-definition returns data in the form of org.w3c.dom.Element. However, if required, the result of processing may be returned if the form of an instance of Java class we call **X-component** instead of an element. Generation of such Java classes is described by special section of X-definition (see Chapter 6 X-components).

3.1 Quantifiers and Validation Methods

To validate an XML document requires check the occurrences of data items as well as check formal correctness of actual data values (a datatype). It is possible to specify in the Script of the attributes and of the text values of elements the description of the properties that they must fulfill. This part of the Script we call the **validation section** of the Script.

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

```
occurs min..max
```

"min" here is an integer expressing minimum limit of occurrences of the item and "max" is maximum limit of occurrences. To make the Script more clear 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 ("*") as max parameter. E.g.:

```
occurs 1..*
```

The validation process is represented by two steps:

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

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

3.2 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. However, the occurrence of 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 many element models may be described.

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

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

The validation method may also have the 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 "int(100,999)". Or, if the minimum length of a character string must be between 2 and 30 characters we can write "string(2,30)". All validation methods are described in the paragraph 4.1.13 Implemented validation methods.

Therefore the model of data value is described by a Quantifier ("occurs 0..1" or "required" or "occurs 1..1" or "optional") and a validation method.

3.4 References and XDPositions

Let us consider an XML data structure describing a specific 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 the father may not exist, while the mother occurrence must be one and only one. Sons and daughters may occur in unlimited number, or may not occur at all.

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

The parts of the Script in a referred object may be overloaded. So e.g. the Script with a reference may contain the specification of its occurrence. Use the delimiting character ";" (semicolon) to separate individual sections of the Script. The occurrence of the element is specified by the keyword "occurs "followed by the interval of the minimal and the maximum number of the expected occurrences separated by ".." (two full stops) If the minimal value is equal to the maximum, then the occurrence is represented by only one number.

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

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

In this way we can create aggregated X-definitions. Consider the model of the family where you have defined unlimited number of sons and daughters. The family may or may not have a father, but must have a mother. The family element model example 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 0..1; ref Person" />
</Family>

<Person firstName  = "string()"
        familyName = "string()"
        birthDate  = "date()" />
```

Defining a family member by referencing to their attributes as a Person, it is possible to re-use the element model Person.

3.5 Extension and modification of referred models

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

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

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

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", it has the additional attribute "Salary" and the child element "Profession".

3.6 Events

In the script language of X-definitions, you can describe what should happen in different events which may occur during the processing of data. For instance, you can describe what to do when a type 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 script by the keyword with the event name, followed by the action command. The example of validation of the item “Salary“ might look like:

```
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 (see) in the event “onFalse”)..

3.7 Groups

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 a X-definition (with the attribute "xs:name" which enables the references to this group from models).

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

3.7.1 Mixed group of nodes (xd:mixed)

Mixed groups are describing 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 nodes must be present in the data. If you want to accept also an empty group specify "optional" in the script:*

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

3.7.2 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 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 action "match" here is a Boolean value that evaluates the attributes of the current element. If there is the 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 4.1.2.6 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>
```

3.7.3 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>
```

3.7.4 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") and then it is possible to use a reference to a group from a model. In this case it must be specified the attribute "xd:name". The name is used then as the reference to the model of a group in the attribute Script.

Example:

```
<xd:def xmlns:xd="http://www.cz.syntea.xdef/3.1">

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

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

3.8 X-definition Header

X-definition is an XML element named "xd:def" where "xd" is the prefix of the namespace "http://www.syntea.cz/xdef/3.1" (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.

3.8.1 xd:name

One X-definition in the project may be unnamed, the other X-definitions compiled in a project (XDPool) must have the 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.

3.8.2 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.

3.8.3 xd:script

The Script in the header of X-definition may contain the actions "init", "onIllegalRoot", "onXmlError" and the specification of options (see Chapter 4.1.22 Options). Here are 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
```

3.8.4 xd:metanamespace

In order 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. In addition, it is possible to refer to these objects.

Example:

```
<meta:def xmlns:meta       ="mynamespace"
          xmlns:xd         =" http://www.syntea.cz/xdef/3.1"
          name             ="dummy"
          xd:metaNamespace ="mynamespace" >

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

</meta:def>
```

3.8.5 xd:include

The "xd:include" attribute contains the list of URL items or pathnames of files with the collections of X-definitions which are imported to the project. The separator of the entries in the list is 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 a local filesystem the wildcard is not allowed.

Example:

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

3.8.6 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 to the compiled pool and it is available in the Script by the method getImplProperty(name) where name is the part of the name of attribute which follows the prefix "impl-".

Example:

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

The processor writes to standard output:

```
Version of this project is: 001.002
```

3.9 Declaration of variables, methods and datatypes (xd:declaration)

The variables, methods and datatypes may be declared in the element "<xd:declaration>" that must be a direct descendant of the X-definition. To specify the scope of accessibility (or say, " visibility") of a declared object is possible to define with the attribute "xd:scope". If the value is "global", the declared variables, methods and datatypes are possible access on any point of the Script from any X-definition. However, if the given attribute is xd:scope="local", the declared variables, methods and datatypes are "visible" only from the X-definition, in which they were declared. The attribute "scope" is optional and the default value is "global". The detailed description of declaration section see paragraphs 4.1.6.1 Declared Variables, 4.1.6.2 Declared methods, 4.1.6.3 Declared datatype.

3.10 Macros (xd:macro)

Macros are used to simplify and help clarify the Script, and easier overall maintenance of the source of the Script in X-definitions. Macro specify a value (a character string), and has the assigned a 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 a 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 Script ax a "${name}". All references to the macro where ever it occurs in a Script are replaced with value of 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 macro is written as "#{paramName}". In the macre reference the parameter is specified in parentheses, and each parameter is referred by 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: If it has to be inside a parameter value to the character " or ', which begins with a declaration of the value, it must be given the escape character ' \ ' before it.

All macros are processed before the Script is compiled (this is provided with 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 in 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 script, even inside the values of constants, keywords or identifiers. For this reason, you should be aware of the possibility of an inadvertent call macros e.g. inside the declaration of character strings. If you do not want that the entry is 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 Script contains a reference to the macro from another X-definition, the name must be introduced by the name of 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 script copied including spaces and new lines. So, with a macro it is possible to insert new rows into the models of attributes.

The 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} ');"
```

The examples of macro reference:

```
<Person firstName="optional ${name}" lastName="${familyName}"/>
<Cover title="required ${colors(p2='red')}"/>
<Desciption 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}'); "/>
```

3.11 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.

3.11.1 Production rule

Each production rule (hereafter "rule") has a name. The name of 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 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:

```
symbol ::= formula
```

3.11.2 Terminal symbol

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

#xN	The 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

3.11.3 Sets 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

3.11.4 **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 may occur more times
A{n}	rule A must occur n times
A{m, n}	rule A may occur minimum m-times and maximum n-times
A{m,}	rule A may occur minimum m-times or more times

3.11.5 **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 the rule A followed by characters which meet rule B. The sequence has a higher priority than the selection. 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.

3.11.6 **Comments and whitespaces**

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

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

3.11.7 **Implemented predefined rules**

Following implemented methods provides parsing of the actual source

$anyChar	parse any character (returns true if a character exists and false if parser reached the end of the parsed string).
$base64	parse base64 format. Parsed text is put to the internal stack as an array of bytes.
$boolean	parse "true" or "false". Parsed text is put to the internal stack as a Boolean value.
$date	parse date according to ISO specification. Parsed text is put to the internal stack as a cz.syntea.xdef.sys.SDatetime value.
$datetime	parse date and time according to ISO specification (argument may be a mask) . Parsed text is put to the internal stack as a cz.syntea.xdef.sys.SDatetime value.
$day	parse day according to ISO specification. Parsed text is put to the internal stack as a cz.syntea.xdef.sys.SDatetime value.
$digit	parse decimal digit.
$duration	parse duration according to ISO specification. Parsed text is put to the internal stack as a cz.syntea.xdef.sys.SDuration value.
$error	writes the error message to the reporter and returns that the rule failed.

$find	skips characters from actual source position until it reaches a string from argument (returns true if string was found and false if not found)
$findOneOfChars	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	parse floating point number without sign (with decimal point and/or exponent). Parsed text is put to the internal stack as a java.lang.Double value.
$hexData	parse hexadecimal format. Parsed text is put to the internal stack as an array of bytes.
$integer	parse integer number without sign (sequence of digits). Parsed text is put to the internal stack as a java.lang.Long value.
$JavaName	parse Java name. Parsed text is put to the internal stack as a java.lang.String value.
$JavaQName	parse Java qualified name (may contain dots). Parsed text is put to the internal stack as a java.lang.String value.
$letter	parse letter
$letterOrDigit	parse letter or digit
$lowercaseLetter	parse lowercase letter
$month	parse month according to ISO specification. Parsed text is put to the internal stack as a cz.syntea.xdef.sys.SDatetime value.
$monthDay	parse month and day according to ISO specification. Parsed text is put to the internal stack as a cz.syntea.xdef.sys.SDatetime value.
$ncnNme	parse NCNAME according to W3C specification. Parsed text is put to the internal stack as a java.lang.String value.
$nmToken	parse NMTOKEN according to W3C specification. Parsed text is put to the internal stack as a java.lang.String value.
$time	parse time according to ISO specification. Parsed text is put to the internal stack as a cz.syntea.xdef.sys.SDatetime value.
$whitespace	parse whitespace according to W3C specification
$year	parse year according to ISO specification. Parsed text is put to the internal stack as a cz.syntea.xdef.sys.SDatetime value.
$yearMonth	parse year and month according to ISO specification. Parsed text is put to the internal stack as a cz.syntea.xdef.sys.SDatetime value.
$xmlChar	parse XML character according to W3C specification
$xmlName	parse XML name according to W3C specification. Parsed text is put to the internal stack as a java.lang.String value.
$xmlNameExtchar	parse following characters of XML name according to W3C specification
$xmlNamestartchar	parse first character of XML name according to W3C specification
$uppercaseLetter	parse capital letter.

Note the "rule" $error do not parse any actual text. However, it forces the parsing process failed at the actual position.

3.11.8 **Implemented methods for handling the internal stack**

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

$pop	removes the item from the top of the internal stack
$push	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 the parsed text.
$clear	clears the internal stack
$code	pushes the code item with the ID from the argument to the internal stack

3.11.9 External rule methods

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

3.11.10 Declaration of used defined methods

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 integer, float, String, Datetime or Duration.

The specification of all external 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
...
```

3.11.11 BNF declaration

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

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

Since the BNF grammar may quite big, it can be also recorded in the auxiliary element (it creates in fact 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 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 of 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" />
```

3.12 **Arbitrary attribute(s) (xd:attr)**

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

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

The selection of those attributes may be described by the "match" action, in which may be specified the Script:

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

Only attribute "b" and any other attributes with an integer value are accepted (at least one such attribute is required in this case).

3.13 **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 can be described in the usual way as in the model of an element. An example of the writing of “any" element:

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

The element in our example can have any name. Attributes and child nodes are not allowed.

The element that has allowed any attributes and child nodes can be written <xd:any> with 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 "ref" with the XDPosition using the name:

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

3.14 **Concatenated text of element (attributes xd:text and xd:textcontent)**

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

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

X-definition:

```
<xd:def xmlns:xd="http://www.cz.syntea.xdef/3.1 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) **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.cz.syntea.xdef/3.1 xd:root="A">
  <A xd:textcontent= "string(10) finally outln(getText());">
    <B/>
  </A>
</xd:def>
```

Data:

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

Prints:

```
text1text2
```

3.15 **Structure of X-definition**

X-definition is an element which starts with the X-definition header and contain 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)
- Thesaurus declarations (not described yet, will be described later)

Any above elements are optional and there may be an unlimited number of them. All of them have a unique name in X-definition and their sequence is arbitrary.

3.16 **Collection of X-definitions**

Larger projects may use many X-definitions. In this case we 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" to 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.syntea.cz/xdef/3.1">

<!-- 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.syntea.cz/xdef/3.1"
   xd:include = "C:/data/xdefA.xdef, C:/common/*.xdef"/>
```

3.17 **XDPosition**

In the previous paragraphs the possibility to specify a reference to an object in X-definitiion was described. The reference may link to the models in the actual X-definition or to another X-definition, The XDPosition 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 a child part of a model. The XDPosition then contains the path to the referred child node (to the second occurrence of the node "x" in the 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 XDPosition of an attribute is recorded with the character "@":

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

So the XDPosition 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>
```

4 Script of X-definition

The Script of X-definition (hereafter the **Script**) is the language used for the description of properties of data objects (both models and their child nodes). The script is written either as a value of the auxiliary attribute ”xd:script” or 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 script we describe the properties of data and the actions which are invoked in different events while an X-definition is processed.

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

1. **Validation section:** The validation formula of the script is different for the description of text values and of 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 the string, including an empty one).
    b. The occurrence of an element is specified by two numbers as the interval of minimal number of occurrences and the maximal number.

Note that X-definition does not use the concept "type of element" which is used in the XML schema. The type of text value of the 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 an 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 script.
3. **Options:** Specification of options starts with the keyword "options" followed by the list of option names separated by a comma. See *Table 36 - Options*. The script can only specify one option list.
4. **References:** By reference we specify the link to the model of an element. This model must exist in X-definition. Within one 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 and also 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 (see chapter 5). The section starts with the keyword "var". This section must be specified before the other sections.

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

Warning: *Line comments used in C or Java which start with the sequence "//" are 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 script are arbitrary with the exception of the declaration of variables, which must be specified before other sections.

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

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

Example of a script with a reference:

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

4.1.1 Identifiers in the Script

In the script identifiers are used for names of X-definitions, methods, variables, events, keywords, etc. The identifier in the 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 national alphabets 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 a 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 the Script

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

4.1.2.1 int (the integer numbers)

Values are in the range:

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

Whole numbers in 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').

Using special predefined constants:

$MININT. the minimum value of an integer (-9223372036854775808).

$MAXINT. the maximum value of an integer (9223372036854775807).

To make an entry clear 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 (value)", 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
- `.` causes the output of the 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 (the floating-point numbers)

Values are in the range:

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

or `0.0` or

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

The specification of numbers with a floating-point number corresponds to commonly used format 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
.	causes the output of the 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.

Using special predefined constants:

$PI the constant pi, the ratio of the circumference of a circle to its diameter (3.141592653589793)

$E the constant e, the base of natural logarithms (2.718281828459045)

$MINFLOAT the smallest positive nonzero value (4.9 E-324)

$MINFLOAT the largest possible positive value (1.7976931348623157 E308)

$NEGATIVEINFINITY negative infinity

$POSITIVEINFINITY positive infinity

$NaN value is not a valid number (Not a Number)

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 (the decimal numbers)**

The decimal numbers in the Script are implemented as java objects system.math.BigDecimal. This number type is given the characters "0 d" or "d" followed by the registration number or numbers with a floating point. In writing it is possible to use an underscore, e.g. "0d123__456_890_999_000_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 **String (the 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 (due to the fact that the values of XML attributes may also be inside quotation marks or apostrophes, you should use another character inside 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 character ' \ before him '. 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)
- \\ back slash ("\", \u005c)

Warning: If the text of Script is specified as the attribute value, the XML processor replaces all occurrences of the new line with a space. Therefore, you must write into the script in attributes to character strings new lines such as "\n". Additionally, you should avoid accidentally calling the macro. The occurrence of the pair of characters "$ {" anywhere in the script is interpreted as the beginning of a macro call, 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".

4.1.2.5 **Datetime (the 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": eg. parsedate ('2004-08-10T13:59:05')-see the description of the implemented features, see 4.1.19. The recommended format is according to ISO8601 (see XXXX), otherwise the function "parseDate" must have 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. dat.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 to the number of times a different meaning (see control characters ' a ', ' E ', ' G ', 'M ', ' y ', ' Z ', ' z '). In addition, 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 own 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 a date, either "1/3/1999" or "1.3.19990").

e. Control characters of the mask. Parser and formatter for date values according to a mask interprets the characters listed in the following table:

Table 1 - Control characters in the date mask

Character	Type	Description	Example:
(and more)	text	information about the part of the day	AM, PM
(D)	number	day of the year without leading zeros	4
DDD	number	day of the year with leading zeros	004
d	number	day of the year without leading zeros	9
DD	number	day of the year with leading zeros	09
E, EE, EEE	text	abbreviated day of the week	Mon, Tue...
EEEE (and more)	text	full weekday name	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
H	number	hours in the range 0-23 without leading zeros	8
HH	number	hours in the range 0-23 with leading zeros	08
h	number	hours in the range 1-12 without leading zeros	9
hh	number	hours in the range 1-12 with leading zeros	09

k	number	hours in the range 0-11 without leading zeros	9
kk	number	hours in the range 0-11 without leading zeros	09
K	number	hours in the range 1-24 without leading zeros	9
KK	number	hours in the range 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	Jan
MMMM	text	full month name	January
m	number	number of minutes (without leading zeros)	1
mm	number	number of minutes (with leading zeros)	1

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

s	number	number of seconds (without leading zeros)	5
ss	number	number of seconds (with leading zeros)	05
S	number	number of milliseconds	123
yy	number	year in two digits form, which is interpreted so that the values of the "01" to "99" are assigned the values and value of 1901 to 1999 and the value "00" is assigned the value 2000	
y	number	year (without leading zeros	1
yyyy (and more)	number	year in four-digit (or more digits) form	1989
Y	number	year according to the ISO specification (may be negative and may have leading zeros)	
YY	number	double digit year; the century is completed from the current date.	
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 a HHmm	+ 0100
ZZZZZZ (and more)	zone	zone in the form of ”+” or ”-” followed by HH: mm	+01:00

4.1.2.6 **boolean (Boolean values)**

The Boolean values may be used in the expressions, parameters of methods and the Script commands in a similar way as in the Java language. The possible values are "true" and "false." Boolean values may be a result of the expression, comparing etc. If in an Boolean expression occurs a reference to the attribute of 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.

4.1.2.7 **Locale (information about 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 and time format etc). Value of this type can be created by 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, country is uppercase two-letter ISO-3166 code and variant is vendor and browser specific code.

4.1.2.8 **Regex (Regular expressions)**

Objects of this type can be created with 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 XML schema.

4.1.2.9 **RegexResult (results of regular expressions)**

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.

4.1.2.10 **Input/Output (streams)**

The objects of this type are used to working 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 default output of the method "putReport"

4.1.2.11 **Element (XML elements)**

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

4.1.2.12 **Bytes (array of bytes)**

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

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

Methods to work with arrays of bytes read see 4.1.19.

4.1.2.13 **NamedValue (named values)**

The named value object is a pair consisting of the name and the assigned Script value (any type of 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.

4.1.2.14 **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 we can create a 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 into the mapped part and the not-named values are stored to 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 value of p, the next object is the Container and the string "abc".

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

The container can occur in Boolean expressions (i.e., it may be in the "match" section or in 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 the 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.

4.1.2.15 **Exception (program exceptions)**

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

4.1.2.16 **Parseresult (results of parsing/validation)**

The 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 in the object, otherwise the value is false (i.e. an automatic call of the method "matches ()").

4.1.2.17 **Report (messages)**

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 get it for example using the "getLastError".

4.1.2.18 **BNFGrammar (BNF grammars)**

This object type is defined by a special declaration. BNF grammar is written in a special declaration in the element "xd:BNFGrammar" (model) or it is possible to create it with the constructor. See 3.11 BNF grammar in X-definition.

4.1.2.19 **BNFRule (BNF grammar rules)**

The reference to a rule of a 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 methods of the method “rule(ruleName)”.

4.1.2.20 **uniqueSet (sets of unique values)**

This type is used to ensure the uniqueness of the set of values (table). It is used in the conjunction with a validation of text values of the attributes or text nodes.

4.1.2.21 **Service (database service; access to a database)**

This object allows you to access the 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 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 user name and finally the password is s4.

4.1.2.22 **Statement (database commands)**

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 the database command:

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

4.1.2.23 **ResultSet (results of the database commands)**

This object contains a result of the database command. In the case of the relational database it is a table whose rows have the named columns. It is possible gradually access lines 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.

4.1.2.24 **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 ofen 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 and it must match the path and the name of the file to which the writing is made. The p2 parameter is the name of the character encoding table and the parameter p3 indicates whether to create the header of the XML document. Example of a typical use in the 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 in the processed document**

In the Script it is also possible to use the values of attributes or text nodes obtained from the processed XML document, e.g. by using the method 'getText', 'getElement', 'getElementText' (see 4.1.19 Implemented 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" there are both attributes "a" and "b" or "c" attribute.

4.1.4 **Local variable of the Script**

In the Script of X-definition it is possible to declare the 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 in 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 The variables of the element model

In the Script of an element, you can declare the variables that are valid (and therefore accessible) only at the time of the processing of the current element. To declare such variables, write them in the Script section "var", which must be written at the beginning of the 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 the 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 that the value of a variable can be set externally before the process of X-definition was started (and therefore 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 this 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, null for the other objects).

All objects that are created in the 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 back 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 a 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 a command the Script).

4.1.6.2 **Declared methods**

Before any method a name of the type of the result must 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, so 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

The name of the type	**Description of the 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`	output stream
`Bytes`	array of bytes
`Container`	array of objects of other types
`Exception`	exception (the parameter of "catch" statement)
`Message`	message
`BNFGramar`	BNF Grammar
`BNFRule`	BNF grammar rule
`XMLOutStream`	stream used to write XML data

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

In the following example the validation methods are declared. These methods must return the value of the ParseResult value or the 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 process. The method "getToday" returns the date in the given format. It is advisable to write the 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 datatype**

In some cases, it is appropriate to declare the datatype. In the declaration section the datatype 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.

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 Script it is possible to use a number of 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 pi, 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, applicable in the Script
`$MININT`	`int`	constant with the highest whole number, applicable in the Script
`$MINFLOAT`	`float`	constant with the largest floating point number, applicable in the Script
`$MINFLOAT`	`float`	constant with the smallest floating-point number applicable in the Script
`$NEGATIVEINFINITY`	`float`	constant corresponding to negative infinity in floating point operations
`$POSITIVEINFINITY`	`float`	constant corresponding to negative infinity in floating point operations

4.1.8 **Expressions**

In the 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 (with reference to 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 with the predefined entities "&", "<", ">", there are defined keywords used as an alternative notation of the operators. That allows you to write the 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	Relation less then.	int, float, String, Datetime, Duration
>	GT	Relation greater then.	int, float, String, Datetime, Duration
<=	LE	Relation less or equal then.	int, float, String, Datetime, Duration
>=	GE	Relation greater or equal then.	int, float, String, Datetime, Duration
==	EQ	Relation equals.	*Any type*
!=	NE	Relation not equals.	*Any type*
<<	LSH	Left shift of integer.	int
>>	RSH	Right shift of integer.	int
>>>	RRSH	Binary zero fill right shift.	int
%	MOD	Aritmetic modulus.	int, float
^	XOR	Logical or bitwise XOR“.	boolean, int

+	*Not exists*	Addition of numers or string concatination.	int, float, String
-	*Not exists*	Substraction.	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 right operand.	*Any type*
+=	*Not exists*	Add to the left operand the right operand.	int, float, String
-=	*Not exists*	Substract from the left operand 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 an 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`	an 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 construction of the corresponding object (an element, attribute, or the 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, there is used the current context (see 5.1 Validation mode). 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 Script of Element. The elements are created according to the nodes from the list. 2. Container. If the action is specified in the 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 named value of the Container which has the same name is used. 3.StatementResult. If the action is specified in the Script of Element, the elements are created from rows of the table. In the case of an attribute it is used the value of the column with the same name (case insensitive) from the actual row. 4. String, if the action is part of the 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 Script of an Element 6. boolean value in the case of the Script of an Element 7. The other values are converted to String value
`default`	this may be defined only in the Script of models of attributes or of 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 the 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 actually processes element and it's attributes (however, not yet processed by X-definition).
`finally`	event "finally" occurs at the end of the processing of an element according to the model. After it, delete from the memory only follows in the case of the action "forget".
`forget`	an 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 placed into computer memory. The "forget" action causes the appropriate element after processing and after all actions (even after the event "finally") is removed from memory. However, the symptoms of examinations and of 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 initialization action that will be performed before further processing of the object. Automatic execution of certain functions can be set using the "options" (see ignoreAttrWhiteSpaces, trimAttr, ignoreTextWhiteSpaces, setAttrLowerCase, setAttrUpperCase, trimText, setTextLowerCase, setTextUpperCase). For the elements in the "init" action the descendant nodes are not yet available (they are not included yet in the resulting document tree), but all attributes are already available (however, not yet processed by X-definition).
`onAbsence`	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 the log file.
`onExcess`	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 the log file.
`onFalse`	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 the log file.

onIllegalAttr	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 the log file.
onIllegalElement	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 the log file.
onIllegalText	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 the log file.
onIllegalRoot	this event description is allowed only in the X-definition 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 the log file.
onStartElement	action of this event is performed after processing of all attributes, but before the processing of the child nodes of the element.
onTrue	action of this event is performed if the result of the validation method is not found with an error (parsing was OK). If no action is specified, the string from the parsed object is stored as text value of the parsed attribute or text node.
onXmlError	action of this event is allowed only in the X-definition Script. The action is performed when the parser detects an error of format of the source XML document. If the event is not specified, an error message is recorded in the log file or, if it is a serious mistake, further processing does not continue, and the program ends with the exception.

4.1.10 **Quantifiers (Specification of occurrence)**

The description of objects in a model requires specification of the limits of occurrence of an object (the **quantifier**). Specifications of the quantifier of an element can be written by 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 (the same thing as occurs 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 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 occurrence may be specified in the Script the 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 write to skip in the quantifier the 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 the attributes and text nodes text values consists of a quantifier and of a validation method. In the case the value is fixed it is possible to specify the keyword "fixed" and a value which must occur in validated data, then the text node or attribute must have this value. If it is missing, the value is inserted to validated data. The specification of quantifier is not allowed here. Therefore, the script with "fixed"

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

In some cases, it is appropriate with the fixed value to also specify the datatype of value. For example:

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

4.1.12 Check 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 the ParseResult object or the Boolean value. If the validation method is not described, the "string()" is used as default. However, if the option "ignoreEmptyAttributes" is specified the attributes with empty strings value are completely ignored.

A set of in-line functions to check the format of the values is commonly encountered in the 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 the datatype is valid or not. If there is no action "onFalse" specified and if the result is false, an error message is recorded into the 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 datatype and the associated actions:

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

In the appropriate implemented validation method is not available, you can declare the custom validation method in the declaration section and refer to it by its name.

4.1.13 Implemented validation methods

The datatypes implemented in X-definition correspond to the types of the XML schema. In 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

The named parameter corresponding to the facet in XML schema	The 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 validation method	i
%length	length of a string, array, etc.	l
%maxExclusive	parsed value of the datatype must be less than the parameter.	m
%maxInclusive	parsed value of the datatype must be less than or equal to the parameter.	m
%maxLength	length of a string, array, etc.	l
%minExclusive	parsed value of the datatype must be greater than the parameter.	m
%minInclusive	parsed value of the datatype 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 checks 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 the validated string. The last column describes the named parameter and sequence parameters.

Table 7 - Validation methods of XML schema datatypes

Method name	Description	The result	Parameters
anyURI	URI.	String	L belp
base64Binary	array of bytes in base64-encoded format.	Bytes	L belp
boolean	Boolean value ("true", "false").	boolean	- p
byte	byte value (an integer).	long	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	double	M bempt
gDate	date.	Datetime	M bempt
gDay	day of the date.	Datetime	M bempt
gMonth	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	array of bytes, in hexadecimal format.	Bytes	L belp
ID	unique value of NCName in the XML document.	String	L belp
IDREF	reference to the unique value in the XML document.	String	L belp
IDREFS	list of the references to the unique values in the XML document.	Container	L belp
integer	integer number	long	M bempt
language	XML schema language specification.	String	L belp
list	array of values.	Container	L beilp
long	integer number	long	M bempt
Name	name (according to the XML name).	String	L
NCName	XML NCName value.	String	L
negativeInteger	negative integer.	long	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 and zero.	long	M bempt
nonPositiveInteger	negative integer and zero.	long	M belpw
normalizedString	character string	String	L bempt
positiveInteger	positive integer	long	M bempt
QName	XML QName.	String	L belp
short	integer number	long	M bempt
string	character string. The named parameter %whiteSpace can only have here a value of "replace" or "collapse". or "preserve". The default is "preserve".	String	L belp
time	time.	Datetime	M bempt
token	XML token (i.e. must not include spaces inside).	String	L
union	union of the types.	Any	-, eip
unsignedByte	value of a byte (integer 0 .. 255)	long	M bempt
unsignedLong	unsigned long	Decimal	M bempt
unsignedInt	unsigned integer value.	long	M bempt
unsignedShort	unsigned short value.	long	M bempt
int	integer value	long	M bempt

In X-definition some other datatypes are implemented and listed in the following table:

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

Method name	Description	The result	Parameters
an	alphanumeric string (only letters or numbers)	String	L
BNF(g, s)	value must match the rule from the BNF Grammar g	String	-
contains(s)	any string that contains s	String	-
Any string that contains s (captualisation is ignored)	string that contains s regardless of upper/lower case	String	-

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
dec	decimal number corresponding to XML schema “decimal" datatype. However, the decimal point can also be recorded as the comma)	Decimal	T efmpt
email	email address	String	L
emailDate	date in the format in email (see RFC822).	String	M
emailList	list of email addresses separated by commas or semicolons	Container	L
ends(s)	value must end with the string value in the parameter s.	String	-
endsi(s)	value must end with the string value s regardless of the upper/lower case.	String	-
enum(s, s1, ...)	value must match with one parameter from the list. Parameters s, s1, ... must be strings.	String	-
enumi(s, s1, ...)	value must match with one parameter from the list regardless of the upper/lower case. Parameters s, s1, ... must be strings.	String	-
eq(s)	value must equal the value of the string s.	String	-
eqi(s)	value must equal the value with the string regardless of upper/lower case.	String	-
file	value must be formally correct file path	String	L
ISOdate	date according to ISO 8601 (also parses the variants, which do not support date in XML schema).	Datetime	M
ISOdateTime	date and time according to ISO 8601	Datetime	M
ISOyear	year according to ISO 8601 (also parses the variants, which do not support gYear datatype in XML schema).	Datetime	M
ISOyearMonth	year and month according to ISO 8601 (also parses the variants which do not support gMonthYear datatype in XML schema).	Datetime	M
languages	list of values separated by a space which are equal to an item from the list of language codes according to ISO 639 or ISO 639-2	Container	-elp
list(s1, s2, ...)	value must be equal to a parameter from the parameter list.	String	-
listi(s1, s2, ...)	value must be equal to a parameter from the parameters list, regardless of the upper/lower case.	String	-
ListOf(t)	value is list of values according to the type of the method parameter (which is a validation method - Parser). Values are separated by white spaces. DEPRECATED, replace the registration list(%item = t)	String	-
MD5	hexadecimal representation of the MD5 checksum (32 hexadecimal digits)	Bytes	- e
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 a sequence of digits.	String	L
QNameList	value must be a list of QName values according to the XML specification. A separator is a white space.	Container	- elp
QNameList(s)	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)	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	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 current element there exists the namespace URI corresponding to the value in the argument s.	String	- elp
pic(s)	value must match the structure of the string s, where '9' means any digit, 'a' means any alphabetic ASCII character, 'X' any alphanumeric (ASCII) character and other characters must match.	String	- elp
regex (s)	value must match the regular expression s. The s must be a regular expression according to 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 ID schema type. However, it does not report an error if the value already exists.	String	-
starts(s)	value must begin with the value of the string s.	String	-
startsi(s)	value must begin with the value of the string s regardless of upper/lower case.	String	-
tokens(s)	value must be equal to any part of the mask s. The individual parts of the mask are separated by the character "|".	String	-
The value must be the equal to any part of the mask s regardless of upper/lower case. The individual parts of the mask are separated by the character "|".	individual parts of the mask are separated by the character "|".	String	-
uri	value must be a formally correct URI, as implemented in Java.	String	-
uriList	formally correct list of URIS, as implemented in Java. The delimiter is a comma or white space.	String	-
url	value must be a formally correct URI, as implemented in Java.	String	-
urlList	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 does 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 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 reformatted according to the mask t.	Datetime	-

4.1.14 **Table of unique values and the type of method, SET the ID, IDREF, and CHKID**

If you require that the value must be unique within a part of the XML document, we declare the table for a set of these unique values. The rows of the 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 may be also the 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, 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 into 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 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 it is necessary for the value to be unique). The methods ID, IDREF, IDREFS, CHKID SET and CLEAR can be called above an uniqueSet,object or any of 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 uses the key value and leaves the key value unchanged. However, if an entry was declared with a question mark, after usage 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>
<Ulice>
  <House xd:script="occurs +" number="house.number.ID() /* the key is stored into the table “house"; entries “person" and “company" are null */ ">
    <Person xd:script="occurs *" name="house.person.ID()   /* save the key with entries “number” and “person”, “company” is null */" />
    <Company xd:script="occurs *" name="house.company.ID()   /* save the key with entries “number” and “company”, “person” is null */" />
  </Dum>
  <Occupant xd:script="occurs *; finally house.IDREF() /* number and person or company */" xd:script="occurs *; finally house.IDREF() /* number and person or company */"></Occupant xd:script="*; finally house.IDREF() /* number and person or company */">
    House = "house.number"
    Person = "optional house.person()"
    Company = "optional house.company() "/>
</Ulice>
```

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 Table of unique values 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 generates automatically one internal variable with the global table that has rows of NCName datatype. However, the user can declare his own table with one not named key entry and to specify the datatype 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 validation method of a key entry from another table. Note that the reference to the entry from 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 nname="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", and 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 in fact copied to the result. This can be achieved by writing the word "template" into the Script of the model of the element. However, if at some point the text value starts with "$$$script:" the value is processed as the 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 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 **Script commands**

The 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 in a similar way as in the language "Java" or "C" (including the compound statement).

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

the variable declaration statement (value types, however, must match the types in the 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 script methods

In the Script a variety of implemented methods can be called. Some of them can only be used in some parts of the 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 9 - Main methods

Method name	Description	The result	Where use
addComment(s)	adds a XML comment node with a value of s to the current element.		attribute, text node, element
addText(s)	adds a text node with a value of s to the current element.		attribute, text node, 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
compilePattern(s)	compiles the regular expression s. Deprecated, replaced by the new Regex(s).	Regex	anywhere
defaultError()	writes a default error message into the 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 the Easter Monday for year n	Datetime	anywhere
error(r)	writes an error message with report r into the temporary report log. The result is Boolean value false.	boolean always false	anywhere
error(s)	writes an error message with text s into the 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 the temporary report log. The result is 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 the 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 script element, the text value or attribute. If the result is null, returns an empty Container.	Container	create element
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 script element, the text value or 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 script of an element, the text value or attribute. If e is null, the result is an empty Container	Container	create attribute create element create text node
fromXQ(s)	returns the Container created after the execution of the xquery expression s in the current context. The method can only be used in the "create" action in the script of an element, the text node or attribute. If the result is null, returns an empty Container. NOTE: this method is implemented only when the Saxon library is available.	Container	create attribute create element create text node
fromXQ (e, s)	returns the Container after the execution of the xquery expression s in the element e. The method can only be used in the "create" action in the script of an element, the text node or attribute. If e is null, the result is an empty Container NOTE: this method is implemented only when the Saxon library is available.	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 current attribute.	String	all the action in the 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
getIem(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 maximum allowed value of year when parsing the date.	int	anywhere
getMinYear()	returns minimum allowed value of 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 the 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 the in the context of the element e. If the namespace URI does not exist, returns an empty string.	String	attribute, text node
getOccurrence()	returns the current number of instances of the object.	int	Element
getParentContextElement()	returns the element of the context of the parent of current element	Element	Element
getParentContextElement(n)	returns the element of the context of n - parent of current element	Element	Element
getParsedBoolean()	returns the Boolean value of the ParseResult (if the previous 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 byte array of ParseResult (if the previous script was a validation method). *Warning: this method must becalled immediately after a call to the validation method, for example, in the section onTrue.*	Bytes	Only after the method type checking

	If this condition is not met, then the result of the method is not defined.		
getParsedDatetime()	returns the value of Datetime from ParseResult (if the previous 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 decimal number of ParseResult (if the previous 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 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 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 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 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.*	AnyValue	Only after the method type checking
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 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 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
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
getText()	returns a string with the value of a current attribute or a text node.	String	attribute text

getXDPosition()	returns a string with the current XDPosition of the X-definition.	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 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)	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 [TH: m: s[.S] [Z]] ").	boolean	attribute, text node
isDatetime(s1, s2)	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 leap year.	boolean	anywhere
IsNumeric(s)	returns true when the string s contains only digits.	boolean	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 ()	output of the new line to the standard output		anywhere
parseBase64 (s)	converts a string to array of bytes If the string is not Base64, then the method returns null.	Bytes	anywhere
parseDate(s)	converts a string 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 a string s with a date in the format of RFC822 to the Datetime value. If the date is not in the specified format	Datetime	attribute, text node

	then the method returns null.		
parseFloat (s)	converts a 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 a string to an integer value If the string s is not an integer number, then the method returns null.	int	attribute, text node
parseDuration(s)	Converts a string 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 a string to an array of bytes If the string is not Base64, then the method returns null.	Bytes	anywhere
pause()	in the debug mode it writes the information about actual processing to the standard output. The program stops and waits for a response in the standard input. If the answer is "go", the program continues. Instead of "go", you can specify the 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 it prints the information line and the text s to the standard output. The program waits for a response on the standard input. If the answer is "go", the program continues. Instead of "go", you can specify the other commands. The list of possible commands will be printed by typing "?".		attribute, text node
printf(s, v1, ...)	prints to the standard output stream a 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 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)	first occurrence of the string s2 in the string s1 are replaced with the string s3.	String	anywhere
returnElement(o)	The result created from the argument o is an Element and it is set as the result of the X-definition process. The process of X-definition will be finished and returns as a reult a 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 year of the validated date.		anywhere
setMinYear(n)	sets the minimum allowed value of year of the validated date.		anywhere
setParsedValue(in)	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 224.1.2 Types of values of variables and expressions in the 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 match 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)	result is an object constructed in accordance with the model of the 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 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	Element	attribute, text node

	the string s1 begins with the character "<", then the parser uses the value of s converted to UTF-8 byte stream.		
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 stream created from s. However, if the string s begins with the character "<", then the parser uses the value of s converted to UTF-8 byte stream.	Element	attribute, text node
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
xpath(s)	returns a Container created after the execution of the xpath expression s on the element created from the Container c. If the result is null, it returns an empty Container.	Container	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 script of an element, the text node or 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 the methods that are implemented by the individual object types. The type of an object is expressed by the name of a type in the Script (e.g. "int", "String", etc. – see 4.1.2 Types of values of variables and expressions in the Script). A number is also assigned to each type (called “Type-ID”). The names of the types and the corresponding identifiers in the Script and in the Java code by enumeration cz.syntea.xdef.XDValueType are in the following table).

Table 10 - Names of the Script types and the corresponding type ID

The Script name	The type ID	The name of the item in the Java enumeration XDValueType
boolean	$BOOLEAN	BOOLEAN
BNFGrammar	BNFGrammar	BNFGRAMMAR
BNFRule	$BNFRULE	BNFRULE
Bytes	$BYTES	BYTES
Container	$CONTAINER	CONTAINER
Datetime	$DATETIME	DATETIME
Decimal	$DECIMAL	DECIMAL
Duration	$DURATION	DURATION
Element	$ELEMENT	ELEMENT
Exception	$EXCPTION	EXCEPTOIN
float	$FLOAT	FLOAT
Input	$INSTREAM	INPUT
int	$INT	INT
NamedValue	$NAMEDVALUE	NAMEDVALUE
Object	$OBJECT	OBJECT
Output	$OUTSTREAM	OUTPUT

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

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

Table 11 - Methods of objects of all types

Method name	Description	The result
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 12 - Methods of objects of the type BNFGrammar

Method name	Description	The result
BNFGrammar x	construction of BNF grammar object x is recorded in the element <xd:BNFGrammar name="x"> Text of BNF grammar specification see 3.11 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 3.11 BNF grammar in X-definition	BNFGrammar
x.parse(s)	returns a parsed value of the attribute of the text node in accordance with 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 13 - Methods of objects of the type BNFRule

Method name	Description	The result
x.parse ()	returns a parsed value of the current attribute of the text node in accordance with grammar rule x	ParseResult

Table 14 - Methods of objects of the type Bytes

Method name	Description	The result
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)	Rturns 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 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 15 - Methods of objects of the type Container

Method name	Description	The result
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 the Container x.	Container
x.getElements (s)	returns the new Container with the XML elements with the name s found in the 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 the Container x.	int
x.getNamedItem(s)	returns the value of a named item in the mapped part of the Container x.	AnyValue
x.getNamedString(s)	returns the value of a named item in the mapped part of the Container x as a string.	String
x.getText()	returns a string concatenated from the string items in the sequential part of the Container x.	String
x.getText(n)	returns a string with the nth item of type string in the sequential part of the Container x.	String
x.hasNamedItem(s)	returns true if the Container x has a named item with the name s.	boolean
x.isEmpty()	returns true if the 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 the 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 the Container x.	
x.sort()	returns the ascending sorted the sequential part of the Container x (according to the compareTo method on the items).	Container
x.sort(s)	returns the ascending sorted sequential part of the 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 the Container x. the direction of 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 the Container x.	Element
x.toElement(s)	creates an element with the name s from the Container x.	Element

x.toElement(s1, s2)	creates an element with the name s2 and namespace s1 from the Container x.	Element

Table 16 - Methods of objects of the type Datetime

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

Table 17- Methods of objects of type Duration (time interval)

Method name	Description	The result
x = new Duration(s)	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 18 - Methods of objects of the type Element

Method name	Description	The result
x = new Element (s)	constructor of Element. Creates new Element with the name s.	Element
x = new Element(s1, s2)	constructor of Element. Creates 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 of the XML tree 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 of the XML tree 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 the string is an indented form of the element x.	String

Table 19 - Methods of objects of the type Exception

Method name	Description	The result
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 20 - Methods of objects of the type Input

Method name	Description	The result
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. The argument s2 specifies the name of encoding.	Input
x=new Input(s1, s2, b)	creates an input stream by the argument s1. The argument s2 specifies the name of encoding. If b is true, it will be read in 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 21 - Methods of NamedValue objects

Method name	Description	The result
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 22 - Methods of objects of the type Output

Method name	Description	The result
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 s2 code page.	Output
x = new Output(s1, s2, b)	creates an output stream by the name of s1 and s2 code page. 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 xa 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 23 - Methods of objects of the type ParseResult

Method name	Description	The result
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 a 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 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 24 - Methods of objects of the type Regex

Method name	Description	The result
x = new Regex(s)	creates a regular expression x from s.	Regex
x.getMatcher(s)	returns the RegexResult created by 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 25 - Methods of objects of the type RegexResult

Method name	Description	The result
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 26 - Methods of objects of the type Report

Method name	Description	The result
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	Report

	parameter s1 is set to s2.	
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 27 - Methods of objects of the type ResultSet

Method name	Description	The result
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 28 - Methods of objects of the type Service

Method name	Description	The result
x=new Service(s1,s2,s3,s4)	creates the 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 user name and s4 is the 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 the query in a database x and returns the ResultSet object.	ResultSet
x.queryItem(s1, s2, s3)	executes the query in a database x and returns a string with the item s3.	String
x.rollback()	executes a rollback in the database x.	
x.setPropertys1, s2)	sets the property s1 to the value of s2 in a database x. Returns true, if the setting has taken place.	boolean

Table 29 - Methods of objects of the type Statement

Method name	Description	The result
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 the 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 the item s1, with parameters s2, and returns a	ResultSet

	ResultSet with the result.	

Table 30 - Methods of the type String

Method name	Description	The result
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 case.	boolean
x.getBytes()	Returns the array of bytes that is created from the string x (uses the current system encoding)	Bytes
x.getBytes(s)	returns the 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 the position n to the end.	String
x.substring(n1, n2)	returns part of the string x starting from the position n1 to the position n2.	String
x.toLower()	returns the string created from x where are all uppercase letters in string x are replaced with lowercase letters.	String
x.toUpper()	returns the string created from x where are all lowercase letters in string x are replaced with uppercase letters.	String
x.trim()	returns a string in which are removed all white spaces at the beginning and end of the string x are removed.	String

Table 31 - Methods of objects of the type uniqueSet

Method name	Description	The result
	the instance of uniqueSet object is created by the declaration statement. See 4.1.14.	uniqueSet
x.CHKID()	checks if the parsed value already exists as an entry in the table x. If not, an error is reported.	ParseResult
x.CHKIDS()	checks if for all parsed values from the list (separator is whitespace) already exist an entry in the 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 an entry in the 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 for all parsed values from the list (separator is whitespace) already exist an entry in the 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 was 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 the table x are unresolved references (by methods IDREF and IDREFS). After all errors are reported all entries of the table x are cleared.	

Table 32 - Methods of objects of the type XmlOutStream

Method name	Description	The result
new XmlOutStream(s)	creates theinstance of 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 script it is possible to use the mathematical methods of the library's "java.lang.Math". These methods are implemented both for the 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 Script is always implemented as the Java "long" and "float" type is always implemented as the Java "double".

Table 33 - Methods of mathematical functions (taken from the class java.lang.Math

Method name	Description	The result
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 Script are:

Table 34 - Methods of mathematical functions (taken from java.math.BigDecimal)

Method name	Description	The result
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 the 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 Script take the form of a usual static method. The types of the values of parameters determined by the table below:

Table 35 - Value types passed to external Java methods

Type name in the Script	Type passed to 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[])
Concept	(cz.syntea.xdef.XDContext)

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 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 "cz.syntea.xdef.XDValue[]". The number of items (i.e. number of actual parameters) in the array depends on the specific method call in the Script. Note: this type of method is selected by the compiler of X-definition when the 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 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 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 description 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 parentheses 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", "cz.syntea.xdef” are default and can be omitted. So, you can write: "(XXElement, String, long)".
- may be followed by an optional key name "as" followed by the alias name of the method (through this name the method will be accessed in the Script). If the alias name is not specified, then just the name of method is used in the Script, as declared in the Java class (not the qualified name). So, in order 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 these to 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

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

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

Table 36 - 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 is
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 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.
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 remains unchanged. This is the default value.
preserveAttrCase	letters in attribute are set before further processing to lower case.
setAttrUpperCase	letters in attributes are set to uppercase before further processing.
setTextLowerCase	letters of value of a text node before further processing are set to lowercase.
setTextUpperCase	letters of 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 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 Script of X-definition header and it causes that the 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 Script of X-definition header and it causes the 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 Script of X-definition header and it causes the 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 Script of X-definition header and it causes the links to external fdata 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 the undeclared objects in the model of element are inserted into the result. By default this option is not set.
ignoreOther	when validating the XML instances of the undeclared objects in the model of element are ignored. By default this option is not set.
cdata	this option causes a text node to be generated as CDATA section. This option is only permitted in the Script of the 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://www.syntea.cz/xdef/3.1" 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 which does not exist and therefore reports an error.

Device X-definitions, 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 script of a model it is possible to describe a link to the models, groups etc. Links makes the source code of X-definition clearer and easier to maintain large projects.

4.1.23.1 Reference to model of element

If we describe a reference of a model we can extend it with the Script sections and, moreover, we can also add the specification of the attributes which are not in specified in the referred model. The unspecified attributes and sections of the Script are taken from the referenced model. The reference starts with the keyword “ref" and follows the specification of 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 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://www.syntea.cz/xdef/3.1"
        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 **Reference to sequence of descendants of model of element**

In addition to links to the models of the elements it is possible to refer 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 Script of model and add a reference to the model, with which it should be compared. It only compares quantifiers and datatypes 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:name="B" ...>
  <Company xd:script="ref Adresa ">
          <Child nodes ...
  </Company>
          Country = "required string (1.30)"
  />
```

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>
```

It the validation section in the Script of an attribute or a text node is missing, it is taken from the referred model. Example:

```
<A xd:sctipt= "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 the model B is copied into the model A. Therefore, it will be “optional int(1,2)”.

5 X-definition 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 "cook book" according to which the resulting object is constructed. The validation mode is useful for validation and processing of input data. The construction mode serves for the construction of resulting data or for 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 which 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 the 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 it 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 the 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 it is possible to invoke the action associated with this event and the user can decide whether the processing will continue or be terminated. If no XML error action is described, then the processor continues when the parser finds light errors (as described in the XML specification).*

5.2 Construction mode

In the construction mode the process is controlled by X-definition (not by the input data as the validation mode). At the beginning a model according to which the construction will start must be specified. Each 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. The introduction how to use the construction mode of the X-definition is available on:

http://xdef.syntea.cz/tutorial/en/userdoc/xdef3.1_construction_mode_eng.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 works by the following steps:

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

2) For all objects specified within X-definition, the action “create” is performe 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 the walking in the tree of the model, the element with the 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 the string (or null value if it is null)

From the value types "Container" and "ResultSet" the 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. The ResultSet returns rows from a table.

5.2.1 Construction of element

The constructed element is given its name according to the name of the model. So, in fact, the only thing necessary to know is whether to construct it or not. If yes, then the 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 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 is 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 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 Script
2. String, the value is set as the value of 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 of the actual row is used
6. any other value is converted to the String and then used for the construction of the text node

6 X-components

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

http://xdef.syntea.cz/tutorial/en/userdoc/xdef3.1_X-component_eng.pdf

6.1 Values in X-component

The text values in the X-component are converted according to the datatype as described in the table:

Table 37 - Conversion of X definition datatypes to X-component types

Datatype in X-definition	X-component
base64Binary, hexBinary, base64, hex	`byte[]`
boolean	`java.lang.Boolean`
byte, short, int, long (and derived types)	`java.lang.Long`
datetime, xdatetime, gDate, gTime, gYear, gMoth, gYearMonth, gMonthDay	`cz.syntea.xdef.sys.SDatetime` `(also available as java.util.Calendar, java.util.Date, java.sql.Timestamp)`
decimal, dec	`java.math.BigDecimal`
duration	`cz.syntea.xdef.sys.SDuration`
enum	`java.lang.String or user declared Java enum`
float, double	`java.lang.Double`
Other datatypes	`java.lang.String`

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

6.2 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 of element in 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 is generated the method add***NAME***. Because there are more possibilities to work with datatypes of datetime the getters are generated in more variants: timestampOf***NAME***, calendarOf***NAME***, and dateOf***NAME***.

6.3 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 (";").

6.3.1 %class

This command specifies generation of the X-component according to the model of element. The "%class" keyword is followed by fully qualified class name and by the keyword "`%link`" that specifies XDPosition of the 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 an Java interface then you can specify "`extends SuperClassName implements InterfaceName`" after the name of Java class, where SuperClassName and InterfaceName must be the fully-qualified names. The syntax is the same as in the declaration of a class in the Java language.

6.3.2 %bind

By the command "%bind" can be set the 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. After the specification of the name the command continues with the keyword "%from", which is followed by the list of XDPositions (separated by 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, you 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 and it uses the implementation of these methods in the ancestor.

6.3.3 %interface

The command "%interface" is used when the (final) model takes the structure of another (referenced) model and, where appropriate, it also adds the 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 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 keyword %link followed by XDPosition of the model in the project.

6.3.4 %ref

It often happens that the project (XDPool) is generated from more X-definitions. In the case the X-component is generated from given X-definition, but XDPool is different (for example, there are some X-definition extra, missing ets), it can be used the already created X-component and it's necessary to prevent its new generation (for example, if the X-component is located in another Jar file). To refer 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 XDPosition of the model in the XDPool.

6.3.5 %enum

If the X-definition the 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 Java enum type. The data type must be defined in the Script section <xd:declaration>. Enum will be generated by using the command %enum followed by the fully qualified name of the enum class and the name of the datatype.

7 Invoking X-definitions from Java

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

"cz.syntea.xdef3.1.jar"

The Java documentation is distributed in the file:

"syntea_xdef3.1_doc.zip"

First you must to compile the X-definition sources and create an instance of the class "cz.syntea.xdef.XDPool" by invoking the static method "compileXD" of the class "cz.syntea.xdef.XDFactory". The parameter list of this method allows you to compile a set of X-definitions (the project) from the files or URL's 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 of the processing by setting of different properties to the parameter "props" in the following example (it is also possible to set it via e.g. java.lang.System.setProperties).

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
cz.syntea.xdef.XDPool xp = cz.syntea.xdef.XDFactory.compileXD(props, xdFiles);
```

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

```
cz.syntea.xdef.reporter.ArrayReporter reporter =
    new cz.syntea.xdef. reporter.ArrayReporter();
```

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

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

A. 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. The second parameter is a ReportWriter object, where the errors detected during validation process are recorded. If this parameter is set to null, then if an error was reported the RuntimeException is thrown. Example:

```
import cz.syntea.xdef.reporter.ArrayReporter;
import cz.syntea.xdef.XDDocument;
import cz.syntea.xdef.XDPool;
import cz.syntea.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 "cz.syntea.xdef.reporter.*" 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 "cz.syntea.xdef.util.*".*

B. Construction Mode

For the construction mode use the method "`xcreate`" of the object `XDDocument`. 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 cz.syntea.xdef.reporter.ArrayReporter;
import cz.syntea.xdef.XDDocument;
import cz.syntea.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.syntea.cz/xdef/3.1" 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>
```

Appendix A: X-definition of X-definition

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 X-definition of X-definition ver 3.1 is has the metanamespace "METAXDef" (prefix "XD31")
-->

<xd:def xmlns:xd  = "meta.namespace"
  name            = "XDefinition_3_1"
  root            = "XD31:def | XD31:collection"
  xmlns:XD31      = "http://www.syntea.cz/xdef/3.1"
  xmlns:w         = "http://www.syntea.cz/xdef/3.1"
  w:metaNamespace = " meta.namespace" >

  <xd:declaration scope = "local">
    String xdef_URI; /* the namespace of parsed X-definition. */

/*****************************************************************************
* 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 elementCreateExpression XDScript.rule('ElementCreateSection');
  type xdIdentifier XDScript.rule('Identifier');
  type xdposition XDScript.rule('XDPosition');
  type booleanLiteral XDScript.rule('BooleanLiteral');
  type bnfGrammar XDScript.rule('BNFGrammar');
  type xcomponent XDScript.rule('XCComponent');
/* will be added! type thesaurus XDScript.rule('Thesaurus'); */

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

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

```
* X-definition of X-definitions (ver 3.1)
*************************************************************************/
  <XD31:collection xd:script = "init xdefURI = @metaNamespace
      ? (String) @metaNamespace : 'http://www.syntea.cz/xdef/3.1';
      options moreAttributes"
    include         = "optional uriList; options acceptQualifiedAttr"
    metaNamespace  = "optional uri; options acceptQualifiedAttr" >
  <!--
     Names of other atributes (option moreAttributes) must start with "impl-"
   -->

    <XD31:def xd:script = "occurs +; ref XD31:def" />

  </XD31:collection>

  <XD31:def xd:script = "init = @metaNamespace
      ? (String) @metaNamespace xdef_URI : 'http://www.syntea.cz/xdef/3.1';
      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">
  <!--
     Names of other atributes (option moreAttributes) must start with "impl-"
   -->
    <xd:mixed>

      <XD31:macro xd:script = "occurs *; options moreAttributes"
        name    = "required QName; options acceptQualifiedAttr"
        xd:attr = "occurs * xdIdentifier" >
        optional string;
      </XD31:macro>

      <XD31:BNFGrammar xd:script = "occurs *"
         extends = "optional xdIdentifier; options acceptQualifiedAttr"
         name = "xdIdentifier; options acceptQualifiedAttr"
         scope = "optional enum('global', 'local'); options acceptQualifiedAttr" >
         required bnfGrammar;
      </XD31:BNFGrammar>

      <XD31:declaration xd:script = "occurs *"
         scope = "optional enum('global', 'local'); options acceptQualifiedAttr">
         required declarationScript;
      </XD31:declaration>

      <XD31:text xd:script = "occurs *"
```

```
         name = "xdIdentifier; options acceptQualifiedAttr">
         optional valueScript;
      </XD31:text>

      <XD31:attr xd:script = "occurs *"
         name = "xdIdentifier; options acceptQualifiedAttr">
         optional attributeScript;
      </XD31:attr>

      <XD31:component xd:script = "occurs *">
         required xcmompoonent;
      </XD31:component>

<!-- will be added!
      <XD31:component xd:script = "occurs *">
         required xcmompoonent;
      </XD31:component>
  -->
      <xd:choice occurs = "*">
          <XD31:choice xd:script = "occurs *; match @name || @XD31:name; ref XD31:choiceDef"
              name = "required QName; options acceptQualifiedAttr" />
          <XD31:mixed xd:script="occurs *; match @name || @XD31:name; ref XD31:mixedDef"
              name = "required QName; options acceptQualifiedAttr" />
          <XD31:sequence xd:script="occurs *; match @name || @XD31:name; ref XD31:sequenceDef"
              name = "required QName; options acceptQualifiedAttr" />
          <XD31:list xd:script="occurs *; match @name || @XD31:name; ref XD31:listDef"
              name = "required QName; options acceptQualifiedAttr" />
          <XD31:any xd:script="occurs *; match @name; ref XD31:anyDef" />
          <xd:any xd:script = "occurs *; ref xelement" />
          optional valueScript;
      </xd:choice>
    </xd:mixed>

  </XD31:def>

  <!-- model of element -->
  <xelement xd:script = "match getNamespaceURI() NE xdef_URI; options moreAttributes"
    xd:attr   = "occurs * attributeScript"
    xd:text   = "occurs * valueScript"
    XD31:script = "optional elementScript" >
    <xd:choice occurs = "*" ref = "xcontent" />
  </xelement>

  <xd:choice name = "xcontent">
    <XD31:choice xd:script = "occurs *; match @ref || @XD31:ref; ref XD31:choiceRef" />
    <XD31:choice xd:script = "occurs *; ref XD31:choiceDef" />
```

```
  <XD31:mixed xd:script  = "occurs *; match @ref || @XD31:ref; ref XD31:mixedRef" />
  <XD31:mixed xd:script  = "occurs *; ref XD31:mixedDef" />
  <XD31:sequence xd:script = "occurs *; match @ref || @XD31:ref; ref XD31:sequenceRef" />
  <XD31:sequence xd:script = "occurs *; ref XD31:sequenceDef" />
  <XD31:list  xd:script = "occurs *; match @ref || @XD31:ref; ref XD31:listRef"/>
  <XD31:any  xd:script = "occurs *; match @XD31:ref; ref XD31:anyRef"/>
  <XD31:any  xd:script = "occurs *; match !@XD31:ref; ref xelement"/>
  <xd:any xd:script  =  "occurs *; ref xelement" />
  <XD31:text> optional valueScript; </XD31:text>
  optional valueScript;
</xd:choice>

<XD31:choiceRef occurs = "optional Occurrence; options acceptQualifiedAttr"
  script = "optional string; options acceptQualifiedAttr"
  create = "optional elementCreateExpression; options acceptQualifiedAttr"
  ref    = "required QName; options acceptQualifiedAttr" />

<XD31:choiceDef occurs = "optional Occurrence; options acceptQualifiedAttr"
  script = "optional groupScript; options acceptQualifiedAttr"
  create = "optional elementCreateExpression; options acceptQualifiedAttr"
  ref    = "illegal; options acceptQualifiedAttr" >
  <xd:choice ref = "xcontent" occurs = "*" />
</XD31:choiceDef>

<XD31:sequenceRef occurs = "optional Occurrence; options acceptQualifiedAttr"
  create = "optional elementCreateExpression; options acceptQualifiedAttr"
  script = "optional groupScript; options acceptQualifiedAttr"
  ref    = "required QName; options acceptQualifiedAttr" />

<XD31:sequenceDef occurs = "optional Occurrence; options acceptQualifiedAttr"
  create = "optional elementCreateExpression; options acceptQualifiedAttr"
  script = "optional groupScript; options acceptQualifiedAttr"
  ref    = "illegal; options acceptQualifiedAttr" >
  <xd:choice ref = "xcontent" occurs = "*" />
</XD31:sequenceDef>

<XD31:mixedRef ref = "required QName; options acceptQualifiedAttr"
  empty  = "optional booleanLiteral; options acceptQualifiedAttr"
  script = "optional groupScript; options acceptQualifiedAttr"
  create = "optional elementCreateExpression; options acceptQualifiedAttr" />

<XD31:mixedDef ref = "illegal; options acceptQualifiedAttr"
  script = "optional groupScript; options acceptQualifiedAttr"
  empty  = "optional booleanLiteral; options acceptQualifiedAttr"
  create = "optional elementCreateExpression; options acceptQualifiedAttr" >
  <xd:choice xd:script = "+; ref xcontent;" />
</XD31:mixedDef>
```

```
  <XD31:listRef ref = "required refName; options acceptQualifiedAttr" />

  <XD31:listDef ref = "illegal refName; options acceptQualifiedAttr">
    <xd:choice xd:script = "*; ref xcontent;" />
  </XD31:listDef>

  <XD31:anyDef name= "required QName"
    script = "optional groupScript; options acceptQualifiedAttr" />
  <XD31:anyRef script = "optional groupScript" />

/*****************************************************************************
* Declaration of BNF grammar of Script
*****************************************************************************/
  <xd:BNFGrammar name = "XDScript">
<![CDATA[

Letter ::= $letter
Char ::= $xmlChar
WhiteSpace ::= $whitespace
Comment ::= "/*" ( [^*]+ | "*" [^/] )* "*/"
S ::= ( WhiteSpace | Comment )+

/**** Keywords ****/

Keyword ::=
  "if" | "else" | "do" | "while" | "continue" | "break" | "switch" | "case" |
  "for" | "return" | "def" | "try" | "catch" | "throw" | "finally" |
  "external" | "new" | "fixed" | "required" | "optional" | "ignore" |
  "illegal" | "occurs" | "onTrue" | "onError" | "onAbsence" | "default" |
  "onExcess" | "onStartElement" | "onIllegalAttr" | "onIllegalText" |
  "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" |
  "$PI" | "$E" | "$MAXINT" | "$MININT" | "$MINFLOAT" | "$MAXFLOAT" |
  "$NEGATIVEINFINITY" | "$POSITIVEINFINITY"

/**** Base symbols ****/

Digit ::= [0-9]

Identifier ::= (BaseIdentifier ( ':' BaseIdentifier)? ) - Keyword
```

```
BaseIdentifier ::= (( Letter | "_" | "$")  ( Letter | Digit | "_" )*)

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

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

BooleanLiteral ::= "true" | "false"

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

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

IntegerLiteral ::= DecimalInteger | ( "0x" | "0X" ) (HexaDigit ("_")* )+
/* 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). */

SignedIntegerLiteral ::= ("+" | "-")? IntegerLiteral

NumberLiteral ::=
  DecimalInteger ( "." DecimalInteger )? ( ("E" | "e") [-+]? DecimalInteger )?

SignedNumberLiteral ::= ( "+" | "-" )? NumberLiteral

SpecChar ::=
  "\" ("\" | [0-7]+ | '"' | "'" | "n" | "r" | "t" ) | UnicodeCharSpecification
/* The octal specification is in interval 0 .. 377*/

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

StringLiteral ::= "'" ( "''" | [^'\] | SpecChar)* "'" |
  '"' ( '""' | [^"\] | SpecChar)* '"'
  /* The opening and closing delimiter must be either """ or "'". The occurrence
     of this delimiter inside of literal can be recorded as 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 XDPosition
```

```
XDefName ::= XMLName

XModelName ::= XMLName

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

XDPositionIndex ::= "[" Digit+ "]"

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

XAnyReference ::= "$any"

XAttrReference ::= "@" XMLName

XTextReference ::= "$text" XDPositionIndex?

RootList ::= S? (XDPosition | "*") (S? "|" S? (XDPosition | "*"))* S?

MethodListItem ::= Identifier S?
  QualifiedIdentifier S? "(" S? MethodListItemParamList? S? ")" S?
  ("as" S? Identifier)? S? ";"

MethodListItemParamList ::= QualifiedIdentifier (S?"[" S? "]")?
  (S? "," S? QualifiedIdentifier (S?"[" S? "]")?)*

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

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

/**** Script language ****/

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

OperatorLevel_1 ::= "OR" | "OOR" | "XOR" | "||" | "|" | "^"

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

OperatorLevel_2 ::= "AND" | "AAND" | "&&" | "&"

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

OperatorLevel_3 ::=
  "LT" | "<" | "GT" | ">" | "==" | "EQ" | "LE" | "<=" | "GE" |
  ">=" | "!=" | "NE" | "<<" | "LSH" | ">>" | "RSH" | ">>>" | "RRSH"
```

```
Expr3 ::= Expr4 (S? OperatorLevel_3 S? Expr4 )*

OperatorLevel_4 ::= "*" | "/" | "%"

Expr4 ::= Expr5 (S? OperatorLevel_4 S? Expr5 )*

OperatorLevel_5 ::= "+" | "-"

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? TypeIdentifier S? ")" S?

UnaryOperator ::= "+" | "-" | "!" | "NOT" | "~"

TypeIdentifier ::= "int" | "String" | "float" | "boolean" | "Datetime" |
  "Decimal" | "Duration" | "Exception" | "Container" | "Element" | "Message" |
  "Bytes" | "XmlOutStream" | "BNFGrammar" | "BNFRule" | "Parser" |
  "ParseResult" | "AnyValue"

Value ::= Constructor | "null" | Method | Increment | VariableReference |
  KeyParameterReference | Literal | AttributeName |
  "$MAXINT" | "$MININT" |
  "$MINFLOAT" | "$MAXFLOAT" |"$NEGATIVEINFINITY" | "$POSITIVEINFINITY" |
  "$PI" | "$E"

Constructor ::= "new" S TypeIdentifier S? "(" S? ParameterList? S? ")" |
  NamedValue | ContainerValue

NamedValue ::= KeyName S? "=" S? Expression

ContainerValueStart ::= (NamedValue (S? "," S? NamedValue)*) | Expression

ContainerValue ::= "[" S? (ContainerValueStart (S? "," S? Expression)*)? S? "]"

KeyParameterReference ::= KeyName

Method ::= QualifiedIdentifier S? ("(" S? ParameterList? S? ")")?

Increment ::=
  ("++" | "--") S? VariableReference | VariableReference S? ("++" | "--")
```

```
  /* The variable must be declared as an integer or a float */

ParameterList ::= (Expression) (S? "," S? Expression)*

VariableReference ::= Identifier - TypeIdentifier

MethodDeclaration ::=
  ( "void" | TypeIdentifier ) S MethodName S? ParameterListDeclaration S? Block

MethodName ::= QualifiedIdentifier

ParameterListDeclaration ::=
  "(" S? ( SeqParameter (S? "," S? SeqParameter )* )? ")"

SeqParameter ::= TypeIdentifier S ParameterName

KeyParameter ::= KeyName S? "=" S? (TypeIdentifier | ConstantExpression)

ParameterName ::= Identifier

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

StatementExpression ::= Expression

Statement ::= S? (Statement1 | Statement2)

Statement1 ::= Block | SwitchStatement | TryStatement

Statement2 ::= ( IfStatement | ForStatement | WhileStatement | DoStatement |
  ReturnStatement | ThrowStatement | BreakStatement | ContinueStatement |
  Method | Increment | AssignmentStatement S? ";" ) | EmptyStatement

EmptyStatement ::= ";"

StatementSequence ::= ( S? VariableDeclaration S? ";" | Statement)*

Block ::= "{" StatementSequence S? "}"

SimpleStatement ::= S? (Statement1 | (Statement2? S? ";" S?)) | EmptyStatement

IfStatement ::=
  "if" ParentalExpression SimpleStatement ( S? "else" S? SimpleStatement)?
  /* Result of ParentalExpression must be boolean. */

ForStatement ::= "for" S? "(" S?  ForInit? S? ";" S?
  Expression? S? ";" S? ForStep? S? ")" S? SimpleStatement
```

```
ForInit ::= AssignmentStatement | VariableDeclaration

ForStepStatement ::= Method | Increment | AssignmentStatement

ForStep ::= ForStepStatement (S? "," S? ForStepStatement)*

WhileStatement ::= "while" ParentalExpression SimpleStatement
  /* Result of ParentalExpression must be boolean. */

DoStatement ::= "do" S? Statement S? "while" ParentalExpression
  /* Result of ParentalExpression must be boolean. */

SwitchStatement ::=
  "switch" ParentalExpression "{" SwitchBlockStatementVariant* S? "}"
  /* Result of ParentalExpression must be integer. Any variant may occur in the
     switch statement only once. */

SwitchBlockStatementVariant ::=
   S? ("default" | "case" S? ConstantExpression) S? ":" S? StatementSequence?
  /* Type  of ConstantExpression must be integer. */

ThrowStatement ::= "throw" S? ExceptionValue

ExceptionValue ::=
  "new" S? "Exception" S? "(" S? (Expression S?)? ")" | Identifier

TryStatement ::= "try" S? "{" S? StatementSequence S? "}" S?
  "catch" S? "(" S? "Exception" S?  Identifier S? ")" S?
  "{" S? StatementSequence S? "}"

ReturnStatement ::= "return" (S? Expression)?

BreakStatement ::= "break" (S? Identifier)?

ContinueStatement ::= "continue" (S? Identifier)?

AssignmentStatement ::= Identifier S? ((AssignmentOperator S? Expression) |
 (("=" S? Identifier))+ S? (AssignmentOperator S? Expression)) |
  S? AssignmentOperator S? Expression

AssignmentOperator ::=
  ("|" | "^" | "+" | "-" | "*" | "/" | "&" | "%") "=" | "="

VariableDeclaration ::= ("final" S | "external" S )*
  TypeIdentifier S VariableDeclarator (S? "," VariableDeclarator)*
```

```
VariableDeclarator ::= S? (AssignmentStatement | Identifier)

Occurrence ::= "required" | "optional" | "ignore" | "illegal" |"*" | "+" | "?"
        | (("occurs" S)? ("*" | "+" | "?" |
  (IntegerLiteral (S? ".." (S? ( "*" | IntegerLiteral))? )?)))
  /* The value of the second IntegerLiteral (after "..")  must be greater or equal
     to the first one. */

ExplicitCode ::= Block
  /* If result value is required it must be returned by the command "return value". */

/**** Script of the header of X-definition ****/

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 ::= XdefOptionItem ( S? "," S? XdefOptionItem )*
  /* Each item can be specified only once. */

XdefOptionItem ::=
  "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 ::= ValueExecutivePart*

ValueExecutivePart::= S? ( ValueValidationSection | ValueInitSection |
  ValueOnTrueSection | ValueOnErrorSection | ValueOnAbsenceSection |
  ValueDefaultSection | ValueCreateSection | ValueFinallySection |
  AttributeOptions | Reference | S | ";" ) S?
  /* Each item can be specified only once. */

ValueValidationSection ::=
 (Occurrence S? CheckValueSpecification?) | CheckValueSpecification

CheckValueSpecification ::= ExplicitCode | Expression | TypeMethodName
  /* Expression or ExplicitCode must return a value of boolean type. */

TypeMethodName ::= Identifier

ValueInitSection ::= "init" S? ( ExplicitCode | Statement )
  /* If Method or ExplicitCode returns a value, it will be ignored. */

ValueOnTrueSection ::= "onTrue" S? ( ExplicitCode | Statement )
  /* If Method or ExplicitCode returns a value, it will be ignored. */

ValueOnErrorSection ::= "onError" S? ( ExplicitCode | Statement )
  /* If Method or ExplicitCode returns a value, it will be ignored. */

ValueOnAbsenceSection ::= "onAbsence" S ( ExplicitCode | Statement )
  /* If Method or ExplicitCode returns a value, it will be ignored. */

ValueCreateSection ::= "create" S? ( ExplicitCode | Expression )
  /* Expression nebo ExplicitCode must return the value of String. */

ValueDefaultSection ::= "default" S? ( ExplicitCode | Expression )
  /* Expression nebo ExplicitCode must return the value of the String. */

ValueFinallySection ::= "finally" S? ( ExplicitCode | Statement )
  /* If Method or ExplicitCode returns a value, it will be ignored. */

AttributeOptions ::= ValueOptions

ValueOptions ::= "options" S? ValueOptionsList

ValueOptionsList ::=  ValueOption ( S? "," S? ValueOption )*
```

```
  /* Each item can be specified only once. */

ValueOption ::= "preserveTextWhiteSpaces" | "ignoreTextWhiteSpaces" |
  "setTextUpperCase" | "setTextLowerCase" | "trimText" | "noTrimText" |
  "preserveAttrWhiteSpaces" | "ignoreAttrWhiteSpaces" |
  "setAttrUpperCase" | "setAttrLowerCase" | "trimAttr" | "noTrimAttr"

/**** Script of elements ****/

ElementScript ::= ElementExecutivePart*

ElementExecutivePart ::= S? ( Occurrence | ElementVarSection |
  ElementMatchSection| ElementInitSection | ElementOnAbsenceSection |
  ElementOnExcessSection | ElementCreateSection | ElementFinallySection |
  ElementOptions | Reference | ElementForgetSection | ElementOnStartSection |
  S | ";" )
  /* Each item can be specified only once.
   * If the occurrence is not specified, the implicite value is "required" */

ElementVarSection ::= "var" S?
  (("{"(S? ElementVarSectionItem S?)* "}") | ElementVarSectionItem S?)

ElementVarSectionItem ::= TypeDeclaration | VariableDeclaration S? ";"  | S? ";"

ElementInitSection ::= "init" S? Statement?

ElementMatchSection ::= "match" S? ( Expression | ExplicitCode )
  /* Expression nebo ExplicitCode must return value of boolean. */

ElementOnStartSection ::= "onStartElement" S? Statement?

ElementOnExcessSection ::= "onExcess" S? Statement?

ElementOnAbsenceSection ::= "onAbsence" S? Statement?

ElementCreateSection ::= "create" S? (Expression | ExplicitCode)
  /* Expression nebo ExplicitCode must return value of Container or Element. */

ElementFinallySection ::= "finally" S? Statement?

ElementForgetSection ::= "forget"

ElementOptions ::= "options" S? ElementOptionsList

ElementOptionsList ::=  ElementOptionItem ( S?  "," S? ElementOptionItem )*
  /* Each item can be specified only once. */
```

```
ElementOptionItem ::= "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" |
  "nillable" | "noNillable"

/**** Script of the declaration part ****/

DeclarationScript ::= (S? (TypeDeclaration | ExternalMethodDeclaration
  | VariableDeclaration S? ";" | MethodDeclaration |  ";"))* S?

TypeDeclaration ::= ("type" S Identifier S?
  ( (Identifier (S? "," S? Identifier)* S? ";" S?) | TypeDeclarationBody ))
  | ("uniqueSet" S Identifier S?  UniqusetDeclarationBody )

TypeDeclarationBody ::= Expression
  /* Expression or ExplicitCode must return value of boolean or ParseResult.*/

ExternalMethodDeclaration ::= "external" S "method"
  ( (S? "{" S? MethodList S? "}" ) | S MethodListItem )

UniqusetDeclarationBody ::= ("{" S? Identifier S? ":" S? "?"? S? Expression
  (S? ";" S? S? Identifier S? ":" S? "?"? S? Expression)? S? ";"? S? "}")
  | Expression
  /* Expression or ExplicitCode must return value of boolean or ParseResult.*/

/*************************************************************************
 * BNF grammar
 *************************************************************************/

BNFGrammar ::= S? BNFMethodDeclarationSection? S? BNFRules S?

BNFMethodDeclarationSection ::= BNFMethodDeclaration (S? BNFMethodDeclaration)*

BNFRuleName ::= RawIdentifier

BNFExternalRuleName ::= "$" BNFRuleName

BNFRuleReference ::= BNFExternalRuleName | BNFRuleName

BNFMethodDeclaration ::= "%define" S BNFExternalRuleName S ":"
```

```
 S "$" QualifiedIdentifier BNFMethodparameters?

BNFMethodparameters ::= "(" S? BNFMethodparameter
 (S? "," S? BNFMethodparameter)* S? ")"

BNFMethodparameter ::= Digit+ | BNFTerminalSymbol

BNFTerminalSymbol ::= "'" (Char - "'")* "'" | '"' (Char - '"')* '"'
  | BNFHexaChracter (S? BNFHexaChracter)*

BNFHexaChracter ::= "#x" HexaDigit+

BNFRules ::= BNFRule (S BNFRule)*

BNFRule ::= BNFRuleName S? "::=" S? BNFExpression

BNFQuantificator ::= S? ("+" | "*" | "?" | BNFExplicitQuantificator)

BNFExplicitQuantificator ::= "{" S? Digit+ (S? "," S? Digit+)? S? "}"

BNFExpression ::= ( "(" S? BNFExpression  S? ")" (S? BNFQuantificator)?
  | BNFRestriction | BNFSelection | BNFSequence ) - ( BNFRuleName S? "::=" )

BNFSet ::= "[" BNFSetBody "]"

BNFSetBody ::= ("^"? Char - "]")+

BNFTerm ::= (BNFTerminalSymbol | BNFSet | BNFRuleReference)
  (S? BNFQuantificator)?

BNFSequence ::= BNFTerm ( S BNFExpression )*

BNFRestriction ::= BNFTerm S? "-" S? BNFExpression

BNFSelection ::= BNFTerm ( S? "|" S? BNFExpression )+

/*************************************************************************
* XComponent
*************************************************************************/

JavaTypeName ::= '<' S? $JavaQName (S? JavaTypeName )? (S? "," S?  JavaTypeName)* S? '>'

JavaTypedQName ::= $JavaQName ( S? JavaTypeName )?

/*JavaTypeName ::= '<' $JavaQName JavaTypeName* ("," JavaTypeName)* '>'
```

```
JavaTypedQName ::= $JavaQName JavaTypedQName?*/

XCComponent ::= ( S? XCComponentCommand )* S?

XCComponentCommand ::= ( XCBind | XCClass | XCEnum | XCInterface | XCRef ) S? ";"

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 $JavaQName S $JavaQName S XCLink

XCLink ::= "%link" S ("*" /*any*/ | XDPosition)
]]>
  </xd:BNFGrammar>

</xd:def>
```

Examples

Example of external methods used in some Syntea projects

`dbTab(t,s)`	values from the database table "t" and column "s"
`dateCurr(c)`	date "c" must be less than the actual date
`batchDate(c)`	date "c" must be less than the date of processed date
`productYear(r)`	date must be in the format "rrrr" and must be in the interval 1950 < value <= actual date
`monthSTK()`	value must be in the year and month in the format "yyy/MM"
`PIDCZ()`	value must be in the form of Czech personal ID
`CIDCZ()`	value must be in the form of company ID in Czech company register

X-definition of the control data of a batch used in projects ČKP and. SKP:

```
<xd:def xmlns:xd="http://www.syntea.cz/xdef/3.1"
        impl-version="2013_11.0"
        impl-date="12.11.2013"
        xd:name ="Batch"
        xd:root ="Batch">

 <Batch Sender        ="required num(1,9999999)"
        Receiver      ="required num(1,9999999)"
        Channel       ="required string(2,5)"
        SeqBatch      ="required int()"
        SeBatchRef    ="optional int()"
        Created       ="required datetime('yyyyMMddHHmmss')"
        Mode          ="required list('STD','TST','DEV')"
        OperationForm ="optional list('TRY','STD','DIR')"
        UserField     ="optional string()">

    <UserAttributes xd:script="occurs 0..1"
                    TransID  ="optional string()"  />

    <File xd:occurs = "1.."
          NazevSouboru  ="required string(1,50)"
          FormatSouboru ="required list('PIPE','XML')"
          DruhSouboru   ="required regex('[A-Z]\\d[A-Z]{1,2}')"
          PocetZaznamu  ="required int(0,99_999_999)">
       <CheckSum Type  ="required list('MD5')"
                 Value ="required string(32)" />
    </File>

 </Batch>

</xd:def>
```

[1]

[2]

[3]

[4]

[5]

[6]