X-definition 3.2

User Manual

Author: **J. Srp**

Translation: **V. Trojan**

This document is designed for users of X-definition

Content

1 Preface

This document is the user manual which contains the description of the programming tool *X-definition*. It is designed for programmers who use this technology in projects with XML data.

The text of this document assumes the basic knowledge of XML language.

The document is divided into several chapters focused on different parts of the X-definition technology. You can read the text either in sequential way or according to keywords presented at the header or the end of a chapter and you can skip to the referenced parts according to your actual knowledge - see green arrows ➡. At the beginning of each chapter see red arrows ➡ pointing to the appropriate chapter describing recommended knowledge.
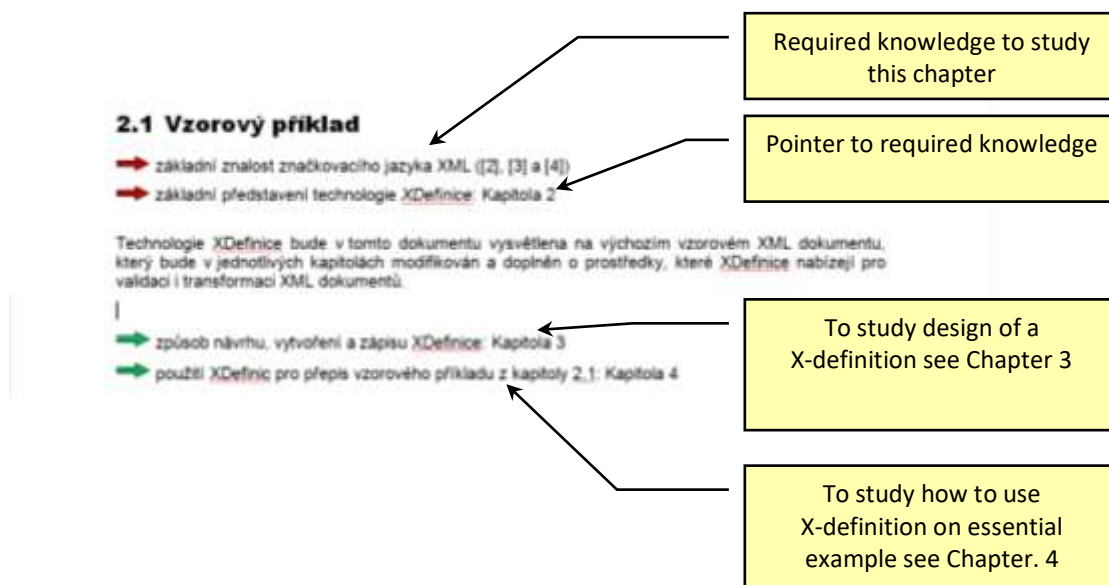
2   X-definition Technology

➡ basic knowledge of XML language ([1], [2]

X-definition is a programming language containing tools for

- Description of structure and values of XML documents and of the processing of XML data;

  o X-definition describes XML document as an object containing root element and child nodes. Elements may have attributes and child nodes may be either elements or text nodes. X-definition also describes types of values of attributes and of the text nodes.

- Validation of source XML data and to process them. This way X-definition can be used instead of XML schema or Schematron.

- Use XQuery format (only the versions distributed with Saxon library)

- With X-definition you can also describe rules for the construction of XML documents from variable resources. This way you can use X-definition also for transformation to a given XML structure (instead of XSLT).

- Generation of X-components - Java source code representing models of elements from X-definition (similar way as JAXB).

- Processing of XML documents in different language variants (see specification of "thesaurus").

- X-definition technology can process XML data of unlimited size (see the X-script command "forget" or "stream mode") [Doc1]

- Process data in the JSON format

X-definition - so as described data – is an XML document belonging to the namespace of X-definitions (ie "http://www.xdef.org/xdef/3.2"). The X-definition highly respects the structure of described data. This way it enables very fast and intuitive way to design the X-definitions even for large group of XML data. This feature facilitates both the design and maintenance of large projects. The X-definition is a meta language used for description of XML documents. In the process of development of X-definition it is possible also to specify commands invoked during processing of data (so called "actions"). Those commands are invoked in different situations during processing of X-definition and data.

The collection of X-definitions makes possible to compile to a project. Within the project may be references from a X-definition to other X-definitions.

The compiler of X-definitions creates from source X-definitions a Java object XDPool. This object enables to create an instance of object XDDocument. With this object it is possible to provide validation and to process the input XML data (by method "xparse") or even to construct the XML data (by method "xcreate"). The result may be either an instance of XML object (org.w3c.dom.Element) or it can be an instance of X-component (see the method "parseXC"). Moreover, result of processed X-definition may be stored to output stream (stream mode).

X-definition may also contain a number of declarations of macros, variables, "types" of text values and methods. The language used for description of programming part we call the "X-s**cript**" of X-definitions..

Syntax of X-script of X-definition is similar to the programming language Java or C. This way a source code is easily understandable for programmers. It is also possible to invoke from X-definition an external method designed in the Java language (see chapter 4.8).

➡ 2.2 X-script of X-definition

➡ 2.4 Exemplary XML Data

➡ 4.8 External (Java) Methods

2.1   Model of XML Element

Basic concept of the X-definition language is "model" of XML element (hereafter only "model"). The model describes structure of an XML element. Let's have following element describing a book:

```
<Book ISBN          = "123456789"
    Title           = "X-definition 3.2"
    PublicationDate = "01.02.2018"
    Editor          = "Syntea Software Group a.s."
    Price           = "195.50" >
  <Author J. Srp </Author>
</Book>
```

Our XML element contains different types of values: isbn, title, date of publication, price, authors). If we have to describe contents of such an element we can simply specify if a value is required or to describe interval of its occurrences (by the "quantifier"). Let's say the attributes "ISBN" and "Title" are required and the others are optional. Let's also accept that a book may have more authors or no author (who is author of The Bible?). Eventually we can specify also the types of text values. Such formal description of a XML element is the model of element "Book". It may look like:

```
<Book ISBN          = "required num(9,10); */ isbn is minumum 9 to 10 digits */"
    Title           = "required string();"
    PublicationDate = "optional date();"
    Editor          = "optional string();"
    Price           = "optional decimal()" >
  <Author xd:script = "occurs 0..*"> string(); </Author>
</Book>
```

As we can see the model is very similar to the described XML data. Values of text items are described by special language called "X-script of X-definition". Note also the quantifier of the element "Author" is written to the special auxiliary attribute „xd:script". This attribute does not belong to any namespace of described XML data. However, the name prefix "xs" defines that it belongs to the special namespace of X-definitions. By means of element models it is possible to describe a general structure of any XML data, including the properties of values, the occurrences of items and even the way of processing or construction of XML data.

2.2   X-script of X-definition

The X-script of the X-definition (hereafter only "X-script") is a language recorded to the X-definition as a text value. In the case of an attribute or of a text node (the child of an element) the X-script is written as a text on the place of the value of this text node. In the case of an element the X-script is written to the auxiliary attribute "xd:script" (where "xd" is the prefix of the X-definition namespace). The X-script enables to describe some other properties of data and/or the actions connected to different events during processing of a project. [Doc1]

The X-script consists of several sections. The order of individual sections in the X-script is arbitrary and none of sections is compulsory. The sections of the X-script are separated by a semicolon (;). At the end of the X-script record the or after the closing brace of a command assigned to a section of the X-script the specification of semicolon may be omitted. The X-script has the free format. The items of the X-script may by separated by unlimited number of spaces, new lines or tab characters (white spaces). [Doc1]

The X-script contains following sections [Doc1]:

1) **Validation section** – the validation section may be declared in the model of an element or in a text node or in an attribute. The validation section defines the quantifier (ie the specification of permitted occurrences of an item). The validation section of the attributes or of text nodes moreover contains also the specification of the "type" of a value, ie rule how to check correctness of the text value. The specification of type is described as an invocation of a method which parses the text value of an item. It returns "true" if the value complies to the parsing rules. Otherwise it returns "false" and it generates an

error message. If validation method is not specified the value of an attribute or text node can be any nonempty string of characters (ie the default validation method is "string()").

2) **Sections of actions** – the description of an action is introduced by a keyword specifying the event or the state of processing of the X-definition code. After the keyword follows a statement which is invoked in connection with the specified event. If no action is connected to an event the X-definition processor provides a default action.

3) **Options** – the specification of options (parameters of processing of X-definition) is introduced by the keyword "options" followed by the list of names separated by comma (",").

4) **Reference** – the reference specifies pointer to a part of the X-definition which is recorded on another place. The reference is introduced by a keyword "ref".

For more details see Chapter 4.7.

2.3 X-definition packages

4.5 Events and Actions

4.7.1 X-script of X-definition

8.1 Reference to another element model in X-definition

14.6 Options

2.3 X-definition Java Packages

The Java package distributed with X-definition 3.2 is:

- **`syntea_xdef3.1.jar`** contains the X-definition compiler and processor, the X-definition converter into the XML schema, and vice versa and few utilities package.

2.4 Exemplary XML Data

basic knowledge of XML language ([1], [2])

Now we'll explain the X-definition technology on the simple example of XML document. In the following chapters it will be modified and supplemented with instruments for the validation and/or the construction mode.

The exemplary XML document describes:

1) a vehicle,

   a) variant with elements and attributes:

```
<Vehicle type      = "SUV"
         vrn       = "1A23456"
         purchase  = "2011-02-01"
         manufacturer = "Škoda"
         model     = "Yeti" />
```

   b) variant using only XML elements:

```
<Vehicle>
    <type>SUV</type>
    <vrn>1A23456</vrn>
    <purchase>2011-02-01</purchase>
    <manufacturer>Škoda</manufacturer>
    <model>Yeti</model>
</Vehicle>
```

Vehicle is determined by the value of registration number of a vehicle (`vrn`). The record contains also the type of vehicle (`type`), the date of purchase of vehicle (`purchase`), the manufacturer of vehicle (manufacturer) and the model of vehicle (`model`).

2) a traffic accident:

a) variant with elements and attributes:

```xml
<Accident id  = "00123"
    date = "2011-05-17"
    injury   = "3"
    death    = "0"
    loss     = "600" >

    <vrn>1A23456</vrn>
    <vrn>1B23456</vrn>
</Accident>
```

b) variant using only XML elements:

```xml
<Accident>
    <vrn>00123</vrn>
    <date>2011-05-17</date>
    <injury>3</injury>
    <death>0</death>
    <loss>600</loss>

    <vrn>1A23456</vrn>
    <vrn>1B23456</vrn>
</Accid
```

The traffic accident is determined by the identification number "id". The record contains also the date of accident ("date"), the number of injured persons ("injury"), the number of death ("death") and total loss ("loss"). The child elements of an element "Accident" are listed as the registration numbers of vehicles participating on an accident.

➡ 3 Way of Usage of X-definition

➡ 14 Appendix C –

2.5 On-line Testing of X-definition

You can try to run an X-definition on WEB:

URL	Mode of test
http://xdef.syntea.cz/tutorial/examples/validate.html	Validation mode.
http://xdef.syntea.cz/tutorial/examples/compose.html	Construction mode.
http://xdef.syntea.cz/tutorial/examples/BNF.html	(E)BNF grammar.
http://xdef.syntea.cz/tutorial/examples/template.html	"Template" mode.

3 Way of Usage of X-definition

➡ basic knowledge of XML language ([1], [2])

➡ 2 X-definition technology

➡ 2.4 Exemplary XML Data

X-definition may be used in two basic processing modes:

a) **validation mode** – the input XML document is validated according to a X-definition. This means the input document is checked according to given X-definition. There are checked the occurrences of all parts of elements, attributes and text nodes and also there are checked the types of all text values.

➡ 4 Description of structure of XML document by X-definition

b) **construction mode** – the XML document (the result of processing) is constructed according to the rules described in a X-definition. In this mode the X-definition is processed as a formula for construction of the result XML document.

➡ 6 Construction Mode of X-definition

4 Description of structure of XML document by X-definition

➡️ basic knowledge of XML language ([1], [2])

➡️ 2 X-definition Technology

➡️ 2.1 Model of XML Element

➡️ 2.4 Exemplary XML data

The validation process of the input XML data starts from an element of input data (usually it is the root element of XML document). In the X-definition it must be declared a model according which the validation starts. The model describes a structure of the XML element where validation starts. In the model are described the attributes and child nodes (see more in chapter 4.6). Since one X-definition may contain more models each model must have an unambiguous name within whole X-definition.

4.1 Model of Exemplary Element

➡️ 2.4 Exemplary XML data

The model of our exemplary element is similar to the data it describes. Note instead of particular values of attributes or text nodes there is specified in the X-script the occurrence limit and the type of described values. The model of the element Vehicle will look as follows:

a) variant with elements and attributes:

```
<Vehicle
  type         = "enum('SUV', 'MPV', 'personal',
                 'truck', 'sport', 'other');"
  vrn          = "string(7);"
  purchase     = "date();"
  manufacturer = "string();"
  model        = "string();"
  />
```

b) variant using only XML elements:

```
<Vehicle>
  <type> enum('SUV', 'MPV', 'personal', 'truck',
    'sport', 'other');
  </type>
  <vrn> string(7); </vrn>
  <purchase> date(); </purchase>
  <manufacturer> string(); </manufacturer>
  <model> string(); </model>
</Vehicl
```

Where data type:

- **enum(p1, …)** – accepts a value from the list of parameters;

- **string()** – accepts any character string;

- **string**(n) - – accepts any character string where string length is equal to n;

- **string(m, n)** – accepts all strings where the string length is greater or equal to m and less or equal to n;

- **date()** – accepts strings with date in the ISO date format (yyyy-MM-dd)

Text values of attributes or of text child nodes ("types") are checked by specification of a "**validation method**". This method is invoked with specified parameters. if parameters are missing the brackets may be omitted – you can specify either "**string**" or "**string()**" when the text value is processed. If no validation method is specified the default type is any nonempty sequence of characters (ie "string").

If the source code of the model is incorrect the compiler of X-definition will report an error message including information about the location in the source data.

If we process the data from our exemplary example then no error will be reported. However, if any value of input data would be incorrect the list of error messages will be reported. Each error message contains the information about the location in source data.

➡️ 4.7 Sample of Complete X-definition

➡️ 4.2 Specification of Quantifier of Attributes and Text Nodes

4.2 Specification of Quantifier of Attributes and Text Nodes

The quantifier (the specification of occurrence) of an attribute or of a text node (the quantifier) may be described as follows:

- `required` – the occurrence of the item is required,
- `optional` – the occurrence of the item is not required,
- `illegal` – the occurrence of the item is illegal,
- `ignore` – the occurrence of the item is ignored.

If the quantifier is not specified the value of quantifier is set to "required" (the default value).

For example, let§ doscribe in our exemplary data that the values of the item "rn" and of the item "purchase" are required. All other items are optional. The model may look like:

a) variant with elements and attributes:

```
<Vehicle type    = "optional enum('SUV', 'MPV',
          'personal','truck', 'sport', 'other')"
     vrn          = "string(7)"
     purchase     = "date()"
     manufacturer = "optional string()"
     model        = "optional string" />
```

b) variant using only XML elements:

```
<Vehicle>
   <type> optional enum('SUV', 'MPV', 'personal',
       'truck', 'sport', 'oother')
   </type>
   <vrn> string(7) </vrn>
   <purchase> date() </purchase>
   <manufacturer> optional string() </manufacturer>
   <model> optional string </model>
</Vehicle>
```

Note that the quantifiers "required" in the model were omitted because this value of quantifier is default.

4.2.1 Fixed Values

If the value of an attribute or of a text node must have only the specified value it is possible to use the validation method "eq" (in following example see validation method of the item "type"):

a) variant with elements and attributes:

```
<Vehicle type        = "eq('personal')"
         vrn         = "string(7)"
         purchase    = "date()"
         manufacturer = "string()"
         model       = "string" />
```

```
<Vehicle>
    <type> eq('personal') </type>
    <vrn> string(7) </vrn>
    <purchase> date() </purchase>
    <manufacturer> string() </manufacturer>
    <model> string </model>
</Vehicle>
```

 In the case the required value in the result of processed data must be present even if it is missing in the input data you can use in the validation the keyword "fixed". The value of such item se set to this value even if it is missing in processed data. However, if the value differs from the required it is reported an error message:

a)   variant with elements and attributes:

```
<Vehicle type        = "fixed 'personal'"
         vrn         = "string(7)"
         purchase    = "date()"
         manufacturer = "string()"
         model       = "string" />
```

b)   variant using only XML elements:

```
<Vehicle>
    <type> fixed 'personal' </type>
    <vrn> string(7) </vrn>
    <purchase> date() </purchase>
    <manufacturer> string() </manufacturer>
    <model> string </model>
</Vehicle>
```

The value of the attribute "type" resp. of element "type" will be „personal" even if it is missing in the input data.

The specification of validation section as "fixed"

```
type     = "fixed 'personal'"
```

can be explicitly expressed as:

```
type     = "required eq('personal'); onAbsence setText('personal')"
```

where "onAbsence" specifies the X-script section which is invoked if the attribute or text node described in the model is missing. The method specification "setText(s)" sets in this situation the value from the parameter "s" to the attribute or text node.

Let's remark the value following the keyword "fixed" can be also a value of a variable (see Chapter 4.7.3):

```
...
<xd:declaration xd:scope="global">
       final String ver = "2.3.0-b02";
</xd:declaration>
...
<Vehicle typ = "fixed ver" ... />
```

4.2.2   Default Value

The specification of default value of an attribute or of text node is possible to set by a command introduced by the keyword "default". Let's say the default value of the type of vehicle is "personal":

a)   variant with elements and attributes:

```
<Vehicle type = "optional enum('SUV', 'MPV',
         'personal','truck', 'sport', 'other');
         default 'personal'"
         vrn         = "string(7)"
         purchase    = "date()"
         manufacturer = "string()"
         model       = "string" />
```

b)   variant using only XML elements:

```
<Vehicle>
    <type> optional enum('SUV', 'MPV', 'personal',
    'truck', 'sport', 'other');
     default 'personal'
    </type>
    <vrn> string(7) </vrn>
    <purchase> date() </purchase>
    <manufacturer> string() </manufacturer>
    <model> string </model>
</Vehicle>
```

4.3   Alternative specification of quantifiers

➡ 2.2 Script of X-definition

Because of backward compatibility with the previous versions of the X-definition the quantifiers may be expressed in different forms. Note the keyword "occurs" is not compulsory, it may be omitted. The meaning is the same:

```
occurs 1..1      required    if not specified the default value is required

occurs 0..1      optional    ?

occurs 0..*      0..*        *

occurs 1..*      1..*        +

occurs m..n      m..n        m..n

occurs n         n           n
```

You can write either "occurs 0..*" or only "*". Instead of specification of "optional string()" you can write just "? string()", instead of "required string()" you can write only "string()".

In the following X-script we will use the simplified version of quantifiers, e.g.:

a) variant with elements and attributes:

```
<List>
  <Vehicle xd:script = "+"
      type     = "? enum('SUV', 'MPV', 'personal',
                 'truck', 'sort', 'other')"
      vrn      = "string(7)"
      purchase = "date()"
      manufacturer = "? string()"
      model    = "? string" />
</List>
```

b) variant using only XML elements:

```
<List>
  <Vehicle xd:script = "+">
    <type> ? enum('SUV', 'MPV', 'personal',
        'truck', 'sport', 'other') </type>
    <vrn> string(7) </vrn>
    <purchase> date() </purchase>
    <manufacturer> ? string() </manufacturer>
    <model> ? string </model>
  </Vehicle>
</List>
```

4.4   Specification of Quantifier of Element

➡ 4.1 Model of Exemplary Element

For each internal element of a model, it is possible to specify quantifier in the "xd:script" attribute. If quantifier is not specified then the number of occurrences is set to one ("required").

If in the list of vehicles and we accept the list can be empty or the number of vehicles can be unlimited the model will be:

a) variant with elements and attributes:

```
<List>
  <Vehicle xd:script = "occurs 1..*"
      type     = "? enum('SUV', 'MPV', 'personal',
                 'truck', 'sort', 'other')"
      vrn      = "string(7)"
      purchase = "date()"
      manufacturer = "? string()"
      model    = "? string" />
</List>
```

b) variant using only XML elements:

```
<List>
  <Vehicle xd:script = " occurs 1..*">
    <type> ? enum('SUV', 'MPV', 'personal',
        'truck', 'sport', 'other') </type>
    <vrn> string(7) </vrn>
    <purchase> date() </purchase>
    <manufacturer> ? string() </manufacturer>
    <model> ? string </model>
  </Vehicle>
</List>
```

Note the element "Vehicle" in the variant b) can have any number of occurrences (even also none). If at least one vehicle is required in the list you may write e.g. plus sign:

```
<List>
  <Vehicle xd:script = "+"
    ...
</List>
```

➡ 4.7 Sample of Complete X-definition

Dokument: **Chyba! Nenalezen zdroj odkazů.***, verze:* **Chyba! Nenalezen zdroj odkazů.***, ze dne:*
REF zedne \h * MERGEFORMAT **Chyba! Nenalezen zdroj odkazů.**

4.5   Events and Actions

➡️ 4.1 Model of Exemplary Element

During processing of elements, attributes or text nodes there arises different events or situations. When such event becomes it is in the X-script possible to specify the action – an executable source code - which is in such situation invoked. Before the specification of an action it must be specified the name of the event. There are following events:

Table of events and actions Doc1]:

Name of event	Action result	Description
Create	Object	This event happens only in the construction mode before it is started the process of construction of an object. In the validation mode this action is ignored. The action must return an object or null. More see Chapter 6.
Default	String	This event is defined only for attributes or text nodes. Result of this action must be a string (an expression with type of result String). This string is set as value of the attribute of text node if it is not present in the source data. The action can be specified only in the X-script of an attribute or text node,
Match	boolean	This event happens before an element, attribute or e text node is processed. Result of this action must be a boolean value. The action has access only to name of the element and to it's namespace and to the list of attributes. If the action returns true then the item is processed. If it is false then this item is not processed. The action can be specified only in the X-script of an attribute or text node,
Finally	void	This event happens after the process of the actually processed element is finished. The action is executed before the processing of an element is finished. The action can be specified only in the X-script of an element, attribute or text node. When invoked first there are executed all "finally" actions of attributes, then actions of text nodes and as the last the finally action of the element itself.
Forget	Action here not allowed	This event means the processed element is connected to the processed data and it is released from the memory. There is not possible to specify any executable code. Usually this action is specified when the input data are too large to be stored to the memory.. Often this action is specified in the stream mode of processing. The action can be specified only in the X-script of elements.
Init	void	This event happens when an item will be processed. The action is invoked to initiate following process of the item. The action can be specified in the X-script of elements, attributes or text nodes.
onAbsence	void	This event happens if the item described by the model not exists or if the number of occurrences is lower then the required number. If the action is specified the error message is not reported (you can report your own message). The action can be specified in the X-script of elements, attributes or text nodes.
onExcess	void	This event happens if the number of occurrences exceeds the allowed maximum. If the action is specified for this event the error message is not reported(you can report your own message). The action can be specified in the X-script of elements, attributes or text nodes.
onTrue	void	This event happens when validation method does not detect an error. The action can be specified only for attributes or for text nodes.

		The action can be specified only in the X-script of attributes or text nodes.
onFalse	void	This event happens when validation method detects an error. If the actionis specified the error message is not reported (you can report your own message). The action can be specified only in the X-script of attributes or text nodes.
onIllegalAttr	void	This event happens when illegal attribute occurs (not declared attribute or illegal attribute). If the action is specified for this event the error message is not reported(you can report your own message). The action can be specified only in the X-script of elements.
onIllegalElement	void	This event happens when illegal element occurs (not declared element or illegal element). If the action is specified for this event the error message is not reported (you can report your own message). The action can be specified only in the X-script of element.
onIllegalText	void	This event happens when an illegal text occurs (not declared text or illegal text). If the action is specified for this event the error message is not reported (you can report your own message). The action can be specified only in the X-script of an element.
onIllegalRoot	void	This event happens when process tarts with an element which is not specified as a root of XML document. If the action is specified for this event the error message is not reported (y report your own message). The action can be specified only in the X-script of X-definition.
onStartElement	void	This event happens when starts proceeding of element body (ie after all attributes were processed. The action is specified in only the X-script of element.
onXmlError	void	This event happens when an XML error occurs, You can this way catch XML error in the input data. Normally the program is finished with the exception. SException. The action can be specified only in the X-script of X-definition.

The examples of using of actions are in following chapters.

4.5.1 Events Connected to Elements, Attributes or Text Nodes.

4.5.2 Events Connected with Element

4.5.3 Events Connected with Text Nodes or Attributes

4.5.1 Events Connected to Elements, Attributes or Text Nodes.

4.1 Model of Exemplary Element

Let's illustrate the actions of two events which may occur:

- onAbsence – the event happens if the required minimum occurrences specified by quantifier is not reached;

- onExcess – the event happens if the maximum occurrence specified by quantifier exceeds (of course this event may happen only with elements),

Let's say the quantifier of elements "Vehicle" in the model "Vehicles" is set to the interval 1 to 10. We can show the usage of actions "onAbsence" and "onExcess" in the following example:

a) variant with elements and attributes:

```
<List>
  <Vehicle xd:script = "1..10;
            onAbsence outln('Missing Vehicle!');
            onExcess outln('Too many Vehicles!');"
       type = "enum('SUV', 'MPV', 'personal',
            'truck', 'sport', 'other')"
       vrn = "string(7);
            onAbsence outln('Missing vrn!')"
       purchase = "date()"
       manufacturer = "string()"
       model = "string" />
</List>
```

b) variant using only XML elements:

```
<List>
  <Vehicle xd:script = "1..10;
       onAbsence outln('Missing Vehicle!');
       onExcess outln('Too many Vehicles!')">
    <type> enun('SUV', 'MPV', 'personal', 'truck',
            'sport', 'other')
    </type>
    <vrn>
       string(7); onAbsence outln('Missing vrn!')
    </vrn>
    <purchase> date() </purchase>
    <manufacturer> string() </manufacturer>
    <model> string </model>
  </Vehicle>
</List>
```

Note that

- The sections of X-script are separated by semicolon (";");

- The action specified for respective use the method `outln`. This method writes to the standard output the line with the string from the argument. This method is invoked only if the respective event happens.

➡ 4.5.2 Events Connected with Element

➡ 4.5.3 Events Connected with Text Nodes or Attributes

➡ 4.7 Sample of Complete X-definition

➡ 4.8 External (Java) Methods

4.5.2   Events Connected with Element

➡ 4 Model of Exemplary Element

We'll illustrate actions connected with two following events:

- `onStartElement` – the action is invoked immediately after all attributes of the element are processed and after check of occurrence interval. However, it happens before child nodes of the element are processed.

- `finally` – the action is invoked when processing of the element and of all it's child nodes is finished, but before it is connected to the parent node (see event "forget"). You can specify the action forget also for attributes and for text nodes. Note the action "forget" if specified is executed first for all text nodes, then for attributes and then for the element itself.

The example of usage:

a) variant with elements and attributes:

```
<Vehicle xd:script = "
        onStartElement outln('Vehicle start');
        finally outln('Vehicle end')"
     type = "enum('SUV', 'MPV', 'personal',
            'truck', 'sport', 'other')"
     vrn = "string(7);
        finally outln('vrn=' + getText())"
     purchase = "date()"
     manufacturer = "string()"
     model = "string" />
```

b) variant using only XML elements:

```
<Vehicle xd:script = "
     onStartElement outln('Vehicle start');
     finally outln('Vehicle end')">
  <type> enum('SUV', 'MPV', 'personal', 'truck',
            'sport', 'other') </type>
  <vrn> string(7);
    finally outln('vrn=' +  + getText())
  </vrn>
  <purchase> date() </purchase>
  <manufacturer> string() </manufacturer>
  <model> string </model>
</Vehicle>
```

Note the method "getText" returns the string with the value of an attribute or a text node. The output will be:
```
Vehicle start
vrn=1234567
Vehicle end
```

4.5.3   Events Connected with Text Nodes or Attributes

➡️ 4 Model of Exemplary Element

In the following example we'll introduce two events and it's actions which are used only with text nodes or with attributes. Those actions are declared in the validation section:

- onTrue – the action is invoked if the value of a text node or of an attribute is correct;

- onFalse – the action is invoked if the value of a text node or of an attribute is incorrect. Note that if this action is specified **no error message is reported**.

For the item "purchase" the specification of above mentioned actions will look like:

a) variant with elements and attributes:

```
<Vehicle type = "enum('SUV', 'MPV', 'personal',
                'truck', 'sport', 'other')"
        vrn = "string(7);"
        purchase = "date();
         onTrue outln('Purhase date: ' + getText());
         onFalse outln('Incorrect date format')"
        manufacturer = "string()"
        model = "string" />
```

b) variant using only XML elements:

```
<Vehicle>
    <type> enum('SUV', 'MPV', 'personal', 'truck',
                'sport', 'other')</type>
    <vrn>string(7)</vrn>
    <purchase>date();
        onTrue outln('Purhase date: ' + getText());
        onFalse outln('Incorrect date format')
    </purchase>
    <manufacturer>string()</manufacturer>
    <model>string</model>
</Vehicle>
```

When the value of the item "purchase" will not be valid according to the required format it will be printed the line "Incorrect date format" on the standard output stream. If the value is correct then it will be printed the line with the value of purchase date.

➡️ finální sestavení X-definice pro proces validace: Kapitola 4.7

➡️ implementace uživatelských metod pro akce událostí: Kapitola 4.8

4.6   Processing of Large XML Data

➡️ 4.5 Events and Actions

The elements from input XML data are normally connected to the result XML document in the memory of the computer. If there are processed very large XML data it may happen the computer have not enough memory to store them. Therefore it is possible to specify the keyword "forget" that cause the processed element will be removed from the memory (not be connected to the result data) after it was processed. However, the information that it was processed (e.g. the counter of occurrences of this element) remains available. Removing of the redundant element happens at the end of processing of the element, even after all events "finally".

In the following example is the illustration how to express that the element "Vehicle" will be removed from memory after it was processed:

a) variant with elements and attributes:

```
<Vehicle xd:script = "*; forget"
        type    = "enum('SUV', 'MPV', 'personal',
                   'truck', 'sport', 'other')"
        vrn     = "string(7)"
        purchase = "date()"
        manufacturer= "string()"
        model   = "string" />
```

b) variant using only XML elements:

```
<Vehicle xd:script = "*; forget">
    <type> enum('SUV', 'MPV', 'personal', 'truck',
                'sport', 'other') </type>
    <vrn>string(7)</vrn>
    <purchase> date() </purchase>
    <manufacturer>string()</manufacurer>
    <model>string</model>
</Vehicle>
```

If you specify the keyword "forget" the element will be removed from the memory of the computer. That is why it should be stored to the result data on the output stream if you need such data. This will be described in detail in the chapter 7.10.

➡ finální sestavení X-definice pro proces validace: Kapitola 4.7

➡ 7.10 Continuous XML document writing

4.7   Sample of Complete X-definition

➡ 4.1 Model of Exemplary Element

The X-definition is the XML document. If it is used for validation purpose it must be specified the name of the model (or names of models) of the root element of processed data. Example of the complete X-definition:

a)   variant with elements and attributes:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name  = "garage"
        xd:root  = "Vehicles">

    <List>
        <Vehicle xd:script = "*"
            type = "enum('SUV', 'MPV', 'personal',
                'truck', 'sport', 'other')"
            vrn     = "string(7)"
            purchase = "date()"
            manufacturer = "string()"
            model = "string" />
    </List>
</xd:def>
```

b)   variant using only XML elements:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name  = "garage"
        xd:root  = "Vehicles">

    <List>
        <Vehicle xd:script = "*">
            <type> enum('SUV', 'MPV', 'personal',
                'truck', 'sport', 'other')
            </type>
            <vrn>string(7)</vrn>
            <purchase> date() </purchase>
            <manufacturer>string()</manufacturer>
            <model>string</model>
        </Vehicle>
    </List>
</xd:def>
```

where:

- The namespace of X-definition is "http://www.xdef.org/xdef/3.2", the namespace prefix is "xd";

- The name of X-definition is specified in the attribute "xd:name" (ie "garage");

- The name of model which describes the root element of processed data is specified in the attribute "xd:root" (ie "List");.

In the X-definiton it may be also included the declaration of variables, methods, BNF grammars, thesaurus and macros. This will be described later.

4.7.1   X-script of X-definition

The places where there is possible to write X-script are marked iń the following example is marked by the red „SCRIPT" word.

As you can see the X-script may be recorded as the value of the attribute "xd:script" at different places of the X-defrinition:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name  = "garage"
        xd:root  = "Vehicles"
        xd:script = "SCRIPT">

    <xd:declaratiodn>
        SCRIPT
    </xd:declaration>

    <List xd:script = "SCRIPT">
        <Vehicle xd:script = "SCRIPT"
            type = "SCRIPT"
            vrn  = "SCRIPT"
            ... />
        SCRIPT
    </List>

</xd:def>
```

4.7.2   Head of X-definition

➡ 2 X-definition Technology

➡ 4.1 Model of Exemplary Element

➡ 4.2 Specification of Quantifier of Attributes and Text Nodes

➡ 4.4 Specification of Quantifier of Element

➡ 4.5 Events and Actions

➡ 4.7 Sample of Complete X-definition

The head of X-definition we consider the set of attributes of the element "xd:def". The attributes are:

xmlns:xd   the namespace of X-definitions, ie http://www.xdef.org/xdef/3.2. The attribute is required.

xd:name   the name of X-definition. One X-definition in the project may be without name. All the other X-definitions must have a name and it must be unambiguous within the project. The name must be a valid identifier.

xd:root   the list of qualified names referring to models of root elements. The names in the list are separated by character "|". Instead of name of root element you can specify also asterisk (*) to specify that it is accepted any other root element not mentioned in the list. [Doc1]. The root list is used only in the validation mode. Example:

```
xd:root = "Vehicle | Accident | List | *"
```

xd:script   you can specify here the X-script used for the whole X-definition (e.g. specification of options). The attribute is not required.

xd:include   the list of URLs of other X-definitions which will be compiled together with this X-definition. The separator of items in the list is comma (,). The attribute is not required.

impl-XXX   not required atributes used for the implementaion information of an application. „XXX" here is the sufix of the name of an implementaion parameter. Number of implementation attributes is not limited. However, the prefix of names must be "impl-". In the X-script you can get the value of the implementation parameter by the method `getImplProperty('XXX')` – the parameter is the sufix of the name of implementation artribute. E.g. the attribute `impl-version`="1.2.3" may be used as the information about the version of project. The value "1.2.3" is retuned by `getImplProperty('version')`.

The example as follows writes at the end of `process the name of author and the implementation version`:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name = "garage"
      xd:root = "Vehicles"
      impl-version = "1.0.53.169"
      impl-author = "J. Srp">

   <List xd:script = "finally { outln('Author: ' + getImplProperty('author'));
                              outln('Version: ' + getImplProperty('version')); }">
      <Vehicle xd:script = "*"
          type  = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
          vrn   = "string(7)"
          purchase = "date()"
          manufacturer = "string()"
          model = "string()" />
   </List>
</xd:def>
```

4.7.3   Declaration Section of X-definition

In the X-definition it tis possible to declare variables, methods and validation types which it is possible to use in the code of the X-script. The scope of visibility of declared objects depends on the parameter in the declaration section. Therefore you use the declared objects either in any X-definition of the project or just in the X-definition where a declaration part is specified.

The X-script of the declaration part is recorded as a text value of the element "xd:declaration". The scope of visibility is specified by attribute "xd:scope" where it's value can be either "global" or "local". The value "local" sets the scope of visibility only to X-definition where the declaration was written. The value "global" sets visibility to all X-definitions. If the attribute "scope" is not specified the default visibility is set to "global". The element "xd:declaration" must be recorded as a direct child of X-definition (ie. of the element "xd:def"). Each X-definition may contain even several declaraton parts. So you can separate different parts of declarations to different sections or even to different X-definitions. This makes it possible eg. to separate "local" objects and "global" objects etc.

In XML language some characters have a special meaning. So it is not possible to write them to the X-script and you must write them to the XML text as entities (e.g.. „&" as „&" or „<" as „<" etc.). To prevent it you can write the X-script of a declaration part to a CDATA section. This way you can write normally those characters. Example:

```
xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
...
  <xd:declaration xd:scope = "global">
  <![CDATA[
      int x = 255 & 7;
  ]]>
  </xd:declaration>
...
  <Vehicle ... vrn = "string(x)" ... />
...
</xd:def>
```

4.7.3.1   Expressions in X-script

The result of an expression in the X-script is a value corresponding to the operation and to data types of operands of an expression. E.g. in the following example you can declare the variable "max" of type int, visible only in the current X definition, and in another declaration part to declare the globaly visible method "maxVRN" which is used in as a parameter of the validation method of the attribute "vrn":

```
...
<Vehicle ... vrn = "string(maxVRN())" />
...
<xd:declaration xd:scope = "local">
      int max = 4;
</xd:declaration>
...
<xd:declaration xd:scope = "global">
      int maxVRN() {
            return max + 3;
      }
</xd:declaration>
```

The list of all operators supported in the X-script expressions is in the table as follows. Note that some operators have an alternative notation to make them more readable in the X-script of attributes or of text nodes:

Binary operators:			
Operator	**Alias**	**Meaning**	**Data types**
&	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

An alternative notation should be preferred outside the CDATA section.

Example of an arithmetic expression written using the alternate notation:

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

The same expression with the base symbols (this should be written in a CDATA section):

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

In the expressions, the priority of operators can be changed by parentheses in the usual way. The priority of the operators is the same as Java or "C". Also, the automated conversions of types run similarly as Java (e.g. "int" or "float"). If one of arguments of an addition operation (ie string concatenation) is a "String" value and the other argument is "int" or "float" (or vice versa), an automated conversion to the "String" type is performed. A conversion can also be forced by typing the type name into parentheses. You can also convert any value type to the String by using the "toString" method.

The X-script also supports the conditional expression that has the form:

```
boolean_value ? value1 : value2
```

where *value1* and *value2* must be values of the same type. If *boolean_value* is true the result of the conditional expression will be *value1* otherwise it will be *value2*.

4.7.3.2 Statement in X-script

The statements in the X-script have very similar syntax and semantics as statements in Java. They are separated by a semicolon and they are always treated regardless of the source text line spacing (ie the spaces, tabs and the new lines are treated as a white space). Unlika In Java the characters of String constructor can be entered on multiple lines between the quotation marks. Note that new lines are translated to spaces.

Unlike Java also in the X-script the labels are not supported. Therefore there are not supported also label references in the commands "break" or "continue". Otherwise the syntax of statements in the X-script is similar to writing in the Java language.

Comments

The comments may appear anywhere in the X-script. Every comment begins with a sequence of * / * and ends * /. However, unlike as in Java the line comments (starting with //) are not allowed.

Identifiers

The identifiers in the X-script (such as naming a variable, method, type, etc.) must either begin with a letter or with the underscore (_) character followed by letters, numbers, or underscores. In addition unlike as in Java in the middle of the identifier, it may be written a dot (.), colon (:), or minus character (-) (as the standard XML naming rules). The names in the X-script are case-sensitive. The names of variables and constants or methods may also contain a dollar character ($). National alphabet characters may also be used.

Variable declaration statement

The variable declaration statement begins with the name of the data type followed by a sequence of variable names (identifiers) separated by comma character. the variable declaration a assignment statement can also be specified (as in Java).

The variable can be declared in the declaration section or within some statements (e.g. compound statements, "for" statement etc.).

 The variable declaration statement can be preceded by a qualifier describing properties of a variable:

- final   the variable has a fixed value in the initialization process. This value can't be changed in other statements of the X-script.

- external   the variable is accessible from the external Java program (this qualifier may only be used in the the declaration part of X-definition).

Example of variable declaration in declaration part:

```
<xd:declaration xd:scope = "global">
    /* Declaration of global variables: */
    int a, b, c = 0;
    final String $count = "120";
    external float _average_02;
</xd:declaration>
```

Example of variable declaration in statement:

```
for (int i=0, max=10; i < max; i++) {
  float x, y;
    ...
}
```

Assignment statement

The assignment statement begins with the name of the variable that we want to assign and it follows value after the equal sign (=). The assignments can be chained:

```
<xd:declaration>
    int a, b;
    ...
    a = b = 20;
    b += 12;
    ...
</xd:declaration>
```

if statement

if (*condition*) *statement*

or

if (*condition*) *statement* else *statement*

while statement

while (*condition*) *statement*

do-while statement

do *statement* while (*condition*)

for statement

for (*statement*; *condition*; *statement*) *statement*

Note the "for each" statement is not supported in the X-script (e.g. for (variable : array) {...}).

switch statement

switch (*expression of the type int or String*) {

      case *value* : *statement (optional, may be repeated)*

      default : *statement (optional)*

}

Compound statement

The compoound statement contains a sequence of statements separated by semicolons between braces

{ *statements* }

Try-catch statement

try { *statements* } catch (Exception *name*) { *statements* }

Exception invocation

An exception in the X-script can be invoked by the "throw" command. Its parameter is an Exception object.

throw new Exception(message);

Example:

```
<xd:declaration>
    void x() {
        try {
            throw new Exception('Exception in method x');
        } catch (Exception ex) {
            outln(ex.getMessage());
        }
    }
</xd:declaration>
```

break statement

The break statement escapes the switch, for, while, do-while, or compound statement.

continue statement

The continue statement jumps to the beginning of the "for", "while" or "do"-"while" statement.

```
return statement
```

The "return" statement escapes the method and it returns process to the code where the method was invoked. If a method should return a value the return statement must be followed by the value of the appropriate type.

4.7.3.3   Declared Variables

The variables listed in the declaration section are visible in an X-definition according to specified scope. We are talking about declared variables. Declaring of a variable can be preceded by a qualifier "final" and/or "external".

The qualifier "external" indicates that the value of a variable can be set externally and it is not initialized by the X-definition processor. The resource allocated as an external variable is not released frm the memory at the end of process (database objects, streams etc.).

The qualifier "final" sets a variable to be constant and thus it is prevented from any further modification. To such object must be assigned a value in the declaration statement.

Example:

```
<xd:declaration xd:scope = "global">
     /* Global variables. */
     external int globalVariable;
     final String const = "KONSTANTA";
     external final extConst;
     int id, start = 0, konec = 50;
     ...
</xd:declaration>
```

To declared variables, as well as Java class objects are assigned to an initial value. That is, the uninitialized global variables are set to the default values (zero for numbers, false for booleans, null for other objects etc.)

All objects that are created in the X-script are, if necessary, automatically closed (by the method "close") immediately after completion of the validation or design: ie. that after returning control code back to Java program where 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 initialized by the X-script command. Eg. in the case of a "Connection" type of object (database) 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 from the Java program where the process was invoked. That is the variable can't be initialized in the declaration statement (nevertheless, the external variable can be also marked as "final" and you can't change it in the X-script).

4.7.3.4   Declaration of Method

➡ 5.5 Container

In declaration part of X-definition can be declared a method. Write syntax of method declaration is based on Java. All the methods declared in a declaration part are visible from the X-script according to "scope" parameter. At the beginning of the method declaration is the identifier of the method result type. It is followed by a name of the method and the comma-separated list of parameters in parentheses. The parameter list may also be empty. In the statement block of a method declaration the statement "return" forces return to the place from which a method was invoked. If the declared method does not return a value (the return type is void) the method may finish with the "return" statement without parameter. If the method should return a value as a result the "return" statement must succeed a value of the result of a method:

```
<xd:declaration>
      void printText(String par1, String par2, String par3) {
        ...
        return;
      }
      int getCount() {
        ...
        return 123;
      }
      boolean compare(String a, int b) {
        ...
        return false;
      }
      ...
</xd:declaration>
```

A special case is a method with a parameter type "Container" specified as the last item of a parameter list. When calling such a method, instead of the last parameter you can specify a comma-separated list of so-called "named values". This is appropriate when it is necessary to declare too many different parameter variants [Doc 1]:

```
<xd:declaration>
      /* Method with parameter Container with named items. */
      String getConVal(int i, Container data) {
          String name = 'p' + i;
          return data.hasNamedItem(name) ? data.getNamedItem(name) : 'null';
      }
      /* This method which invokes method getConVal with different parameters and prints them. */
      void printConVal() {
          outln( getConVal(%p1='DATA', %p2='Secret') );
          outln("==========");
          outln( getConVal(%a='A', %b='B', %cD.eF:1-3='Some value') );
      }
      ...
</xd:declaration>
```

The output will be after calling the method `PrintConVal()`:

```
%p1=DATA
%p2=Secret
=======
%a=A
%b=B
%cD.eF:1-3=Some value
```

Note that the variable declared in a body of a method, in a statement block (between the opening and closing brackets), in a "for" statement etc. - so as in Java - is not initially set to an approptiate initial value and therefore attempting to access an uninitialized variable causes an error at the compile time of X-definition.

4.7.3.5   Declaration of Validation Type

As we have already shown a validation method provides parsing of the text values and we speak about the "type" of a value. In order not to specify validation methods always with the parameter list (and also to increase clarity of source code) it is possible to declare certain type of value as a named type. The declaration is written in the declaration part starting with the keyword "type". So as the other declarations in declaration section the visibility is set according to parameter "scope". The syntax of type declaration is described in [Doc 1]:

type *name* type_spacification (parameters);

Example:

```
...
<xd:declaration>
      type string7 string(7);
      type rgbColor enum("red", "green", "blue");
      ...
</xd:declaration>
...
<Item id="string7" color="optional rgbColor" />
...
```

The declared type can also be written as method invocation with parameters in brackets. However, the list of parameters must be empty:

```
...
<Item id="string7()" color="optional rgbColor()" />
...
```

4.7.3.6   Command Block in X-script Section

➤ 2.2 X-script of X-definition

➤ 4.5 Events and Actions

In a X-script section you can write a statement block (ie, a sequence of statements) which is bounded by brackets ("{" and "}"). The statement block is similar to a statement block in the method declaration (ie it may contain a "return" statement):

```
<Vehicle xd:script = "occurs 1..100;
         onAbsence {
             outln('Element Vehicle is missing.');
             outln('Reqiuired minimum 1 occurence.');
             return;
         }
         onExcess { outln('Maximum 100 occurences.'); }"
    ...
```

4.7.3.7   Statement Block in the X-script Sections Requiring Return Value

➤ 6.1 Create Section in X-script

If a X-script statement block is specified in a section that requires a value (such as the create section, the match section or a validation method), the corresponding data must be returned with the "return" statement followed by the result value:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name  = "accidents"
      xd:root  = "Accidents">

   <Accidents xd:script = "
             create {
                 outln('Konstructin of element Accident.');
                 return true;
             }">
        <Accident xd:script = "occurs 0..*;
             create {
                 i++;
                 return 3;
             }" ...>
          ...
        </Accident>
   </Accidents>

   <xd:declaration>
      int i = 0;
   </xd:declaration>
</xd:def>
```

If the return statement is not specified, or if return value is missing the X definition compiler reports an error.

4.7.3.8 Ambiguous Objects in the X-Definition

If an ambiguous declaration of an model appears in the X-definition, the X-definition compiler will report an appropriate error. A typical cause of an ambiguous object is specification of two successive elements with the same name when the first element is specified with variable number of occurrences, e.g.:

```
<Accidents>
  <Accident xd:script = "occurs 0..*" ... >
  <Accident ... >
</Accidents>
```

It is obvious that the second "Accident" model will never be applied. However, if the first model was specified with a fixed number of occurrences, the X-definition would be unambiguous.

```
<Accidents>
  <Accident xd:script = "occurs 2" ... >
  <Accident ... >
</Accidents>
```

Note: if the "match" section is specified in the first element model with variable number of occurrences then the ambiguity error is not reported.

4.7.3.9 Varibale Declaration in Elements

If you need to have a variable that is bound with an instance of processed element in a model you can declare such variable in the variable declaration section of the X-script of element. Variable declaration section starts with the keyword "var" and it MUST be specified before any executive section of the X-script of element (only quantifier can be specified before it):

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:root = "Measurements">

<Measurements>
    <Measurement xd:script="occurs 1..*;
                var { int count = 0; float total = 0; } /* MUST be before other sections! */
                finally outln(@date +'; measurements: ' + count + ', average=' + (total/count));"
                date = "required date()" >
      <Value xd:script = "occurs 1..*; finally count++;">
        required double; onTrue total += (float) getParsedValue();
      </Value>
    </Measurement>
  </Measurements>
</xd:def>
```

Note that "@date" refers to the attribute "date" of the current element (ie "Measurement"). In this context it is the string which is value of the attribute "date". The method "getParsedValue()" returns parsed value from the result of validation method (in this case the float value of the text of element "value" - the result of method "double" is converted to the X-script value "float").

If input data are:

```
<Measurements>
  <Measurement date="2017-08-10T11:31:05">
    <Value>10</Value>
    <Value>11.8</Value>
    <Value>9.4</Value>
  </Measurement>
  <Measurement date="2017-08-10T13:01:27">
    <Value>12.35</Value>
  </Measurement>
</Measurements>
```

it will print the messages:

```
2017-08-10T11:31:05; measurements: 3, average=10.4
2017-08-10T13:01:27; measurements: 1, average=12.35
```

4.7.3.10 Specification of X-definition in XML Document

➡ 2 X-definition Technology

➡ 4.1 Model of Exemplary Element

It is possible to specify an X-definition that handles XML data directly in an XML document in the "location" attribute from the "http://www.syntea.com/xdef/instance" namespace. Let the prefix of this namespace be "xdi". You can specify the URL of the X-definition by the "xdi:location" attribute in the XML document element. You can also specify a list of X-definitions URLs separated by the character comma (,). In the "xdi:xdefName" attribute you can specify the name of X-definition that describes the corresponding XML object. [Doc1]:

```
<?xml version="1.0" ?>

<Vehicles xmlns:xdi  = "http://www.cz.syntea/xdef/instance"
       xdi:location   = "/path/to/Vehicles.xdef"
       xdi:xdefName    = "garage" >
  <Vehicle ... >
        ...

  <Vehicle ... >

</Vehicles>
```

4.8   External (Java) Methods

➡ 4.5 Events and Actions

➡ 4.7 Sample of Complete X-definition

In the X-script in addition to built-in methods you can also specify and in to use "external" methods which are embedded in a Java class accessible to X-definition processor. In order to invoke an external method in a X-script it is necessary to declare external methods in "xd:declaration" element. Declared external methods can be used in the X-script of X-definition. The list of external methods is privided by the command "external method {list}" where the list contains items separated by semicolon in the following format:

```
ReturnType FullyQualifiedMethodName(parameter list) [as AliasName];
```

where

- **The return type** of value returned from the method (see the complete list of data types defined for the X-definition X-script in Chapter 14.2).

- **Fully qualified method name**, ie, package, class name, and method name separated by a dot character.

- **The parameter list** is described by the data types of the method parameters. The parameter list in in parenthesis contains the specification of parameter types separated by the comma character. It must be listed in the same order of the order as the parameters specified in the method header declared in the Java code.

- **Alias name** (optional):

- o If the alias name is specified it must precede the keyword "as". The alias name must be of course unambiguous in the entire project.

- o The alias thus represents a new name for the appropriate external method. The alias name overlays the original name of an external method. Thus, if an alias is used, it is no longer possible to invoke the external method with its original name.

- o As an alias you can use the original name of another method declared in the attribute "methods". In this case the alias must have also the second one method; otherwise the X-definition compiler will report an error.

- o If the alias is not specified the external method is referred in the X-script with its unqualified name (ie without the name of the package and the class name).

If the list contains only one item, the brackets "{" and "}" may be omitted. So as all objects declared in "xd:declaration", the declared external methods have visibility according to the paramater "scope".

Example:

```
<xd:declaration scope = "global">
  external method {
    void myorg.project.xd.util.Support.printItemPrice(String, String) as print;
    float myorg.project.xd.util.Data.price(String);
  }
...
  external method double myorg.Calculation.getDiscount(float);
</xd:declaration>
```

The X-definition resolves overloading of methods similarly as Java. Therefore, in the list of declared external methods can be specified methods with the same name differing only in the parameter list. The X-definition compiler ensures that the corresponding method is called from the appropriate Java class. You can also use the alias name to distinguish overloaded methods.

In the code of Java class each method declared in the X-script must be declared as a "public" and "static". The type of return value must match a type specified for the X-script. The validation methods must return "ParseResult" value.

Types of parameters of an external method must also match a data type available as X-script data type. As the first parameter it is possible to specify the parameter with the data type org.xdef.proc.XXNode (or the derived type org.xdef.proc.XXData or the type org.xdef.proc.XXElement), which represents the currently processed XML node. It makes available the value of an attribute or a text node in the external method. If this first parameter is specified in the declaration of the external method the X-definition passes to the external method the object corresponding to the current processed XML.

In the X-Definition the numeric data types "int" and "float" are implemented as 64bit values (corresponding to Java types "long" and "double"). However, it is also possible to specify the types "long" and "double" in the X-script, but the result is same as "int" and "float". It is necessary to take this into account when implementing the external method and passing parameters between the X-definition and the external Java method. Parameter conversion to int and float to external methods is done automatically, but numeric loss can occur. An example of use is given in Chapter 4.7.4. For a complete list of all data types used in X-definition and their equivalents between Java and X-definition; see Chapter 14.3.3.

➡ 4.8.1 External Method with Parameter

➡ 4.8.2 External Method with XXNode Parameter

➡ 4.8.3 External Method with Array of Values

➡ 4.7 Sample of Complete X-definition

4.8.1   External Method with Parameter

➡ 4.5 Events and Actions

As an example of use of an external method with a parameter will be used the method providing printing of the text that is passed to the method through the parameter. In our example we use both methods "print": without parameter and also with a parameter.

Java source code:

```java
package myProject;
public class MyClass {

        public static void print() {
            System.out.println("Processing of the element X started");
        }

        public static void print(String msg) {
            System.out.println(msg);
        }
}
```

X-definition:

```xml
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:root = "X" >

<xd:declaration>
  external method {
    void myProject.MyClass.print();
    void myProject.MyClass.print(String);
  }
</xd:declaration>

  <X xd:script = "onStartElement print(); finally print('Processing of the element X finished')" />
</xd:def>
```

The example above can be modified: for the "print" method we set the alias name "pr". The "print" method can therefore be called in the X-definition using the "pr" alias name:

```xml
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:root = "X" >

<xd:declaration>
  external method void myProject.MyClass.print() as pr;
  external method void myProject.MyClass.print(String) as pr;
</xd:declaration>

  <X xd:script = "onStartElement pr(); finally pr('Processing of the element X finished')" />
</xd:def>
```

The output will be:

```
Processing of the element X started
Processing of the element X finished
```

4.8.2   External Method with XXNode Parameter

The X definition at work internally creates control objects that relate to the currently processed part of an XML object. All work objects are derived from the common interface `org.xdef.proc.XXNode`. The object `org.xdef.proc.XXElement` that is created at the start of processing of XML elements and the `org.xdef.proc.XXData` is created when processing text values of XML objects (ie the attributes and text nodes). These objects are created dynamically during processing of XML document and after processing expire. The mentioned objects are accessible in external Java method invoked from the X-script. External method can use

them to get detailed information about processed data or to manipulate processing. Therefore, the object can be passed to external method via parameter of XXNode type (or XXElement or XXData). [Doc3]

As an example the external method will now be used to print the text that will be passed through its String parameter and to read and print out value of "vrn" attribute through XXNode object from the Vehicle element:

```
public static void myPrint(XXNode xnode, String msg) {
        String regn = xnode.getXXElement().getAttribute("vrn");
        System.out.printf("Info=%s; Registred number=%s", msg, regn);
}
```

In we declare the external method with XXNode parameter. However, in the X-script the first parameter is not specified; it is automatically added:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:root = "Vehicles" xd:name = "Vehicles">

    <xd:declaration>
      external method void myProject.MyClass.myPrint(XXNode, String);
    </xd:declaration>

    <List>
        <Vehicle xd:script = "occurs 0..*; onStartElement myPrint('Element Vehicle started')"
            type  = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
            vrn   = "string(7)"
            purchase = "date()"
            manufacturer = "string()"
            model = "string()" />
    </List>
</xd:def>
```

The X-definition compiler adds to the external method automatically the value XXNode of parameter declared as the first parameter.

In external method sometimes you need to get values of element models from X-definition or from the project. The object containing the X-definition information is implemented on the interface org.xdef.model.XMDefinition. This object can be obtained from XXNode object by method "getXMDefinition". The org.xdef.model.XMElement object contains information about the element model and can be obtained from XXNode by method "getXMElement". An object with cz.syneta.xdef.model.XMData interface contains information about the attributes and child nodes. It can be obtained by various methods from object org.xdef.model.XMElement:

```
public static void myPrint(XXNode xnode, String msg) {
        String regn = xnode.getXXElement().getAttribute("vrn");
        System.out.printf("Info=%s; Registred number=%s", msg, regn);

        /* Root elements of actual X-definition */
        for (XMElement xelem : xnode.getXMDefinition().getRootModels()) {
                System.out.println("Root Element: " + xelem.getName());
        }

        /* Name and quantifier of the model of element where was invoked this method myPrint(): */
        System.out.println("Model of element " + xnode.getXMElement().getName() +
                " occurs " + xnode.getXMElement().minOccurs() + ".." + xnode.getXMElement().maxOccurs());

        /* Attributes declared in the model of actual element (types and names) */
        for (XMData xdata : xnode.getXMElement().getAttrs()) {
                System.out.println((xdata.isOptional() == true ? "optional " : "required ") +
                        xdata.getValueTypeName() + " " + xdata.getName() + ";");
        }
}
```

➡ 7 Using of X-definitions in Java code

4.8.3 External Method with Array of Values

➡ 4.5 Events and Actions

➡ 4.7 Sample of Complete X-definition

➡ 4.8 External (Java) Methods

4.8.1 External Method with Parameter

As an example will be presented an external method that has a single parameter with the array of values of general data types supported by X-definitions. For this purpose the X-definition offers the interface "org.xdef.XDValue", which represents the basis of all supported data types of X-script values. The external method in our example prints the value of the data types passed by the array in the method parameter and lists whether this is or is not a String data type:

```java
public static void printParam(XDValue[] values) {
    for(XDValue par:values) {
        /* Get data type id of the item. */
        short dataType = par.getItemType();

        /* Print out the data type ID and the value of the item. */
        System.out.print(dataType + " (" + par.stringValue() + "), ");

        /* print if it is or isn't a string value. */
        if(dataType == XDValueTypes.STRING_VALUE) {
            System.out.println("value is String.");
        } else {
            System.out.println(" value is not String.");
        }
    }
}
```

This external method can be called as any external method, except that any number of parameters (including none, then length of XDValue [] parameter will be zero). All passed values correspond to the XDValue interface, from which it can be cast in the external method into a given data type:

```xml
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:root    = "X" >

    <xd:declaration> external method void myProject.MyClass.printParam(XDValue[]) as pp; </xd:declaration>

    <X xd:script = " finally {int min = 5; pp('aaa', min, 1);}" />
</xd:def/>
```

When passing an object, its data type is retained. That is, for example, when calling the pp ('aaa', '5', 1), the first and second parameters are represented as string values (ie, the value '5' will be understood as a string containing the character 5) and the third parameter is represented as the number 1.

Each data type supported by X-definitions is implemented by a specific Java class that implements the XDValue interface. X definition distinguishes between data types using numeric constants. Constants for each type are declared in the class cz.synea.xdef.XDValueTypes. For the String data type, it is XDValueTypes.STRING_VALUE.

It is also possible to specify as the first parameter the XXNode object that is passed to the method automatically. Its use is identical to the example in Chapter 4.7.3.

7 Using of X-definitions in Java code

4.9   Declaration of X-script Methods

4.5 Events and Actions

4.7 Sample of Complete X-definition

4.7.3.4 Declaration of Method

In addition to the built-in (Chapter 14.6) and external (Chapter 4.8) methods, the user-defined methods may be declared in the X-definition declaration part (as opposed to the external methods defined in the Java class) can also be called in the appropriate X-Script sections. Each method declared in the declaration part has the visibility according to "scope" parameter of declaration section. User defined methods are written to X-script od declaration part as the text value (or CDATA section) of an element <xd:declaration>.

4.8 External (Java) Methods

4.8.1 External Method with Parameter

4.8.2 External Method with XXNode Parameter

4.8.3 External Method with Array of Values

→ 7 Using of X-definitions in Java code

4.9.1   Methods without Parameter

→ 4.7.3.4 Declaration of Method

→ 4.8 External (Java) Methods

Let's define now a method that only prints a text to the standard output:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:root   = "X" >

    <xd:declaration xd:scope = "local" >
        void myPrint() {
            outln("Metoda myPrint() was invoked");
        }
    </xd:declaration>

    <X xd:script = "finally myPrint();" />

</xd:def/>
```

Note that compared to Chapter 4.8, where the implementation of an identical but external method is presented the "myPrint" method is declared in the declaration part X of the definition.

→ 7 Using of X-definitions in Java code

4.9.2   Methods with Parameter

→ 4.7.3.4 Declaration of Method

→ 4.8 External (Java) Methods

We now show a declaration of a method that lists the text passed to it through its String parameter:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:root   = "X" >

    <xd:declaration>
        void myPrint(String msg) {
            outln(msg);
        }
    </xd:declaration>

    <X xd:script = "finally myPrint('method myPrint called from the element X');" />

</xd:def/>
```

The method will print a text from the parameter to the standard output.

→ 7 Using of X-definitions in Java code

4.9.3   Value of Attribute or Text Node in the X-script

→ 4.7.3.4 Declaration of Method

→ 4.8 External (Java) Methods

In methods defined in the X-definition declaration part the attributes of the currently processed element can be processed similarly as in the case of external methods (through the XXNode object - see Chapter 4.7.3). You can get the value of an attribute from a currently processed attribute or text node by the method getText (). A value of the attribute with a given name in a currently processed element can be obtained in the script by method getAttr(attribute_name).The following example shows how to get the value of the attribute "manufacturer" while processing the element "Vehicle" and how to get a value of the actually processed attribute:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:name = "garage" xd:root = "Vehicles">

   <List>
      <Vehicle
          xd:script = "occurs 0..*; finally print(getAttr('manufacturer'));"
          type     = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
          vrn      = "string(7)"
          purchase = "date()"
          manufacturer = "string()"
          model    = "string; onTrue print(getText());" />
   </List>

   <xd:declaration>
       void print(String s) {
          outln(s);
       }
   </xd:declaration>

</xd:def>
```

Methods "getText" and "getAttr" can also be used in the body of a declared method. They are applied to the currently processed object (analogically, in external methods these values are available from the XXNode, XXData, XXElement arguments):

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage"
        xd:root = "Vehicles">

   <List>
      <Vehicle
          xd:script = "occurs 0..*; finally printAttr('manufacturer');"
          type     = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
          vrn      = "string(7)"
          purchase = "date()"
          manufacturer = "string()"
          model    = "string; onTrue printText();" />
   </List>

   <xd:declaration>
       void printAttr(String s) {
          outln(getAttr(s));
       }
       void printText() {
          outln(getText());
       }
   </xd:declaration>

</xd:def>
```

➡ 7 Using of X-definitions in Java code

4.10 User Defined Methods for Checking Data Types

➡ 4.5 Events and Actions

➡ 4.7 Sample of Complete X-definition

➡ 4.7.3.4 Declaration of Method

➡ 4.8 External (Java) Methods

In some situations, the built-in methods for checking the data types used in the X-script may not be sufficient, and it is necessary to check the data values by the user resources. For this purpose, user-defined methods can be used: either the external methods (see Chapter 4.8) or the methods defined in the X-definition declaration section (see Chapter 4.9).

The return value of all data type parsing methods can be either boolean or ParseResult (if the parsed badge is not a String and needs to be used in other part of X-script). The number of parameters can be arbitrary. The method thus defined is used to check (parse) the value of the attribute or text node in the element model.The following example demonstrates the use of the Vehicle Registration Number type where the correct number of characters must be 7. In the case of a malformed format, the message put error message and returns false, otherwise it

returns true. The method is defined in the following example in the declaration part of the X-definition and is named "vrn":

a) variant with elements and attributes:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage" xd:root = "Vehicles">

    <List>
        <Vehicle xd:script = "occurs 0..*"
             typ    = "enum('SUV', 'MPV', 'personal',
                  'truck', 'sport', 'other')"
             vrn    = "vrn()"
             porchase = "date()"
             manufacturer = "string()"
             model  = "string" />
    </List>

    <xd:declaration>
        /* Method checks data type of Vehicle
         * Registration number.
         */
        boolean vrn() {
            String text = getText();
            if (text.length() != len) {
                return error(text +
                    " is not valid. Required " +
                    len + " characters.");
            }
            return true;
        }
    </xd:declaration>
</xd:def>
```

b) variant using only XML elements:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage" xd:root = "Vehicles">

    <List>
        <Vehicle xd:script = "occurs  0..*">
            <type>enum('SUV', 'MPV', 'personal',
                  'truck', 'sport', 'other')
            </type>
            <vrn>vrn(7)</vrn>
            <purchase>date()</purchase>
            <manufaccturer>string()</manufacturer>
            <model>string</model>
        </Vehicle>
    </List>

    <xd:declaration>
        /* Method checks data type of Vehicle
         * Registration number.
         */
        boolean vrn() {
            String text = getText();
            if (text.length() != len) {
                return error(text +
                    " is not valid. Required " + len +
                    " characters.");
            }
            return true;
        }
    </xd:declaration>
</xd:def>
```

In the example above, several embedded X-script methods are used:

• getText () method returns the value of the currently processed attribute or text node.

• length() - the method of the String object in the X-script returns the number of characters of a given string. (ie it is similar to the java.lang.String.length() method).

• error() - the method returns allways the boolean value false and it puts the text specified in the parameter to the error reporter (details and other variants of the error method see chapters 7 and 11.2).

If the above-mentioned "vrn" method for checking the data type of a vehicle registration number returns the boolean value true, then the valid value is considered to be correct (and the onTrue event arises). If it returns false, then it false it writes the the error message to log file and the onFalse event occurs.

▷ 7 Using of X-definitions in Java code

▷ 7.8 Validation of data with a database in a XML document

▷ 11.3 The error method

4.10.1 Unique Value of Attribute or Text Node

▷ 4.10 User Defined Methods for Checking Data Types

If an attribute (or text node) whose value is to be a unique NCName value within the whole XML document, the uniqueness can be ensured using the ID() method (similar to XML schema or DTD). If this value occurs in the XML document multiple times, an error is reported. If there is required a reference to a unique value you can use the validation method IDREF(). The validation method IDREFS() checks the list of references to unique values separated by a space.

In the X definition it is possible to specify a set of any type of unique values. You can declare the "uniqueSet" object to which the appropriate validation method is attached. The values stored in this object can be viewed as a

table of unique values. The unique value can be viewed as a key to the table. The uniqueSet object is declared by the following command:

```
<xd:declaration>
  uniqueSet obj validation_method;
</xd:declaration>
```

The following methods are defined on "uniqueSet" objects (below "obj"). Note the key in a table can generally be composed of multiple items (this will be described in next chapter):

- `obj.ID()` will first check the formal correctness of the value and then checks the uniqueness of the key and adds the key to the table. If the value is incorrect or unambiguous, an error is reported.

- - `obj.SET()` is similar to the ID method but it does not report an error if the key in the table already exists, ie this key can be repeated in the table.

- `obj.IDREF()` checks whether the value is formally correct and whether the value of the actual key is in the table. If not an error is reported. The reference may occur before the a unique value is inserted to the table (by methos ID or SET), it may be inserted later.

- `obj.IDREFS()` performs the IDREF operation on a list of values separated by spaces.

- `obj.CHKID()` checks whether the value is formally correct and whether the value of actual key is in the table. If not, an error is reported. However, unlike the IDREF method the key must be inserted to table before the reference.

- `obj.CHKIDS()` performs the CHKID operation on a list of values separated by spaces.

- `obj.CLEAR()` clears all entries of the table. If the table contains the references to a value (by method IDREF or IDREFS) that has not been inserted yet to the table by the ID of SET method, the appropriate errors are reported. This feature can typically be used at the end of the element processing in the model to limit the validity range of the respective set of keys.

In following example, the X definition describes the "Accident" model which contains a list of vehicles and a list of accidents. Both, the vehicle registration number and the identification number of an accident, must be unique. The reference to the vehicle registration number involved in the accident must correspond to some of the vehicle registration number ("vrn") from the list of elements "Vehicle":

a) variant with elements and attributes:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:root = "Accidents">

  <xd:declaration>
    uniqueSet vehicles string(7);
    uniqueSet accidents int();
  </xd:declaration>

  <Accidents>

    <Vehicle xd:script = "occurs 0..*"
         type   = "enum('SUV', 'MPV', 'personal',
                  'truckí', 'spor', 'other')"
         vrn    = " vehicles.ID() "
         purchase = "date()"
         manufacturer = "string()"
         model  = "string()" />

    <Accident xd:script = "occurs 0..*"
      id = " accidents.ID(); "
      date = " date(); "
      injury = " int(); "
      death = " int(); "
      loss = "decimal()" >
      <vrn xd:script = "occurs 0..*" >
        vehicles.IDREF();
      </vrn>
    </Accident>
  </Accidents>

</xd:def>
```

b) variant using only XML elements:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:root = "Accidents">

  <xd:declaration>
    uniqueSet vehicles string(7);
    uniqueSet accidents int();
  </xd:declaration>

  <Accidents>

    <Vehicle xd:script = "occurs 0..*">
      <type>enum('SUV', 'MPV', 'personal',
                 'truck', 'sport', 'other')
      </type>
      <vrn> vehicles.ID() </vrn>
      <purchase> date() </purchase>
      <manufacturer>string()</manufacturer>
      <model>string()</model>
    </Vehicle>

    <Accident xd:script = "occurs 0..*">
      <vrn> accidents.ID(); </vrn>
      <date> date(); </date>
      <injury> int(); </injury>
      <death> int(); </death>
      <loss> decimal()"</loss>
      <vrn xd:script = "occurs 0..*" >
        vehicles.IDREF();
      </vrn>
    </Accident>

  </Accidents>

</xd:def>
```

Furthermore, we show more complex situaltion. Let's have a street where are houses and in houses are apartments. House numbers are unique across the street. In each house there are apartments that must be unique within the house and not within the street. Uniqueness is ensured by validation method ID(). Using the CLEAR() method invoked at the end of processing of the house, we will ensure the uniqueness of the apartment numbers within the house (and not within the streat):

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:root = "Street">

  <Street>
    <House xd:script = "occurs 0..*; finally apartements.CLEAR(); "
         number = "houses.ID()">
      <Apartement xd:script = "occurs 0..*;"
           number = "apartements.ID()" />
    </House>
  </Street>

  <xd:declaration>
    uniqueSet houses int();
    uniqueSet apartements int();
  </xd:declaration>
</xd:def>
```

➡ 7 Using of X-definitions in Java code

➡ 7.8 Validation of data with a database in a XML document

➡ 11.3 The error method

4.10.2 Unique Values Table with Multiple Items Key

In the X definition you can declare tables with keys composed from more key items. The uniqueness of a key in a table is determined by a set of key items. If more than one key item is specified in the declaration of a table, these key items must be named. The key item name ends with a colon (":") followed by the specification of a validation

method. Key items are written into the compound parentheses ("{" and "}") and they are separated by a semicolon (";"). The declaration of such a table (set of houses and apartements) with multiple key items has the following form:

```
<xd:declaration>
  uniqueSet street { house: int(); apartement: int() }
</xd:declaration>
```

An example from the previous chapter can be written:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:root = "Houses">

    <xd:declaration>
        uniqueSet houses { house: string(); apartement: string(); }
    </xd:declaration>

    <Houses>
        <House xd:script = "occurs 0..*"
            number = "houses.house.ID()" >
          <Apartement xd:script = "occurs 0..*; "
            number = "houses.apartement.ID()" />
        </House>
    </Houses>
</xd:def>
```

The validation method ID stores to the table an entry with all the key items that have been previously set. If a key item is not set at this time the null value is used.

If no method is used to check the key value in the table, only the relevant key items of the current key are set. E.g. if you write

```
number = "houses.house()"
```

house number is saved in the current key, but it does not perform any operation with the table "houses". However, if you specify the ID method, for example the acctual key with all items is stored in the table. Thus if the value of the "apartement" item has not been set before the following value of the key containing the pair of house and apartement will be stored in the table "houses":

```
street.house=number, streat.apartement=null
```

To understand better the uniqueSet activity here's another example. Let's have an XML document containing countries, cities, streets house numbers and apartement numbers. In the element address we must take in mind that the order of processing it's attribytes id not defined. So we prepare values of key items by invoking just the method on the appropriate key item (it provides formal check of the value and saves it to the key item). To ensure the preared key is completed we ca't use the method IDREF at the attribute "house" but the method IDREF must be invoked in the "finally" section of the element "Address" the on the table "address" (it checks the whole prepared key composed from attributes):

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:root = "Addresses">

  <xd:declaration>
    uniqueSet address { country: string(); town: string(); street: string(); house: int();}
  </xd:declaration>

  <Addresses>

    <Country xd:script = "occurs 0..*;" name = "address.country();">
      <Town xd:script = "occurs 0..*;" name = "address.town();">
        <Street xd:script = "occurs 0..*;" name = "address.street();">
          <House xd:script = "occurs 0..*;"  number = "address.house.ID();" />
        </Street>
      </Town>
    </Country>

    <Address xd:script = "occurs 0..*; finally address.IDREF();"
        country = "address.country(); "
        town    = "address.town(); "
        street  = "address.street(); "
        house   = "address.house(); " />

  </Addresses>

</xd:def>
```

Note that the "ID" method for the "address" table is specified only in the "House" element. In the case of a village, the value of the "street" item in the key will be null. The previous key items are set to the actual key and only the ID method puts the completed key with all the items to the table. In the "Address" element, the "IDREF" method is called at the end of the element processing on the competed actual key "address". If this method was specified e.g. for the "number" attribute, an error could occur because the order of processing of an attribute is not defined and some key items might not have been set at the time the attribute "number" was processed.

If a key item is optional (ie it will be set to it authomatically a "null" value if it doesn't occur in the data) specify a question mark ("?") before the validation method. E.g.:

```
uniqueSet street { house: string(); apartement: ? string(); }
```

If the question mark is not specified in the named item, the value previously set remains unchanged, the null value is set when a question mark is specified.

4.10.3 Nested Key

In some cases, we can declare the key as a key entry. To understand our example we have to know that the result of methods "ID", "IDREF", "SET" and "CHKID" returns as a result (like any validation method) a "ParseResult" object. So the expression on the line 4 first cheks if a value is in the table "A" and then the value sets to the key part "b" in table "B":

```
1      uniqueSet A { a: string() }
2      uniqueSet B { b: A.a; c: string() }
3      ...
4      B.b(A.a().ID())
```

Here's how the following example works. Let's have XML with an overview of staff, projects, teams, and staff activity report - ie how many hours the individual staff worked on. Let's define the table "Person" with the employee codes. Let's define the table "Project" with the projects codes. And define the table "Team" which contains the touples of the employee code project code. The X-definition will be:

```
 1 <xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:root = "Overview" >
 2
 3   <xd:declaration xd:scope = "local" >
 4     type codePrj string(4);
 5     type codeUsr string(2, 50);
 6     uniqueSet Person { usr: codeUsr };
 7     uniqueSet Project { prj: codePrj };
 8     uniqueSet Team { prj: Project.prj; usr: User.usr }
 9   </xd:declaration>
10
11   <Overview>
12
13     <Employee xd:script = "occurs +"
14        UserID = "required User.usr.ID()" Name = "required string()" />
15
16     <Project xd:script = "occurs +"
17          ProjectCode = "/* Check the unique code in Project and set the prj item in the Team table. */
18                       required Team.prj(Project.prj.ID());"
19          ProjectName = "required string()" >
20      <Team xd:script = "occurs +"
21         UserID = "required Team.usr.ID()" />
22     </Project>
23
24     <ProjectActivity xd:script = "occurs +"
25          ProjectCode = "/* Check if Project code exists and set prj item in Team table. */
26                       required Team.prj (Project.prj.IDREF())" >
27      <EmployeeActivity xd:script = "occurs +"
28         UserID = "required Team.usr.IDREF()" >
29       <Activity xd:script = "occurs +"
30         Date = "required date()"
31         Hours = "required int()" />
32      </EmployeeActivity>
33     </ProjectActivity>
34
35   </Overview>
36
37 </xd:def>
```

The "ProjectTeam" table has a key composed of two entries "prj" and "user" that refer to the "User" and "Project" key entries (see line 8). Line 7 ensures unambiguous employee code. Line 14 adds value of the key to the table "User". On the line 18, the unique code is set to the "Project" table, and at the same time the key part "prj" of "Team" is set. On the line 21, the usr entry is stored in the "Team" and ensures the uniqueness of the key in the "Team" table (ie the "prj" and "usr" pair). The line 26 will check if the key is in the "Project" table and set "prj" in the "Team" key. Line 28 adds the "prj" key part to the "Team" key and check if the pair exists in the "Team" table.

Example of valid data:

```
<OverWiew>
   <Employee UserID="JBROWN" Name="John Brown"/>
   <Employee UserID="KSMITH" Name="Karl Smith"/>
   <Employee UserID="VTELL"  Name="Vilem Tell"/>
   <Project ProjectCode="ALFA" ProjectName="Project ALFA">
     <Team UserID="JBROWN"/>
     <Team UserID="KSMITH"/>
     <Team UserID="VTELL"/>
   </Project>
   <Projekt ProjectCode="BETA" ProjectName="Project BETA">
     <Team UserID="JBROWN"/>
   </Project>
   <Project ProjectCode="GAMA" ProjectName="Project GAMA">
     <Team UserID="KSMITH"/>
     <Team UserID="JBROWN"/>
   </Project>
   <ProjectActivity ProjectCode="ALFA">
     <EmployeeActivity UserID="KSMITH">
        <Activity Date="2016-06-15" Hours="2"/>
     </EmployeeActivity>
     <EmployeeActivity UserID="JBROWN">
        <Activity Date="2016-06-15" Hours="2"/>
        <Activity Date="2016-06-15" Hours="6"/>
     </EmployeeActivity>
   </ProjectActivity>
   <ProjectActivity ProjectCode="BETA">
     <EmployeeActivity UserID="JBROWN">
        <Activity Date="2016-06-15" Hours="4"/>
     </EmployeeActivity>
   </ProjectActivity>
   <ProjectActivity ProjectCode="GAMA">
     <EmployeeActivity UserID="KSMITH">
        <Activity Date="2016-06-15" Hours="2"/>
     </EmployeeActivity>
   </ProjectActivity>
</OverWiew>
```

4.11 Group Specifications

➡️ 4.7 Sample of Complete X-definition

The order of attributes in an element is always arbitrary, so the order of the attributes specified in the X definition is not respected in a valid XML document. The same statement is not true for child elements and text nodes.

This chapter describes how to define **groups of elements and text nodes**. An important feature of groups is that they can also act as models. That is, they can be refered from different parts of the X-definition. A group may describe different combinations of elements and text nodes.

➡️ 4.11.1 Strict Order of Group Items (xd:sequence)

➡️ 4.11.2 Arbitrary Order of Group Items (xd:mixed)

➡️ 4.11.3 Selection of Item from a Group (xd:choice)

➡️ 4.11.5 Groups with Text Node

4.11.1 Strict Order of Group Items (xd:sequence)

➡️ 4.7 Sample of Complete X-definition

The fact that the order of the elements in a valid XML document matches the order declared in the model is implicit. However, the group of text node elements in the X-definition can be explicitly defined using a special "xd:sequence" element. Our exempary example will have then the following form:

```
<Vehicle xd:script = "occurs 0..*">
  <xd:sequence>
    <type>enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')</type>
    <vrn>string(7)</vrn>
    <purchase>date()</purchase>
    <manufacturer>string()</manufacturer>
    <model>string()</model>
  </xd:sequence>
</Vehicle>
```

In the example it was explicitly stated that the order of items in the group is fixed.

➡ 4.11.2 Arbitrary Order of Group Items (xd:mixed)

➡ 4.11.3 Selection of Item from a Group (xd:choice)

4.11.2 Arbitrary Order of Group Items (xd:mixed)

➡ 4.7 Sample of Complete X-definition

The X-Definition allows you to allow the occurrence of declared items also in any order. Elements (elements or the text nodes) that may occur in an arbitrary sequence must be declared in the X-definition special element "xd:mixed". The example in Chapter 4.7 will be in such case as follows:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage"
        xd:root = "Vehicles">

    <List>
        <Vehicle xd:script = "occurs 0..*">
            <xd:mixed>
                <type>enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')</type>
                <vrn>string(7)</vrn>
                <purchase>date()</purchase>
                <manufacturer>string()</manufacturer>
                <model>string()</model>
            </xd:mixed>
        </Vehicle>
    </List>
</xd:def>
```

In the example it was declared that the order of the sub-elements of the Vehicle element does not matter.

➡ 4.11.3 Selection of Item from a Group (xd:choice)

4.11.3 Selection of Item from a Group (xd:choice)

➡ 4.7 Sample of Complete X-definition

Selecting an element from a given group can be written in the X-definition using the "xd:choice" element. We extend our example so that we add the element "owner" which have a child element either "person" or "company":

```
<Vehicle>
    <type>enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')</type>
    <vrn> string(7) </vrn>
    <purchase> date() </purchase>
    <manufacturer> string() </manufacturer>
    <model>string()</model>
    <owner>
      <xd:choice>
        <person firstName = "string()" lastName = "string()" />
        <company name = "string()" />
      </xd:choice>
    </owner>
</Vehicle>
```

If we specify in the X-script of element "xd:choice" a qualifier with "occurence 0..1" the element "owner" may be empty or it may contain a child according to "xd:choice":

Dokument: **Chyba! Nenalezen zdroj odkazů.***, verze:* **Chyba! Nenalezen zdroj odkazů.***, ze dne:*
REF zedne \h * MERGEFORMAT **Chyba! Nenalezen zdroj odkazů.**

```
<owner>
    <xd:choice xd:script="occurs 0..1">
        <person firstName = "string()"lastName = "string()" />
        <company name = "string()" />
    </xd:choice>
</owner>
```

Then it will be valid following three XML data:

```
<owner>
    <person firstName = "John" lastName = "Brown" />
</owner>
```

or

```
<owner>
    <company name = "Syntea" />
</owner>
```

or

```
<owner/>
```

If we include to a selection group a sequence containing at least 2 elements then at least one occurrence of the first item in the sequence must be mandatory (selection will be made according to this item). The following example is model that allows either a touple of elements "type" and "vrn" or one of the "purchase", "manufacturer" or "model" elements to be selected in the "Vehicle" element:

```
<Vehicle>
  <xd:choice>
    <xd:sequence>
      <type>enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')</type>
      <vrn>string(7)</vrn>
    </xd:sequence>
    <purchase>date()</purchase>
    <manufacturer>string()</manufacturer>
    <model>string()</model>
  </xd:choice>
</Vehicle>
```

Therefore, the following two elements will be valid for the above model:

```
<Vehicle>
  <type>SUV</type>
  <vrn>1A23456</vrn>
</Vehicle>
</Vehicles>
<Vehicle>
  <manufacturer>Škoda</manufacturer>
</Vehicle>
```

4.11.4 Selection with Action "match"

In some cases, you need to select elements according to a rule (eg the existence of a certain attribute). For this purpose, it is possible to specify a "match" action in the Skript. The "match" action is a boolean expression that specifies whether to select an item [Doc1].

The "match" action in the X-script that evaluates whether the element contains an attribute has the following form:

```
match @attributeName
```

If the attribute exists in an element, the expression will be "true", otherwise "false".

Let's show it on the following example. In the selection group, the element of the same name is specified more than once and the match action determines which variant of the model element will be selected. In our example, there are two variants of the "owner" element (note the second "element owner" element does not have an action "match" - if the first option was not chosen, it will be the second):

```
  <xd:choice>
    <owner xd:script = "match @firstName AND @lastName"
      firstName = "string()"
      lastName = "string()" />
    <owner company = "string()" />
  </xd:choice>
```

Therefore, the two following XML document samples will be valid for the selection group:

```
<owner firstName = "Jan" lastName = "Novak" />
```

```
<owner company = "Syntea" />
```

A X-script that evaluates whether a particular element contains a given attribute with a specific value has the following form:

```
match @attributeName EQ 'attribute value'
```

The use of action "match" demonstrates the following example, which distinguishes two different models of the "Vehicle" element according to the name of manufacturer:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:name = "garage" xd:root = "Vehicles">

   <List>
      <xd:choice xd:script = "occurs 0..*">
         <Vehicle xd:script = "match @manufacturer EQ 'VW'; "
            type = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
            vrn = "string(7)"
            purchase = "date()"
            manufacturer = "fixed 'VW'"
            model  = "enum('Golf', 'Passat', 'Polo')" />
         <Vehicle xd:script = "match @manufacturer EQ 'Škoda';"
            type = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
            vrn = "string(7)"
            purchase = "date()"
            manufacturer = "fixed 'Škoda'"
            model  = "enum('Fabia', 'Octavia', 'Yetti')"/>
      </xd:choice>
   </Vehicles>

</xd:def>
```

The example shows that if the "manufacturer" attribute of the "Vehicle" element contains a "Škoda" value, the element model will check the "Škoda" models if it contains "VW" value then the Volkswagen models are allowed.

Additionally, the example shows that the value of attribut "manufacturer" has been replaced by a fixed value (see chapter 4.2.2) corresponding to the value in the match action condition. If instead of the fixed 'value' the type string method remained, the result would be the same.

Therefore, the two following elements "Vehicle" will be valid for the X-definition above:

```
<List>
    <Vehicle type    = "SUV"
        vrn          = "1A23456"
        purchase     = "2011-02-01"
        manufacturer = "VW"
        model        = "Golf" />
    <Vehicle type    = "SUV"
        vrn          = "2B34567"
        purchase     = "2009-10-30"
        manufacturer = "Škoda"
        model        = "Yetti"/>
</List>
```

Note: the specification of @attributeName returns a value of the attribute from the currently processed element. This value is interpreted as true in a logical expression if the attribute exists, otherwise it is false. If the attribute exists and the specification is followed by an comparation operator (e.g., EQ 'value'), the corresponding expression is evaluated with the value of the appropriate attribute.

➡ 7 Using of X-definitions in Java code

4.11.5 Groups with Text Node

All the group statements described above may not necessarily be a combination of elements only, but may also include a text node declaration.

For example, if the "Vehicle" element contained a vehicle registration number or a text node with a description, the "xd: choice" element may have the form:

```
<Vehicle>
  <xd:choice>
    <vrn>string(7)</vrn>
    string() /* description */
  </xd:choice>
</Vehicle>
```

Both following elements will be valid:

```
<Vehicle>
  <vrn>1A23456</vrn>
</Vehicle>
```

```
<Vehicle>
   Unknown vehicle!
</Vehicl>
```

Similarly, the text node can be included in the "xd:sequence" or "xd:mixed" groups.

4.11.6 "Named" Group as Model and Reference to Group.

Each of the "xd:sequence", "xd:mixed" and "xd:choice" groups can be specified as a self-standing model in the X-definition. In order to be able to refer to the appropriate group within the X-definition, a group must be a direct descendant element "xd:def" and must be named with the "xd:name" attribute. The following example demonstrates how to use a group model and corresponding reference:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:name = "garage" xd:root = "Vehicles" >

    <List>
        <Vehicle xd:script = "occurs 0..*" >
            <xd:sequence xd:script="ref VehicleInfo" />
        </Vehicle>
    </List>

    <xd:sequence xd:name="VehicleInfo" >
        <type>enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')</type>
        <vrn>string(7)</vrn>
        <purchase>date()</purchase>
        <manufacturer>string()</manufacturer>
        <model>string()</model>
    </xd:sequence>

</xd:def>
```

With the "ref" keyword in the X-script is specified link to a group model.

Group models serve in particular to increase the clarity of the X-definition, or to allow the sharing of a single group model in multiple element models, thereby helping to make X-definitions easier to read and maintain.

4.11.7 Events and events of groups

For the "xd:sequence", "xd:mixed" and "xd:choice" groups you can specify the "xd:script" attribute for writing X-script, which can also define some actions like in the X-script of element ("init", "finally", "create"). The exception

is "onStartElement" which can not be used for groups. The following example prints the "Group is processed" when the group is process finish:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:name = "garage" xd:root = "Vehicles">
    <List>
        <Vehicle xd:script = "occurs 0..*">
            <xd:mixed xd:script = "finally outln('Group is processed')">
                <type>enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')</type>
                <vrn>string(7)</vrn>
                <purchase>date()</purchsee>
                <manufacturer>string()</manufacturer>
                <model>string()</model>
            </xd:mixed>
        </Vehicle>
    </List>
</xd:def>
```

4.1 XML Namespace in Models

If there are elements or attributes from a particular namespace in a model, you need to use the "xmlns" attributes in the header of the X-definition to describe all the namespaces and the corresponding prefixes. In models the namespace is then specified by the appropriate prefix. In the following example the "Vehicle" element and all internal elements and attributes are from the namespace "http://cz.vehicle.registr" to which the prefix "reg" is assigned. The element "accident" and all internal elements are from the namespace "http://cz.vehicle.accident" with the prefix "acc", but in our sample the attributes of this element have no namespace assigned. The corresponding X definition will be:

a)   variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:root  = "Acceidents"
        xmlns:reg = "http://cz.vehicle.registr"
        xmlns:acc = "http://cz.vehicle.accident" >

<Accidents>
    <Vehicles>
        <reg:Vehicle xd:script = "occurs 0..*"
            reg:type = "enum('SUV', 'MPV','personal',
                 'truck', 'sport', 'other')"
            reg:vrn          = "string(7)"
            reg:purchase = "date()"
            reg:manufacturer = "string()"
            reg:model = "string()" />
    </Vehicles>

    <acc:accident xd:script = "occurs 0..*" >
        id      = "int()"
        date    = "date()"
        injury  = "int()"
        death   = "int()"
        <acc:vrn xd:script = "occurs 1..*" >
         string(7)
       </acc:vrn>
    </acc:accident>

</Accidents>

</xd:def>
```

b)   variant using only XML elements:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:root  = "Acceidents"
        xmlns:reg = "http://cz.vehicle.registr"
        xmlns:acc = "http://cz.vehicle.accident" >

<Accidents>
    <Vehicles>
        <reg:Vehicle xd:script = "occurs 0..*">
            <reg:type>enum('SUV', 'MPV', 'personal',
                 'truck', 'sport', 'other')
            </reg:typ>
            <reg:vrn> string(7) </reg:vrn>
            <reg:purchase> date() </reg:purchase>
            <reg:manufacturer> string()
        </reg: manufacturer >
            <reg:model>string()</reg:model>
        </reg:Vehicle>
    </Vehicles>

    <acc:accident xd:script = "occurs 0..*" >
        <acc:id> int() </acc:id>
        <acc:date> date(); </acc:date>
        <acc:injury> int(); </injury>
        <acc:death> int(); </acc:death>
        <acc:vrn xd:script = "occurs 1..*">
          string(7)
        </acc:vrn>
    </acc:accident>

</Accidents>

</xd:def>
```

5 Types of Values and Objects in X-script

➡️ Basic knowledge of XML markup language ([1], [2])

➡️ 2 X-definition Technology

➡️ 2.4 Exemplary XML Data

➡️ 4 Description of structure of XML document by X-definition

This chapter describes the types of values and data objects and some useful methods implemented in X-definitions.

5.1 Basic Types of Values

int - integer numbers are implemented in the X-script as 64-bit values, ie as "long" Java type. You can write the value of a number as decimal (eg "123456") or as hexadecimal if the entry begins with "0x" or "0X", eg "0xABCDEF123". In hexadecimal form it is possible to use both upper and lower case letters (eg "0xabdDEF123"). For greater readability, it is possible to write an underscore character ("_") at any time after the first digit; Ie "1__234_56" or "0xAB_CD_EF". In the X-script, the following int constants are defined by identifiers:

```
$MININT               minimal value of integer number (-9223372036854775808)
$MAXINT               maximal value of integer number (9223372036854775807)
```

float - floating-point numbers are implemented in the X-script as 64-bit values, ie they correspond to Java type "double". The value can be written with decimal point and / or exponent. The exponent is indicated by the letter "e" or "E". Similarly as integers, it is possible to enter the "_" character at any position after the first digit. The following entries represent the same value: "3.14159", "0.0314159e2", "0.0314159E + 2", "314159E-5", "31_41.5__9E-3".$PI … konstanta pi, poměr obvodu kruhu k průměru (3.141592653589793). In the X-script, the following float constants are defined by identifiers:

```
$PI                   the ratio of the circumference of a circle to its diameter (3.141592653589793)
$E                    e, the base of natural logarithms (2.718281828459045).
$MINFLOAT             smallest positive nonzero value (4.9E-324) .
$MAXFLOAT             largest positive value (1.7976931348623157E308).
$NEGATIVEINFINITY     negative infinity.
$POSTIVEINFINITY      positive infinity.
$NaN                  not valid value (Not a Number).
```

Decimal decimal numbers with decimal development are implemented in the X-script as java.math.BigDecimal. This type of number is written with prefix "0d" or "0D" followed by an integer or floating-point number. An underscore can be used in the entry, eg. "0d123__456_890_999_000_333". Decimal values are only possible to compare in the expressions, other operations must be performed using the appropriate methods.

boolean - logical values. For writing values you can use identifiers "true" and "false".

String - arrays of characters. Character strings can contain any characters that are allowed in XML documents. Values of strings strings are written into apostrophes or quotation marks (since XML attribute values can also be inside quotation marks or apostrophes, it is appropriate to use another character inside the attribute, ie if the attribute value is bounded with quotes, we write apostrophes and vice versa). If there is a character inside the string that is a string (ie an apostrophe or quote), the '\' character is preceded by a character. The occurrence of the '\' character itself is written in duplicate as '\\'. Using the "\" character, you can further describe any Unicode 16 character by writing "\ uxxxx" where x is hexadecimal digits. You can also write some special characters using the following two characters:

```
\n    linefeed (LF == \u000a)
\r    carriage return (CR == \u000d)
\t    horizontal tab (HT == \u0009)
\f    form feed (FF == \u000c)
\b    backspace (BS == \u0008)
```

Warning: If the X-script text is listed as an attribute value, the XML processor will replace all occurrences of the linefeeds by spaces. Therefore, new lines need to be written to like "\ n" in X-script of the attribute. Additionally, it is necessary to avoid unintentional macros. The occurrence of a pair of characters "${" anywhere in the X-script is interpreted as the beginning of a macro call, so inside the character strings, in this case, the "$" character of that character pair must be written using an escape sequence such as "\ 44" or "\ u0024")

Datetime - date and time. Values representing date and time. Contains year, month, and day. They can be written to the constructor as a string of characters according to ISO8601. E.g.

```
Datetime dt = new Datetime("2005-11-31T14:35+0200");
```

The Datetime object may not contain all items so it can serve to preserve the date or time individually:

```
Datetime d = new Datetime("2005-11-31");
Datetime t = new Datetime("14:34:07.234");
```

Bytes - array of bytes. This object may be the result of the parseBase64 or parseHex method. The ten-byte field constructor is (all bytes are set to zero):

```
Bytes bb = new Bytes(10); /* Cleaned array of 10 bytes */
```

See 14.2.12 Bytes (array of bytes)

Element – XML elements. Objects of this type refer to the value of type "org.w3c.dom.Element". They can, for example, be the result of the "getElement", "XPath" methods or the methods above the "Container" objects.

Report - message. This type of object represents a parameterizable and linguistically customizable message. Messages are, for example, bug reports in the X-script. We can create a message:

```
Report r = new Report("REP005", "Unknown error!");
```

The last error reported during processing can be obtained, for example, by the "getLastError()" method.

Regex - regular expression. Objects of this type can be created using the new Regex (s) constructor, where s is a string with the regular expression source. Regular expression conforms to XML schema specification.

RegexResult - resut of regular expression. Objects of this type arise as a result of the getMatcher method.

Input/Output - files, streams. Objects of this type are used to work with files and streams. There are predeclared objects "$stdOut" (corresponding to java.lang.System.out), 1"$stdErr" (corresponding to java.lang.System.err) and "$stdIn" (corresponding to java.lang.System.in). The files can be used as the first parameter of the "putReport" method. The "$stdOut" file is automatically used in "out" and "outln" methods. Similarly, the file "$stdErr" is used in the "putReport" method.

Note: If the "out" or "outln" method is called in the X-script, the stream "$stdOut" is automatically added as the first parameter. (For example, for output to standard output, only the "outln ('My Text') can be written in the X-script.)

Exception - program exception. This object is passed when a program exception (bug) is caught in the construct "try {...} catch (Exception (ex)) {...}". An exception can also be caused by the "throw" command. You can create an exception object in a X-script using the new Exception ("text of error message") constructor.

BNFGrammar - extended Backus-Naur form. This type of object is defined by the variable in the declaration section or by the special declaration element:

```
<xd:BNFGrammar xd:name="name">
  rule := ...
  ...
</xd:BNFGrammar>
```

See 14.2.19 BNFGrammar (BNF grammars).

BNFRule - rule from BNF grammar. The BNF grammar rule can be used, for example, to validate text values of attributes or text nodes. The rule can be obtained from a BNFGrammar object using the method rule(name). See 14.2.20 BNFRule (BNF grammar rules).

5.2 ParseResult

ParseResult values are the results of X-script validation methods. They contain a string that has been processed by the validation method and the result value of parsing (this may be the original string, or the number, date, etc., depending on the type of validated value). If an error occurred while validating, the PrseResult value may contain a list of errors found during validation. The ParseResult object can be also created by a constructor with a string parameter. When created, this string is also set as a parsing result. Methods above the ParseResult type see DOPLNIT!. Example:

```
ParseResult p = new ParseResult("123");
p.setValue(0d123);  /* set decimal value. */
p.error("Error occured"); /* set error message. */
if (p.matches()) outln("OK");
if (p.error()) outln("error");
```

If this object occurs in the boolean expression, it is converted to a boolean type, and the result is true if errors are not reported in the object, otherwise false (ie. it is authomatically invoked the "match" method):

```
String s = "12456.78";
ParseResult p = double(s); / * The type method with a string parameter performs check and conversion. * /
float x =  (p) ? (double) p.getValue() : $NaN; /* if error set Not a Number value. */
```

5.3 Reference to the Attribute of the Current Element

In the X-script of elements and attributes it is possible to express a reference to the attribute of the current element by writing "@ attribute_name". If this entry appears in a boolean expression, then the value is true if the attribute with that name exists, otherwise the value is false. If this entry is given in string expressions, the result is an attribute value or an empty string.

```
<A xd:scriot = "match (@a AND @b OR @c); finally outln('Attribute a is: ' + @a);"
   a = "optional string" b = "optional string" c = "optional int" ...
```

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

5.4 Named Value

Named value represents a pair of name and value. The name can be any string and the value can be any value implemented in X definitions. The name of this type is "NamedValue". Sample of declaration and use:

```
NamedValue x = new NamedValue("abc", 123);
int i = (int) x.getValue();
String s = x.getName();
x.setValue("xxx");
namedValue y = %yyy=3.14;
```

In the first line, we created the named value "x" with the name "abc" and the value 123. In the second line, we store this value in the variable "i" using the "getValue ()" method. But we had to cast it to an integer, because the getValue () method returns the value of the general X-definition type "Any". In the third line, we saved the name "abc" to the variable "s" using the "getName ()" method. In the fourth line, we set the named variable "x" to string "xxx" using the "setValue" method. In the fifth line, a constructor example is given for the named value "y", the name being "yyy" and the value 3.14 (the name in this case must be the valid identifier of the X definitions).

5.5 Container

The "Container" data object can contain a number of arbitrary values supported in the X-definition, ie also the "Container" data type, which is also the supported value. Objects of this type may be the result of some X-script

methods (e.g. xPath, XQuery, etc.). Items stored in "Container" may be named or unnamed. The "Container" data type in X-definitions thus represents a type that merges the Java properties of the java.utils.Map class (the name of the named item in the "Container" serves as the key to retrieve its value) and java.utils.List (unnamed items are ordered sequentially so as elements in the field). Each unnamed item is then indexed and each named item is designated by its name. "Container" can represent data similar way as an XML element: attributes are as named items and unnamed items match the descendants of an element.

An example of using the Data Type Container is in the examples in chapters 6.2.3, 6.4 where it is used to prepare the data retrieved from the relational database and the chapter 7.7 where it is used to store the part of the XML document specified by the selected element. A more complete example of container use is given in chapter 12.

The declaration of "Container" and how to work with it see chapter 5.5.1.

➡ 5.5.1 Working with Container in X-script.

➡ 5.5.2 Container in external Java method

➡ 6 Construction Mode of X-definition

➡ 7 Using of X-definitions in Java code

➡ podrobné informace o typu Container: Kapitola 14.2

5.5.1 Working with Container in X-script.

The Container can be created in the X-script using a constructor defined by pairs of square brackets "[" a and "]". Inside brackets are the comma-separated values. When writing an unnamed item, only its value is entered. When writing a named item it starts with the percentage symbol ("%") followed by the name of the item, the equation character ("=") and the value. Unnamed items are stored in the order listed. The first unnamed item index is 0. Example:

```
Container measurements = [%locality = "Prague", %value="temperature", 11.3, 12.9, 15, 10.5 8.7];
Container empty = [];
```

Add item:

- add unnamed item with Conteiner method addItem(*value*),

- add named item with Conteiner method setNamedItem(*name,value*).

Read item:

- Read value of unnamed item from given index use method item(*index*),

- Read value of named item with given name use method getNamedItem(*name*).

See other methods of Conteiner in the table DOPLNIT!.

The following two examples describe two different ways how to create a container.

a) Container created by the constructor:

```
<xd:declaration>
    int injCnt = 3; /* Number of injures */

    /* Returns vehicle registration number. */
    String getVRN() {
        return "1B23456";
    }

    /* Data for element Accident. */
    Container accident = [
        %id      = "00123",
        %date    = "2011-05-17",
        %injury  = injCnt, /* value from variable */
        %death   = 0, /* value directly */
        %loss    = "600", /* value directly */

        [ /*Container in Container (element vrn) */
            "1A23456" /* value directly */
        ],
        [
            getVRN() /* value by method. */
        ]
    ];
</xd:declaration>
```

b) Container created using methods:

```
<xd:declaration>
    int injCnt = 3; /* Number of injures */

    /* Returns vehicle registration number. */
    String getVRN() {
        return "1B23456";
    }

    /* Prepare the empty Container for Accident. */
    Container accident = [];

    /* Set values to Container */
    accident.setNamedItem("id", "00123");
    accident.setNamedItem("date", "2011-05-17");
    accident.setNamedItem("injury", injCnt);
    accident.setNamedItem("death", 0);
    accident.setNamedItem("loss", "600");

    /* Prepare the Container of element "vrn" */
    Container vrn = ["1A23456"];
    /* Insert it as unnamed item to Container */
    accident.addItem(rz);

    vrn = [getVRN()];
    accident.addItem(vrn);
</xd:declaration>
```

Example b) can be used for example when data for the Container is obtained by successive reading of values e.g. from ResultSet from JDBC. The example a) can be used when all values are available.

➡ 5.5.2 Container in external Java method

5.5.2   Container in external Java method

➡ 5.5.1 Working with Container in X-script.

The Container can also be used in an external Java method. In Java, the Container implementation is represented by the org.xdef.XDContainer interface and can be created using the static "createXDContainer" method in the XDFactory class. See methods of the interface org.xdef.XDContainer (addXDItem, setXDNamedItem, getXDItem, getXDNamedItem etc.).

The "accident" example we can write as Java external method which returns an accident in the Container object:

```
import org.xdef.XDContainer;

public class ContainerTest {

    /** Return Container with Accident. */
    public static Container createAccident() {
        /* Prepare empty Container "accident". */
        XDContainer accident = XDFactory.createXDContainer();
        accident.setXDNamedItem("id", "00123");
        accident.setXDNamedItem("date", "2011-05-17");
        accident.setXDNamedItem("injury", 3);
        accident.setXDNamedItem("death", 0);
        accident.setXDNamedItem("loss", "600");
        /* Prepare Container with one string item. */
        XDContainer vrn = XDFactory.createXDContainer("1A23456");
        /* Insert the container "vrn" to "accident". */
        accident.addXDItem(rz);
        /* Prepare other Container "vrn". */
        vrn = XDFactory.createXDContainer(getVRN());
        /* Insert the container "vrn" to "accident". */
        accident.addXDItem(vrn);
        return accident;
    }
```

```
    /** Return vehicle registration number. */
    private static String getVRN() {
        return "2B23456";
    }

}
```

5.6   Working with Text Values

While processing an XML document, you can in the X-script refer to the currently processed XML items such as the element, attribute, or node text node using the following methods:

- getText() returns the text value of the currently processed attribute or text node. If this method is called out the attribute, or the text node it returns null

- setText(string) sets the value to the currently processed attribute or text node. If this method is called outside the attribute or text node, the exception occurs.

- getElement() returns the currently processed element. The method can be called from an X-Script of element, its attributes, or a text nodes.

Example of use:

```
<Vehicle xd:script = "occurs 0..*; finally printElem()"
         type = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
         vrn = "string(7); finally printVrn()"
         purchase = "date()"
         manufacturer = "string()"
         model = "string()" />

<xd:declaration>
    /** Print the actual element.*/
    void printElem() {
        Element el = getElement();
        outln(el);
    }
    /** Print text value of the processed attribute or the text node.*/
    void printVrn() {
        String vrn = getText();
        if (vrn != null) {
            outln(vrn);
        }
    }
</xd:declaration>
```

Printed result:

```
1A23456
<Vehicle typ="SUV" vrn="1A23456" ... />
```

5.1   Objects for Working with Databases

To work with different types of databases, the following types of objects are defined in the X-script:

Service. An object to connect to the services of different databases. In most cases it is passed to the X definitions as an external variable. However, the constructor can also be used in the X-script:

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

where parameter s1 is a type of database (e.g., "jdbc", "eXist" etc.), s2 is a database URL, s3 is a username, and s4 is a password.

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

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

ResultSet. Object containing the result of a database command. In the relational database instance, it contains a table whose rows have named columns. Lines can be accessed by the "next()" method. In the case of an XML database, the result of a Statement depends on the command, for example, this may be the Container object.

6 Construction Mode of X-definition

➡ Basic knowledge of XML markup language ([1], [2])

➡ 2 X-definition Technology

➡ 2.4 Exemplary XML Data

➡ 4 Description of structure of XML document by X-definition

The X-definition can serve as a rule for constructing an XML document or its specific part (eg selected elements) or as a rule for transforming an XML input document to another XML structure.

The description of the result XML document structure in construction mode is specified in the "create" section of the X-script. The create section is introduced by the keyword create and must provide data for creating the appropriate XML object. If the creation section is not defined, then the default operations are performed.

When creating the final XML document, the validation of the result is performed at the same time according to the model. The X-definition design for the construction mode is therefore adding the "create" section to the X-script where it is needed.

XML document construction is done starting with the root element. In order for the construction to be executed it is first necessary to specify the model of root element which will be used for creating the resulting XML document.

➡ 6.1 Create Section in X-script

➡ 6.7 Example of XML transformation into HTML

➡ 6.2 Value of attribute or text node created from value of X-script variable

➡ 6.3 Linking databases with X-definitions

➡ 6.4 Construction of template ("fixed") XML documents

➡ 6.5 Construction of groups

➡ 6.6 Combination of validation and construction mode

➡ 7 Using of X-definitions in Java code

6.1 Create Section in X-script

➡ 4 Description of structure of XML document by X-definition

➡ 6 Construction Mode of X-definition

Before selecting a specific example to create an XML document in the construction mode, it is necessary to familiarize yourself with the X-script writing capabilities and its meaning in the "create" section of the X-script. The "create" section provides the creation of the desired element, attribute or text node and their values. This section is introduced by the key word "create", followed by a command that provides the value by which the corresponding object is created. However, if the "create" section is not specified, a default operation is performed.

The command in the "create" section can return a value of several types. The table below provides an overview of the types of values that can be returned by the "create" section to construct an element, attribute, or text node. Each of the options is either added to a chapter with a more detailed description, or refers to a website where the X-definition in the design mode can be tested:

The type of value in the "create" section of the X-	Description	Example	Result of the example

	script			

create boolean	The value indicates whether the given element (can only be used for the element) will be created.
	
	If the value is true, then the maximum number (from quantifier) of elements will be created. Therefore, when the number of occurrences is unlimited, this X-script invokes an endless cycle, which will end with a lack of memory.
	
	If the value is false, then the element will not be created at all.
		`<vrn xd:script="`
		  `occurs 1..*;`
		  `create true">`
		
		  `string(7)`
		
		`</vrn>`
			***Error**: Outofmemory Exception (because of creating an infinite number of elements "rz").*

		`<vrn xd:script="`
		  `occurs 1..*;`
		  `create false">`
		
		  `string(7)`
		
		`</vrn>`
			Error: missing required element "rz" (not creates any element "rz").

	Online Demo: http://xdef.syntea.cz/tutorial/examples/C201.html		

create int	Non-negative integer value indicates the number of resulting elements that are created.
	
	When you use the attributes and text nodes the value is created to the String.
		`<vrn xd:script="`
		  `occurs 1..*;`
		  `create 2 ">`
		
		  `string(7)`
		
		`</vrn>`
			`<vrn/>`
			`<vrn/>`
			
			Error: missing text node.
			Error: missing text node.

	Online demos: http://xdef.syntea.cz/tutorial/en/example/C202.html http://xdef.syntea.cz/tutorial/en/example/C203.html		

create String	A string value that is used to create a text node or attribute values. A string value is always validated, ie verifies to the defined datatype (number, date, etc.) of an attribute or a text node.
	
	If the string value is used in the X-script of the element it will be created exactly one element (string value can be further used as a context, see chapter 6).
	
	A string constant must be enclosed in double quotation marks or apostrophes (because it is itself the attribute value enclosed in quotation marks, were used to define string apostrophes – with this combination, you must follow the correct pair of quotation marks or apostrophes) and must be non-empty.
		`<vrn xd:script="`
		  `occurs 1..*;`
		  `create 1 ">`
		
		  `string (7);`
		  `create ' 1A23456 '`
		`</vrn>`
		
		`<Accident xd:script="`
		  `occurs 0 .. *;`
		  `create 1 "`
		  `injury = "`
		    `int (0, 9999);`
		    `create 1 "`
		`/>`
			`<vrn>`
			  `1A23456`
			`</vrn>`
			
			`<Accident`
			  `injury = "12"`
			`/>`

	Online Demo: http://xdef.syntea.cz/tutorial/en/example/C209.html		

create null	Null values behaves like a value of boolean false, so the element is not created.	`<vrn xd:script="`
		  `occurs 1..*;`
		  `create 1 ">`
		
		  `string (7); create null`
		`</vrn>`
			`<vrn/>`
			
			Error: missing text node.

	Online Demo: http://xdef.syntea.cz/tutorial/en/example/C208.html		

create Context	The type of Context represents in the construction mode an object whose	`<vrn xd:script="`
		  `occurs 1..*;`
		  `create [true,`
			`<vrn>`
			  `1A23456`
			`</vrn>`

Dokument: **Chyba! Nenalezen zdroj odkazů.***, verze:* **Chyba! Nenalezen zdroj odkazů.***, ze dne:*
REF zedne \h * MERGEFORMAT **Chyba! Nenalezen zdroj odkazů.**

	values are used as a prototype for the construction of elements.
	
	The number of created elements corresponds to number of sequential items in the Context.
		`    false, true] ">`
		
		`   string (7);`
		`   create ' 1A23456 '`
		
		`</vrn>`
			`<vrn>`
			`   1A23456`
			`</vrn>`

	More in the chapter 6.1		
`create Element`	More in the chapter 6.1.		
`create` `XPath_result`	More in the chapter 6.1.4.1		

`create`
`ResultSet`
	The type of the ResultSet is a similar type of Context. The Resultset is used as an iterator and the number of its iterations specifies the number of elements that will be created.
	
	The entries of ResultSet can be all of the types, or null.
		`<vrn xd:script="`
		`   occurs 1..*;`
		`   create rs ">`
		
		`   string (7);`
		`   create rs.`
		`     getItem ('rzID')`
		`</vrn>`
		
		`<xd:declaration>`
		`external Service s;`
		`ResultSet rs =`
		`   s. query ('SELECT rzID`
		`     FROM RZ_Table');`
		`</xd:declaration>`
			`<vrn>`
			`   1A23456`
			`</vrn>`
			`<vrn>`
			`   9B87654`
			`</vrn>`

	More in the chapter 6.1.5		
`create` `XQuery_result`	In an implementation that supports XQuery can be the value of a sequence that is the result of the XQuery expression. This implementation contains the code of Saxonica standard and it is not included to the distribution of X-definition. For using of XQuery and XPath version 2.0 you need the software of the company Saxonica or some other software. Description of the use of XQuery is not included in this documentation.		

For the the construction of attributes and text nodes it can be used as a data source only numeric or string value or null.

➡ 6.1.1 Construction of elements

➡ 6.1.2 Construction of attributes and text nodes

➡ 6.1.3 Construction of element from Container

➡ 6.1.4 Construction of Element from Element

➡ 6.1.5 Construction of element from ResultSet

➡ 6.1.6 The source data used as the context used for the construction of XML document

➡ 6.7 Example of XML transformation into HTML

➡ 6.2 Value of attribute or text node created from value of X-script variable

➡ 6.3 Linking databases with X-definitions

➡ 6.4 Construction of template ("fixed") XML documents

➡ 6.5 Construction of groups

➡ 6.6 Combination of validation and construction mode

6.1.1 Construction of elements

If it had been known in advance the number of constructed elements, you can create an XML document with accidents according to the following X-definition (for the time being were omitted attributes and text nodes whose values can only be created when in the create section of the X-script returns a string). It is assumed that in the root element of an accident will be created 3 elements "Accident":

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name = "accidents"
      xd:root = "Accidents">

   <Accidents xd:script = "create true">
      <Accident xd:script = "occurs 0..*; create 3" ...>
         ...
      </Accident>
   </Accidents>
</xd:def>
```

The generated XML file then it will have the following form:

```
<Accidents>
   <Accident ...>
      ...
   </Accident>

   <Accident ...>
      ...
   </Accident>

   <Accident ...>
      ...
   </Accident>
</Accidents>
```

You can test it online here: http://xdef.syntea.cz/tutorial/en/example/C202.html.

The same result occurs when the boolean value true is replaced by the numeric value of 1:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name = "accidents"
      xd:root = "Accidents">

   <Accidents xd:script = "create 1">
      <Accident xd:script = "occurs 0..*; create 3" ...>
         ...
      </Accident>
   </Accidents>
</xd:def>
```

You can test it online here: http://xdef.syntea.cz/tutorial/en/example/C203.html.

However, these cases occur very rarely, because usually the number of elements is not known in advance. It is also the case in the situation when working with relational databases and reading the results of a SQL query via the interface ResultSet when elements in the result set are undergoing a throwaway prototype and in each iteration method "hasnext()" to determine whether the appropriate set contains more some other, yet unread, element.

➡ 6.1.2 Construction of attributes and text nodes

➡ 6.1.3 Construction of element from Container

➡ 6.1.4 Construction of Element from Element

➡ 6.1.5 Construction of element from ResultSet

➡ 6.1.6 Construction of element from ResultSet

6.1.2 Construction of attributes and text nodes

In the examples with X-definitions (the attribute "loss" was to demonstrate the use of null values in the create section changed to optional), you can now add the create section for the attributes and text nodes. To create sections will be given constant values, ie, that their values have been in all the elements of the resulting XML file the same and, therefore, the following sample is rather illustrative and, in practice, usable only in special cases. The example of the construction of the numeric attribute values from both a number and a string:

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name  = "accidents"
        xd:root  = "Accidents">

    <Accidents xd:script = "create 1">
        <Accident xd:script = "occurs 0..*; create 3"
            vrn      = "string(5) ; create '00123'"
            date     = "date();
                        create '03.04.2013'"
            injury   = "int(0, 9999); create 2"
            death    = "int(0, 9999); create '1'"
            loss     = "optional int(0, 100000000);
                        create null">

            ...
        </Accident>
    </Accidents>
</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name  = "accidents"
        xd:root  = "Accidents">

    <Accidents xd:script = "create 1">
      <Accident xd:script = "occurs 0..*; create 3">
            <vrn>string(5); create '00123'</vrn>
            <date>date(); create '03.04.2013'</date>
            <injury>int(0, 9999); create 2</injury>
            <death>int(0, 9999); create '1'</death>
            <loss>
                optional int(0, 100000000);
                create null
            </loss>
            ...
        </Accident>
    </Accidents>
</xd:def>
```

The resulting XML document will contain the three identical elements Nehoda. The attribute "loss" in the XML document will be not contained, because it was in its create section specified the null value (ie in the situation when an attribute or a text node is not present its value not exists). The element "loss" int the variant b) in the output of the XML document will be present, but its text value will be empty. And because an attribute or a text node of the element is optional, the   X-definition above is correct and it will construct the following XML documents:

a) variant with elements and attributes

```
<Accidents>
    <Accident
        vrn     = "00123"
        date    = "2013-04-03"
        injury  = "2"
        death   = "1" >

        ...
    </Accident>

    <Accident ...>
        ...
    </Accident>

    <Accident ...>
        ...
    </Accident>
</Accidents>
```

b) variant only with elements

```
<Accidents>
    <Accident>
        <vrn>00123</vrn>
        <date>2013-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <loss />
        ...
    </Accident>

    <Accident>
        ...
    </Accident>

    <Accident>
        ...
    </Accident>
</Accidents>
```

As you can see from the preceding examples, if the create section of the selected element is not defined, then this element is created automatically, even if the specified subtree is empty or does not contain in the resulting XML document neither attribute.

This procedure you can test online here:

- http://xdef.syntea.cz/tutorial/en/example/C211.html;

- http://xdef.syntea.cz/tutorial/en/example/C212.html;

- http://xdef.syntea.cz/tutorial/en/example/C213.html;

In general, you can write to the create section of attributes and text nodes any object or a method that returns an object. This object is in the construction mode converted to type String (that is, it is automatically called an internal method "toString"). After the conversion above are tests of the accuracy of the format of the string that corresponds to the datatype specified for that attribute or a text node.

Therefore, in the examples above it was possible to create a numeric type of attribute value from both, a string and a number. In the case of numeric value in the create section is this value converted to a string and then verified that the appropriate value corresponds to the datatype. Therefore, the following X-definition for an attribute or a text node of the element "id" caused an error:

```
"string(5); create 00123"
```

The specified number 00123 is first converted to a string. However, because it is a number, the conversion removes the leading zeros, thus it is converted to the string "123". However, the definition of the datatype requires a string length of 5 in this case, but the string length was 3 only. If the "id" was the "int" data type, it eliminates this problem, which you can verify online on the example:

- http://xdef.syntea.cz/tutorial/en/example/C214.html.

➡ 6.1.3 Construction of element from Container

➡ 6.1.4 Construction of Element from Element

➡ 6.1.5 Construction of element from ResultSet

➡ 6.1.6 The source data used as the context used for the construction of XML document

6.1.3 Construction of element from Container

➡ 5.5 Container

➡ 6.1.1 Construction of elements

➡ 6.1.2 Construction of attributes and text nodes

The objects of the type Container are objects that can be called an iterator for the gradual acquisition of their items, which can be used in the construction of the elements, attributes, and text nodes and their values. In general it can be said that each element (object), which is stored in the Container is used to create a single element. This means that the Container containing for example. 3 elements will allow the construction of 3 elements. Data from each of the object (ie an item of the Container) are also used to construct the attribute values and text nodes.

In doing so, the items of Container used to construct elements can be unnamed, or named not necessarily depending on the name (see below). While for the construction of attribute values is in general required that the values of the items of the Container to have same name as the element attributes.

The following topics in this chapter explain how to use the data type of the Container in the construction mode, first on simple examples, followed by a more complex example, and a more complex use of Context.

Because the Container object implements an iterator, is its usage and behavior similar to the Element and to the ResultSet.

The simplest example of usage of Container is to create a set of boolean values that indicates whether or not the element to create. The following example builds on the example from the chapter 6.1 when it uses for the construction of the three elements "Accident" the field with three values "true", ie. it exploits the Container [true, true, true]:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script = "create true">
        <Accident xd:script = "occurs 0..*; create [true, true, true]" ...>
            ...
        </Accident>
    </Accidents>
</xd:def>
```

The generated XML file then will contain the 3 elements `Accident`, as in the example in chapter 6.1:

```
<Accidents>
    <Accident ...>
        ...
    </Accident>

    <Accident ...>
        ...
    </Accident>

    <Accident ...>
        ...
    </Accident>
</Accidents>
```

The X-definition understands the Context as a set from which it selects one element after another. And even if it is in the X-definition defined model of element "Accident" only once, by the number of items in the Context will determine how many of the resulting elements will be created. The source code:

```
<Accident xd:script = "occurs 0..*; create [true, true, true]" ...>
```

You can, from the perspective of the create section, see it from the pseudocode as follows:

```
for-each (AnyValue value : Container ([true, true, true]) ) {
    // for each item 'value' of the Container construct element(s)
    if (value != null & value != false) { // for null or false do not create anything
        if (is-instance-of-integer(value)) {
            // the integer value specifies the number of constructed elements
            while (value-- > 0) {
                <Accident ... />
            }
        } else {
            // only one element is created
            <Accident ... />
        }
    }
}
```

Such a mechanism you can be verify online here: http://xdef.syntea.cz/tutorial/en/example/C204.html.

On the contrary, the following X-definition constructs only one element "Accident", because if the value of the entry (even if it is part of the Context) is false or null, leads to the fact that the element or attribute or text node will not be created:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script = "create true">
        <Accident xd:script = "occurs 0..*; create [false, true, false]" ...>
            ...
        </Accident>
    </Accidents>
</xd:def>
```

The generated XML file will contain only one element "Accident":

```
<Accidents>
   <Accident ...>
      ...
   </Accident>
</Accidents>
```

Check it online here: http://xdef.syntea.cz/tutorial/en/example/C205.html.

The analogy between boolean and numeric value in the create section can also be converted to the Context, and therefore, instead of the three values "true" will lead to the creation of 3 elements of an accident the following X-definition:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name = "accidents"
      xd:root = "Accidents">

   <Accidents xd:script = "create true">
      <Accident xd:script = "occurs 0..*; create [1, 2]" ...>
         ...
      </Accident>
   </Accidents>
</xd:def>
```

Here is online example: http://xdef.syntea.cz/tutorial/en/example/C206.html.

Now add the construction of the remaining structure - attributes, and text nodes using the Context (there will be also explained a similar mechanism for the use of Element). In the case of the Context type (or Element) in the current entry of the Context (or in the currently processed element) are searched the named data items (or attribute names) corresponding to the names of the attributes in the X-definition.

The pseudocode described above can therefore be supplemented with this finding as follows:

```
for-each (AnyValue value : Container ([true, true, true]) ) {
   // for each item 'value' of the Container construct element(s)
   if (value != null && value != false) { // for null or false do not create anything
      if (is-instance-of-integer(value)) {
         // the integer value specifies the number of constructed elements
         while (value-- > 0) {
            <Accident id="value.get('id')" date="value.get('date')" ... />
         }
      } else {
         // for other types of value create one element with the item
         <Accident id="value.get('id')" date="value.get('date')" ... />
      }
   }
}
```

The listed properties with automatic lookup values for attributes and text nodes can be used as shown in the following example (in fact it already is a basic usage of the context described in the chapter 6.1.6). The Container will be built manually with one element, which will also be the type of Container (will correspond to the element "Accident", which will be constructed), the named items (they will match the attributes) and other elements of the Container (these will correspond to the child elements "vrn" in the element of Accident). Unnamed elements in the Container generally match the values of text nodes.

Important note: For the construction of an element which in any depth of the XML tree contains at least two child elements with that are siblings in the XML tree, you must build the Container from the context values or from elements (see chapter 6.1.4). It is not enough to use the Container containing the simple data types, as in the following example. Therefore, the following example describes the construction of XML document using attributes.

For the construction of Container will be used an example from Chapter 5.5.1 Working with Container in X-script. in combination with X-definition described above (the comprehensive examples of work with the context will be extended X-definition – into the element "vrn" the attribute "price" will be added):

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
       xd:name = "accidents"
       xd:root = "Accidents">

   <Accidents>
       <Accident xd:script = "occurs 0..*; create accident"
           vrn     = "string(5)"
           date    = "date()"
           injury  = "int(0, 9999)"
           death   = "int(0, 9999)"
           loss    = "int(0, 100000000)">

           <vrn xd:script="occurs 1..*"
               price = "int(1, 100000)">
               string(7)
           </vrn>
       </Accident>
   </Accidents>

   <xd:declaration>
       /* value ca be stored in variable. */
       int injury = 3;

       /* Data returned by a method. */
       String getVRN() {
           return "1B23456";
       }

       /* Data for the element Accident. */
       Container accident = [
           %id    = "00123", /* named item used as value of attribute. */
           %date  = "2011-05-17",
           %injured = zrCnt, /* Value from variable injuryCnt. */
           %killed = 0, /* Hodnota jako číslo. */
           %loss  = "600", /* value as string*/

           /* Contexts for child elements "vrn". The item %price cannot be placed on the same level as
             the item %loss, because the processor of X-definition would not find it. */
           [
               %price = "240", /* Named value for the construction of attribute 'price'. */
               "1A23456" /* Unnamed value prepared for construction of textového node. */
           ],

           [%price = "130", getVRN()]
       ];
   </xd:declaration>
</xd:def>
```

The above described property simplified the X-definition and allowed to write only one single create section to the entire model (load data from the Container will ensure that by default). The output of the specified X-definition will be the following XML file:

```
<Accidents>
   <Accident vrn= "00123"
       date    = "2011-05-17"
       injury  = "3"
       death   = "0"
       loss    = "600" >

       <vrn price="240">1A23456</vrn>
       <vrn price="130">1B23456</vrn>
   </Accident>
</Accidents>
```

See online example: http://xdef.syntea.cz/tutorial/en/example/C346.html.

Tip: the value of named and unnamed items of context it is possible to retrieve from a relational database – see chapter 6.1.5.

6.1.4 Construction of Element from Element

6.1.5 Construction of element from ResultSet

6.1.6 The source data used as the context used for the construction of XML document

6.1.4 Construction of Element from Element

➡️ basic knowledge of XPath expression

➡️ 5.5 Container

➡️ 6.1.1 Construction of elements

➡️ 6.1.2 Construction of attributes and text nodes

The construction of an element from an element is very similar to the construction of the element from Container. If an element is used for the construction of the element according to X-definition, it behaves as if the source element in its place was the Container object containing this element as a single item. The X-script with Container containing the single element for the construction of the element "Accident":

```
<Accident xd:script = "occurs 0..*; create [ELEMENT]" ...>
```

you can specify directly in the create section the element with source data:

```
<Accident xd:script = "occurs 0..*; create ELEMENT" ...>
```

Usage of element as the data source (or an element as an item of Container), however, allows you to construct the element that contains on the same the depth level of XML tree more siblings.

The Example from Chapter 6.1.1, where was described the mechanism of construction of elements regardless to the attributes and text nodes, can be modified so that when you request the creation of one element "Accident" (for simplicity without attributes or child elements) will be inserted instead of the values "true" or "1" the source element from create section:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script = "create new Element('A')">
...
    </Accidents>
</xd:def>
```

In the above example was in the X-script created a new object of Element by using the command "new".

As with the item in the Container nor the source element, the item is used regardless of name (**Note**: it doesn't matter the name of the root element, the names of the child elements should be accepted!). In the above example has a source element name "A", but it will still be used for the construction of the element "Accident", which will be created from it. The result of the specified X-definition is the following document.

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C207.html.

For simplicity, the XML source data will be presented in separate XML files and in the appropriate X-definition will be accessible through the external variable of "source", which can be set from Java code. Let the XML input data be available:

A typical example of using the element as the data source for the construction of element is the use of the content of one XML document to another XML document structure (see chapter 6.7). In the source XML document, you can use the XPath expression to select the specific part (a particular source element) that will be used as the source data for the construction of the element.

For simplicity, the XML source data will be presented in separate XML files and in the appropriate X-definition will be accessible through the external variable of "source", which can be set from Java code. Let the XML input data be available:

```
<root>
    <rec vrn  = "00123"
        date   = "2013-04-03"
        injury = "2"
        death = "1"
        loss   = "120">
        <car vrn="1C23456" />
    </rec>

    <rec vrn       = "00345"
        date   = "2012-04-03"
        injury = "2"
        death = "1"
        loss   = "150">
        <car vrn="1B23456" />
    </rec>

    <rec vrn  = "00456"
        date   = "2011-04-03"
        injury = "2"
        death = "1"
        loss   = "30">
        <car vrn="1A23456" />
        <car vrn="2A34567" />
    </rec>
</root>
```

```
<root>
    <rec>
        <vrn>00123</vrn>
        <date>2013-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <loss>120</loss>
        <car>1C23456</car>
    </rec>
    <rec>
        <vrn>00345</vrn>
        <date>2012-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <loss>150</loss>
        <car>1B23456</car>
    </rec>
    <rec>
        <vrn>00456</vrn>
        <date>2011-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <loss>30</loss>
        <car>1A23456</car>
        <car>2A34567</car>
    </rec>
</root>
```

➡ 6.1.5 Construction of element from ResultSet

➡ 6.1.6 The source data used as the context used for the construction of XML document

➡ 6.1.4.1 Use of xpath method

➡ 6.1.6.1 Using of the method `from`

6.1.4.1   Use of xpath method

The first will be for the construction elements of the source elements described the use of the xpath method.

The xpath method as input parameters to enter the XPath expression and the source element in which the XPath expression according to the specified will be searched for the source of the element (or elements); the method returns as a result the list of found elements as the NodeList. Calling the "xpath" method does not change the current context and, therefore, to use the xpath method is especially useful in those cases where a specific element is loaded the source XML document, the values of attributes, and text nodes, and its child elements are used in a form of the source element; or in the case where the source XML document is used to retrieve a specific attribute value, or a text node (see Chapter 7.7).

The first example will be simple. Using the newly defined X-definition reads from the source file the list of all the elements "car":

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script="create source">
        <vrn xd:script="occurs 1..*;
            create xpath('/root/rec/car', source)"
            vrn="string(7)" />
    </Accidents>

    <xd:declaration>
        /* Root element */
        external Element source;
    </xd:declaration>
</xd:def>
```

c) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script="create source">
        <vrn xd:script="occurs 1..*;
                create xpath('/root/rec/car', source)">
                string(7)
        </vrn>
    </Accidents>
    <xd:declaration>
        /* Root element */
        external Element source;
    </xd:declaration>
</xd:def>
```

As can be seen, the X-definition is based on the X-definition where has been left only the element "vrn", and has been modified so that the structure match the source XML document.

If the element "Accident" is not the root element, then the section create must be specified. On the root element is not the create section need to define, and it is therefore included here only for sample purposes (the global variable "source" contains one element and therefore creates a single element "Accident").

Xpath method ('/root/rec/car ', source) finds in the root element all elements "car". Processor of X-definition finds elements "car" and uses them to create elements of the "vrn". When you create an element "vrn" and the attribute "crn" is used as the value of attribute "vrn" in the element "car" (ie, the behavior is similar to the behavior of Container and thanks to the same name of attribute "vrn" is not necessary to define the create section for this attribute. The result after construction will be the following:

a) variant with elements and attributes

```
<Accidents>
    <vrn vrn="1C23456" />
    <vrn vrn="1B23456" />
    <vrn vrn="1A23456" />
    <vrn vrn="2A34567" />
</Accidents>
```

b) variant only with elements

```
<Accidents>
    <vrn>1C23456</vrn>
    <vrn>1B23456</vrn>
    <vrn>1A23456</vrn>
    <vrn>2A34567</vrn>
</Accidents>
```

A special case of the construction of the same XML document but only for a specific element of the source XML data. If it was required to make the 2nd element "rec", just modify the XPath expression as follows:

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script="create source">
        <vrn xd:script="occurs 1..*;
            create xpath('/root/rec/car[2]', source)"
            vrn="string(7)">
        </vrn>
    </Accidents>

    <xd:declaration>
        external Element source;
    </xd:declaration>
</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script="create source">
        <vrn xd:script="occurs 1..*;
            create xpath('/root/rec/car[2]', source)">
                string(7)
        </vrn>
    </Accidents>

    <xd:declaration>
        external Element source;
    </xd:declaration>
</xd:def>
```

Constructed document will be:

a) variant with elements and attributes

```
<Accidents>
    <vrn vrn="1B23456" />
</Accidents>
```

b) variant only with elements

```
<Accidents>
    <vrn>1B23456</vrn>
</Accidents>
```

In practice, however, it is not usually known sequence number of element and its specific selection, therefore, the search for a key value may be specified in XPath expression. A key value used in the source XML document is the value of the attribute "id" (or text node of element "id"). Identically constructed document as is mentioned above, therefore, the X-definition will be (the xpath method returns only those elements "rec", which has the value of the attribute "id" "00345"):

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name  = "accidents"
      xd:root  = "Accidents">

   <Accidents xd:script="create source">
      <vrn xd:script="occurs 1..*;
         create xpath(
              '/root/rec[@id=\'00345\']/car',
              source)"
         vrn="string(7)">
      </vrn>
   </Accidents>

   <xd:declaration>
      external Element source;
   </xd:declaration>
</xd:def>
```

In the case that the xpath method returns a single element, you can use it to implement even simple transformations. If it is for a variant a) the aim to print the registration number of that record as the value of text node of element, and not as the attribute value. The X-definition can be as follows:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name  = "accidents"
      xd:root  = "Accidents">

   <Accidents xd:script="create source">
      <vrn xd:script="occurs 1..*;
         create xpath('/root/rec[@id=\'00345\']/car/@vrn', source)">
         string(7); create xpath('/root/rec[@id=\'00345\']/car/@vrn', source)"
      </vrn>
   </Accidents>

   <xd:declaration>
      external Element source;
   </xd:declaration>
</xd:def>
```

The parameter of the method "xpath" that searches the element "car" with the value of the attribute of the parent element and from the element "car" reads the value of the attribute "spz" which returns as a result. This will be result:

```
<Accidents>
   <vrn>1B23456</vrn>
</Accidents>
```

If the method "xpath" returned more results, then this example will not work and you must use the method "from".

The method "xpath" try on simple examples online here:

- http://xdef.syntea.cz/en/tutorial/example/C210.html;

- http://xdef.syntea.cz/tutorial/en/example/C215.html.

For transformations between two XML documents it is usually better to use the method "from" that is described in the chapter 6.1.6.1.

➡ 6.1.6.1 Using of the method `from`

➡ 7.7 External instance methods

6.1.5 Construction of element from ResultSet

➡️ basic knowledge of JDBC

➡️ 6.1.1 Construction of elements

➡️ 6.1.2 Construction of attributes and text nodes

Another typical case and often used in practice is the structure of an XML document and its elements on the basis of data obtained as the result of a database command. Because the most common type of the database is relational database, the following samples will be carried out via the SQL commands over relational databases.

For this purpose it is possible to use in the create section the ResultSet as a data source (from JDBC) that has an iterator through which gradually offers its items. The ResultSet returns in the base case, the method "query" called on the Service object (which is an instance of JDBC java. sql. Connection).

For simplicity, assume that the goal will be to construct an XML document containing a list of registered vehicles, ie the aim of the construction element "Vehicle" from the variant a) example in Chapter 7.7.

Let there be a relational table VozidloTable:

id *(primary key)*	type	date	vendor	model
1A23456	SUV	14-07-2011	Škoda, a.s.	Yeti
2B34567	sport	21-12-2007	Toyota	RX8
3C45678	MPV	03-02-2013	VW	Golf

SQL statements of the relational database, you can enter by using JDBC directly from external methods in the Java code. In this documentation in the following example are entered and processed directly from X-definition.

Connections to the database it is appropriate to obtain or create in the Java code and to pass into the X-definition as an external variable — in a XDService object that in the X-definition acts as the data type Service – see chapter 7.6.1.

Let's hahe the external variable which represents a connection to a relational database:

```
...
<xd:declaration>
    ...
    external Sdervice service;
    ...
</xd:declaration>
...
```

To obtain the ResultSet, from which it would be possible to construct the resulting XML document, you can use the method "query" on the object "service" that passes as a parameter the SQL command:

```
ResultSet cars = service.query('SELECT * FROM CarsTable');
```

ResultSet "Vehicles" can now be used as an iterator to the "create" section. Constructs so many elements, how much the ResultSet contains items. The boolean method "next()" shifts the cursor to the next row of the ResultSet and it is called implicitly before the construction of an element. The attribute values are created from the columns of the row returned from ResultSet. The value is obtained by method "getItem" with the name of column.

However, the next() method on the ResultSet you can call explicitly in the X-script. On the ResultSet is also available the boolean method "hasNext()", which returns "true" if the ResultSetu has a next row, otherwise it returns false.

The X-definition which of the data obtained from a relational database constructs an XML document with the elements of the Vehicle is based on the X-definition referred to in chapter 4.7 is listed below:

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "Vehicles"
        xd:root = "Vehicles">

    <List>
        <Vehicle xd:script = "occurs 0..*;
            create cars; finally cars.close()"
            type = "enum('SUV', 'MPV', 'personal',
                'truck', 'sport', 'other');
                create cars.getItem('type')"
            vrn = "string(7);
                create cars.getItem('id')"
            purchase = "date();
                create convertDT(
                    cars.getItem('date'))"
            producer = "string();
                create cars.getItem('vendor')"
            model = "string;
                create cars.getItem('model')" />
    </List>

    <xd:declaration>
        external Service service;
        ResultSet cars = service.query(
            'SELECT * FROM VozidloTable');

        /** Method to convert a time from the format
         * dd-MM-yyyy to xd:dateTime. */
        String convertDT(String dt) {
            Datetime in = parseDate(dt, "dd-MM-yyyy");
            return in.toString("yyyy-MM-dd");
        }
    </xd:declaration>
</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "Vehicles"
        xd:root = "Vehicles">

    <List>
        <Vehicle xd:script = "occurs 0..*;
            create cars; finally cars.close()">
            <type>enum('SUV', 'MPV', 'personal',
                'truck', 'spor', 'other');
                create cars.getItem('type')
            </type>
            <vrn>string(7);
                create cars.getItem('id')
            </vrn>
            <purchase>
                date();
                create convertDT(
                    cars.getItem('date'))
            </purchase>
            <producers>string();
                create cars.getItem('vendor')
            </producer>
            <model>string;
                create cars.getItem('model')
            </model>
        </Vehicle>
    </List>

    <xd:declaration>
        external Service service;
        ResultSet cars = service.query(
            'SELECT * FROM VozidloTable');

        /** Method to convert a time from the format
         * dd-MM-yyyy to xd:dateTime. */
        String convertDT(String dt) {
            Datetime in = parseDate(dt, "dd-MM-yyyy");
            return in.toString("yyyy-MM-dd");
        }
    </xd:declaration>
</xd:def>
```

Because the date format in the database is different from the format required by the X-definition, it was necessary to ensure the conversion to the desired format. The appropriate conversion ensures user method "convertDT", which is declared in the X-script in the xd:declaration section. The created XML document will have the following content:

a) variant with elements and attributes

```xml
<List>
    <Vehicle type = "SUV"
        vrn = "1A23456"
        purchase = "2011-07-14"
        producer = "Škoda, a.s."
        model  = "Yeti" />

    <Vehicle type = "sport"
        vrn = "2B34567"
        purchase = "2007-12-21"
        producer = "Toyota"
        model  = "RX8" />

    <Vehicle type = "MPV"
        vrn = "3C45678"
        purchase = "2013-02-03"
        producer = "VW"
        model  = "Golf" />
</List>
```

b) variant only with elements

```xml
<List>
    <Vehicle>
        <type>SUV</type>
        <vrn>1A23456</vrn>
        <purchase>2011-07-17</purchase>
        <producer>Škoda, a.s.</producer>
        <model>Yeti</model>
    </Vehicle>

    <Vehicle>
        <type> sport</type>
        <vrn>2B34567</vrn>
        <purchase>007-12-21</purchase>
        <producer>Toyota</producer>
        <model>RX8</model>
    </Vehicle>

    <Vehicle>
        <type>MPV</type>
        <vrn>3C45678</vrn>
        <purchase>2013-02-03</purchase>
        <producer>VW</producer>
        <model>Golf</model>
    </Vehicle>
</List>
```

If for example, the value of attribute "model" in a relational database is empty (it should be default NULL), then in the variant b) creates the element "model", but its text node will have an empty value.

In addition, if the X-definition of variant b) requested an optional element "model", then it would not be this element being created. In order to ensure that behavior it is required to add the create section also to the X-script of element "model". Therefore we add to the create section of the X-script of this element the the test if the value is NULL (whose result will be true or false):

```xml
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
    xd:name  = "Vehicles"
    xd:root  = "Vehicles">

  <List>
      <Vehicle xd:script = "occurs 0..*; create cars; finally cars.close()">
          <type>
              enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other'); create cars.getItem('type')
          </type>
          <vrn>string(7); create cars.getItem('id')</vrn>
          <purchase>date(; create convertDT(cars.getItem('date'))</purchase>
          <producer>string(); create cars.getItem('vendor')</producer>
          <model xd:script = "occurs 0..1; create cars.getItem('model') != null">
              string; create cars.getItem('model')
          </model>
      </Vehicle>
  </List>

  <xd:declaration>
      external Service service;
      ResultSet cars = service.query('SELECT * FROM CarsTable');

      String convertDT(String dt) {
          Datetime in = parseDate(dt, "yyyy-MM-dd");
          return in.toString("yyyy-MM-dd ");
      }
  </xd:declaration>
</xd:def>
```

The example how to use `ResultSet` in the X-definition `garageDB` and `garagDBElem`:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name  = "garage"
      xd:root  = "List">

   <List>
      <Vehicle xd:script = "occurs 0..*; create cars; cars.close()" ... >
         ...
      </Vehicle>
      ...
   </List>
   <xd:declaration>
      external Service service;
      ResultSet cars = service.query('SELECT * FROM VozidloTable');
      ...
   </xd:declaration>
</xd:def>
```

The variable "cars" (of the type `ResultSet`) is replaced by the method "query". In this case the `ResultSet` is created internally. The method `getItem` is not invoked on the `ResultSet` but as a global method in the X-script and it is applied on the actual context (the internally created `ResultSet`). After the SQL is processed it is authomatically ensured it external resources are closed (see the chapter 6.3.2 - as you can see in the following example):

a)   variant with elements and attributes

b)   variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name  = "garageDB"
      xd:root  = "List">

   <List>
      <Vehicle xd:script = "occurs 0..*;
         create service.query(
            'SELECT * FROM VozidloTable')"
         type  = "enum('SUV', 'MPV', 'personal',
            'truck', 'sport', 'other');
            create getItem('type')"
         vrn = "string(7);
            create getItem('id')"
         purchase = "date();
            create convertDT(getItem('date'))"
         manufacturer = "string();
            create getItem('vendor')"
         model  = "string;
            create getItem('model')" />
   </List>

   <xd:declaration>
      external Service service;

      /** Conversion of datetime from the form
       * yyyy-MM-dd to the form yyyy-MM-dd. */
      String convertDT(String dt) {
         Datetime in = parseDate(dt, "yyyy-MM-dd");
          return in.toString("yyyy-MM-dd");
      }
   </xd:declaration>
</xd:def>
```

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name  = "garageDBElem"
      xd:root  = "List">

   <List>
      <Vehicle xd:script = "occurs 0..*;
         create 'SELECT * FROM VozidloTable'">
         <type>enum('SUV', 'MPV', 'personal',
            'truck', 'sport', 'other');
            create getItem('type')
         </type>
         <vrn>string(7);
            create getItem('id')
         </vrn>
         <purchase>
            date();
            create convertDT(getItem('date'))
         </purchase>
         <manufacturer>string();
            create getItem('vendor')
         </manufacturer>
         <model>string;
            create getItem('model')
         </model>
      </Vehicle>
   </List>

   <xd:declaration>
      external Service service;

      /** Conversion of datetime from the form
       * yyyy-MM-dd to the form yyyy-MM-dd. */
      String convertDT(String dt) {
         Datetime in = parseDate(dt, "dd-MM-yyyy");
          return in.toString("yyyy-MM-dd ");
      }
   </xd:declaration>
</xd:def>
```

The method `query` may be used also for SQL statements with parameters. The method query has in such case more parametrers. The first parameter is then the SQL query statement and and the parameters follows: `query(SQL_statement, parameter_1, parameter_2, …, parameter_n)`:

```
...
<xd:declaration>
   String type = "osobní";
   String model = "Yeti";
   ...
   external Service s;
   ResultSet rs = s.query('SELECT rzID FROM RZ_Table WHERE typ=? AND model=?', type, model);
   ...
</xd:declaration>
...
```

The arguments of the SQL statement are in the same sequence as the symbols "?" in the SQL statement.

➡ 6.1.6 The source data used as the context used for the construction of XML document

➡ 6.3 Linking databases with X-definitions

6.1.6   The source data used as the context used for the construction of XML document

➡ 6.1.1 Construction of elements

➡ 6.1.2 Construction of attributes and text nodes

➡ 6.1.3 Construction of element from Container

➡ 6.1.4 Construction of Element from Element

➡ 6.1.5 Construction of element from ResultSet

In the chapter 6.1 in most cases were the source data explictitly specified excplicitně specifikovala for each part of the constructed XML document, ie. for each element, attribute or the text node. For some examples (especially for Context and Element), options have been described for defining the data source for these two data types. In this chapter, the implicit behavior will be specified and generalized.

When explicitly specifying a data source that typically produces the entire resulting XML document, it is usually used an input data source (Element, Context, ResultSet, etc.). The source data will now be used as the context. The context is usually the element passed to the create section.

The simple examples are given in Chapter 6.1.4, with the create section used only for elements, but not for attributes or text values. The value of attributes and text nodes was automatically taken from the source data specified in the create section of the element. In this case, the source data of the respective element has become a context for creating an attribute value, respectively. text node.

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C341.html

The another property of the create section is to automatically create a text value of an element when only the text value is entered into the create section. The X-definition list from chapter 6.1.4 can therefore be overwritten as follows:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
     xd:name = "accidents"
     xd:root = "Accidents">

   <Accidents xd:script="create source">
     <vrn xd:script="occurs 1..*; create xpath('/root/rec[@id=\'00345\']/car/@vrn', source)">
         string(7)
     </vrn>
   </Accidents>

   <xd:declaration>
     external Element source;
   </xd:declaration>
</xd:def>
```

From the text value of the vrn attribute, a new element with a text node value is generated equal to this text value, and the whole element becomes the context for the vrn element whose value of the text node will be created.

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C342.html

The same scenario occurs when you obtain data from the database through ResultSet:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents>
        <vrn xd:script="occurs 1..*; create service.getItem('rz')">
            string(7)
        </vrn>
    </Accidents>

    <xd:declaration>
        external Service service;
    </xd:declaration>
</xd:def>
```

Let the following input files be inserted in the next explanation (the root element of the given DOM) into the corresponding X-definitions as an external Element variable:

a) variant with elements and attributes   b) variant only with elements

```
<rec
    id     = "00123"
    date   = "2013-04-03"
    injury = "2"
    death = "1"
    loss   = "120">

    <vrn>1B23456</vrn>
    <vrn>1C23456</vrn>
</rec>
```

```
<rec>
    <vrn>00123</vrn>
    <date>2013-04-03</date>
    <injury>2</injury>
    <death>1</death>
    <loss>120</loss>

    <vrn>1B23456</vrn>
    <vrn>1C23456</vrn>
</rec>
```

As you can see, the names of the attributes as well as the internal elements are the same as the X-definitions below:

a) variant with elements and attributes   b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents>
        <Accident xd:script = "occurs 0..*;
                    create source"
            id      = "string(5)"
            date    = "date()"
            injury  = "int(0, 9999)"
            death   = "int(0, 9999)"
            loss    = "int(0, 100000000)">

            <vrn xd:script="occurs 1..*">
                string(7)
            </vrn>
        </Accident>
    </Accidents>

    <xd:declaration>
        external Element source;
    </xd:declaration>
</xd:def>
```

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents>
        <Accident xd:script = "occurs 0..*;
                    create source">
            <vrn>string(5)</vrn>
            <date>date()</date>
            <injury>int(0, 9999)</injury>
            <death>int(0, 9999)</death>
            <loss>int(0, 100000000)</loss>
            <vrn xd:script="occurs 1..*">
                string(7)
            </vrn>
        </Accident>
    </Accidents>

    <xd:declaration>
        external Element source;
    </xd:declaration>
</xd:def>
```

It differs only in the name of the root element, which does not matter for the intended purpose (see chapter 6.1.3). Therefore, you can specify the create section only for the Accident element. For its sub-elements and all attributes (all of which do not have the create section), the data will be taken from the context of the data listed in the create section of one of the parent elements, ie as the context set the input file represented in the X-definition variable. That is, the source context automatically becomes a context for the vrn elements as well.

The result will be the following XML documents (up to the order of attributes):

a) variant with elements and attributes

b) variant only with elements

```
<Accidents>
    <Accident
        id     = "00123"
        date   = "2013-04-03"
        injury = "2"
        death  = "1"
        loss   = "120">

        <vrn>1B23456</vrn>
        <vrn>1C23456</vrn>
    </Accident>
</Accidents>
```

```
<Accidents>
    <Accident>
        <vrn>00123</vrn>
        <date>2013-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <loss>120</loss>

        <vrn>1B23456</vrn>
        <vrn>1C23456</vrn>
    </Accident>
</Accidents>
```

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C343.html

As you can see, those descendents (elements) from the context that had the same name (in the example above were vrn elements) or the text value when a text node was created were automatically used.

When working with a context, it does not matter to the default element, which is defined in the create section (not necessarily the root element), but the names of its sub-elements and all the attributes that are not listed in the create sections do matter.

In another example, it can be seen that it does not matter the order of attributes or elements in source data, but only on their names. The order of the elements is given by the order in the X-definition, not the order in the source data. The order of the attributes is taqken from the X-definition. Therefore, if the X-definitions listed above (list of X.def and theExampleFile list) are used for the following input:

a) variant with elements and attributes

b) variant only with elements

```
<rec death = "1"
    date   = "2013-04-03"
    injury = "2"
    id     = "00123"
    loss   = "120">
    <vrn>1B23456</vrn>
    <vrn>1C23456</vrn>
</rec>
```

```
<rec>
    <date>2013-04-03</date>
    <death>1</death>
    <injury>2</injury>
    <vrn>1C23456</vrn>
    <loss>120</loss>
    <vrn>00123</vrn>
    <vrn>1B23456</vrn>
</rec>
```

outputs will be retrieved again by the NodeCreateOutput2.xml list and the NodeCreateElemOutput2.xml.

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C344.html

Similar rules are also applied by using Context as a data source in the create section. The context is generally used as a context for creating an element. If the context contains another context as an element, this nested context is used as a context for subelement construction. Named values are used to construct attributes and the unnamed string value from the sequential part of the Container object is used to construct the value of text node.

Using the following modified X-definition for the Accident element (sub-element vrn must have at most one occurrence; for multiple occurrences, the following sample will not work and the procedure in Chapter 6.1.3):

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name = "accidents"
      xd:root = "Accidents">

   <Accidents>
      <Accident xd:script = "occurs *; create source"
         id      = "string(5)"
         date    = "date()"
         injury  = "int(0, 9999)"
         death   = "int(0, 9999)"
         loss    = "int(0, 100000000)">

         <vrn>
               string(7)
         </vrn>
      </Accident>
   </Accidents>

   <xd:declaration>
      Container source = [[
         %id='12345',
         %date = '2000-01-10',
         %injury = '4',
         %death = '2',
         %loss = '1240',
         '1A23456' // the unnamed value will be used to construct the text value of the vrn element
      ]];
   </xd:declaration>
</xd:def>
```

The result will be following XML document:

```
<Accidents>
   <Accident
      id      = "12345"
      date    = "2000-01-10"
      injury = "4"
      death = "2"
      loss    = "1240">
      <vrn>1A23456</vrn>
   </Accident>
</Accidents>
```

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C345.html.

The use of Container in the context for the construction of elements and their sub-elements is described in the chapter 6.1.3.

➡ 6.1.6.1 Using of the method `from`

6.1.6.1   Using of the method `from`

➡ 6.1.4.1 Use of xpath method

➡ 6.1.6 The source data used as the context used for the construction of XML document

When constructing more complex elements, it is very often necessary to select a specific object (string value, element, etc.) from the context and set a new context for the child's construction of the currently processed element. The from method can be used when the context for creating an element is some other element. In this case, the method from invokes the xpath expression, but the corresponding XPath expression specified as its parameter is performed above the current context rather than the entire source XML document.

Using the from method also sets a new context for all the descendants of the currently processed element. Therefore, if the method returns an element, this element will be used as the context for all descendants until the context is changed in a child by anoter calli of the from method.

For further examples, the source XML document will be utilized as follows:

a) variant with elements and attributes

```
<root>
    <rec
        date   = "2013-04-03"
        injury = "2"
        death  = "1"
        total  = "120">
        00123
        <car vrn="1C23456" />
    </rec>
    <rec
        date   = "2011-04-03"
        injury = "2"
        death  = "1"
        total  = "150">
        00345
        <car vrn="1B23456" />
    </rec>
    <rec
        date   = "2011-04-03"
        injury = "2"
        death  = "1"
        total  = "30">
        00456
        <car vrn="1A23456" />
        <car vrn="2A34567" />
    </rec>
</root>
```

b) variant only with elements

```
<root>
    <rec>
        <date>2013-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <total>120</total>
        00123
        <List>
            <car>1C23456</car>
        </List>
    </rec>
    <rec>
        <date>2011-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <total>150</total>
        00345
        <List>
            <car>1B23456</car>
        </List>
    </rec>
    <rec>
        <date>2011-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <total>30</total>
        00456
        <List>
            <car>1A23456</car>
            <car>2A34567</car>
        </List>
    </rec>
</root>
```

An X-definition that converts the data from the example into the desired output according to Chapter 2.4. Let the above XML input documents be passed through their root element to the following X-definitions as an external source variable:

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script = "create source">
        <Accident xd:script = "occurs 0..*;
                create from('rec')"
            id      = "string(5);
                create from('text()')"
            date    = "date()"
            injury  = "int(0, 9999)"
            death   = "int(0, 9999)"
            loss    = "int(0, 100000000);
                create from('@total')">

            <vrn xd:script="occurs 1..*;
                create from('car')">
                string(7); create from('@vrn')
            </vrn>
        </Accident>
    </Accidents>

    <xd:declaration>
        external Element source;
    </xd:declaration>
</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script = "create source">
        <Accident xd:script = "occurs 0..*;
                create from('rec')">
            <vrn>string(5);
                create from('text()')
            </vrn>
            <date>date()(</date>
            <injury>int(0, 9999)</injury>
            <death>int(0, 9999)</death>
            <loss xd:script = "
                create from('total')">
                int(0, 100000000)
            </loss>
            <vrn xd:script="occurs 1..*;
                create from('cars/car')">
                string(7)
            </vrn>
        </Accident>
    </Accidents>

    <xd:declaration>
        external Element source;
    </xd:declaration>
</xd:def>
```

Now it will be explained in detail the context setting in the above-mentioned X-definitions for the selected create sections:

- **create** source sets the context Element to the root of the data source element, ie to the root element;

- create from('rec') the method from executes the XPath expression 'rec' in the current context, ie in the root element, ie, the result of the XPath expression will be all 'rec' elements from the source XML document (returned as a NodeList from the xpath method). Subsequently, the method from sets the context of the created Accident element to the root/rec element of the source document (after creating one Accident element, the context moves to the following root/rec element and the construction of the next Accident element begins);

- create from('text()') expression 'text ()' in the current context, ie the method from returns the value of the root node text node and the id attribute sets the context to the text value obtained by the XPath expression;

- create from('@total') the method executes the executes the XPath expression '@total' in the current context, ie the method from returns the value of the total attribute of the root/rec element and the "loss" attribute sets the context to the value of the "total" attribute;

- create from('cars/car') The method from executes the XPath expression 'cars/car' in the current context, ie the method returns a list of all /root/rec/cars/car elements and sets it as the context of the vrn element. The list of elements returned by the method from is subsequently iteratively scrolled and the each one vrn element is created for a car element in the list.

The elements, attributes, and text nodes for which no create section is specified will be created from the current context that has been set up by their predecessors. An XML document created based on the X-definition and input data will look like this:

a) variant with elements and attributes

```
<Accidents>
    <Accident
        id    = "00123"
        date  = "2013-04-03"
        injury = "2"
        death = "1"
        loss  = "120">
        <vrn>1C23456</vrn>
    </Accident>
    <Accident
        id    = "00345"
        date  = "2012-04-03"
        injury = "2"
        death = "1"
        loss  = "150">
        <vrn>1B23456</vrn>
    </Accident>
    <Accident
        id    = "00456"
        date  = "2011-04-03"
        injury = "2"
        death = "1"
        loss  = "30">
        <vrn>1A23456</vrn>
        <vrn>2A34567</vrn>
    </Accident>
</Accidents>
```

b) variant only with elements

```
<Accidents>
    <Accident>
        <vrn>00123</vrn>
        <date>2013-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <loss>120</loss>
        <vrn>1C23456</vrn>
    </Accident>
    <Accident>
        <vrn>00345</vrn>
        <date>2012-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <loss>150</loss>
        <vrn>1B23456</vrn>
    </Accident>
    <Accident>
        <vrn>00456</vrn>
        <date>2011-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <loss>30</loss>
        <vrn>1A23456</vrn>
        <vrn>2A34567</vrn>
    </Accident>
</Accidents>
```

The described principle can be tested online here:

- http://xdef.syntea.cz/tutorial/en/example/C501.html

- http://xdef.syntea.cz/tutorial/en/example/C502.html

- http://xdef.syntea.cz/tutorial/en/example/C503.html

A similar but somewhat more complicated example will solve the situation where the input data has a slightly different meaning: the first record determined by the rec element will represent the highway accident, the second on the 1st class road and the other on the other roads. In addition, the attributes "date" or the elements "date" will be created. The date of creation of the output document will be generated in the element "Accidents" by the built-in method "now()" and it will be converted to the desired format by the method toString (...):

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

   <Accidents xd:script = "create source"
       date = "date();
           create toString(now(), 'yyyy-MM-dd');">

       <Highway xd:script = "occurs 0..*;
               create from('rec[@id=\'00123\']')"
           id      = "string(5);
                   create from('text()')"
           date    = "date()"
           injury  = "int(0, 9999)"
           death   = "int(0, 9999)"
           loss    = "int(0, 100000000);
                   create from('@total')">

           <vrn xd:script="occurs 1..*;
                   create from('car')">
               string(7); create from('@vrn')
           </vrn>
       </Highway>

       <FirstClassRoad xd:script = "occurs 0..*;
               create from('rec[2]')"
           id      = "string(5);
                   create from('text()')"
           date    = "date()"
           injury  = "int(0, 9999)"
           death   = "int(0, 9999)"
           loss    = "int(0, 100000000);
                   create from('@total')">

           <vrn xd:script="occurs 1..*;
                   create from('car')">
               string(7); create from('@vrn')
           </vrn>
       </FirstClassRoad>

       ...
   </Accidents>

   <xd:declaration>
      external Element source;
   </xd:declaration>
</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

   <Accidents xd:script = "create source">
       <date> date();
           create toString(now(), 'yyyy-MM-dd');
       </date>
       <Highway xd:script = "occurs 0..*;
           create from('rec[@id=\'00123\']')">
           <vrn>string(5);
               create from('text()')
           </vrn>
           <date>date()</date>
           <injury>int(0, 9999)</injury>
           <death>int(0, 9999)</death>
           <loss xd:script = "
               create from('total')">
               int(0, 100000000)
           </loss>
           <vrn xd:script="occurs 1..*;
                   create from('cars/car')">
               string(7)
           </vrn>
       </Highway>

       <FirstClassRoad xd:script = "occurs 0..*;
           create from('rec[2]')">
           <vrn>string(5);
               create from('text()')
           </vrn>
           <date>date()</date>
           <injury>int(0, 9999)</injury>
           <death>int(0, 9999)</death>
           <loss xd:script = "
               create from('total')">
               int(0, 100000000)
           </loss>
           <vrn xd:script="occurs 1..*;
                   create from('cars/car')">
               string(7)
           </vrn>
       </FirstClassRoad>

       ...
   </Accidents>

   <xd:declaration>
      external Element source;
   </xd:declaration>
</xd:def>
```

In the above X-definition, the XPath expression in the from method was changed to achieve the desired result: In the X-definition above, the XPath expression was changed to achieve the desired result:

- the method from('rec[@id=\'00123\']') in the current context, it searches for and returns all rec elements, whose id attribute is 00123;

- the method from('rec[2]') returns the second occurrence of the rec element in the current context.

The generated XML document then has the following contents:

```
<Accidents datum = "22.4.2013">
    <Dalnice
        id    = "00123"
        date  = "2013-04-03"
        injury = "2"
        death = "1"
        loss  = "120">
        <vrn>1C23456</vrn>
    </Dalnice>
    <PrvniTrida
        id    = "00345"
        date  = "2012-04-03"
        injury = "2"
        death = "1"
        loss  = "150">
        <vrn>1B23456</vrn>
    </PrvniTrida>

    ...
</Accidents>
```

```
<Accidents>
    <date>22.4.2013</date>
    <Dalnice>
        <vrn>00123</vrn>
        <date>2013-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <loss>120</loss>
        <vrn>1C23456</vrn>
    </Dalnice>
    <PrvniTrida>
        <vrn>00345</vrn>
        <date>2012-04-03</date>
        <injury>2</injury>
        <death>1</death>
        <loss>150</loss>
        <vrn>1B23456</vrn>
    </PrvniTrida>

    ...
</Accidents>
```

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C504.html

Other variants of the method from are:

a) **fromElement(Element e)** – similar to method from () when element e is used as the current context;

b) **fromAttr(c, name)** – the method returns the value of the attribute "name" from the Container c.

6.1.6.2 Variants of the method from (fromDB) for a database

a) **fromDB(DBConnection con, String SQL_statement [, String parameters])** – method with similar functionality as the query method (see chapter 6.1.5) that executes the SQL statement specified on the database connection and returns a ResultSet object. The typical use is in the create section, where the resulting element is constructed from the a ResultSet. The DBConnection (URL, login, password) object can be created in the X-Script in the declaration part by the constructor for DBConnection:

```
...
<xd:declaration>
    DBConnection con = new DBConnection('jdbc:derby://localhost:3309/sample', 'admin', '123456');
</xd:declaration>
...
<Dalnice xd:script="create fromDB(con, 'SELECT * FROM RoadTable WHERE type=?', 2)" ...>
    ...
</Dalnice>
...
```

b) **fromDB(DBStatement con [, String parameters])** – similar to the previous method, but as a parameter accepts a SQL statement (DBStatement) with parameters:

```
...
<xd:declaration>
    DBConnection con = new DBConnection('jdbc:derby://localhost:3309/sample', 'admin', '123456');
    DBStatement st = new DBStatement(con, 'SELECT * FROM RoadTable WHERE type=?');
</xd:declaration>
...
<Dalnice xd:script="create fromDB(st, 2); finally {st.close(); con.close();}" ...>
    ...
</Dalnice>
...
```

c) **fromDBItem(DBConnection con, String SQL_command, String attribute_Name_from_DB [, String parameters_of_SQL_command])** – the method uses the database connection con and executes the SQL_command with the_SQL_command parameters. The result is the value of attribute_name from the SQL query. The following sample lists the highway identifiers in the text node:

```
...
<xd:declaration>
    DBConnection con = new DBConnection('jdbc:derby://localhost:3309/sample', 'admin', '123456');
</xd:declaration>
...
<Dalnice xd:script="finally con.close()" ...>
    create fromDBItem(con, 'SELECT id, death FROM RoadTable WHERE type=?', 'id', 2)
</Dalnice>
...
```

6.2 Value of attribute or text node created from value of X-script variable

➡ 4.7.3 Declaration Section of X-definition

➡ 6.1 Create Section in X-script

In the chapter 6.1, a global variable defined in X-definition in the element xd:declaration was used in the create section a data source for element construction. Typically, this was a variable containing value of an element or a Service object.

Similarly (by specifying the name of the variable in the create section), globally defined variables can also be used to construct attribute values and text nodes, as shown in the following example. The variable, defined in X-Script globally, has global visibility in all parts of X-Script, not only in the xd: declaration declaration section but also in the xd: X-script attributes as well as in the text values of attributes and text nodes in the element model :

a) variant with elements and attributes

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name  = "accidents"
        xd:root  = "Accidents">

   <Accidents>
      <Accident xd:script = "occurs 0..*;
               create true"
         id    = "string(5); create id"
         datum = "date(); create datum"
         injury = "int(0, 9999); create injury"
         death = "int(0, 9999); create death"
         loss  = "int(0, 100000000); create loss">
         ...
      </Accident>
   </Accidents>

   <xd:declaration xd:scope = "global">
      String id = "00123";
      String datum = "22.04.2013";
      String injury = "2";
      String death = "0";
      String loss = "124";
   </xd:declaration>

</xd:def>
```

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name  = "accidents"
        xd:root  = "Accidents">

   <Accidents>
     <Accident xd:script = "occurs 0..*;
                            create true">
       <vrn>string(5); create id</vrn>
       <date>date(); create datum </date>
       <injury>int(0, 9999); create injury; </injury>
       <death> int(0, 9999); create death </death>
       <loss> int(0, 100000000); create loss </loss>
       ...
     </Accident>
   </Accidents>

   <xd:declaration xd:scope = "global">
      String id = "00123";
      String datum = "22.04.2013";
      String injury = "2";
      String death = "0";
      String loss = "124";
   </xd:declaration>

</xd:def>
```

Since the values of both the attributes and the text nodes are text, the variables that are used to construct them are the String data type.

Variable values can be obtained by different calculations, conversions, e.g. in user methods, etc. Therefore, they are usually not specified as constants as in the above example.

Using this X-definition, the following document will be generated:

a) variant with elements and attributes

```
<Accidents>
    <Accident id = "00123"
        datum  = "2013-04-22"
        injury = "2"
        death  = "0"
        loss   = "124">
        ...
    </Accident>
</Accidents>
```

b) variant only with elements

```
<Accidents>
    <Accident>
        <vrn>00123</vrn>
        <date>2013-04-22</date>
        <injury>2</injury>
        <death>0</death>
        <loss>124</loss>
        ...
    </Accident>
</Accidents>
```

6.3   Linking databases with X-definitions

➡ 6.1 Create Section in X-script

➡ 5.5 Container

➡ basic knowledge of database technology and JDBC

This chapter serves as a more comprehensive example of using THE relational databases in X-definitions and using the context to construct XML documents.

6.3.1   Statement

In X-script, you can first prepare a java.sql.Statement object that can then be sent to the database engine. Statement is created by the DBStatement prepareStatement method using the DBConnection object and by calling the queryItem method (String Attribute_name[, String Parameters…]), you can obtain attribute values and specify the SQL statement parameters:

```
...
<xd:declaration>
    DBConnection con = new DBConnection('jdbc:derby://localhost:3309/sample', 'admin', '123456');
    DBStatement st = con.prepareStatement('SELECT id, death FROM RoadTable WHERE type=?');
    ResultSet rs = st.query(2);
</xd:declaration>
...
<Highway xd:script="finally {rs.close(); st.close(); con.close()}"...>
    create st.queryItem(id, 2);
</Highway>
...
```

The meaning of the DBConnection constructor parameters is given in variants of method z in Chapter 6.1.6.2.

The query method called on the Statement accepts only the SQL statement parameters.

6.3.2   Closing resources

When using a database interface in X-definitions, the external resources are opened, which is usually a SQL statement and a pointer to the result table, ie DBStatement, ResultSet and DBConnection. Closing of the resources in question is done either automatically or the programmer has to call the close() method on given object.

The automatic close occurs when the DBStatement, ResultSet, and DBConnection objects are not available in X-Script via a variable (see the detailed example in Chapter 6.1.5) or when this variable is not delared as external. ResultSet and DBStatement are closed automatically when the end of result is reached or when the processing of an element is finished. In general, resources will be closed at the latest end of the X-definition process:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name = "garage"
      xd:root = "Vehicles">
   <Vehicles>
      <Vehicle xd:script = "occurs 0..*; create service.query('SELECT * FROM VozidloTable')"
          type  = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
          vrn   = "string(7)"
          purchase = "date()"
          manufacturer = "string()"
          model  = "string" />
   </Vehicles>
</xd:def>
```

However, if the DBStatement, ResultSet, and DBConnection are stored in an external X-Script variable, the source close must be done by the programmer:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name = "garage"
      xd:root = "Vehicles">

   <Vehicles>
      <Vehicle xd:script = "occurs 0..*; create cars; finally cars.close()" ... />
   </Vehicles>

   <xd:declaration>
      external Service service;
      external ResultSet cars = service.query('SELECT * FROM VozidloTable');
      ...
   </xd:declaration>

</xd:def>
```

If the programmer does not close system resources consistently, they will run out and the system may report an error. If the programmer closes a source that were closed automatically the close command is ignored.

6.4 Construction of template ("fixed") XML documents

➡ 6.1 Create Section in X-script

All of the previous demos of the X-definitions assumed that the constructed XML document is very variable and has relatively few fixed (unchanging) data. These examples are typically characterized by creating an X-Script section in many elements, attributes, and text nodes.

The opposite is the construction of XML documents that have a "static" character and, on the contrary, very few elements are variables in them. For this purpose, the so-called "template model" may be used.

The template model differs from the current element model by adding the template keyword to the X-Script of an element model. In the template model, all attributes and text node values are then understood as the constants that are copied to the result XML document being created (in fact, all constants are automatically converted to X-Script sections); the structure and number of occurrences of all sub-elements of a given template model are also constant (again, in fact, the models are generated as required).

If you need to insert a non-constant value in some part od the template model, you need to insert X-Script with a special keyword $$$script: followed by a X-script to specify a nonconstant value. If the X-script entry puts the X-Script element, then its attributes, text nodes, and all sub-elements are automatically converted from the template model.

By using the template model, the X-definition from Chapter 6.7.2 would be simplified and made more concise in the form below. For simplicity, the iterative construction of the tr elements was modified so that the create sections were moved from the element xd: sequence directly to the tr element:

a) the variant for transforming of an XML document into HTML using an element and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
  xd:name = "AccidentsHtml"
  xd:root = "html">

<html xd:script="template">

  <head>
    <title>List of accidents</title>
  </head>

  <body>
    <table border="1">
      <tr>
        <th>ID</th>
        <th>Date</th>
        <th>Injured [persons]</th>
        <th>Death [persons]</th>
        <th>Loss [1000 KC]</th>
          <th>Participants [vrn]</th>
      </tr>

      <tr xd:script="$$$script: occurs 0..*;
           create from('//Accidents/Accident')">
        <td>string(5); create from('@id') </td>
        <td>date(); create from('@date') </td>
        <td>int(0, 9999);
            create from('@injury')
        </td>
        <td>int(0, 9999);
            create from('@death')
        </td>
        <td>int(0, 100000000);
            create from('@loss')
        </td>
        <td>
          <xd:sequence xd:script="occurs 0..*;
            create from('./rz')">
          string; create from('./text()');
            <span>create ', '</span>
            </xd:sequence>
        </td>
      </tr>
      </table>
    </body>
  </html>
</xd:def>
```

b) the variant for transforming of an XML document into HTML using only elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
  xd:name = "AccidentsHtml"
  xd:root = "html">

<html xd:script="template">

  <head>
    <title>List of accidents</title>
  </head>

  <body>
    <table border="1">
      <tr>
              <th>ID</th>
        <th>Date</th>
        <th>Injured [persons]</th>
        <th>Death [persons]</th>
        <th>Loss [1000 KC]</th>
        <th>Participants [vrn]</th>
      </tr>

      <tr xd:script="$$$script: occurs 0..*;
           create from('//Accidents/Accident')">
        <td> string(5);
            create from('./id/text()')</td>
        <td> date();
            create from('./date/text()')</td>
        <td>int(0, 9999); create
            from('./injury/text()')</td>
        <td>int(0, 9999); create
            from('./death/text()')</td>
        <td>int(0, 100000000);
            create from('./loss/text()')</td>
        <td>
          <xd:sequence xd:script="occurs 0..*;
            create from('./rz')">
          string; create from('./text()');
            <span>create ', '</span>
          </xd:sequence>
        </td>
      </tr>
      </table>
    </body>
  </html>
</xd:def>
```

Template model can be tested online here:

- http://xdef.syntea.cz/tutorial/en/example/C701.html
- http://xdef.syntea.cz/tutorial/en/example/C801.html

6.5 Construction of groups

➡ 4.11 Group Specifications

➡ 6.1 Create Section in X-script

This chapter describes how to construct xd:choice, xd:mixed and xd sequence groups in the X-definition. which allow you to define a more complex XML document structures.

6.5.1 Strict order of elements (group xd:sequence)

In the xd:sequence group model, which determines the strict order of items in the group, it is also possible to write the xd:script attribute containig a create section. The value in the create section will then be used to controll the construction the entire group, ie to create all of its items.

The first example is based on the example of Chapter 6.1.1, which for the purpose of this chapter is extended to include several sub-elements of the Accident element:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script = "create 1">
        <xd:sequence xd:script="occurs 2..2; create 2">
            <Accident_A xd:script = "occurs 1..1; create 1" ...>
                ...
            </Accident_A>

            <Accident_B xd:script = "occurs 1..1; create 1" ...>
                ...
            </Accident_B>
        </xd:sequence>
    </Accidents>
</xd:def>
```

The generated XML element will then take the following form:

```
<Accidents>
    <Accident_A ...>      ...</Accident_A>

    <Accident_B ...>      ...</Accident_B>

    <Accident_A ...>      ...</Accident_A>

    <Accident_B ...>      ...</Accident_B>
</Accidents>
```

As you can see from the example, the elements of the group (Accident_A and Accident _B elements) were created in the order in which they were created and just twice (create 2 in the element xd: sequence).

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C311.html.

If a section of element xd:sequence is used as the value to create a list of elements obtained by the xpath method, the group of elements will be created as many times as the element list contains. Using the from method, the current XPath context specified in the method in question is set for all elements of the group. An example of use is given in chapter 6.7. You can also try the following online here:

- http://xdef.syntea.cz/tutorial/en/example/C312.html

- http://xdef.syntea.cz/tutorial/en/example/C411.html.

- http://xdef.syntea.cz/tutorial/en/example/C412.html.

6.5.2 Arbitrary order of elements (group xd:mixed)

To the xd:mixed group model, which specifies the arbitrary order of its items in the group, it is also possible to write the xd:script attribute containing a create section. The value in the create section will then be used to controll construction of the entire group, ie to create all of its items.

The first example is based on the example of Chapter 6.1, which for the purpose of this chapter is extended to include several sub-elements of the Accident element:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script = "create 1">
        <xd:mixed xd:script="occurs 1..2; create 2">
            <Accident_A xd:script = "occurs 1..1; create 1" ...>
                ...
            </Accident_A>

            <Accident_B xd:script = "occurs 1..1; create 1" ...>
                ...
            </Accident_B>
        </xd:mixed>
    </Accidents>
</xd:def>
```

The generated XML element will then take the following form:

```
<Accidents>
    <Accident_A ...>        ...</Accident_A>

    <Accident_B ...>        ...</Accident_B>

    <Accident_A ...>        ...</Accident_A>

    <Accident_B ...>        ...</Accident_B>
</Accidents>
```

As you can see from the example, the elements of the group (Accident_A and Accident _B) were created just twice (create 2 in element xd:mixed). Even though the order of the elements in the group is free, they were created in the order of the X-definition. The construction is therefore similar as xd: sequence group.

Limitations: The minimum number of occurrences listed in the X-script of xd: mixed may be at most 1.

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C321.html.

If in the xd:mixed group the list of elements obtained from the the xpath method is used, then the elements are created as many times as the element list contains. Using the from method, the current XPath context specified in the method in question is set for all elements of the group. You can also try the following examples online:

- http://xdef.syntea.cz/tutorial/en/example/C322.html

- http://xdef.syntea.cz/tutorial/en/example/C341.html

- http://xdef.syntea.cz/tutorial/en/example/C432.html

At this point, there will be discussed a difference in XML document processing for xd:mixed in validation and construction mode. Suppose the following document is available:

```
<Accidents>
    <Accident_B ...>        ...</Accident_B>
    <Accident_A ...>        ...</Accident_A>
</Accidents>
```

And the X-definition with the create section:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name = "accidents"
      xd:root = "Accidents">

   <Accidents xd:script = "create 1">
      <xd:mixed xd:script="occurs 1..2; create 2">
         <Accident_A xd:script = "occurs 1..1; create xpath('Accident_A', source)" ...>
            ...
         </Accident_A>

         <Accident_B xd:script = "occurs 1..1; create xpath('Accident_B', source)" ...>
            ...
         </Accident_B>
      </xd:mixed>
   </Accidents>

   <xd:declaration>
      external Element source;
   </xd:declaration>
</xd:def>
```

If this X-definition was applied to the input document in the validation mode (ie all create sections would be ignored), the order of the elements in an XML document obtained by the X-definition corresponds to the validated input document (the process is controlled by the input data):

```
<Accidents>
   <Accident_B ...>      ...</Accident_B>
   <Accident_A ...>      ...</Accident_A>
</Accidents>
```

But in a construction mode where the process is controlled by the X-definition, the order of the elements of the constructed document will match the order in the X-definition:

```
<Accidents>
   <Accident_A ...>      ...</Accident_A>
   <Accident_B ...>      ...</Accident_B>
</Accidents>
```

6.5.3   Choice of elements (xd:choice group)

In the xd:choice group model, which allows you to select just one of variants, you can also write the xd:script containing a create section. The value in the create section is then used to create the entire group, ie to create one of its variants.

The first example for the purpose of this chapter contains several sub-elements of the Accident element:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name = "accidents"
      xd:root = "Accidents">

   <Accidents xd:script = "create 1">
      <xd:choice xd:script="occurs 2..2; create 2">
         <Accident_A xd:script = "occurs 1..1; create 1" ...>
            ...
         </Accident_A>

         <Accident_B xd:script = "occurs 1..1; create 1" ...>
            ...
         </Accident_B>
      </xd:choice>
   </Accidents>
</xd:def>
```

The generated XML element will then take the following form:

```
<Accidents>
    <!-- Constructed according to the first elementi in the group. -->
    <Accident_A ...>      ...</Accident_A>
    <!-- Also constructed according to the first elementi in the group. -->
    <Accident_A ...>      ...</Accident_A>
</Accidents>
```

As you can see from the example, the first possible element was always created in the group (the element for which the sections were created to contain the data to create it) and the other are ignored. This element has always been the element Accident_A. The element was created two times (see create 2 in element xd:choice).

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C331.html.

If you need to determine which element in the group is to be created, you need to use specific conditions as in the following example:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script = "create 1">
        <xd:choice xd:script="occurs 2..2; create 2">
            <Accident_A xd:script = "occurs 1..1; create c++ == 0" ...>
                ...
            </Accident_A>

            <Accident_B xd:script = "occurs 1..1; create c == 2" ...>
                ...
            </Accident_B>
        </xd:choice>
    </Accidents>

    <xd:declaration>
        int c = 0;
    </xd:declaration>
</xd:def>
```

The generated XML element will then take the following form:

```
<Accidents>
    <Accident_A ...>      ...</Accident_A>
    <Accident_B ...>      ...</Accident_B>
</Accidents>
```

The reason is that the X definition processor moves the element after the element in the appropriate group until it manages to construct an element (or a text node). Other elements skip over the current iteration of the group construction. Therefore, the Accident _A element was created at the first creation of the group, the test c++ == 0 returned the true value for the create section and the variable c had the value 1. While the second construct of the group, the test c++ == 0 returned false and increased the value of the variable c will be 2, the element Accident_A was not created and therefore continued with another element; test for Accident_B, since c == 2, iss true, and element will be constructed.

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C332.html.

If the the value from the xpath method in the xd:choice group is used, the list of elements obtained by the XPath expressionwill be used for construction of items of the group. Using the from method, the current XPath specified in the method is set for the construction of all elements of the group.

The described principle can be tested online here:.

- http://xdef.syntea.cz/tutorial/en/example/C421.html

- http://xdef.syntea.cz/tutorial/en/example/C422.html.

The following example shows a simple example of use. Suppose, first of all, that the following document is available:

```
<Accidents>
    <Accident_B ...>      ...</Accident_B>
    <Accident_A ...>      ...</Accident_A>
</Accidents>
```

And the corresponding X-definition with the create sections:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script = "create 1">
        <xd:choice xd:script="occurs 2..2; create 2">
            <Accident_A xd:script = "occurs 1..1; create xpath('Accident_A', source)" ...>
                ...
            </Accident_A>

            <Accident_B xd:script = "occurs 1..1; create xpath('Accident_B', source)" ...>
                ...
            </Accident_B>
        </xd:choice>
    </Accidents>

    <xd:declaration>
        external Element source;
    </xd:declaration>
</xd:def>
```

Since element Accident _A, which is first handled with the X-definition (in the design mode does not specify the order of the elements of the input document), is found in the source XML document, this element is also created in the xd: choice:

```
<Accidents>
    <Accident_A ...>      ...</Accident_A>
</Accidents>
```

However, if another X-definition is used for the same source data:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents xd:script = "create 1">
        <xd:choice xd:script="occurs 2..2; create 2">
            <Accident_A xd:script = "occurs 1..1; create xpath('Accident_C', source)" ...>
                ...
            </Accident_A>

            <Accident_B xd:script = "occurs 1..1; create xpath('Accident_B', source)" ...>
                ...
            </Accident_B>
        </xd:choice>
    </Accidents>

    <xd:declaration>
        external Element source;
    </xd:declaration>
</xd:def>
```

the resulting document will be different because the Nehoda_C element was not found in the source document and therefore the Accident_A element of the group is not created:

```
<Accidents>
    <Accident_B ...>      ...</Accident_B>
</Accidents>
```

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C333.html.

6.6   Combination of validation and construction mode

➡️ 6.1 Create Section in X-script

The validation mode and the construction mode of X-definitions can be interconnected. The link is implemented from the X-definition so that when using it in the validation mode, it is possible to invoke the construction of another element from its X-Script, or vice versa, when using the given X-definition in the construction mode, the XML data can be validated in its X-Script. Both options will be explained on simple examples.

Important note: If the X-definifion contains the X-Script creation sections describing the construction of elements, attributes, text nodes and their values is invoked in the validation mode, all created sections will be ignored and only the validation will be performed.

6.6.1   Validation in the construction mode

➡ 4.7.2 Head of X-definition

The first example presents a simpler variant, which is validation in the X-definition the construction mode. Let's take the following X-definition, which contains the source data directly in X-Script (based on the example in the chapter 6.1.6.1):

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Nehody | root">

    <Accidents xd:script = "create source">
        <Accident xd:script = "occurs 0..*;
                create from('rec')"
            id      = "string(5) "
            datum   = "date()"
            injury  = "int(0, 9999)"
            death   = "int(0, 9999)"
            loss    = "int(0, 100000000)">

            <vrn xd:script="occurs 1..*">
                string(7)
            </vrn>
        </Accident>
    </Accidents>

    <root>
        <rec xd:script = "occurs 0..*"
            id      = "string(5) "
            datum   = "date()"
            injury  = "int(0, 9999)"
            death   = "int(0, 9999)"
            loss    = "int(0, 100000000)">

            <vrn xd:script="occurs 1..*">
                string(7)
            </vrn>
        </rec>
    </root>

    <xd:declaration>
        <![CDATA[
            /* Source XML data. */
            String data = "
                <root>
                    <rec id='00123'
                        date = '2012-05-03'
                        injury = '0'
                        death = '0'
                        loss = '90'>

                        <vrn>1A23456</vrn>
                    </rec>
                </root>
            ";

            /* Validation of source data. */
            Element source = xparse(data, "*");
        ]]>
    </xd:declaration>
</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Nehody | root">

    <Accidents xd:script = "create source">
        <Accident xd:script = "occurs 0..*;
                create from('rec')">
            <vrn>string(5)</vrn>
            <date>date()</date>
            <injury>int(0, 9999)</injury>
            <death>int(0, 9999)</death>
            <loss>int(0, 100000000)</loss>
            <vrn xd:script="occurs 1..*">
                string(7)
            </vrn>
        </Accident>
    </Accidents>

    <root>
        <rec xd:script = "occurs 0..*">
            <vrn>string(5)</vrn>
            <date>date()</date>
            <injury>int(0, 9999)</injury>
            <death>int(0, 9999)</death>
            <loss>int(0, 100000000)</loss>
            <vrn xd:script="occurs 1..*">
                string(7)
            </vrn>
        </rec>
    </root>

    <xd:declaration>
        <![CDATA[
            /* Source XML data. */
            String data = "
                <root>
                    <rec>
                        <vrn>00123</vrn>
                        <date>2012-05-03</date>
                        <injury>0</injury>
                        <death>0</death>
                        <loss>90</loss>

                        <vrn>1A23456</vrn>
                    </rec>
                </root>
            ";
            /* Validation of source data. */
            Element source = xparse(data, "*");
        ]]>
    </xd:declaration>
</xd:def>
```

In the X-definition, an Accident element is created in the construction mode (the Java Code requires the construction of the Accident element according to the X definition, see the chapter 7.2). The data for its construction is obtained from the element source. However, as seen from variants a) and b), the source element was obtained directly in the X-Script of the X-definition. The xparse method returns the value of the variable source and therefore this method must be called first before the result element Accident is constructed.

To create the source element, the xparse method has two variants:

- xparse(*input_data*), creates element as the result of parsing of the input data,

- xparse(*input_data, name_of_x-definition*), also creates element from the input source data,. However, it vlalidates the input source according to the model of element declared as root in the X-definiition with the name from the second parameter. If the actual X-definition should be used you can specify asterisk ("*") instead of the name..

Because the xparse method with two parameters was used in the X-definition above, and an asterisk was given as the second value, a root element model was used according to the X-definition header in the xd:root attribute.

The described principle can be tested online here: http://xdef.syntea.cz/tutorial/en/example/C601.html.

6.6.2 Construction in the validation mode

The second example illustrates the case of element construction during processing of validation mode. Suppose the following X-definition validates the XML input data specified by the root root element. Initial XML validation is performed first, and after finishing it (the event event is finally was invoked), the construction of the Accident element is started:

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
       xd:name = "accidents"
       xd:root = "root">

   <Accidents xd:script = "create from('/root')">
      <Accident xd:script = "occurs 0..*;
            create from('rec')"
            id      = "string(5) "
            datum   = "date()"
            injury  = "int(0, 9999)"
            death   = "int(0, 9999)"
            loss    = "int(0, 100000000)">

            <vrn xd:script="occurs 1..*">
                  string(7)
            </vrn>
      </Accident>
   </Accidents>

   <root xd:script = "finally
         returnElement(xcreate('Nehody'))">
      <rec xd:script = "occurs 0..*"
            id      = "string(5) "
            datum   = "date()"
            injury  = "int(0, 9999)"
            death   = "int(0, 9999)"
            loss    = "int(0, 100000000)">

            <vrn xd:script="occurs 1..*">
                  string(7)
            </vrn>
      </rec>
   </root>
</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
       xd:name = "accidents"
       xd:root = "root">

   <Accidents xd:script = " create from('/root')">
      <Accident xd:script = "occurs 0..*;
            create from('rec')">
            <vrn>string(5)</vrn>
            <date>date()</date>
            <injury>int(0, 9999)</injury>
            <death>int(0, 9999)</death>
            <loss>
                  int(0, 100000000)
            </loss>
            <vrn xd:script="occurs 1..*">
                  string(7)
            </vrn>
      </Accident>
   </Accidents>

   <root xd:script = "finally
         returnElement(xcreate('Nehody'))">
      <rec xd:script = "occurs 0..*">
            <vrn>string(5)</vrn>
            <date>date()</date>
            <injury>int(0, 9999)</injury>
            <death>int(0, 9999)</death>
            <loss> int(0, 100000000)</loss>
            <vrn xd:script="occurs 1..*">
                  string(7)
            </vrn>
      </rec>
   </root>
</xd:def>
```

In the X-definition header, it is sufficient to specify the root element in the xd:root attribute. As a context, a parsed XML document, ie a root element, is automatically used, so the create from ('/root') can be skipped because the root element does not matter in the root element name.

In the X-definition, the xcreate method was used to construct the Accident element. This will construct the element whose name is specified as its parameter, according to the model of this element in the X-definition.

In addition, the returnElement method was used to ensure that the call to the xparse method on the XDDocument (see Chapter 7.1) does not return the DOM for the parsed root element, but the DOM for the generated Accident element.

The described principle can be tested online here:

- http://xdef.syntea.cz/tutorial/en/example/C602.html

- http://xdef.syntea.cz/tutorial/en/example/C603.html

- ➡ příklad, který realizuje konstrukci v rámci validace: Kapitola 12

6.7 Example of XML transformation into HTML

➡️ 4 Description of structure of XML document by X-definition

➡️ 6 Construction Mode of X-definition

➡️ 6.1 Create Section in X-script

V této kapitole bude ukázán způsob, jak mohou být X-definice využity pro transformaci XML souboru z ukázkového příkladu se seznamem nehod z kapitoly 2.4 pro něž byla v kapitole 6.1 sestavena X-definice do struktury jiného XML dokumentu. Předmětným cílovým dokumentem bude HTML dokument s následující strukturou:

This chapter will be deomonstrated how the X-definitions can be used to transform an XML file from an example with an accident list from Chapter 2.4 for which an X-definition has been built into the structure of another XML document in Chapter 6.1. The target document will be the HTML document with the following structure:

```html
<html>
    <head>
        <title>Accidents list</title>
    </head>

    <body>
        <table border="1">
            <tr>
                <th>ID</th> <th>Date</th> <th>Injured [person]</th> <th>Death [person]</th>
                <th>Loss [thousends]</th> <th>Prticipants [VRN]</th>
            </tr>

            <tr>
                <td>00123</td> <td>17.05.2011</td> <td>3</td> <td>0</td> <td>600</td>
                <td>1A23456, 1B23456</td>
            </tr>

            <tr>
                <td>07045</td> <td>30.11.2012</td> <td>5</td> <td>1</td> <td>1300</td>
                <td>2A34567, 2B34567, 2C34567</td>
            </tr>
        </table>
    </body>
</html>
```

In a Web browser, the bulle displayed as the following table:

ID	Datum	Zraněno [osoby]	Usmrceno [osoby]	Škoda [tis. Kč]	Účastníci nehody [RZ]
00123	17.05.2011	3	0	600	1A23456, 1B23456
07045	30.11.2012	5	1	1300	2A34567, 2B34567, 2C34567

Now, for the target HTML (XML) document, an appropriate X-definition will be constructed and then an X-Script section will be benerated to describe the attributea and text node values for the resulting document and how to obtain the relevant values.

➡️ 6.7.1 X-definition of HTML document

➡️ 6.7.2 Create section used to construct a HTML document

➡️ 7 Using of X-definitions in Java code

6.7.1 X-definition of HTML document

➡️ 6.7 Example of XML transformation into HTML

In Chapter 6.7, the desired target HTML document structure was defined on the sample example, resulting in the transformation of an XML document with a list of accidents. The X-definition for the required XML document will be as follows:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" xd:name = "seznamNehodHtml" xd:root = "html">
    <html>
        <head>
            <title> String constant generated here </title>
        </head>
        <body>
            <table border=" String constant generated here ">
                <tr>
                    <th> generated String constant </th> <th> generated String constant </th>
                    <th> generated String constant </th> <th> generated String constant </th>
                    <th> generated String constant </th> <th> generated String constant </th>
                </tr>
                <tr xd:script="occurs 0..*">
                    <td>string(5); value taken from the input XML document </td>
                    <td>xdatetime('dd.MM.yyyy'); value taken from the input XML document </td>
                    <td>int(0, 9999); value taken from the input XML document </td>
                    <td>int(0, 9999); value taken from the input XML document </td>
                    <td>int(0, 100000000); value taken from the input XML document </td>
                    <td>string; value taken from the input XML document </td>
                </tr>
            <table>
        </body>
    </html>
</xd:def>
```

As you can see, the target HTML document will be largely made up of constant entities. The only variable entity is the second tr element in the table, which can be repeated in the resulting element (or may not be present at all).

Values that are constant may not be validated using data type checking methods. In this case, the correct X-definition may be corrected by the programmer.

For variables, the situation is different:

- If it is guaranteed that the input XML document to be transformed into HTML is valid, then in this X-definition it is no longer necessary to validate the data types using validation methods

  o the XML input document for the assurance of the assumption is first validated by the X-definition defined in the chapter 6.1

- Otherwise, it is possible to check the data types even in the validation mode as in this case.

Instead of the constant or variable values of the attributes and text nodes to be generated, temporary text labels have been used in the X-definition. In the next chapter, the above-mentioned X-definition will be added the create sections that will ensure the generation of the resulting HTML document.

➡️ 6.1 Create Section in X-script

➡️ 7 Using of X-definitions in Java code

6.7.2 Create section used to construct a HTML document

➡️ 6.7 Example of XML transformation into HTML

In Chapter 6.7.1, an X-definition was designed to describe the HTML document. The X-definition describes the structure of the resulting document and the data types of the selected text nodes (attributes would be similar). In order for the resulting document to contain some specific data (and not just empty elements and attributes) in its attributes and text nodes, its generation must be defined using X-Script in its create sections. The following example is performed in case the transformation is performed from a varnable a) XML document, ie proposed by means of elemets and attributes, and in the case where the transformation is performed from an XML document of variant b), that is, designed exclusively by means of elements. The target HTML document will be the same for both variants:

a) variant with elements and attributes

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
  xd:name = "dislapyAccidenta"
  xd:root = "html">

  <html>
    <head>
      <title>create 'Accidents list'</title>
    </head>
    <body>
      <table border="create '1'">
        <tr>
          <th>create 'ID'</th>
          <th>create 'Date'</th>
          <th>create 'Injured [persons]'</th>
          <th>create 'Death [persons]'</th>
          <th>create 'Loss [thousends]'</th>
          <th>create 'Paricipants [VRN]'</th>
        </tr>

        /* The element tr is "wrapped" with an
           Element xd:sequence by which they will be
           iteratively create tr elements including
           their sub-elements. */
        <xd:sequence xd:script="occurs 0..*;
          create from('//Accidents/Accident')">
          <tr>
            <td> string(5);
                 create from('@id')</td>
            <td> xdatetime('dd.MM.yyyy');
                 create from('@datum')</td>
            <td> int(0, 9999);
                 create from('@injury')</td>
            <td> int(0, 9999);
                 create from('@death')</td>
            <td> int(0, 100000000);
                 create from('@loss')</td>
            <td>
            /* Using the xd: sequence element
               iteratively pass all elements rz
               For the currently processed element
               Accident. */
            <xd:sequence xd:script="occurs 0..*;
                         create from('./vrn')">
              <xd:text>
                string; create from('./text()');
              </xd:text>
              <span>create ', '</span>
            </xd:sequence>
            </td>
          </tr>
        </xd:sequence>
      </table>
    </body>
  </html>
</xd:def>
```

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
  xd:name = "displayAccidents"
  xd:root = "html">

  <html>
    <head>
      <title>create 'Accidents list'</title>
    </head>
    <body>
      <table border="create '1'">
        <tr>
          <th>create 'ID'</th>
          <th>create 'Date'</th>
          <th>create 'Injured [persons]'</th>
          <th>create 'Death [persons]'</th>
          <th>create 'Loss [thousends]'</th>
          <th>create 'Paricipants [VRN]'</th>
        </tr>

        /* The element tr is "wrapped" with an
           Element xd:sequence by which they will be
           iteratively create tr elements including
           their sub-elements. */
        <xd:sequence xd:script="occurs 0..*;
          create from('//Accidents/Accident')">
          <tr>
            <td>string(5);
                create from('./id/text()')</td>
            <td> xdatetime('dd.MM.yyyy');
                 create from('./datum/text()')</td>
            <td> int(0, 9999); create
                 from('./injury/text()')</td>
            <td> int(0, 9999); create
                 from('./death/text()')</td>
            <td> int(0, 100000000);
                 create from('./loss/text()')</td>
            <td>
            /* Using the xd: sequence element
               iteratively pass all elements rz
               For the currently processed element
               Accident. */
            <xd:sequence xd:script="occurs 0..*;
                         create from('./vrn')">
              <xd:text>
                string; create from('./text()');
              </xd:text>
              <span>create ', '</span>
            </xd:sequence>
            </td>
          </tr>
        </xd:sequence>
      </table>
    </body>
  </html>
</xd:def>
```

In the example above, the construction mode was used to create the X-Script section:

- create 'co*onstant*' – is used to construct attribute values and text nodes, the subject value will be a string specified as a constant;

- create from('*xpath*') – is used to create the appropriate XML document fragment based on the context that is obtained after the xpath expression invoked in the method from (String). XPath is executed on the current context, which must be set before executing the X-definition construction process

  o from('//Accidents/Accident') – returns the context;

  o from('./vrn') – returns the context specified by the relative path from the currently processed element to the vrn element;

  o from('@id') – returns the context specified by the relative path from the currently processed element to the vrn element

- `text()` – returns the value of the text node for the currently processed element.

The create section specified in the element xd:sequence passes the current context (eg an XML document that is being transformed) iteratively and sequentially returns the context (ie a specific element including its attributes and subelements) corresponding to the defined XPath to which the X-definition given as the content of the element xd:sequence. For the example above, this means that xd:sequence passes through all of the vrn elements of the transformed XML document sequentially, lists the value of their text node, and enters the comma symbol (,). An alternative X definition to the above is presented in Chapter 6.4.

Note: If the create section is listed for elements, text nodes and attributes and the X-definition is processed in the validation mode, then all create sections are ignored

7 Using of X-definitions in Java code

➡ 4 Description of structure of XML document by X-definition

➡ 6 Construction Mode of X-definition

The processor of X-definition runs from the Java program for both validation and construction mode. The following examples show how to create a Java class to run a validation or construction based on a pre-assembled X definition.The following classes will be required in the Java class you created available in libraries provided with X definitions:

```java
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDDocument;
import org.xdef.XDPool;
```

Some parameters for processing X-definitions (eg language mutation, default code page for files, language, report messages, etc.) can be set before running the X definition processes (see chapter 11.2.3) and affect with System Properties (see chapter 10). E.g. if the System property "xdef.warnings" is set to true, the warning messages are reported. If the parameters are not set, the default setting is used.

➡ 7.1 Running the validation mode

➡ 7.2 Start the XML document construction

7.1 Running the validation mode

➡ 4 Description of structure of XML document by X-definition

First, in the Java program compile the selected set of X-definitions (in our example it is in the file Garage.xdef):

```java
/* Compile the XDPool from the X-definition – here it contains just one X-definition.
 * However, it may be more X=definitions. */
XDPool xpool = XDFactory.compileXD (null, "/path/to/Garage.xdef");

/* Create the instance of XDDocument from the X-definition "garage" from the XDPool. */
XDDocument xdoc = xpool.createXDDocument("garage");
```

The X-definition (as a document) is loaded from the xdef file and compiled to XDPool object. Then it is created the object XDDocument used to run a process (eg. validation). The XDPool object is reentrant, so you can use it to create mode XDDocument objects. Note the XDPool containing compiled X-definition is fully reentrant (ie it all values are constants (can not be changed), So you can use this object more times (and even it can be stored to a static variable).

➡ see 7.6 External variables

The errors detected during the validation process can be written to the standard output, ie System.out, or to a special stream called ReportWriter that allows you to report errors eg to an external file with reports. Refer to Chapter 10 for information. To use the default Reporter, you can use the following code:

```java
/*The reporter where are store the report messages generated durnig process. */
ArrayReporter reporter = new ArrayReporter();
```

➡ see 11.1 Generate XML file with errors

Now you can start the process ov validation of XML document:

```java
/* Parsing of the source XML data will be controlled by xdoc object (ie. Ccording to trhe X-definition 'garage')
 * All recognized errors will be stored to the reporter. */
xdoc.xparse("/path/to/garage.xml", reporter);
```

Now check for errors. If so, you can print them. Otherwise, the validated XML document corresponds to the X-definition:

```
/* The methoda errorWarnings() returns true if and only if some errors were generated during process. */
if (reporter.errorWarnings()) {
        System.err.println("Errors found in the input date.");
        PrintStream ps = new PrintStream("/path/to/errors.txt");
        reporter.printReports(ps);
        ps.close();
} else {
        /* Store parsed XML element to the file 'xmlResult.xml'. */
        KXmlUtils.writeXml("/path/to/xmlResult.xml", xdoc.getElement());
        System.out.println("OK");
}
```

The getElement() method invoked on the XDDocument object returns the root element of the resulting XML document. The writeXml method of the KXmlUtils class writes the XML document specified by the argument into the file. Note that when you validate an XML input document, all create sections of X-Script are skipped.

➡ details and a complete overview of all the features that X-definitions are listed in the chapter 14

7.2   Start the XML document construction

➡ prepared X-definition for construction mode: Chapter 6

First, the selected X-definition is loaded into the Java program (it is assumed that the generated X-definition is stored in the file displayAccidents.xdef):

```
/* Compile the file to XDPool. */
XDPool xpool = XDFactory. compileXD(null, "/path/to/displayAccidents.xdef");

/* Create the instance of XDDocument created from the X-definition 'displayAccidents'. */
XDDocument xdoc = xpool.createXDDocument("displayAccidents");
```

The XDPool containing compiled X-definition is fully reentrant (ie it all values are constants (can not be changed), So you can use this object more times (and even it can be stored to a static variable). More in Chapter 7.6.

The errors detected during a validation or construction process can be written to a standard output, ie System.out, or to a special file For more information, see Chapter 10. Now create the repoter object:

```
/* Create ArrayReporter for writing errors and messages. */
ArrayReporter reporter = new ArrayReporter();
```

See also 11.1 Generate XML file with errors

Now construct the XML document:

```
/* Before the process of constructions set to the XDDocument the context data used for the construction.
 * As the context will be used the XML document stored in the file accidents.xml. */
xdoc.setXDContext("/path/to/seznamNehod.xml");

/* The result will be constructed according to the model "html". The error messages will be stored to
   'reporter'. */
xdoc.xcreate("html", reporter);
```

Check if errors were reported. If yes store them to the file 'errorOutput.txt'. If not, then store the constructed element to the file 'xmlResult.xml':

```
/* Check if errors or warnings were reported */
if (reporter.errorWarnings()) {
        System.err.println("Errors reported.");
        /* Store error reports to the file */
        PrintStream ps = new PrintStream("/path/to/errorOutput.txt");
        reporter.printReports(ps);
        ps.close();
} else {
        /* Store created XML document to the file 'xmlResult.xml'. */
        KXmlUtils.writeXml("/path/to/xmlResult.xml", xdoc.getElement());
        System.out.println("OK");
}
```

The getElement() method invoked on the XDDocument object returns the root element of the constructed XML document. The writeXml method of the KXmlUtils class writes the XML document into the file xmlResult.xml.

details and a complete overview of all the features that X-definitions are listed in the chapter 14

➡️

7.3 Alternate creation of XDPool

➡️ 7.1 Running the validation mode

➡️ 7.2 Start the XML document construction

If you need to create XDPool from different sources - such as files stored on a local disk, files stored in a database or X-definition written in the String variable or even from an InputStream, you can use the org.xdef.XDBuilder class in which you can use the setSource method incrementally add different X-definition source data. The XDPool object is then created from the XDBuilder object by the compileXD() method as shown in the following example:

```
/* Create instance of XDBuilder. */
XDBuilder builder = XDFactory.getXDBuilder(prpos); // props may be null or a Properties object

/* Create reporter for error messages about compiling of X-definitions. */
ArrayReporter reporter = new ArrayReporter();

/* Set reporter to bulder. */
builder.setReporter(reporter);

/* Set sources of X-definitions to builder. */
builder.setSource("/path/to/vozovyPark.xdef");
builder.setSource(new URL("http://path.to/vozovyPark.xdef"));
builder.setSource("<xd:def xmlns:xd='http://www.xdef.org/xdef/3.2' xd:name='vozovyPark' xd:root='Vozidlo'>
        <Vehicle typ='SUV' rz='1A23456' purchase='01.02.2011' výrobce='Škoda' model='Yeti' /> </xd:def>");

/* Create XDPool; and the next procedure is the same as in the chapters 7.1 and 7.2. */
XDPool xpool = builder.compileXD ();

/* Throw an exception if th reporter contains errors or warnings.*/
reporter.checkAndThrowErrorWarnings();
```

7.4 Build XDPool with classes containing external metods

➡️ 4.8 External (Java) Methods

➡️ 7.2 Start the XML document construction

➡️ 7.3 Alternate creation of XDPool

The Java classes in which external methods used in X-definitions are located, which are by default searched by the default class loader. In the case you need to use another class loader to retrieve and load thise Java classes, this loader must be connected to the X-Definition builder. The class loader can be connected to the XDBuilder by the method setClassLoader::

```
...

/* Create instance of XDBuilder. */
XDBuilder builder = XDFactory.getXDBuilder(null);

/* Set your class loared*/
builder.setClassLoader(new MyClassLoader());

/* Set source input data. */
builder.setSource("/path/to/vozovyPark.xdef ");
builder.setSource(new URL("http://path.to/vozovyPark.xdef"));
...
```

You can also add classes with external methods to the compiler as the array of classes and normally compile X-definitions:

```
/* Array with classes containing the external methods. */
Class[] ext = new Class[] {examples.tutorial.VozovyPark.class, tasks.SimpleAccident.class};
/* The compiler will search the external methods in the array of classes. */
XDPool xpool = XDFactory.compileXD(null, "/path/to/vozovyPark.xdef", ext);
```

7.5   Get result of X-definition process

➡️ run the validation or the construction mode chapter 7.1 or 7.2

The metods xparse or xcreate return the processed element as the return value. So you do net need to use the method getElement():

```
Element e;
e = xdoc.xparse("/path/to/garage.xml", reporter);
e = xdoc.xcreate("html", reporter);
e = xdoc.getElement();
```

Výjimkou je použití metody returnElement v X-Scriptu (viz kapitola 6.6.2), která nastaví element, který vrátí metody xparse, xcreate i getElement objektu XDDocument.

7.6   External variables

➡️ external (java) methods chapter 4.8.

➡️ run the validation or the construction mode. Chapter 7.1 or 7.2

Similarly as was shown invoking of external methods in the Chapter 4.8 in the corresponding Java classes, the so called external variables that are declared in the X-definition can be used in the X definition, but their value is set from the Java code.

The declaration of an external variable in the X-definition must be introduced to the keyword 'external'. From the Java code, the value of the external variable can be set before invoking of the validation or construction process by the 'setVariable' method on the XDDocument object.

The following example will expand the example in Chapter 4.8, which called myPrint() external Java method at the onStartElement event. An integer variable 'version' value will be set from the Java program and the myPrint() method will print it to the standard output:

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage"
        xd:root = "Vehicles">

   <Vehicles>
      <Vehicle
         xd:script = "occurs 0..*;
            onStartElement myPrint();
            finally version = 1;"
         type  = "enum('SUV', 'MPV', 'personal',
            'truck', 'sport', 'other')"
         vrn   = "string(7)"
         purchase = "date()"
         manufacturer = "string()"
         model = "string()" />
   </Vehicles>

   <xd:declaration>
      /* External variable. */
      external int version;

      void myPrint() {
         outln("Vrsion is:" + version);
      }
   </xd:declaration>

</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage"
        xd:root = "Vehicles">

   <Vehicles>
      <Vehicle
         xd:script = "occurs 0..*;
            onStartElement myPrint();
            finally version = 1;"
         <type> enum('SUV', 'MPV', 'personal',
            'truck', 'sport', 'other')
         </type>
         <vrn> string(7) </vrn>
         <purchase> date() </purchase>
         <manufacturer> string() </manufacturer>
         <model> string() </model>
      </Vehicle>
   </Vehicles>

   <xd:declaration>
      /* External variable. */
      external int version;

      void myPrint() {
         outln("Vrsion is:" + version);
      }
   </xd:declaration>

</xd:def>
```

Set the value of an external variable to the xdoc object in the Java code before the validation or construction process begins. See the following snippet of code:

```
XDDocument xdoc = ...
/* Set value of variable 'version' to XDDocument. */
xdoc.setVariable("version", 5);
...
xdoc.xparse ...
```

On the other hand, the value of an external variable can be obtained after completion of the validation or construction process by the getVariable method:

```
xdoc.xparse ...
/* Get value of variable "version" from XDDocument, */
XDValue x = (XDValue) xdoc.getVariable("version");
/* Get integer value from XDValue, */
int ver = x.integerValue();
```

The getVariable method returns an object from the XDValue interface that needs to be changed (ie cast) to the appropriate data type corresponding to the given variable from the X-Definition. A list of all data types is in the chapter 14.2.

➡ 7.7 External instance methods

➡ 14 Appendix C – supplementary description of X-definitions

7.6.1   Connection to the reelational database

If the data type Service (which corresponds to a JDBC connection type) is required in the X-definition, the java.sql.Connection data type connection object must be encapsulated in the XDService object (this corresponds to the Service type in X-definitions) and paste it into an external service data type service variable in X definition:

```
Java.sql.Connection connection;
...
xdoc.setVariable("service", XDFactory.createSQLService(connection));
```

As can be seen from the following example, encapsulation can be accomplished by the method creatingSQLService from the XDFactory class.

The XDService object can also be obtained from the XDFactory class by entering the URL database, logins name and password using the createSQLService method, which has the similar parameters as the java.sql.DriverManager.getConnection (url, user, password) method:

```
XDService service = XDFactory.createSQLService(url, user, password);
xdoc.setVariable("service", service);
```

Declaration of the external variable 'service' in the X-definition:

```
...
<xd:declaration>
    ...
    external Service service;
    ...
</xd:declaration>
...
```

7.7   External instance methods

Several variants of implementation and use of external X-definition methods have been described in Chapter 4.8. In all cases these were static methods (methods with the keyword static). This chapter describes how to **use objects and their instance methods** that can be called from X-Definition using external (static) methods.

Let's have the Java class with the myPrint class method, the getInstanceName instance method, and the instanceName instance variable:

```java
public class Garage {

/** Instance field. */
private final String instanceName;

/**
 * Construktor.
 * @param name the name assigned to this instance of this class.
 */
public Garage(String name) {
        instanceName = name;
}

/**
 * Instance method.
 * @return string wirh the name from the instance of this class.
 */
public String getInstanceName() {
        return instanceName;
}

/**
 * External method used in the X-definition.
 * @param xnode the object representing the current processed node.
 * @param output text to be printed.
 */
public static void myPrint(XXNode xnode, String output) {
        // Get the user object containing the instance of this class.
        Object obj = xnode.getUserObject();
        if (obj != null) {
                String iName = ((Garage)obj).getInstanceName();
                System.out.printf("The name of instance: %s \n", iName);
        }
        String spz = xnode.getXXElement().getAttribute("rz");
        System.out.printf("Info=%s; Vehicle registration=%s", output, spz);
}
```

As you can see, compared to the version in Chapter 4.8.2, the myPrint method will be called from the X-definition in the same way. In the body of the method, the getUserObject() method is called on the XXNode object, which

returns any object (java.lang.Object) associated with the appropriate X definition (the association must be performed before the validation or construction of the XML document - see below). After casting it on *Garage*, you can call its instance method getInstanceName().

Any object can be associated with X-Definition by the setUserObject(Object) method through XDDocument (association is done after creation of XDDocument but before invoking the X-definition process (ie. xparse or xcreate method):

```
...

/* It will be used the X-definition 'garage'. */
XDDocument xdoc = xpool.createXDDocument("garage");

/* Assign the user object ot the XDDocument. */
xdoc.setUserObject(new Garage("Instance 1"));

...
```

When calling an external (class) myPrint method in the following X-definition, it invokes the instance method getInstanceName through which will be accessed the instanceName variable:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage"
        xd:root = "Vehicles">
   <xd:declaration>
      external method void examples.tutorial.Garage.myPrint(XXNode, String);
   </xd:declaration>

   <Vehicles>
      <Vehicle
         xd:script = "occurs 0..*; onStartElement myPrint('Started process of the element Vehicle. ')"
         type  = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
         vrn   = "string(7)"
         purchase = "date()"
         manufacturer = "string()"
         model = "string" />
   </Vehicles>
</xd:def>
```

It will print:

```
JThe name of instance: Instance 1
Info Started process of the element Vehicle; Vehicle registration=...
```

7.8   Validation of data with a database in a XML document

➡ 4.10 User Defined Methods for Checking Data Types

➡ 5.5 Container

➡ 7.6 External variables

The following example will expand the example of the X-definition for the list of traffic accidents in Chapter 4.1. For the registration tag element vrn, a validation method will be added to verify whether the registration tag in the validated document according to the X-definition list is in the XML document:

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents>
        <Accident xd:script = "occurs 0..*"
            id      = "string(5)"
            datum   = "date()"
            injury  = "int(0, 9999)"
            death   = "int(0, 9999)"
            loss    = "int(0, 100000000)">
        <vrn xd:script="occurs 1..*">
          checkVrn()
        </vrn>
        </Accident>
    </Accidents>

    <xd:declaration>
        external Element vehicles;
        Container vehicle;

        /* Check if VRN is in the database */
        boolean checkVrn() {
            /* Value of VRN. */
            String vrn = getText();
            vozidlo = xpath(
             'Vehicle[@rz="'+vrn+'"]', vehicles);
            if (vozidlo.getLength() == 0) {
                outln('VRN' + vrn +
                    ' is not registred.');
                Return false
            }
            /* VRN was found in the database. */
            return true;
        }
    </xd:declaration>
</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

    <Accidents>
        <Accident xd:script = "occurs 0..*">
          <vrn>string(5)</vrn>
          <date>date()</date>
          <injury>int(0, 9999)</injury>
          <death>int(0, 9999)</death>
          <loss>int(0, 100000000)</loss>
          <vrn xd:script="occurs 1..*">
             checkVrn()
          </vrn>
        </Accident>
    </Accidents>

    <xd:declaration>
        external Element vehicles;
        Container vehicle;

        /* Check if VRN is in the database */
        boolean checkVrn() {
            /* Value of VRN. */
            String vrn = getText();
            vozidlo = xpath(
             'Vehicle[@rz="'+vrn+'"]', vehicles);
            if (vozidlo.getLength() == 0) {
                outln('VRN' + vrn +
                    ' is not registred.');
                Return false
            }
            /* VRN was found in the database. */
            return true;
        }
    </xd:declaration>
</xd:def
```

In the example above, the checkVrn method was defined to check whether the currently processed registration tag (from the vrn attribute value or the vrn element text value) occurs in the XML element 'vehicles'. You can also use the method error('...') which writes the the text in its parameter into the error report file and returns false:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">
    ...
    <xd:declaration>
        external Element vehicles;
        ...
        boolean checkVrn() {
            String vrn = getText();
            Container vehicle = xpath('Vehicle[@rz="' + rz + '"]', vehicles);
            if (vehicle.getLength() == 0) { // not found
                return error('VRN' + vrn + ' is not registred.');
            }
            return true;
        }
    </xd:declaration>
</xd:def>
```

The check is performed using xpath, which searches an element in the XML document with the list of vehicles, the vehicle whose attribute, element vrn has a value corresponding to the currently processed vehicle registration from the list of accidents. The xpath (XPath, Element) method accepts two parameters, a path in an XML document specified by xpath expression and the default element from which to search by the specified xpath.

If a Vehicle tem exists in the XML document, the result of XPath expression is inserted into the Context data structure (see Chapter 5.5). If the desired element is not found, the Context data type variable is left empty, ie the number of items obtained by the getLength () method is zero.

The XML document is accessible in X-definition the external variable vehicles. The specified variable must be set from the Java class to run validation of the appropriate XML document with the list of crashes before the validation starts:

```
/* XML parser reads the XML document with the list (a database) of vehicles and its vrn. */
org.w3c.dom.Element el = KXmlUtils.parseXml("/path/to/garage.xml").getDocumentElement();

/* The root element is set to the external variable vehicles in the XDDocument (X-definition). */
xdoc.setVariable("vozidla", el);
```

➡️ 8 Structuring of X-definitions

➡️ 12 Appendix A – complete example

➡️ 14 Appendix C – supplementary description of X-definitions

7.9   XDPool in a binary data file

➡️ 7.1 Running the validation mode

➡️ 7.2 Start the XML document construction

The X-definition technology allows you to save the compiled XDPool object created fro source X-definitions into a binary file (or generate a Java class) and its further use when processing XML data. This property is especially important when X-definitions source data are extensive or repeatedly used, and the continued translation of X-definitions into an XDPool object would be time consuming. In such cases, it is preferable to use the result of X-definition, which is stored in a binary file whose read-in is faster than the compilation of X-definitions.

The generated XDPool is saved to the binary file by the writeXDPool(OutputStream) method:

```
/* Compile X-definition and create the XDPool object. */
XDPool xpool = XDFactory.compileXD(null, "/path/to/vozovyPark.xdef");

/* Save the XDPool object to the file. */
OutputStream out = new FileOutputStream("/path/to/vozovyPark.xp");
xpool.writeXDPool(out);
out.close();
```

For further use, it is enough to create XDPool from the given binary file. The time to create an instance of the XDPool object will be much shorter by doing this because the X-definition source code will not be re-compiled. The readXDPool(InputStream) method is used to read:

```
/* Read XDPool from the file. */
InputStream in = new FileInputStream("/path/to/vozovyPark.xp");
XDPool xpool = XDFactory.readXDPool(in);
in.close();

/* Now you can work with the xpool object in a common way, ie you can create XDDocument, etc. */
...
```

The version and build number of X-definition library must always be the same when creating a binary file for XDPool. With each new version and build number of X-definition library, it is therefore necessary to recreate the binary file from the source X-definitions data.

7.10 Continuous XML document writing

➡️ 7.1 Running the validation mode

➡️ 7.2 Start the XML document construction

When processing an XML document, the processed DOM objects are constructed and stored by default. Only after completing the validation or construction process the entire XML document is returned as a result. For very large files, the default behavior needs to be modified (due to a lack of memory).

One possible modification is the use of the forget keyword in the X-script of element models to ensure that the corresponding DOM object is released from memory after processing of the model- see Chapter 4.6. In order for a processed element whose model has the keyword forget to be written to the output before its release from memory, it is necessary to ensure that the X-definition processor writes DOM objects continuously to an external file.

The java.io.OutputStream (in Java) object is reserved for this entry. XmlOutStream (in X-Script), which corresponds to the interface org.xdef.XDXmlOutStream. [Doc1].

7.10.1 Automatic write to OutputStream

By using the OutputStream, it is possible to ensure automatic recording of the processed DOM objects. The generated OutputStream (e.g. file) is passed by the void setStreamWriter(OutputStream out, String encoding, boolean writeDocumentHeader) to the XDDocument object that will automatically continuously write all processed objects to the specified output stream. In addition, if the keyword forget is in the element models, the processed and written elements are released from memory after being procewssed:

```
...

/* Create XDDocument from XDPool. */
XDDocument xdoc = xpool.createXDDocument("garage");

/* Prepare output stream. */
OutputStream out = new FileOutputStream(new File("AccidentsOut.xml"));

/* Assign the output stream to the XDDocument. */
xdoc.setStreamWriter(out, "UTF-8", true);

/* Start X-definition process xdoc.xparse(...) or xdoc.xcreate(...). */

/* Close the output stream. */
out.close();
...
```

7.10.2 Incremental writing using XmlOutStream

➡ 7.6 External variables

➡ 14.7 Methods implemented in X-Script

Output stream can also be written "manually" from X-script. In the X-script, an XmlOutStream variable vcan be created for this purpose:

a) either directly in the X-script:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

   <Accidents xd:script = "finally closeStream()">
      <Accident xd:script = "occurs 0..*; finally writeToStream()"
         id      = "string(5)"
         datum   = "date()"
         injury  = "int(0, 9999)"
         death   = "int(0, 9999)"
         loss    = "int(0, 100000000)">

         <vrn xd:script="occurs 1..*">string(7)</vrn>
      </Accident>
   </Accidents>

   <xd:declaration>
      /* Create the output stream. */
      XmlOutStream out = new XmlOutStream("AccidentesOut.xml");
      boolean first = true;
```

```
        ...

        /* The method called in the finally action, ie. after the element is processd. */
        void writeToStream() {
            if (first) {
                /* Write the root element. */
                out.writeElementStart(new Element("OutputResult"));
                first = false;
            }

            /* Write the subelement. You can also add attributes and text nodes. */
            Element el = new Element("Nehoda");
            el.setAttr("id", toString(@id));
            ...
            out.writeElement(el);
        }

        /* The method invoked in the action finally sekci after the whole XML document was processed. */
        void closeStream() {
            /* Write the end tag of the root element. */
            out.writeElementEnd();

            /* Close the output stream. */
            out.closeStream();
        }
    </xd:declaration>
</xd:def>
```

The output XML file can look like this:

```
...
<OutputResult>
        <Accident id="00123" ... />
        <Accident id="00234" ... />
        ...
</OutputResult>
```

b)   or add as an external variable:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name  = "accidents"
        xd:root  = "Accidents">

    ...

    <xd:declaration>
        /* The output stream is in the external variable. */
        external XmlOutStream out;

        ...

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

```
...

/* Prepare the output stream. */
XDXmlOutStream out = XDFactory.createXDXmlOutStream("AccidentsOut.xml", "UTF-8", true);[1]

/* Set the external variable to the XDDocument. */
xdoc.setVariable("out", out);

...
```

---

[1] The createXDXmlOutStream (String file, String encoding, boolean writeDocumentHeader) method has parameters: file containing a path to the generated stream; encoding indicates a code table where characters are stored in the stream (the default is UTF-8). The writeDocumentHeader parameter, if true, ensures that the XML header is written to the output stream.

8 Structuring of X-definitions

➡ 4 Description of structure of XML document by X-definition

Extensive X-definitions may become less well-arranged, and therefore need to be internally structured. For this purpose, X-definition technology have options to extract part of the element model description and move it to another location, take the element models from another X-definition, or replace parts of the X-definition macros.

To ensure these modifications and constructions, the X-script links are used in the X-definitions, which generally contain references to the parts of X-definitions listed elsewhere (in another part of the same file or in another file). The link then takes the values from the place to which it refers.

8.1 Reference to another element model in X-definition

➡ 4.7 Sample of Complete X-definition

The first example will be to move a part of the element model to another location within the X-definition. Specifically, it will move the model of the Vehicle element from examples to construct an XML document. For this purpose, the keyword ref can be used, referring to the referenced model for the relevant element model. The X-definition in Chapter 4.7 will have the following form, after maintaining the same functionality:

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name  = "garage"
        xd:root  = "Vehicles">

    <Vehicles>
        <Vehicle xd:script = "occurs 0..*; ref Vuz"/>
    </Vehicles>

    <Vehicle type= "enum('SUV', 'MPV', 'personal',
                   'truck', 'sport', 'other')"
          vrn    = "string(7)"
          purchase = "date()"
          manufacturer = "string()"
          model  = "string" />
</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name  = "garage"
        xd:root  = "Vehicles">

    <Vehicles>
        <Vehicle xd:script = "occurs 0..*;
                ref Vuz" />
    </Vehicles>

    <Vehicle>
        <type>enum('SUV', 'MPV', 'personal',
              'truck', 'sport', 'other')
        </type>
        <vrn>string(7)</vrn>
        <purchase> date() </purchase>
        <manufacturer>string()</manufacturer>
        <model>string</model>
    </Vehicle>
</xd:def>
```

From the example above, it is clear that the reference helps keep the X-definition entry readable by allowing the structural part to be separated (the XML tree defined by the Vehicles element) from its full content (element Vehicle).

The above example evaluates the X-definition in such a way that the Vehicle element will inherit all of its sub-elements as well as all its attributes, including all definitions of data types, actions, etc. If the Vehicle element has the number of occurrences, this number will be covered by the number given for the element Vehicle. For this reason, the number of occurrences is not specified in the Vehicle model.

References between element models also allow reuse of one element model and to specify extension of the referenced model. If it is now considered a case where it will be necessary to add the element Bus, whose model will be identical to the model for the Vehicle element, respectively, in the above-mentioned X-definition. for the Vehicle, except that it is necessary to specify its capacity for the number of passengers for the Bus element, the reference can be used to inherit all attributes, model elements for the Vehicle and add new features:

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name = "garage"
      xd:root = "Vehicles">

  <Vehicles>
   <Vehicle xd:script = "occurs 0..*; ref Vehicle"/>
   <Bus xd:script = "occurs 0..*; ref Vehicle"
        passenger="int(0,200)"/>
  </Vehicles>

  <Vehicle type  = "enum('SUV', 'MPV', 'personal',
                    'truck', 'sport', 'other')"
           vrn   = "string(7)"
           purchase = "date()"
           manufacturer = "string()"
           model  = "string()" />
</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name  = "garage"
      xd:root  = "Vehicles">

    <Vehicles>
        <Vehicle xd:script = "occurs 0..*;
                ref Vehicle" />
        <Bus xd:script = "occurs 0..*; ref Vehicle">
              <passenger>int(0,200)</passenger>
        </Bus>
    </Vehicles>

    <Vehicle>
        <type>enum('SUV', 'MPV', 'personal',
            'truck', 'sport', 'other')
        </type>
        <vrn>string(7)</vrn>
        <purchase> date() </purchase>
        <manufacturer> string() </manufacturer>
        <model>string()</model>
    </Vehicle>
</xd:def>
```

Without using a reference mechanism between element models, the same model that was already created for the Vehicle element would have to be described for the Bus element model, with the addition of only the attribute or attribute. elelement traveling.

In the same way, it is possible to refer to groups of elements that can act as a model. The following example demostrates how a reference is made to the model of the xd:mixed group of the Accident element sub-element:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name  = "accidents"
      xd:root  = "Accidents">

    <Accidents xd:script = "create 1">
        <xd:mixed xd:script="ref Accident" />
    </Accidents>

    <xd:mixed xd:name="Accident" xd:script="occurs 1..2; create 2">
        <Accident_A xd:script = "occurs 1..1; create 1" ...>
            ...
        </Accident_A>

        <Accident_B xd:script = "occurs 1..1; create 1" ...>
            ...
        </Accident_B>
    </xd:mixed>
</xd:def>
```

In order for the group to act as a model and be referenced, it must specify the xd:name attribute specifying its name.

In the example above, the entry is equivalent to the entry:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

   <Accidents xd:script = "create 1">
      <xd:mixed xd:script="occurs 1..2; create 2">
         <Accident_A xd:script = "occurs 1..1; create 1" ...>
            ...
         </Accident_A>

         <Accident_B xd:script = "occurs 1..1; create 1" ...>
            ...
         </Accident_B>
      </xd:mixed>
   </Accidents>
</xd:def>
```

Even for element groups, the element (group of element models) to which it refers must be the direct descendant of element xd: def.

➡ 8.2 Collection of X-Definitions

➡ 8.4 Macros

8.1.1 Alternative references to groups

The example from the previous chapter can also be written so that the xd:mixed element will act as a sub-element of another element to which the reference inserting its descendants will be directed by using the element xd:includeChildNodes and the ref attribute, ie xd:mixed:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "accidents"
        xd:root = "Accidents">

   <Accidents xd:script = "create 1">
      <xd:includeChildNodes ref="ref NehodyA_B" />
   </Accidents>

   <AccidentsA_B>
      <xd:mixed xd:script="occurs 1..2; create 2">
         <Accident_A xd:script = "occurs 1..1; create 1" ...>
            ...
         </Accident_A>

         <Accident_B xd:script = "occurs 1..1; create 1" ...>
            ...
         </Accident_B>
      </xd:mixed>
   </AccidentsA_B>
</xd:def>
```

8.2 Collection of X-Definitions

➡ 4.7 Sample of Complete X-definition

➡ 8.1 Reference to another element model in X-definition

In the case that a task is more extensive and more separate X-definitions have to be created for this purpose, it can be placed in separate xdef files or embedded into a single xdef file. The collection of X-definitions is determined by the xd:collection element. As a sample example, X-definitions are recorded using elements into a single file:

```
<xd:collection xmlns:xd = "http://www.xdef.org/xdef/3.2">
    <!-- First X-definition, garage. -->
    <xd:def  xd:name = "garage"
             xd:root = "Vehicles">

        <Vehicles>
            <Vehicle xd:script = "occurs 0..*"
                type     = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
                vrn      = "string(7)"
                purchase = "date()"
                manufacturer = "string()"
                model    = "string" />
        </Vehicles>
    </xd:def>

    <!-- Second X-definition, garageElem. -->
    <xd:def  xd:name = "garageElem"
             xd:root = "Vehicles">

        <Vehicles>
            <Vehicle xd:script = "occurs 0..*">
                <type>enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')</type>
                <vrn>string(7)</vrn>
                <purchase>date()</purchase>
                <manufacturer>string()</manufacturer>
                <model>string</model>
            </Vehicle>
        </Vehicles>
    </xd:def>
</xd:collection>
```

As can be seen above, the X-definition collection contains two X-definitions that differ in structure. Therefore, the whole collection can be modified so that the definition of the content of the X-definitions, defined attributes, respectively. elements and their data types, for example, to move X-Definition to the X-Definition of the VehiclePark and, with reference, the Vehicle element will refer to its content in the X-Definition of the Porsche. The reference to the Element element element found in the XDef X-definition is written using ref as **ref XDef#Element**.

The modified example will look like this:

```
<xd:collection xmlns:xd = "http://www.xdef.org/xdef/3.2"

    <xd:def  xd:name = "garage"
             xd:root = "Vehicles">

        <Vehicles>
            <Vehicle xd:script = "occurs 0..*; ref Vuz" />
        </Vehicles>

        <Vehicle type = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
            vrn  = "string(7)"
            purchase = "date()"
            manufacturer = "string()"
            model= "string" />

        <VehicleElem>
            <type>enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')</type>
            <vrn>string(7)</vrn>
            <purchase>date()</purchase>
            <manufacturer>string()</manufacturer>
            <model>string</model>
        </VuzElem>
    </xd:def>


    <xd:def  xd:name = "garageElem" xd:root = "Vehicles">
        <Vehicles>
            <Vehicle xd:script = "occurs 0..*; ref garage#VehicleElem" />
        </Vehicles>
    </xd:def>

</xd:collection>
```

➡ 8.3 X-definitions in separate files

8.2.1 Scope (visibility) of global variables and methods

Variables and methods that are defined in the X-script of one X-definition as global are also visible from other X-definitions that are compiled together. From the X-script of an another X-definition, these variables and methods are accessed as if they were explicitly defined in the X-script of that X-definition.

For this reason, two different X-definitions with two identically named global variables or methods will report an error.

Dependencies between variables and methods from different X-definitions will attempt to resolve them automatically, and X-script commands will reorganize in sequential order so values of all desired variables are available at the level of each command. If this can not be done (typically when a cycle is detected between variables dependencies), an error is generated. After automatic reordering of commands, initialization of variable variables is always first performed. The reason is that the X-script of all X-definitions is merged into a block.

8.3 X-definitions in separate files

➡ 7 Using of X-definitions in Java code

➡ 8.2 Collection of X-Definitions

If both X-definitions in Chapter 8.2 where they are part of one collection, are placed in separate xdef source files, then these are reflected only in the way of creating the XXDPool in the Java program. If both of these X-definitions are located in xdef files as follows:

```xml
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2
        xd:name = "garage"
        xd:root = "Vehicles">

    <Vehicles>
        <Vehicle xd:script = "occurs 0..*; ref Vuz" />
    </Vehicles>

    <Vehicle typ      = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
        vrn  = "string(7)"
        purchase = "date()"
        manufacturer = "string()"
        model= "string" />

    <VehicleElem>
        <type>enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')</type>
        <vrn>string(7)</vrn>
        <purchase>date()</purchase>
        <manufacturer>string()</manufacturer>
        <model>string</model>
    </VuzElem>
</xd:def>
```

```xml
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garageElem"
        xd:root = "Vehicles">

    <Vehicles>
        <Vehicle xd:script = "occurs 0..*; ref garage#VehicleElem" />
    </Vehicles>
</xd:def>
```

the compileXD method will be used to create the XDPool object in the Java program, which accepts a field of paths to xdef source files with individual X-definitions:

```java
/* Create XDPool from all X-definitions. */
XDPool xpool = XDFactory.compileXD(null, "/path/to/garage.xdef", "/path/to/garageElem.xdef");
```

All X-definitions compiled to the XDPool are visible in all other X-definitions from this pool.

The other way to add another X-definition is to use the xd:include attribute in the X-definition header that allows you to join a specific X-definition source (specified in the xd:include attribute using the URL). In the xd:include attribute, you can specify multiple comma-separated URLs or file paths to add multiple X-definitions:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garageElem"
        xd:root = "Vehicles"
        xd:include = "path/to/garage.xdef">

        <Vehicles>
            <Vehicle xd:script = "occurs 0..*; ref garage#VehicleElem" />
        </Vehicles>
</xd:def>
```

```
/* Create XDPool – note only one source X-definition is specified in the parameter. */
XDPool xpool = XDFactory.compileXD(null, new String[] {"/path/to/vozovyParkElem.xdef"});
```

Example of compilation of more X-definitions:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garageElem"
        xd:root = "Vehicles"
        xd:include = "path/to/garage.xdef,
            http://www.myexamples.com/xdef/example/TestDef.xdef,
            file://path/to/Test2Def.xdef">
...
```

If an X-definition file is located in a local file system, the prefix "file: //" can be omitted, and for wildcard specification can be used (ie. "*" for any sequence of characters and "?" for one character). Wildcards are only applied to the appropriate directory. E.g. write path /to/*.xdef will select all files with xdef file name extension from the ./path/to directory.

➡ 8.4 Macros

➡ 8.5 Structure comparison

8.4   Macros

➡ 4 Description of structure of XML document by X-definition

Macros in X-definitions are used to write strings or parameterized strings that replace all calls to the appropriate macros before X-definitions are compiled. In a X-definition, any number of macros that can be distinguished by their name can be defined. Each macro is defined using the element xd:macro which is in the xd:declaration element with the required attribute name:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garageElem"
        xd:root = "Vehicles"
        xd:include = "path/to/garage.xdef"><xd:def>
  <xd:declaration>
    <xd:macro name="reqStr7">
        required string(7);
    </xd:macro>
  </xd:declaration>
...
</xd:def>
```

Note: to ensure the backward compatibility with the older versions of X-definition the macros may be declared also as the direct descendant of element xd:def:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2" ...>
  <xd:macro name="reqStr7">
    required string(7);
  </xd:macro>
...
</xd:def>
```

The example below defines a macro with the name `reqStr7`, when the call to the subject macro will be replaced with the string "`required string(7);`" which in the following example is used to define the data type of the attribute, respectively element vrn:

a) variant with elements and attributes

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
     xd:name  = "garage"
     xd:root  = "Vehicles">

  <Vehicles>
     <Vehicle xd:script="occurs *; ref Vehicle"/>
  </Vehicles>

  <Vehicle type = "enum('SUV', 'MPV', 'personal',
              'truck', 'sport', 'other')"
       vrn = "${reqStr7}"
       purchase = "date()"
       manufacturer = "string()"
       model = "string" />

  <xd:declaration scope = "local">
    <xd:macro name="reqStr7">
      required string(7);
    </xd:macro>
  <xd:declaration>

</xd:def>
```

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
     xd:name  = "garage"
     xd:root  = "Vehicles">

  <Vehicles>
     <Vehicle xd:script = "occurs; ref Vehicle" />
  </Vehicles>

  <Vehicle>
     <type>enum('SUV', 'MPV', 'personal',
          'truck', 'sport', 'other')
     </type>
     <vrn>${reqStr7}</vrn>
     <purchase>date()</purchase>
     <manufacturer>string()</manufacturer>
     <model>string</model>
  </Vehicle>

  <xd:declaration scope = "local">
    <xd:macro name="reqStr7">
      required string(7);
    </xd:macro>
  <xd:declaration>

</xd:def>
```

The macro is referenced by entering its name into compound parentheses "{" and "}", which are denoted by the "$" (dollar) symbol as shown in the example above. E.g. in variant a), the vrn = "${reqStr7}" attribute value will be replaced by vrn = "required string (7);".

Therefore, the use of macros can also be used to provide greater readability and easier maintenance of the X-definition code, but also for repeated and consistent declarations of variable types or other constant strings. You can also place a function call in a macro, such as calling a query with a SQL query into a relational database that returns ResultSet with source data for the construction of the elements of the Accident:

```
...
<Accident xd:script="0..*; create ${m_si}">
        ...
</Accident>

...
<xd:declaration>
  <xd:macro name="m_si">
    service.query("SELECT ...");
  </xd:macro>
<xd:declaration>
...
```

8.4.1   Macros with parameters

For strings that have a fixed structure up to the final number of substrings (parameters), parameterized macros can be used. Each parameterized macro has additionally attributes corresponding to the names of the overriding parameters in its element, and their value represents the default value of the relevant parameter. The parameterized macro is called with a comma-separated list of parameter names with their values as shown in the following example, which in addition to the previous example of the X-definition defines the default values of the individual attributes:

a) variant with elements and attributes

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage"
        xd:root = "Vehicles">

  <Vehicles>
      <Vehicle xd:script = "occurs *; ref Vuz"/>
  </Vehicles>

  <Vehicle type = "enum('SUV', 'MPV', 'personal',
               'truck', 'sport', 'other');
                 onFalse ${typErr(attr='type')}"
           vrn  = "${reqStr7}; onFalse
                   ${typErr(attr='vrn',msg='7 chars')}"
           purchase = "date()"
           manufacturer = "string()"
           model = "string" />

  <xd:declaration>
    <xd:macro name="reqStr7">
      required string(7);
    </xd:macro>

    <xd:macro name="reqStr7" atr="?" msg="">
      outln("Incorrect attribute value: " +
            #{atr} + " (" + "#{msg}" + ").");
    </xd:macro>
  </xd:declaration>

</xd:def>
```

b) variant only with elements

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage"
        xd:root = "Vehicles">

  <Vehicles>
      <Vehicle xd:script="occurs *; ref Vehicle" />
  </Vehicles>

  <Vehicle>
      <type>enum('SUV', 'MPV', 'personal',
              'truck', 'sport', 'other');
            onFalse ${typErr(attr='type')}
      </type>
      <vrn>${reqStr7}; onFalse ${typErr(attr='vrn',
              msg='7 chars')}</vrn>
      <purchase>date()</purchase>
      <manufacturer>string()</manufacturer>
      <model>string</model>
  </Vehicle>

  <xd:declaration>
    <xd:macro name="reqStr7">
      required string(7);
    </xd:macro>

    <xd:macro name="reqStr7" atr="?" msg="">
      outln("Incorrect attribute value: " +
            #{atr} + " (" + "#{msg}" + ").");
    </xd:macro>
  </xd:declaration>

</xd:def>
```

The parameterized macro in the example above is called typeErr and has two parameters, atr and msg. The default value of the atr parameter is "?" (question mark) and msg is a blank string. In the macro definition (in the xd:macro element), the "#" and brackets "{" and "}" are accessed to the individual parameters, with the name of the specific parameter.

If the value of any of the parameters is not specified when a macro is called, its default values are used. For example, calling the ${typErr (atr = 'type')} will be replaced with the string outln("Incorrect attribute value: " + "type" + " (" + "" + ")."); . If the parameter "atr" is not specified, then the macro definition will be replaced with a string containing only the symbol "?".

8.5   Structure comparison

➡ 4 Description of structure of XML document by X-definition

➡ 8.1 Reference to another element model in X-definition

The X-definition technology allows you to develop X-definitions separately (eg multiple development teams). Because these separate teams can share some parts of the X-definitions, but at the same time need to have the right to change these parts, it will solve this request by allowing each team to use their own X-definition implementation, and by linking all the X-definition will ensure that the shared parts of the XML document match. In order to ensure compliance, use either the implements or uses keyword, which, like the reference (see chapter 8.1), refer to the specified element model and are referenced in the relevant X-Script section.

When using the specification of referencing "**implements**", the names, namespaces, and occurrences of all objects in the element model are compared. Attributes and text values also compare their data types.

By using the specification of referencing "**uses**", unlike "**implements**", the names and occurrences of the referenced element model may vary. However, the names and occurrences of his descendants and attributes must be the same.

For both types of references, the structure (hierarchy) of the referenced models must match, the declaration of attributes and text nodes, their data types, and the same number of reported occurrences must be identical.

The following example is based on chapter 8.3, where two X-definitions are placed in separate files. In contrast to the initial example of this chapter, the following example is performed only for the X-definition assuming an XML document design using elements, with one attribute added to the VehicleElem element to demostrate the complexity of the example (for the X-definition case for XML documents made up of elements and attributes the solution will be similar):

```xml
<xd:def  xmlns:xd = "http://www.xdef.org/xdef/3.2
        xd:name = "garageElemOne"
        xd:root = "Vehicles">

    <VehicleElem xd:script="occurs *" id="int()">
        <type>enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')</type>
        <vrn>string(7)</vrn>
        <purchase>date()</purchase>
        <manufacturer>string()</manufacturer>
        <model>string</model>
    </VuzElem>
</xd:def>
```

```xml
<xd:def  xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garageElemTwo"
        xd:root = "Vehicles">

    <VehicleElem xd:script="occurs *; implements garageElemOne#VehicleElem" id="int()">
        <type>enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')</type>
        <vrn>string(7)</vrn>
        <purchase>date()</purchase>
        <manufacturer>string()</manufacturer>
        <model>string</model>
    </VuzElem>
</xd:def>
```

In the example above, there are two X-definitions, each one can be developed individually by another programmer. The programmer of the X-Definition garageElemTwo uses the "implements" section which ensures that its VehicleElem model structure will always be the same as the VehicleElem model from the X-Definition garageElemOne. In the case of a disagreement, an error is detected that is detected already in the X-definition compilation, thus notifying the different implementation of the model in question.

A similar statement is valid to the reference specification "uses", except that the root element VuzElem in the second X-definition can have any name and also a different number of occurrences.

9 Events in X-Script

➡ 4 Description of structure of XML document by X-definition

➡ 6 Construction Mode of X-definition

This chapter thoroughly describes how to validate and construct an XML document and tells what X-Script events are generated in each phase. Since the validation and design modes vary, both modes are described separately.

9.1 Events in the validation mode

In the validation mode, the validation process is controlled by an XML input document. When validating an XML document, the X processor follows the definitions in the following order:

1. **Beginning of root element processing**: This is the moment when you start the validation from a Java class or from X-definition by th xparse methods. At this point, only the name of the root element of an XML document is known, and the corresponding X-definition is selected in which the model in question is marked as root.

   Events that may occur are:

   - `match`: occurs before further processing when a model is selected;

   - `init`: occurs after match;

   - `onIllegalRoot`: occurs if an element model whose name is not found in the list of root elements specified in xd:root attributet in the X-definition header.

   Both events can be described in the X-definition header in element xd:def in the xd:script attribute. At this point, only the name of the root element and its attributes is available to X-script commands. According to the X-definition found, further processing continues. If an X-definition is not found and the onIllegalRoot event is not reported, an error will be reported and the validation will terminate immediately.

2. **Beginning of an element processing**: At this point, processing of an element from the input validated XML document begins.

   Events that may occur are:

   - `match`: occurs if the model is searched in the X-definition;

   - `init`: occurs if the model is found in the X-definition;

   - `onExcess`: occurs after checking whether the occurrence of the element exceeds the maximum allowed number. If an onExcess section in the X-Script of the element is not specified, an error is reported, otherwise the action specified in this section is performed;

   - `onIllegalElement`: occurs if the element description is not found in the X-definition or when this element is defined as "illegal". If this event is described in X-Script, the action will be performed, otherwise an error will be reported.

   X-Script now has all the attributes of the element, but still in its unprocessed form.

3. **Process the list of attributes:** The next step is processing of the list of attributes in the order how they was declared in the model.

   Events that may occur are:

   - `match`: it occurs if an attribute is searched in the X-definition.;

- onIllegalAttr: it occurs if an attribute is not acceptable by definition. If the attribute is not declared in the definition or if it is declared illegal, the specified action is called. If the action is not declared, an error is reported;

- onTrue: followed by the invocation of the content validation code. If the validation result is "true", this event occurs and the specified action if specified, is performed;

- onFalse: the event occurs again after validating the content with the negative result. If the onFalse action is not described, an error is reported, othewrwise the specified action is performed. The onFalse event occurs only if the attribute is contained in the input data but its value is empty and the option "acceptEmptyAttributes" is true;

- onAbsence: the event occurs if the attribute is described in the X-definition but is missing in the input data. When the onAbsence action is described, the value of the attribute can be set from X-Script by the specified action for example (however, in this case, it is recommended to clear the actual reporter's error using the clearReports method, otherwise an error will be reported). If an action is not described, an error is reported.

4. **Element Content Processing** (onStartElement event): After onset of element start and list of attributes, the onStartElement event occurs. At this point, the resulting element is already available after checking and processing its attributes, but still without descendants of this element. In the action that you invoke, you can, for example, process relationships between attributes etc. This event occurs even if the element has no child nodes.

5. **Element occurrence.** An element that has been read in the input data as the descendant of the currently processed element continues processing the subject under step 2.

6. **Text value occurrence**: The occurrence of an element's child text node is handled in a similar way as the occurrence of a new attribute. If the text value is defined as illegal, the onIllegalText event (instead ofIllegalAttr for attributes) occurs. Other events and actions are the same as for attributes.

   The text value is handed over to the processing only when the links to any entities have been resolved. If multiple text values follow in the input data, these values are concatenated in one value, which also applies to CDATA sections.

7. **Exit element processing**: After processing the inner nodes of the element, it is checked that the processing result contains the required minimum number of inner elements and text values. If not, an onAbsence events, as described in the X-Script of the element model, occurs. If not described, relevant errors are reported.

   After checking the element's content, the event **finally** occurs. The actions of event "finally" are first done for all attributes and eventually. Finally, if the **forget** action is specified, the contents of the element are deleted from the memory and deleted from the parent XML node (ie, this element will not be part of the created XML document). However, the number of occurrence counters will remain set.

8. If an XML document is processed in source form, an **onXmlError** event may occur. Actions for serving this event can be specified in the xd:script attribute of X-definition header. If it is written in the X-definition, it is possible to decide in the X-script whether the processing should proceed or the error may be treated in some way. If the action is not specified, processing is terminated as a fatal error and an error record is written to the reporter.

Upon completion of root element processing, the action of the event **finally** described in the X-script of the relevant X-definition is called. If validation is invoked from the program, the generated XML document is returned.

9.2   Construction mode events

In the construction mode, the construction process is controlled by X-definition. Instead of processing and controlling the input data, the resulting object is created according to the instructions described in the X-definition (in the Create Sections). The input data may have a completely different structure or even may not exist at all. If it exists, the input data can be passed as an XML element object..

In constructive mode, the onExcess, onIllegalElement, onIllegalAttr, and onIlelegalText events should not occur for a properly defined X definition. The processing steps are as follows:

1. **Element mode selection**: In the X-definition, the model of the constructed root element is selected according to the parameter in the xcreate method. If the element's root model is not found, the onIllegalRoot event occurs, the corresponding action is executed and the process is terminated. Otherwise, the corresponding element model is selected and proceeds to step 2.

2. The action **create** is executed from the X-script of root element model. If the action is not described, an element of the corresponding name is searched for in the input data (if such elements in the input data are more than the maximum number of occurrences allowed, only the corresponding number is selected). Otherwise, the same events occur as in the validation mode in step 2 (except the onExcess event that does not occur in this mode).

3. The **create** actions described in the element model attribute list are performed. After the list of attributes is created, it continues in the same way as in the validation mode in step 3.

4. The actions **create** for all descendants of the element model will be executed. Then, the processing follows as in steps 4 through 6 in validation mode.

5. The processing is the same as in step 7 in validation mode.

10 Debugger

➡ 4 Description of structure of XML document by X-definition

➡ 9 Using of X-definitions in Java code

Since X-definitions contain the functional code, X- definition offers a debugger that allows you to stop executing commands defined in the X-script in the specified locations, and to track the data content of the individual variables.

Two basic steps are required to run debugging mode:

1) Turn on the **debugging mode with IDE editor**. This mode is enabled by setting the property "xdef.debug" to the value "true" and the property "xdef.display" on "errors". Then if there are found errors during compilation of X-definitions a window with the source of X-definition and with error list is shown. You can correct errors, recompile X-definitions and run the project with corrected errors. Acter compilation the debugger window is shoen and you can set break points. The program is stopped at a break point and you can see actual data.The property you can pass to the Java in the Properties object:

```
...
Properties props = new Properties();
props.setProperty("xdef.debug", "true");
props.setProperty("xdef.display", "true");
XDPool xp = XDFactory.compileXD(props, "/path/to/*.xdef");
...
```

Note if the argument with Properties in the compilation command is null then properties are retrieved from System.getProperties (), which can be set from the command line.

2) Turn on the **debugging mode in the system console**. If the property "xdef.display" is set to "false" (it is default value) and the debugging mode is enabled only by setting the property "xdef.debug" to the value "true" the console debugger is invoked. You can add breakpoints

3) Add breakpoints or debug prints to X-Script..

You can also add special "trace()" or "trace(String)" methods to X-Script, which in the debug mode causes the program to print on the standard console or on the debugger window the information about the location where it was called and eventually the value specified as its parameter. You can also write a "pause()" method that prints the information and stops the program and waits for the user to continue. Both methods are ignored if the debug mode is not set.

In the above examples of command statements, the Java program from Chapter 4.8.1 was added, with debugging support commands as outlined above. An X-definition with debugging methods was used:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage"
        xd:root = "Vehicles">

  <Vehicles>
    <Vehicle
        xd:script = "occurs 0..*;
          onStartElement {
              trace('This is before method myPrint()');
              myPrint('my input')
          }"
        type   = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
        vrn    = "string(7)"
        purchase = "date()"
        manufacturer = "string()"
        model  = "string" />
  </Vehicles>
```

```
    <xd:declaration>
        int count = 0;

        void myPrint(String s) {
            if (1 &lt; 2) {
                pause();
                outln(s);
                count++;
            }
        }
    </xd:declaration>
</xd:def>
```

The following is displayed on the console or in the debugger window:

```
TRACE "This is before method myPrint()"; /Vehicles/Vehicle[1]; pc=6
PAUSE /Vehicles/Vehicle[1]; pc=1
```

The lines with the word "TRACE" are result of the `trace()` method. The lines with "PAUSE" are result of the `pause()` method. The position is written as xpath and pc is a program counter of the compied code. In the pause you can enter other debugger commands.

11 Processing and reporting errors

➡ 4 Description of structure of XML document by X-definition

➡ 7 Using of X-definitions in Java code

This chapter discusses the possible processing of error messages generated during the compilation of X-definition, validation, or XML document writing, and is based on and quoted from [Doc4].

Error and information messages are passed through a reporter that allows you to write a processing report in the form of org.xdef.sys.Report to the files or work memory of your computer. In chapters 7.1 and 7.2, the org.xdef.sys.ArrayReporter was used to process error messages, which stores all messages into the computer's memory.

The X-definition technology offers several other ways to report errors, warnings, or information reports. One option is to use the X-Script method "error" (see chapter 11.3), or to construct an XML document containing a description of the detected errors (see chapter 11.1) or to use the generally applicable X-definition error message mechanism, ie. reporter (see chapter 11.2).

Although the reporter is not required to use X-Definitions, this mechanism is used to report X-definition or XML document processing errors through X definitions. Therefore, it is recommended to read at least the chapters 11.1 and 11.2 for understanding error and warning messages.

11.1 Generate XML file with errors

➡ 5.5 Container

➡ 6 Construction Mode of X-definition

➡ 10 Debugger

An error or information message can be generated and written into a separate XML file directly when performing the validation or construction of the required XML document. Generating an XML file with reports can be done using construction mode and, together with the processing of the main XML document, is a combination of validation and construction (see chapter 6.6). Normally, the X-definition processor generates file with the recognized errors and warnings. However, you can also create your own report file as is described in the following chapters.

11.1.1 Creating of error data (two-phase)

The following sample will generate XML with errors in the following form:

```xml
<Errors>
    <Error Vrn="1A23456" CodeError="12" />
    <Error Vrn="2B34567" CodeError="27" />
</Errors>
```

Because the XML file will be created in constructive mode, it is necessary to add a bug element model to the appropriate X definition. It will also define the Vehicles element model in Chapter 4.7. Additionally, an error method will be defined in the declaration section, which will construct one error record, ie. one Error element (the variant for XML document design only by the elements will look similar):

```xml
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage"
        xd:root = "Vehicles">

   <Vehicles>
       <Vehicle xd:script = "occurs 1..*; onStartElement vrn=(String) @vrn; onAbsence myError(501)"
           type     = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other'); onFalse myError(10)"
           vrn      = "string(7) ; onFalse myError(11)"
           purchase = "date(); onFalse myError(12)"
           manufacturer = "string(); onFalse myError(13)"
           model    = "string; onFalse myError(14)" />

       <Errors xd:script = "create errors">
           <Error xd:script = "occurs +"
               Vrn = "string(7)"
               CodeError= "int()"
               Line = "int()"
               Column = "int()" />
       </Errors>
   </Vehicles>

    <xd:declaration>
           /* Create an empty Container. */
           external Container errors = [];

           /* Saved value of vrn */
           String vrn;

           /** Write item to the Container "errors".
            * @param code error code.
            */
           void myError(int code) {
               Context c = []; /* temporary Container */
               /* Set values from parameters to the temporary Contaner. */
               c.setNamedItem("Vrn", vrn);
               c.setNamedItem(CodeError", code);
               /* Add the temporary Container c to errors. */
               errors.addItem(c);
           }
    </xd:declaration>
</xd:def>
```

The "myError" method is called as an action of onFalse in the case of an error value of the attribute, onAbsence in the case of a missing element etc. The individual error messages are stored in the external variable "errors" of the data type Container using the "myError" method.

In the X-Script, two embedded methods, getSourceLine and getSourceColumn, were used to return the current parser position (row and column) of the XML document.

The onStartElement event was used to save the vehicle registration number, in which the value of the vrn attribute was stored in a variable of the same name.

In the first phase, X-definition is first used to validate an XML document with a list of vehicles, and in the second phase, it is subsequently verified whether any errors have been generated. If so, then the XML program explicitly generates XML document construction with errors:

```java
import org.xdef.sys.ArrayReporter;
import org.xdef.xml.KXmlUtils;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintStream;

public class GarageError {
    public static void main (String[] args) {
        /* Compile X-definition source to xpool. */
        XDPool xpool = XDFactory.compileXD(null, "/path/to/garage.xdef");
```

```
    /* Create xdoc for validation of input data. */
    XDDocument xdoc = xpool.createXDDocument("garage");

    /* Create ArrayReporter where to write recognized errors. */
    ArrayReporter reporter = new ArrayReporter();

    /* Validate the input data in 'garage.xml'. */
    xdoc.xparse("/path/to/vozovyPark.xml", reporter);

    /* Get the variable "errors" from xdoc. */
    XDContainer errors = (XDContainer) xdoc.getVariable("errors");

    /* Check if the container errors is not empty. */
    if (errors.getXDItemsNumber() > 0) {
        System.err.println("Incorrect imput data.");

        /* Create an XML document with errors. The variable errors still exists in the xdoc. */
        result = xdoc.xcreate("Errors", reporter);

        /* Write XML document with errors to a file. */
        KXmlUtils.writeXML("/path/to/garageErrors.xml", result);
    } else {
        /* Write validated document to the file 'xmlResult.xml'. */
        KXmlUtils.writeXml("/path/to/xmlResult.xml", xdoc.getElement());
        System.out.println("OK");
    }
  }
}
```

11.1.2 One step construction of error data

➡️ 11.1.1 Creating of error data (two-phase)

In the example from Chapter 11.1.1, a list of error messages was created during validation, and the XML report file was generated in the next step in the Java program. The reported error file can be created directly using the X-script and without calling the next step.

The solution is to add a method to the X-script that invokes the appropriate XML file construction and call this method from the event of the X-definition root element:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage"
        xd:root = "Vehicles">

   <Vehicles>
       <Vehicle xd:script = "occurs 1..*; onAbsence myError(501); onStartElement vrn=(String)@rz;
                             finally createErrors()"
           type   = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other'); onFalse myError(10)"
           vrn    = "string(7) ; onFalse myError(11)"
           purchase = "date() ; onFalse myError(12)"
           manufacturer = "string(); onFalse myError(13)"
           model  = "string; onFalse myErrora(14)" />

       <Errors xd:script = "create errors">
           <Error xd:script = "occurs 1..*"
              Vrn = "string(7)"
              CodeError = "int()"
              Line = "int()"
              Column = "int()" />
       </Errors>
   </Vehicles>

 <xd:declaration>
        /* Create an empty Container. */
        external Context errors = [];

        /* Saved value of vrn */
        String vrn;

        /* The XML element with errors. */
        external Element xErrors;
```

```
        /** Write item to the Container "errors".
         * @param code error code.
         */
        void myError(int code) {
            Context c = []; /* temporary Container */

            /* Set values from parameters to the temporary Contaner. */
            c.setNamedItem("Vrn", vrn);
            c.setNamedItem(CodeError", code);
            c.setNamedItem("Line", getSourceLine());
            c.setNamedItem("Column", getSourceColumn());

            /* Add the temporary Container c to errors. */
            errors.addItem(c);
        }

        /**
         * Check if an error message was generated by the X-definition processor or if tne Container errors
         * is empty and start the construction mode with the model "Errors". If the container is empty but
         * an error was generated by the X-definition processor an empty XML document is generated to the
         * xErrors.
         */
        void createErrors() {
            if (errors() || errors.getLength GT 0) {
                xErrors = xcreate("Errors");
            } else {
                xErrors = null;
            }
        }
    }
  </xd:declaration>
</xd:def>
```

The error() method returns the true value if an error message was generated by X-definition processor. Therefore, if there are such errors and the error method has not been invoked, an empty XML document xErrors will be created.

In the Java code will be data taken from the xErrors:

```java
import org.xdef.sys.ArrayReporter;
import org.xdef.xml.KXmlUtils;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintStream;

public class GarageErrors {
    public static void main (String[] args) {
        /* Compile X-definition source to xpool. */
        XDPool xpool = XDFactory.compileXD(null, "/path/to/garage.xdef");

        /* Create xdoc for validation of input data. */
        XDDocument xdoc = xpool.createXDDocument("garage");

        /* Create ArrayReporter where to write recognized errors. */
        ArrayReporter reporter = new ArrayReporter();

        /* Validate the input data in 'garage.xml'. */
        xdoc.xparse("/path/to/garage.xml", reporter);

        /* Get the variable "errors" from xdoc. */
        XDContext errors = (XDContext) xdoc.getVariable("errors");

        /* Check if the container errors is not empty. */
        if (errors.getXDItemsNumber() > 0) {
            System.err.println("Incorrect imput data.");
```

```
            /* Write XML document with errors to a file. */
            KXmlUtils.writeXML("/path/to/garageErrors.xml", xdoc.getVariable("xErrors").getElement());
        } else {
            /* Write validated document to the file 'xmlResult.xml'. */
            KXmlUtils.writeXml("/path/to/xmlResult.xml", xdoc.getElement());
            System.out.println("OK");
        }
    }
}
```

Another procedure for constructing an XML file with error messages has been chosen in the example in Chapter 12 where XML is made by writing XML objects through the output stream in external methods.

Another option would be to construct an element, ie a DOM object, in an external method and pass it on to the KXmlUtils utility.

11.2 Reporter

10 Debugger

In case you need to define a custom set of error messages generated in validtion or constructing of XML documents, the X-definition technology provides a mechanism for defining multilingual error messages. There are three main tools for this purpose: **reports**, **report tables**, and **reporters**. This tools are so general that they allow the use of reported messages from X definition definitions or in any application, not just when processing XML data.

This whole chapter is based on and quotes from [Doc4].

11.2.1 Reports

Reports are objects that are used to handle text messages. Each report is:

- *modifiable* – ie. it may contain a modifiable part, the so-called **model text**. Modifications are provided by modifiable text values and parameters with a specific value;

- *localizable* – ie. it may exist in multiple language variants;

- *unifiable* – ie each report can be uniquely identified using identifiers.

Each report is of a type that expresses the purpose for which the report serves (e.g., an error, warning, or an informative message). Language localization of reports is done through so-called **report tables** (see chapter 11.2.2), which are managed by the system manager (see chapter 11.2.3).

11.2.1.1 Report identifiers

Each report is uniquely identified by its identifier. The identifier is made up of:

- *prefix* – serves to identify the report table to which the report belongs. The prefix must consist of large ASCII characters with a length of 2 to 10 characters;

- *identification part* – is a unique identifier within the report table. It must start with an ASCII character that is not a capital letter, followed by any sequence of small as well as large ASCII letters, digits, or underscores (_)..

Examples of valid identifierfor the prefix „VHCL": VHCL001, VHCL_vrn, VHCLcolor, VHCL0120_1 etc.

11.2.1.2 Model text, parameters and links

The report model template is the text string can contain **parameters** and possibly **links to other reports**. A text string can contain characters in any UTF alphabet.

The **parameter** of report template model starts with "&" and bracket "{" followed by parameter name and ending with closing bracket "}". Usually the parameter name is a number corresponding to the report mehod argument number. However, the parameter name may also begin with an ASCII letter, a digit, or an underscore "_" followed by any sequence of ASCII letters, digits, or underscores.

Examples of text models with parameters:

```
Vehicle "&{0}" was not found in the database &{1}
```

In order to be able to handle cases where the parameter is not specified for the model text, the parameter can be added to separate brackets after the parameter the given parameter only. This text is writte to the resut text only if the value of the parameter is missing:

```
Vehicle "&{0}" was not found&{1}{ in the database }
```

If it is now assumed that the parameter &{0} with the value "1A23456" and &{1} with the "alpha" value will now be assumed in the modification text (text containing the parameter values, see chapter 11.2.1.3), then the following text will be reslut:

```
Vehicle "1A23456" was not found in database alfa

Vehicle "1A23456" was not found in database alfa
```

However, if the parameter &{1} is not included in the modifying text, then the result is the following:

```
Vehicle "1A23456" was not found in database

Vehicle "1A23456" was not found
```

For a given parameter, it is possible to include another text in the model text into another pairs of brackets (second in the order), which will then be listed after the relevant parameter if included in the modification text:

```
Vehicle "&{0}" was not found &{db}{ in database }{ of the application beta.}
```

For parameter &{0} with the value "1A23456" and &{1} with the value "alpha", after the modification, the resulting text will be as follows:

```
Vehicle "1A23456" was not found in database alfa of the application beta.
```

Without the & {1} listed parameter, the result will be:

```
Vehicle "1A23456" was not found
```

References to other reports assume the existence of the appropriate report table to which the referenced reports belong. Writing a link to another report model is similar to a parameter, but starts with a three-character "&{#" followed by the identifier of the report and is completed with a bracket "}".

Let there be a table of XX reports with an XX123 report that will list information about the position of an source XML object:

XX123: `&{line}{ on the line }&{xp}{, xpath=}`

Next, we have the following model of text referenced to report XX123:

```
Unknown vehicle registration number '&{0}'&{#XX123}
```

For parameters &{0} with "2B34567", &{line} with "125" and &{xp} with "/Vehicles/Vehicle[17]" the result will be:

```
Unknown vehicle registration number '2B34567' na řádce 125, xpath=/Vozy/Vozidlo[17]
```

11.2.1.3 Modification of the model text of the report

Chapter 11.2.1.2 lists various variants of model texts containing parameters and ways of their construction. Below is a structure of modification strings that is used to assign values to parameters in model text.

In the text model frm the previous chapter:

```
Vehicle "&{0}" was not found in database &{1}
```

the entry of the **modification string** that assigns the value "1A23456" of the parameter &{0} and the value "alpha" of the parameter &{1} is given below:

```
&{0}1A23456&{1}alfa
```

As can be seen from the example, the modification string only lists the parameters "&{" followed by the parameter name, bracket "}", and the value of the parameter. Parameters are not separated by any additional characters.

11.2.1.4 Reporting Links in the Modification String

In some cases, it is also necessary to refer to the report directly from the modification chain. The entry of the link is inserted as a sequence of characters "&{#" followed by the identifier of the report ending with the parenthesis "}".

The following sample assumes the existence of the STT02 report whose model text value is:

```
crashed&{state}{/}
```

We also have a text model report:

```
State of vehicle: &{state}
```

For the modification string:

```
&{state}{#STT02}
```

The result after modification will be:

```
Stav vozidla: crashed
```

After adding any parameters for the STT02 report to the modification string:

```
&{state}{#STT02 &{state}OK}
```

The result after modification will be:

```
State of vehicle: crashed/OK
```

The X-Definition resource library has eg. a message table with a "SYS" prefix, in which the SYS000 report is declared, which adds the line number, column, and xpath of the currently processed location in the XML document. In the previous chapters, it is similar to the sample report XX123 in chapter 11.2.1.2 whre the mpodel text can be modified as follows:

```
Unknown vehicle registration number '&{0}'&{#SYS000}
```

11.2.1.5 Report Types

Report type specifies the purpose for which the report serves. X-definition technology supports the following types of reports:

- ERROR – error message;

- FATAL – fatal error;

- LIGHTERROR – less serious errors;

- WARNING – warning messages;

- EXCEPTION – exceptions;

- KILL – report about the violent end of the program;

- TRACE – the program debugging report;

- AUDIT – auditing information;

- TEXT – text message;

- INFO – information message;

- UNDEF – nedefinovaný typ reportu.

- The setting of the desired report type is described in chapter 11.2.1.8.

When reporting a report, its type is always the first letter of the type name followed by the report identifier and the modified text.

The following example lists the ERROR report with VHCL001::

```
E VHCL001 Vehicle "1A23456" was not found in database alfa
```

For the following report type, the initial letter type and the report identifier is not printed; only the text is printed after the modification is made:

- TEXT – report contains a general text;

11.2.1.6 User report tables

Report tables are stored in source form as properties files in the resources library in the `org.xdef.msg` package, which is a part of the distributed X-definition file jar. The name of the file with report table starts with the prefix name follpwed with the character "_" (underscore) and the 3 letters ISO name of the language. The name extension must be "properties" (see chapter 11.2.1.9).

Example of the source report table, the English version file SYS_eng.properties:

```
# Description of messages.
SYS_DESCRIPTION=Messages for basic messages of the cz.syntea.xdef package
# Prefix of messages.
_prefix=SYS
# ISO name of language of this reports.
_language=eng
# ISO name of the default language.
_defaultLanguage=eng
# Localized name of language.
SYS_LANGUAGE=English

# *************** Messages: ***************
SYS000=&{line}{; line=}&{column}{; column=}&{sysId}{; source="}{"}&{xpath}{; xpath=}&{xdpos}{; X-position=}
SYS010=Compiled: &{c}, build version: &{v}, date: &{d}
SYS012=Errors detected&{0}{: }
SYS013=Too many errors
...
SYS015=Can't access system seq. ID file &{0}
SYS033=Parser can't continue; too many nested includes&{#SYS000}
SYS034=IO error detected on &{0}&{1}{, reason: }
...
SYS064=Datetime mask error: missing closing character&{0}{: "}{"}&{1}{, position: }
...
```

The user can also create a similar custom report table which must be available in the classpath `org.xdef.msg`.

11.2.1.7 Timestamp

The X definition technology allows you to attach the time information when a report was created (by the `setTimestamp` method on the report object) to the report. The time stamp is inserted into the report object as a long integer corresponding to the number of milliseconds beginning in 1990. If the time report is not connected to the report, the value is set to zero.

11.2.1.8 Programing of reports in Java

The reports are implemented using the org.xdef.sys.Report class. For the most common case, the constructor has the following form:

```
Reporter(byte type, String reportID, String text, String modification)
```

- *type* – corresponds to one of the constants for types: Report.ERROR, Report.FATAL, etc;

- *reportID* – report identifier, eg VHCL001;

- *text* – the default model text to be used when a report is not found in any table of reports (may be null if the report model exists);

- *modification* – modification string, eg &{0}1A23456&{line}125&{column}10&{xpath}{/root}.

Reports can be created using static methods of the class Report. The following sample creates a report with the ERROR type:

```
Report r1 = Report.error("VUZ001",
  "Unknown vehicle '&{0}' in database &{1}", "1A23456","ALFA" &{line}125&{colunn}10&{xpath}{/root}");
```

Note that the parameter names with the sequence number only are set as parameters of the method. The named parameters can be added as a parameters after the list of sequential parameters. However, you can write the whole modification text as only one parameter:

```
Report r1 = Report.error("VUZ001",
  "Unknown vehicle '&{0}' in database &{1}", "1A23456&", "ALFA&{line}125", "&{column}10&{xpath}{/root}");
```

Note instead of the method error you can use eg. method fatal, warning etc.

This way, you can use reports that are not yet in the spreadsheets and can be added later in the development of the application. These are so-called unregistered reports. If a table already exists, it is allowed instead of the report text id to specify a null value that causes the report to be generated from the text model in the report table (see chapter 11.2.2).

11.2.1.9 Registered and unregistered form of reports

Chapter 11.2.1.8 lists examples of creating reports using identifiers defined as text strings. However, most applications use a stable set of reports, so it is also possible to use constant identifiers instead of text values - but first, you need to "register" reports through report tables.

Report tables in which reports are contained are written in their basic form as a property file (see chapter 11.2.2). However, access to the appropriate files when running an application may not be effective. Therefore, the report table can be registered and a Java class that is automatically generated from the source property file (see chapter 11.2.2.1). Registered reports are then accessible more quickly and with certainty that they exist, so when using registered reports, the model text is not mentioned because it has to be in the corresponding table of reports in the generated Java class. Instead of a string-shaped identifier, the Report ID is used to construct a report (which is a long number and is taken from a registered table in the generated Java class. The Java identifier of a report corresponds to the report identifier.

If the report table VHCL was not registered the Java code will be:

```
import org.xdef.sys.Report;

public class RegisteredReportExample {
        ...
        // The report identifier must be in the string form and there must be also
        // the pasrameter with the report model text. However, if the table or reports existe
        // the parameter with the report model text may be null.
        Report r = Report.fatal("VHCL001", "Unknown vehicle '&{0}'", "1A23456");
        ...
```

Example of the Java source using a registred report table:

```
import org.xdef.msg.VHCL; // registred reports with the prefix VHCL
import org.xdef.sys.Report;

public class RegisteredReportExample implements VHCL {
        ...
        // Text with the report model text is missing because the report is in the registered table
        // Ihe identifier of the report is VHCL001
        Report r = Report.fatal(VHCL001, "1A23456");
        ...
```

11.2.2 Tables of reports

Report tables describe the text versions of individual reports in different language versions. The report table is in the form of properties in its source form.

For each report identifier, there is just one item in the report table, designated by the property name which matches the appropriate report identifier. The property value ontains a report text model.

Each report table must have a property item with the name of the table prefix (property "_prefix") and the property item describing the language version of text models in the form of ISO code (property "_language"). The language code must be ISO-639-2, ie it must be three letters with small ASCII letters (for example, English "eng", Czech "ces", German "deu", etc; see http://www.loc.gov/standards/iso639-2). The name of items describing the text models reprezents the reports in specific language, as shown in the following samples of the report tables

Each language version of the table should have defined all reports. As the default language, English is set, so there must be an English mutation for each report table. If the desired language mutation is not found by the application, the default language version of the report table will be used.

Example of the English version of table (file name: VHCL_eng.properties):

```
_language=ces
_prefix= VHCL
VHCL001 eng=Vehicle "&{0}" wasn't found &{1}{ in database }
VHCL002 eng=Vehicle "&{0}" wasn't found &{1}{ in directory }
VHCL016 eng=Database &{0} not found.
      ...
VHCL237 eng=Vehicle type &{t} not supported.
      ...
VHCL580a eng=The total damage by accident &{0} CZK
VHCL580b eng=The total damage by accident &{0} EUR
VHCL580c eng=The total damage by accident &{0} USD
      ...
VHCL905 eng=System error
```

The Czech version of report table VHCL (file name: VHCL_ces.properties):

```
_language=ces
_prefix=VHCL
VHCL001=Vozidlo "&{0}" nebylo nalezeno &{1}{ v databázi }
VHCL016="Databáze &{0} nenalezena.
VHCL237=Vozidlo typu &{0} není podporováno.
VHCL580a=Celková škoda při nehodě &{0} Kč
VHCL580b=Celková škoda při nehodě &{0} EUR
VHCL905=Chyba systému
```

Note that in the English version of the report table is also a VUZ580c report, which will be used even if the target language is set to Czech (because this report is missing in the Czech version).

11.2.2.1 Registering report tables

You can register the report tables using the `org.xdef.sys.RegisterReportTables` class by running the main methods. You can define the following parameters::

- **-i** followed by a list of source XML files with report tables ("wildcard" characters can be used to select multiple files, ie "*", "?", etc.);

- **-o** followed by a directory where the Java classes are to be generated;

- **-c** determining the encoding of the input file (default setting is taken from the system setting);

- **-d** sets the default report language (default is English).

The `RegisterReportTables` utility will generate Java-classes with report tables in the specified output directory. In order to ensure uniqueness or, the uniqueness of the report tables, the generated Java classes should be placed in the org.xdef.msg package. Each prefix corresponds to one class with a list of report identifiers. The name of the generated class matches the table prefix name.

For the example from Chapter 11.2.2, the following class would be generated:

`org.xdef.msg.VUZ.java`

11.2.3 System report manager

Access to reports from their report taebles is provided by the system manager through the `org.xdef.sys.SManager` class. When JVM starts, only one instance of this class is created automatically. The system manager ensures availability of reports tables, setting the anguage environment, system properties, and more. The manager also allows users to add or delete the actual report tables. Initial settings are taken over by the manager from system properties. The manager instance is accessible through the `SManager.getInstance()` method.

The following example shows how to obtain a manager instance and add report tables with the VHCL prefix for English and Czech, and set up Czech as the default language. Report tables can be imported from both the local repository or from the Internet, for example:

```
SManager manager = SManager.getInstance();
manager.setProperty("xdef.msg.VHCL_eng", "http://www.syntea.cz/xdef/VHCL_eng.properties");
manager.setProperty("xdef.msg.VHCL_ces", "C:/path/to/VHCL_ces.properties");
manager.setLanguage("ces");
```

All properties that have been set up by the system manager may, of course, also be taken over before being initialized from `System.properties`.

In the case of multi-threaded programming, when each thread uses a different language, it is necessary to synchronize the part of the program that works with the system manager settings.

11.2.4 Reporters

Reports (Chapter 11.2.1) and report tables (Chapter 11.2.2) allow you to create reports within a given application. These rports can be handled as character strings, or you also need to create files of such messages. To create report files are available so called *reporters*

Reporters are available as Java-classes, which implements following interfaces:

a) `org.xdef.sys.ReportWriter;`

b) `org.xdef.sys.ReportReader.`

X-definition has the following implementations of reporters:

- `org.xdef.sys.ArrayReporter` implements both interfaces, `ReportWriter` and the `ReportReader.` This reporter writes messages to the memory and it is able both, to write reports or to read reports.

- `org.xdef.sys.FileReportWriter` writes reports to a file;

- `org.xdef.sys.FileReportReader` reads reports from a file;

- `org.xdef.sys.NullReportWriter` this reporter reports "throws away";

11.2.5 Reports in exceptions

In the case of generating a program exception, it is necessary to pass the generated report describing the cause of the error to the parent control structure just by the exception. Reports can pass on those exceptions that implement the `org.xdef.sys.SThrowable` interface.

The text that the program exception carries will be the relevant report, and it is possible to continue working with the report when the exception is captured. The X-definition library has the following exceptions that can handle reports (this is an extension of the basic exceptions from the j `java.lang` package.

- `org.xdef.sys.SError;`
- `org.xdef.sys.SException;`
- `org.xdef.sys.SExternalError;`
- `org.xdef.sys.SIllegalArgumentException;`
- `org.xdef.sys.SIOException;`
- `org.xdef.sys.SNullPointerException;`
- `org.xdef.sys.SParseException;`
- `org.xdef.sys.SRuntimeException;`
- `org.xdef.sys.SUnsupportedOperationException;`
- `org.xdef.sys.SDOMException.`

The following example shows using a report exception:

```
try {
        String pojistovna;
        ...
        throw new SException(VUZ057, "&{as}" + pojistovna);
        ...
} catch (SException ex) {
        if ("VHCL057".equals(ex.getID())) { ... }
}
```

11.2.6 Example of use

The following is a sample of the simple reporting of error reporting reports.

The sample tables (in Czech and English versions) as defined in chapter 11.2.2 will be used as report tables, with only those reports explicitly mentioned here.

Consequently, it is appropriate to register the tables of reports, which are now only in the source form in XML files (see chapter 11.2.2.1):

```
java -cp org.xdef.sys.RegisterReportTables -i /path/to/VHCL*.properties -o /path/to/src/org/xdef/msg
```

Now you can use the reports in the following program demo that searches for the entered registration mark in the database and calculates the total damage recorded for the vehicle from traffic accidents:

```
import org.xdef.msg.VHCL; // registered reports with the VHCL prefix
import org.xdef.sys.*;

public class VehicleAccidents implements VHCL {

        public static void main(String[] args) {
                // Create reporter
                ArrayReporter reporter = new ArrayReporter();

                // Put the unregistered message do reporter
                reporter.putReport(Report.text(null, "Start programu", null));
```

```
                // check the vehicle registration number
                String vrn = "1A23456";

                // Run validation of XMLdata
                XDPool xpool = XDFactory.compileXD(null, "/path/to/garage.xdef");
                XDDocument xdoc = xpool.createXDDocument("garage");
                xdoc.xparse("/path/to/vozovyPark.xml", reporter);

                // add user report messages to the reporter
                try {
                        // the method getTotalLoss should return the total loss for the given vrn
                        // The method throws am exception the vrn is not returned from the database.
                        int total = getTotalLoss(rz);
                        // zápis informace o celkové škodě do reportéru
                        report.putReport(Report.info(VHCL580a, "&{total}" + total));
                } catch (Exception ex) {
                        if (ex instanceof NoSuchRzException) {
                                // vrn not found
                                reporter.putReport(Reporter.error(VHCL001, "&{rz}" + rz));
                        } else if (ex instanceof DatabaseException) {
                                // the databse is not available
                                // in the report VHCL016 will not be the database name and the parameter
                                // will be ignored
                                reporter.putReport(Reporter.error(VUZ016, null));
                        }
                }

                // check if an error report was generated
                if (reporter.errors()) {
                        // write mesage
                        System.err.println(reporter.printToString());
                } else {
                        // No error reports generated; write the report SYS069 ("No errors found")
                        report.putReport(Report.text(SYS069));
                        System.out.println(reporter.printToString());
                }
        }
}
```

Both exceptions (`DatabaseException` a `NoSuchRzException` must implement the `SThrowable` interface, see 11.2.5).

Through the reporter, you can also access individual reports as shown in the following code snippet [Doc1]:

```
import org.xdef.msg.VHCL; // registered reports with the VHCL prefix
import org.xdef.sys.*;

public class VehicleAccidents implements VHCL {

        public static void main(String[] args) {
            // Create reporter
            ArrayReporter reporter = new ArrayReporter();

            ...
            // chack if there are error reports
            if (reporter.errors()) {
                // prepare report reader
                ReportReaderInterface reader = reporter.getReportReader();
```

```
                // print reports
                Report rep;
                while ((rep = reader.getReport()) != null) {
                    System.err.println(rep.toString());
                }
            } else {
                // No error reports; write the report SYS069 ("No errors found")
                report.putReport(Report.text(SYS069));
                System.out.println(reporter.printToString());
            }
        }
    }
}
```

In the example of `VehicleAccidents.java` listed above, an interface implementation technique was used to use the VHCL repository table. Such a solution is too static, and it is better to use the system manager to retrieve the corresponding report tables, as shown in the following example:

```java
public class VehicleAccidents2 {

        public static void main(String[] args) {
                // prepeare files with the reports
                String[] messages =
                        new String[] {"/path/to/VHCL_ces.properties", "/path/to/ VHCL_ces.properties "};

                // Create report tables – (the default language will be english (eng)
                ReportTable[] rt = SManager.createReportTables(messages, "eng", null);

                // prepare report manager
                SManager sm = SManager.getInstance();

                // add report tables
                sm.addReportTables(rt);
                rt = null;

                // set supported languages
                sm.setLanguage("eng");
                sm.setLanguage("ces");

                // prepare reporter
                ArrayReporter reporter = new ArrayReporter();

                // put the unregistered report to the reporter
                reporter.putReport(Report.text(null, "Programu started", null));

                // the vehicle registration number
                String vrn = "1A23456";

                // zde může probíhat běžná validace nebo konstrukce XML dokumentu – hlášení bude do
                // reportu vkládat automaticky nebo budou generována metodou error z X-Scriptu
                XDPool xpool = XDFactory.compileXD(null, "/path/to/garage.xdef");
                XDDocument xdoc = xpool.createXDDocument("garage");
                xdoc.xparse("/path/to/garage.xml", reporter);

                // add messages to the reporter
                try {
                        // Suppose the method getTotalLoss returns for the vrn the total loss.
                        // The metoda throws an exception if the databes is not available
                        // (DatabaseException) or if the vrn not exists (NoSuchVRNException)
                        int total = getTotalLoss(vrn);
                        // write report
                        report.putReport(Report.info(VHCL580a, "&{total}" + total));
                } catch (Exception ex) {
                        if (ex instanceof NoSuchVRNException) {
                                reporter.putReport(Reporter.error(VHCL001, "&{vrn}" + rz));
                        } else if (ex instanceof DatabaseException) {
                                reporter.putReport(Reporter.error(VHCL016, null));
                        }
                }
```

```
                       // check if an error was reported
               if (reporter.errors()) {
                       System.err.println(reporter.printToString());
               } else {
                       report.putReport(Report.text(SYS069));
                       System.out.println(reporter.printToString());
               }
       }
}
```

11.3 The error method

➡ 10 Debugger

➡ 11.2 Reporter

The reports can also be written from X-Script to the reporters. For this purpose, the error method is reserved in the following variants:

- error(String s) – writes an error message to a reporter specified when running a validation or construction from a Java program and returns a value of false;

- error(String s, String m, String t) – writes an error message corresponding to the report with identifier s, modification text m and modifier t to the reporter and returns false. The modification text is used in case the report is not found in the report table.

An example of using the method is given in the following example (X-definition is based on the example of chapter 4.10 and the table of reports are taken from chapter 11.2.2):

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
       xd:name = "garage"
       xd:root = "Vehicles">

   <Vehicles>
       <Vehicle xd:script = "occurs 0..*"
           type    = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other'); onFalse typErr()"
           vrn     = "vrn(7)"
           purchase = "date()"
           manufacturer = "string()"
           model   = "string" />
   </Vehicles>

   <xd:declaration>
       /** Check the form of a vehicle registration number.
        * @param len number of characters for VRN.
        * @return true, if VRN is correct, otherwise return false.
        */
       boolean vrn(int len) {
          String text = getText();
          if (text.length() != len) {
             return error(text + " has incorrect format. Required is " + len + " characters.");
          }
          return true;
       }

       /** Put the error the vehicle is incorrect. */
       void typErr() {
          String typ = getText();
          // error(String s, String m, String t)
          error("VHCL237", "Vehicle type &{t} is not supported", "&{t}" + typ);
       }
   </xd:declaration>
</xd:def>
```

Input XML document:

```xml
<Vehicles>
    <Vehicle
        Type = "traktor"
        vrn = "1A23456"
        purchase = "2011-02-01"
        manufacturer = "Škoda"
        model = "Yeti" />
</Vehicles>
```

The error outout will be:

```
E VRN237: The vehicle type tractor is not supported.
```

11.4 Automatically generated errors

The errors that are detected by an X-definition tool when validating or constructing an XML document, such as type checking, and not treated in X-Script, such as onFalse, are automatically written to the appropriate reporter that was specified as the xparse or xcreate method parameter of the XDDocument object.

If a null value has been entered instead of a reporter, then if they have been generated some errors, they are returned from the xparse or xcreate method as an exception to SRuntimeException.

11.4.1 Generating into a reporter

For a Java code when a reporter is assigned to the X-definitions, all messages are written to the appropriate reporter:

```java
// Create ArrayReporter
ArrayReporter reporter = new ArrayReporter();

// Validate XML document according to the X-definition
XDPool xpool = XDFactory.compileXD(null, "/path/to/garage.xdef");
XDDocument xdoc = xpool.createXDDocument("garage");
xdoc.xparse("/path/to/garage.xml", reporter);

...

// using the error method, you can see if any error reports have been written to the reporter
if (reporter.errors()) {
        // In the reporter are FATAL, ERROR, or LIGHTERROR reports, the printToString method converts all
        // reports into text and in the specified language
        System.err.println(reporter.printToString("ces"));
} else {
        // No errors were found in the reporter.
        // For the listing, the registered report model SYS069, which contains "No errors found" is used.
        report.putReport(Report.text(SYS069));
        System.out.println(reporter.printToString("ces"));
}
```

The output of the report reporter, for example:

```
E XDEF525: Attribute "xxx" not allowed; line=1; column=10; xpath=/Vehicles/Vehicle[1]
```

11.4.2 Generate an exception listing error

However, if no reporter is reported (i.e., is null), an exception is generated containing the relevant reports after the xparse or xcreate method ends:

```java
// Validate XML document according to the X-definition
XDPool xpool = XDFactory.compileXD(null, "/path/to/garage.xdef");
XDDocument xdoc = xpool.createXDDocument("garage");
try {
    xdoc.xparse("/path/to/garage.xml", null);
} catch(Exception es) {
    System.err.println(ex);
}

...
```

It will be printed:

```
org.xdef.sys.SRuntimeException: E XDEF525: attribute "xxx" not allowed; line=1; column=10;
xpath=/Vehicles/Vehicle[1]
```

A similar result can be achieved by calling the checkAndThrowErrorWarnings() method on a reporter that converts all reports into SRuntimeException:

```
// Create ArrayReporter
ArrayReporter reporter = new ArrayReporter();

// Validate XML document according to the X-definition
XDPool xpool = XDFactory.compileXD(null, "/path/to/garage.xdef");
XDDocument xdoc = xpool.createXDDocument("garage");
xdoc.xparse("/path/to/garage.xml", reporter);
...
// it the reporter contains errors the exception with a meesage about errors will be thrown.
reporter.checkAndThrowErrorWarnings();
```

Example of the message of the thrown exception:

```
org.xdef.sys.SRuntimeException: E SYS012: Errors detected:
E XDEF525: attribute "xxx" not allowed; line=1; column=10; xpath=/Vehicles/Vehicle[1]
```

11.5 Errors when compiling X-definition

An error may be generated when you translate the X-definition from the source code.

Let's have an X-definition that contains errors in X-script:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
        xd:name = "garage"
        xd:root = "Vehicles">

    <Vehicles>
        <Vehicle xd:script = "ocur 0..*"
            type    = "enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
            vrn     = "string(7); onTrue *"
            purchase = "date(); onFalse myPrint()"
            manufacturer = "string()"
            model   = "string" />
    </Vehicles>
</xd:def>
```

There are the following errors in the X-definition:

- On the line 7 "ocur" (should be "occurs"
- On the line 9 "onTrue *" is not correct written action (should be a statement)
- On the line 10 the method "myPrint" is not declared

Suppose the specified X-definition is in the /path/to/garageErr.xdef file. The message of the exception SRuntimeException thrown by the method compileXD will be:

```
org.xdef.sys.SRuntimeException: E SYS012: Errors detected:
E XDEF425: Script error; line=7; column=30; source="/path/to/garageErr.xdef "
E XDEF426: Action statement expected; line=9; column=26; source="/path/to/garageErr.xdef "
E XDEF443: Unknown method: 'myPrint()'; line=10; column=54; source="/path/to/garageErr.xdef"
```

12 Appendix A – complete example

➡ 4 Description of structure of XML document by X-definition

➡ 5.5 Container

➡ 6 Construction Mode of X-definition

➡ 7 Using of X-definitions in Java code

➡ 8 Structuring of X-definitions

➡ 11 Processing and reporting errors

Appendix A summarizes in a simple example all the basic options that X-definitions offer for processing XML files, both in validation mode and in construction mode.

The sample below handles a fictitious and very simplified traffic accident record that is first validated (both the accuracy of the XML document structure and the accuracy of the document data types, including validation of the values against the existing vehicle registration value database). The validation process calculates statistics using user methods directly in X-definition as well as using external methods. The results of computation are gradually inserted into a Context data object object. It is then used after the processing of the relevant element from the input file to construct part of the output XML document with a summary of accidents.

The first is the Java class, including its external methods for use in X-Definitions, which will prepare the required environment (load and set the necessary objects) and start validation of the `simpleAccident.xml` file:

```
src/accident/SimpleAccident.java
```

```java
package accident;

import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import org.xdef.XDXmlOutStream;
import org.xdef.proc.XXNode;
import org.xdef.sys.ArrayReporter;
import org.xdef.xml.KXmlUtils;
import java.net.URL;
import java.net.URLClassLoader;
import java.util.Random;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
```

```java
/**
 * Simple example of using X-definition for validating and processing input data and generating new data.
 *
 * @author Jaroslav Srp
 */
public class SimpleAccident {

    /* Run SimpleAccident project
     * @param inData path to the input XML file.
     * @param cars path to the file with data about vehicles.
     * @param outData path to the output XML file
     * @param errData path of the generated error XML file.
     * @throws Exception if an errors occurs.
     */
    public static void process(String inData,
        String cars,
        String outData,
        String errData) throws Exception {
        // get URL location of the X-definition
        URL url = URLClassLoader.getSystemResource("accident/simpleAccident.xdef");
        // compile X-definition
        XDPool xpool = XDFactory.compileXD(null, url);
        // create XDDocument
        XDDocument xdoc = xpool.createXDDocument("simple");
        // Read and validate the input document with data about vehicles
        Element el = KXmlUtils.parseXml(cars).getDocumentElement();
        // Set the parsed data tu the field "cars" in the X-definition
        xdoc.setVariable("cars", el);
        // Set the "out" X-definition variable as a stream to the outData file, where
        // will be written the generated output with statistics
        xdoc.setVariable("out", XDFactory.createXDXmlOutStream(outData, null, true));
        // an instance of this class that contains the external method "error" for writing errors
        // into the output XML errData file
        SimpleAccident writer = new SimpleAccident(errData);
        xdoc.setUserObject(writer);
        // Create the reporter
        ArrayReporter reporter = new ArrayReporter();
        // Validace the inData file
        xdoc.xparse(inData, reporter);
        // Check validation errors
        if (reporter.errorWarnings()) {
            // Display error reported by X-definition validation
            System.err.println(reporter);
        } else {
            // No reported by X-definition validation
            System.out.println("No input data format errors detected");
        }
        // Check if errors were detected by the process
        int errs = xdoc.getVariable("errs").intValue();
        if (errs != 0) {
            System.err.println("Errors detected in the input data.");
        }
        // Close the stream with errors
        writer.closeAll();
    }

    ////////////////////////////////////////////////////////////////
    // Methods called from the X-definition and the variables
    ////////////////////////////////////////////////////////////////

    // File name of the file with errors
    private final String _errorFile;

    // writer
    private XDXmlOutStream _errorWriter;

    // XML document with errors
    private Document _errorDoc;
```

```
    /**
     * A constructor for creating an user object that is passed to the X-definition for use in an external
     * method "error".
     * @param errorFile the path to an XML file to which errors will be written.
     */
    public SimpleAccident(String errorFile) {
        _errorFile = errorFile;
        _errorWriter = null;
    }

/**
     * The constructor to create a new object in this class. It has only an auxiliary purpose.
     */
    public SimpleAccident() {
        _errorFile = null;
    }

    // total number of injured persons
    private static int sumInjured = 0;

    // total number of killed persons
    private static int sumKilled = 0;

    // total loss
    private static int total = 0;

    /**
     * An external method that adds the number of injured and killed from the currently processed element to
     * total totals.
     * @param xnode currently processed element.
     */
    public static void infoPeople(XXNode xnode) {
        String s = xnode.getXXElement().getAttribute("injury");
        sumInjured += Integer.valueOf(s);
        s = xnode.getXXElement().getAttribute("death");
        sumKilled += Integer.valueOf(s);
    }

    /**
     * External method that returns a string value that contains total statistics.
     * @return string with total statistics..
     */
    public static String statistics() {
        return String.valueOf("Total injured: " +sumInjured + " and killed: " + sumKilled
                + ". Total loss: " + total + " thousand.");
    }

    /**
     * External method that generates a random ID to record the resulting statistics.
     * @return ID.
     */
    public static String generateIdStats() {
        Random rn = new Random();
        int id = rn.nextInt(999);
        return String.valueOf(id);
    }

    /**
     * External method that adds to the total damage caused by the accident being handled.
     * @param xnode the currently processed element in which this method was invoked.
     * @param loss the loss of the accident.
     */
    public static void loss(XXNode xnode, long loss) {
        total += loss;
    }
```

```java
    /**
     * External method that writes an error into an XML file.
     * @param xnode the currently processed element in which this method was invoked.
     * @param code error code.
     */
    public static void myError(XXNode xnode, long code) {
        // Get user object from the X-definition
        SimpleAccident x = (SimpleAccident) xnode.getUserObject();
        // create an XML writer for errors
        if (x._errorWriter == null) {
            try {
                // writer has not yet been created, so open a stream to write errors to an XML file
                x._errorWriter = XDFactory.createXDXmlOutStream(x._errorFile, "windows-1250", true);
            } catch (Exception ex) {
                throw new RuntimeException(ex.getMessage());
            }
            // a document with a root element Errors is created
            x._errorDoc = KXmlUtils.newDocument(null, "Errors", null);
            // write XML header and a root element
            x._errorWriter.writeElementStart(x._errorDoc.getDocumentElement());
        }
        // auxiliary element for writing an error
        Element el = x._errorDoc.createElement("Error");
        el.setAttribute("ErrorCode", String.valueOf(code));
        // find the parent element Accident in which the id attribute is available
        while (xnode != null && !"Accident".equals(xnode.getElement().getNodeName())) {
            xnode = xnode.getParent();
        }
        el.setAttribute("id", xnode.getElement().getAttribute("id"));
        el.setAttribute("XPos", String.valueOf(xnode.getXPos()));
        x._errorWriter.writeNode(el);
    }

    /**
     * Close the output stream for writing errors.
     * (This method is not called from the X definition.)
     */
    public void closeAll() {
        if (this._errorWriter != null) {
            this._errorWriter.writeElementEnd(); // end the root element
            this._errorWriter.closeStream(); // close writing errors
        }
    }

////////////////////////////////////////////////////////////////////////////////////////////////
//  Run this example twith given data from resources.
////////////////////////////////////////////////////////////////////////////////////////////////
    public static void main(String[] args) throws Exception {
        SimpleAccident.process("resources/input/data.xml",
            "resources/input/cars.xml",
            "resources/output/result.xml",
            "resources/output/errors.xml");
    }

}
```

X-definition, which ensures validation of input and the constructed output. The root element is the element Accidents:

resources/accident/simpleAccident.xdef

```xml
<xd:def xmlns:xd="http://www.xdef.org/xdef/3.2"
    xd:name="simple"
    xd:root="Accidents">

    /* Root element. */
    <Accidents xd:script="finally close()">

        /* The element Accident contains the date and ID of the accident. */
        <Accident xd:script="occurs 1..*; onStartElement saveAccident(); finally calculate();"
            date="date(); onFalse myError(10)"
            id="uniID.ID(); onFalse myError(11)">

            /* The element Persons contains the number of injured and killed persons in an accident. It may
                also contain an ill attribute that is not currently being used. */
```

```
        <Persons xd:script="finally injuryDeath(); onStartElement infoPeople();"
            injury="int(0,1000); onFalse myError(12)"
            death="int(0,1000); onFalse myError(13)" >
            optional string();
        </Persons>

        /* Element Vehicles element contains information about vehicles that were involved in a traffic
           accident. */
        <Vehicles xd:script="occurs 1..*">

            /* Use the xd:attr attribute to allow the occurrence of another arbitrarily named attribute
               with value string(0.64) in the Vehicle element.
               The element Vehicle also contains the vehicle's VRN and loss on the vehicle. */
            <Vehicle xd:script="occurs 1..*; finally computeLoss();"
                vrn="vrn(); onFalse myError(501)"
                loss="int(0,100000); onFalse myError(502)"
                xd:attr="? string(0,64)" />
        </Vehicles>
    </Accident>
</Accidents>

/********** Output elements. **********/

/* Element Calculation contains attributes with accident date, accident ID, cumulative number of injured
   and killed and the total damage. */
<Calculation xd:script="create calc; finally outln('****** Accident ' + @id + ' *******');"
    date="date()"
    id="uniID.IDREF()"
    injury_death="int(0,2000)"
    loss="int(0,1000000)">

    /* Element Vehicles contains a summary information of vehicles that have been involved in an accident.
       The attributes include the number of cars from the accident and the ID of the calculation, which
       must match the ID from the Calculation element. */
    <Vehicles xd:script="create vehicles; finally loss(totalLoss)"
        number="int(0,500);"
        id="uniID.IDREF()">

        optional string; options setTextUpperCase;

        /* Element Vehicle contains information on the purchase price of the car, the damage caused,
           the registration number, the year of manufacture and, optionally, the warn attribute. This
           element is created in the design mode from the Container crashedVehicles. */
        <Vehicle xd:script="occurs 0..*; create crashedVehicles"
            price="int()"
            loss="int()"
            vrn=" string(7)"
            model="optional string()"
            year="optional int(1900,2100)"
            warn="optional string()">
        </Vehicle>
    </Vehicles>

    /* Element Now contains an information of the date and time of the assembly of the element and the
       timeout  (the actual date shifted by one year in advance). The actual date and time is saved
       to the variable n of the element Now. */
    <Now xd:script="var Datetime x = now(); /*x is the current time when this element was processed*/ "
        createtime="optional dateTime(); create x.toString('yyyy-MM-dd\'T\'HH:mm:ss'); /*datetime form*/ "
        timeout="optional date(); create create x.addYear(1).toString('yyyy-MM-dd'); /*year in advance*/ "/>

</Calculation>

/* The Element Stats contains basic information about statistics such as the 'statistics' name that is set
   to the fixed (static) value in the resulting XML document using the fixed option. It also contains
   generated IDs and a text node with a description. */
<Stats name="fixed 'statistics'"
    id="int(0,999); create generateIdStats();">
    string(); create statistics();
</Stats>
```

```
/* Declaration of variables and methods. */
<xd:declaration>
    /* Declaration of external methods. */
    external method {
      void accident.SimpleAccident.infoPeople(XXNode);
      String accident.SimpleAccident.statistics();
      String accident.SimpleAccident.generateIdStats();
      void accident.SimpleAccident.loss(XXNode, long);
      void accident.SimpleAccident.myError(XXNode, long);
    }

    /* The identifier must be unique and it must be a three-digit number */
    uniqueSet uniID num(3);

    /* ********* Variables used in methods. ********* */

    /* XML document with vehicle database. The value is set by the Java program. */
    external Element cars;
    /* The number of actual errors and the number of errors in validating the previous element. */
    int errs = 0, errsPrev = 0;
    /* Output XML stream with errors. */
    XmlOutStream errStream;
    /* Number of statistics generated. */
    int cnt = 0;
    /* Output stream with generated statistics. */
    external XmlOutStream out;
    /* Container with a vehicle. */
    Container vehicle;
    /* Container used for calculation of statistics. */
    Container calc;
    /* Total loss, the total purchase price of the vehicles and the total number of vehicles. */
    int totalLoss = 0;
    int totalSum = 0;
    int cntVehicles = 0;
    /* Container for vehicles. */
    Container vehicles;
    /* Container for crashed vehicles. */
    Container crashedVehicles;

    /* The method closes both streams - both XML and bugs, with generated statistics. */
    void close() {
        if(errs != 0) {
            /* write the end tag of the element to the errStream */
            errStream.writeElementEnd();
            /* close the stream */
            errStream.close();
        }

        if(cnt > 0) {
            out.writeText("Computed staristics");
            /* Create the "Stats" element, which will be created in construction mode with the
               method xcreate */
            out.writeElement(xcreate("Stats"));
            out.writeElementEnd();
            out.close();
        }
    }

    /* Check the type of vehicle registration mark. */
    boolean vrn() {
        String vrn = getText();
        vehicle = xpath('Vehicle[@vrn="' + vrn + '"]', cars);
        if(vehicle.getLength() == 0) {
            errs++;
            return error("VRN '" + vrn + "' is not in the database.");
        }
        return true;
    }
```

```
          /* The method inserts to the first element of the Container calc an attribute about the
             cummulative number of injured and killed persons. */
          void injuryDeath() {
              /* The toString method converts the value of the attribute in the argument into a string data
                 type.*/
              int z = parseInt(toString(@injury));
              int u = parseInt(toString(@death));
              calc.getElement(0).setAttr("injury_death", toString(z + u));
          }

          /* Save details of the accident and clear Containers.*/
          void saveAccident() {
              /* initialize the Container "calc"; after initialisation it will contain a single element
                 that will be the currently processed element. */
              calc = [getElement()];
              /* initialization of the "crashedVehicles" Container; after initialization it will be empty */
              crashedVehicles = new Container();
          }

          /* This method inserts to the first element of the Container cars teh  attributes with the
             registration number of the crashed car, its year of production and the total damage. It also writes
             a warning to the output XML file if the damage to the vehicle exceeds its acquisition value. */
          void computeLoss() {
              int aLoss = parseInt(toString(@loss));
              totalLoss += aLoss;
              cntVehicles++;
              int aCena = parseInt(vehicle.getElement(0).getAttr("price"));
              totalSum += aCena;
              /* Search in the database Vehicle  (external variable cars) a vehicle with the same VRN. */
              String vrn = vehicle.getElement(0).getAttr("vrn");
              Element typ = xpath('Vehicle[@vrn="' + vrn + '"]/Type', cars).getElement(0);
              /* From the "vehicle" container, the first element (ie the element with the index 0) is taken and
                 the necessary attributes are gradually added to it. */
              vehicle.getElement(0).setAttr("model", typ.getAttr("model"));
              vehicle.getElement(0).setAttr("year", typ.getAttr("year"));
              vehicle.getElement(0).setAttr("loss", toString(@loss));
              /* note: you can write the relation "&gt;" also as GT */
              if(aLoss &gt; aCena) {
                  vehicle.getElement(0).setAttr("warn", "Loss on the vehicle is higher the the price!");
              }
              crashedVehicles.addItem(vehicle.getElement(0));
          }

          /* The method calculates the ratio of total damage to the total purchase price of crashed vehicles
             and rounds the result to 2 decimal places. */
          String damageRatio() {
              float ratio = ((float)totalLoss / (float)totalSum)*100;
              return toString(ratio, "##0.00' %'");
          }
```

```
            /* The method writes the calculated statistics into the output file. */
        void calculate() {
            if(errs != errsPrev) {
                errsPrev = errs;
            } else {
                calc.getElement(0).setAttr("loss", toString(totalLoss));
                /* creates an empty Temp element */
                Element e = new Element("Temp");
                /* add the appropriate attributes to the element */
                e.setAttr("number", toString(cntVehicles));
                e.addText("Total loss on vehicles is " + totalLoss + " thousands, "
                    + "which is " + damageRatio() + " and crashed "
                    + cntVehicles + " vehicle" + (cntVehicles != 1 ? "s" : ""));
                e.setAttr("id", calc.getElement(0).getAttr("id"));
                /* New Container "Vehicles" with element "e". */
                vehicles = [e];
                if(cnt++ == 0) {
                    /* Set output indentation. */
                    out.setIndenting(true);
                    /* Write header ot the root element. */
                    out.writeElementStart(getRootElement());
                }
                /* For the currently processed element (can be obtained by the getElement method,
                   the Calculation element is created in construction mode. */
                out.writeElement(xcreate('Calculation', getElement()));
            }

            /* Crear counters. */
            cntVehicles = 0;
            totalLoss = 0;
            totalSum = 0;
        }
    </xd:declaration>

</xd:def>
```

Example of vehicle database:

```
resources/input/cars.xml
```

```xml
<Cars>
    <Vehicle price="600" vrn="1A01122">
        <Type model="Audi" doors="4" year="2011" />
    </Vehicle>

    <Vehicle price="70" vrn="4B35500">
        <Type model="Opel" doors="3" year="2003" />
    </Vehicle>

    <Vehicle price="240" vrn="4C35500">
        <Type model="Škoda" doors="5" year="2009" />
    </Vehicle>

    <Vehicle price="30" vrn="4E35500">
        <Type model="Mazda" doors="3" year="1994" />
    </Vehicle>

    <Vehicle price="170" vrn="4M35500">
        <Type model="VW" doors="4" year="2006" />
    </Vehicle>

    <Vehicle price="470" vrn="4T35500">
        <Type model="Mercedes Benz" doors="2" year="2007" />
    </Vehicle>
</Cars>
```

Input XML document:

```
resourtces/input/data.xml
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Accidents>
    <Accident date="2012-04-01" id="001">
        <Persons injury="-2" death="1" />
        <Vehicles>
            <Vehicle vrn="1A01122" loss="160" />
            <Vehicle vrn="4B35500" loss="50" vin="000491024560" />
        </Vehicles>
    </Accident>

    <Accident date="2012-04-17" id="002">
        <Persons injury="2" death="0" />
        <Vehicles>
            <Vehicle vrn="4B35500" loss="15" />
        </Vehicles>
    </Accident>

    <Accident date="2012-05-11" id="003">
        <Persons injury="14" death="3" />
        <Vehicles>
            <Vehicle vrn="4B35500" loss="150" />
            <Vehicle vrn="4C35500" loss="14" />
            <Vehicle vrn="4E35500" loss="360" />
            <Vehicle vrn="4M35500" loss="240" />
            <Vehicle vrn="4T35500" loss="190" />
        </Vehicles>
    </Accident>
</Accidents>
```

Generated XML document with statistics according to the above X-definition and XML input files:

resources/output/resault.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Accidents>
  <Calculation date"2012-04-01"
    id="001"
    injury_death="-1"
    loss="210">
    <Vehicles id="001"
      number="2">
      TOTAL LOSS ON VEHICLES IS 210 THOUSANDS, WHICH IS 31.34 % AND CRASHED 2 VEHICLES
      <Vehicle loss="160"
        model="Audi"
        price="600"
        vrn="1A01122"
        year="2011"/>
      <Vehicle loss="50"
        model="Opel"
        price="70"
        vrn="4B35500"
        year="2003"/>
    </Vehicles>
    <Now createtime="2018-10-12T13:33:32"
      timeout="2019-10-12"/>
  </Calculation>
  <Calculation date="2012-04-17"
    id="002"
    injury_death="2"
    loss="15">
    <Vehicles loss="15"
      number="1">
      TOTAL LOSS ON VEHICLES IS 15 THOUSANDS, WHICH IS 21.43 % AND CRASHED 1 VEHICLE
      <Vehicle loss="15"
        model="Opel"
        price="70"
        vrn="4B35500"
        year="2003"/>
    </Vehicles>
    <Now createtime="2018-10-12T13:33:32"
      timeout="2019-10-12"/>
  </Calculation>
  <Calculation date="11.5.2012"
    id="003"
    injury_death="17"
    loss="954">
    <Vehicles id="003"
      number="5">
      TOTAL LOSS ON VEHICLES IS 954 THOUSANDS, WHICH IS 97.35 % AND CRASHED 5 VEHICLES
      <Vehicle loss="150"
        model="Opel"
        price="70"
        vrn="4B35500"
        warn="Loss on the vehicle is higher the the price!"
        year="2003"/>
      <Vehicle loss="14"
        vrn="4C35500"
        model="Škoda"
        year="2009"
        price="240"/>
      <Vehicle loss="360"
        model="Mazda"
        price="30"
        vrn="4E35500"
        warn="Loss on the vehicle is higher the the price!"
        year="1994" />
      <Vehicle loss="240"
        model="VW"
        price="170"
        vrn"4M35500"
        warn="Loss on the vehicle is higher the the price!"
        year="2006"/>
      <Vehicle loss="190"
        model="Mercedes Benz"
        price="470"
        vrn="4T35500"
```

```
        year="2007"/>
    </Vehicles>
    <Now createtime="2018-10-12T13:33:32"
      timeout="2019-10-12"/>
  </Calculation>
  Computed staristics
  <Stats id="760"
    name="statistics">
    Total injured: 14 and killed: 4. Total loss: 1179 thousand.
  </Stats>
</Accidents>
```

The enerated XML document with recgnized errors (these are only errors that were detected by the appropriate X-Script actions and processed by the error method). Each error record contains the error code (according to the error method parameter) and the accident ID, the line and the error detection XML position in the XML entry:

```
resources/output/errors.xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Errors>
    <Error ErrorCode="10" XPos="/Accidents/Accident[1]/@date" id="001" />
    <Error ErrorCode="18" XPos="/Accidents/Accident[1]" id="001"/>
    <Error ErrorCode="10" XPos="/Accidents/Accident[2]/@date" id="002"/>
    <Error ErrorCode="10" XPos="/Accidents/Accident[3]/@date" id="003"/>
</Errors>
```

The example prints to the output streams System.out (black) and System.err (red):

```
****** Accident 001 *******
****** Accident 002 *******
****** Accident 003 *******
E XDEF813: Value of 'int' doesn't fit to 'minInclusive'; line=9; column=7;
source="D:\cvs\DEV\java\examples\resources\input\data.xml"; xpath=/Calculation/@injury_death; X-
position=simple#Calculation/@injury_death
```

13 Appendix B – frequently asked questions (F.A.Q.)

➡ 4 Description of structure of XML document by X-definition

➡ 6 Construction Mode of X-definition

Appendix B summarizes the most frequently resolved issues and the most frequently asked questions related to the X-definition technologies

13.1 Data content, types

Is it possible to declare the mandatory empty string?

When enabling zero-length string values, you need to distinguish between the attribute value and the text node. While with the `string(0, N)` type method is possible for the attributes, the zero length of the string at the text nodes has no justification for their zero length.

The following example allows the value of the `id` attribute to be from 0 to 5 characters long:

```
...
<Accident xd:script = "occurs 0..*"
      id = "string(0, 5)"
</Accident>
...
```

If you need to enable the zero string string length of a text node, you must define this node as optional:

```
<Accident xd:script = "occurs 0..*">
      <vrn>optional string(5)</vrn>
</Accident>
```

You can not use the **string(0, N)** method for declaring a text node type.

Is it possible to set the connection to a database Java code and set it to the X-script?

Yes, you can use the `org.xdef.XDFactory` and set it do the XDDocument object as a external variable (see chapter 7.6.1).

Why does X-definition report an error when declaring the variable of the type "long" in X-Script?

The data type long is not supported in X-Script. Instead, you can use an int type that has a 64-bit range in X-Script, and matches the long type in Java, see chapter 14.2.

How to declare an array in X-Script?

You can declare the Container data type, which integrates field implementation with ArrayList and the map using HashMap, see chapter 5.5.

For example, a field that contains eg numbers 1 and 2 and the character 'a', X-Script can be created using the constructor as follows:

```
Container array = [1, 2, 'a'];
```


Can I convey information about the element and its attributes in the design mode to an external method?

For example, for writing `<A b='create b()' c='create c()'/>` you can pass the information about element A and the contents of the attribute b to the method "`c()`"?

Yes, but only through the XXNode object passed to the external method, see chapter 4.8.2.

```
<A b='create b()' c='create c(toString(@b))' />
```

How vary the cases when using the create command in the validation mode (XDDocument.xparse) in the construction mode (XDDocument.xcreate)?

There is practically no difference because the resulting generated document will always have the same content. The main difference, therefore, is whether the generation is run from X-script or from Java program. For more information, see chapter .

What data will contain the generated (output) element of a given model if the X-definition processing is run in validation mode, when this model validates some XML input data and also defines the construction of the new (output) element in the create section?

For example, the model may have the following form:

```
<Vehicle xd:Script="finally xcreate('Vozidlo'); create true" id="int; create '12'" />
```

Notwithstanding the fact that some data is validated using the model in question, the output, ie the newly generated element, contains just the data that is intended to create the X-Script sections. That is, according to the model, the following element will always be generated:

```
<Vehicle id="12" />
```

13.2 Error reporting

Why do not I know the Reporter about the name of the X-definition file or the XML entry where the error was detected?

If the X-definition XML entry document was used as an InputStream type parameter instead of a file path, such as a String or an XML document, the system does not have the source file location information. However, it is possible to add a name to the X-definition translation when using the XDBuilder object.

Why is XDEF227 reported ("Ambiguous repeated declaration of the external method")?

Is the following error reported to you?

```
org.xdef.sys.SRuntimeException: E SYS012: Errors detected:
E XDEF227: Ambiguous repeated declaration of the external method 'method_name'; line=nnn; column=nnn
```

The reasons why this error is reported may be several:

1) External methods are declared by at least two external methods of the same name with identical parameters:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/3.2" xd:name="..." >
  <xd:declaration>
    external method {
            void example.VozovyPark.myPrint(String);
            void example.SeznamNehod.myPrint(String);
    }
  </xd:declaration>
      ...
</xd:def>
```

2) A XDPool has been generated containing at least with two X-definitions that declaration of the external methods with the same name and parameters:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/3.2" xd:name="garage" xd:root="Vehicles" >
  <xd:declaration>
    external method {
        void exampole.VozovyPark.myPrint(String);
    }
  </xd:declaration>
  ...
</xd:def>
```

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/3.2" xd:name="garage1" >
  <xd:declaration>
    external method {
        void exampole.SeznamNehod.myPrint(String);
    }
  </xd:declaration>
      ...
</xd:def>
```

```
...
XDPool pool = XDFactory.compileXD(null, new String[] {"VozovyPark.xdef", "SeznamNehod.xdef"});
...
```

In the case of an XDPool object consisting of multiple X-definitions, the content of all declaration section X-definitions is combined into one and the result is therefore the situation in (1).

Possible corrections of the cases described:

1) Alias - The original names of external methods can be overlayed using aliases, see chapter 4.8.

2) Renaming methods - you can observe the convention that external methods from a given class will have a prefix, for example, by the name of that class.

3) Inheritance - if possible, and if the external methods that have the same name have the same implementation, then it is appropriate to move these methods into a common ancestor. This will allow the X-definition pool to see only one implementation of the method, and the error will no longer be reported.

4) Set the "local" scope range in the xd:declaration section:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/3.2" xd:name="garage1" >
  <xd:declaration scope="local">
    external method {
        void exampole.SeznamNehod.myPrint(String);
    }
  </xd:declaration>
      ...
</xd:def>
```

13.3 How to create an X-definition from given XML data

This chapter is a guide how to design the X-definition describing given XML data:

1) From the described XML document, a structure description is first created, ie it is necessary to assemble the relevant element models. Data structures that do not exist in a particular instance of data have to be described. If any value is not relevant for processing (e.g., it is counted in the future), then its occurrence in X-Script is described by the ignore key. If there may be attributes that are not known at that time, you can describe this situation as an attribute xd: attr. (See chapters 4 and 14.5.)

2) Describe how to check attribute values and text nodes. If you need to activate some operations depending on the result of the checks, the onTrue and onFalse actions may be added to the type validation expression and eventually describe actions for different events..

If a fixed value is required for an attribute or text node, write the `fixed` `keyword` X-Script (see chapter 4.2.1).

3) If the other elements than those that have been described are allowed, add the x-Script element xd:any or set the option moreElements.

4) Repetitive and similar inner parts of the elements are replaced by the creation of auxiliary element models and relevant references. Referred element models can also be written in different X-definitions (see Chapter 8).

5) If the order of elements placed within a parent element does not have to be respected, the description of these parts can be inserted into the mixed group in the `xd:mixed` element (see chapter 4.11.2).

6) The selection of one of the multiple variants is written using the selection section bounded by the `xd:choice` (see chapter 4.11.3).

   Note: Selection sections may be nested into a mixed section.

7) If more XML documents are to be described in one X-definition, you need to use the `xd:root` attribute to describe which elements can occur as root elements (see chapter 4.7.2). In the multiple X-definitions project it is possible to refer to all X-definitions of a given file set.

8) If the application requires it, it is possible to describe global variables and user validation methods and eventually for frequently used actions (see chapters 4.7.3.3 a 4.10).

9) When processing large files with XML source text, it is possible to tell the X-definition processor that when the element is processed, it can be "forgotten" to unnecessarily vacate the memory. This fact will be written as the keyword `forget` (see chapter 4.6).

14 Appendix C – supplementary description of X-definitions

➡ basic knowledge of XML ([1], [2]

➡ 2 X-definition Technology

➡ 4 Description of structure of XML document by X-definition

➡ 6 Construction Mode of X-definition

➡ 7 Using of X-definitions in Java code

This Appendix o adds the functionality and possibilities of the X-definitions described in the preceding chapters and summarizes the remaining functionality that has not yet been mentioned.

14.1 Utitlites.

X-definition technology offers programming tools available for use with XML data in the library `org.xdef.xml` [Doc3]:**:**

- `KXmlUtils` – collection of static methods for working with XML data (e.g;
  - `nodeToString` - conversion of XML node into a string value, eg.:
    - nodeToString(Node node, boolean indent);

    if the parameter indent is true then result will be indented.
  - `writeXML` writes a DOM object as a XLM file (see chapter 7). The method `writeXML` can be invoked eg as:
    - writeXML(String fileName, Node rootElement);
    - writeXML(String fileName, Node rootElement, boolean indenting, boolean comments);

    The DOM represented by rootElement is stored in a fileName, including / without comments (true / false) and indented (ie including XML formatting characters) or indentation (true or false).
- `KDOMUtils` – collection of static methods extending the methods from the org.w3c.dom interface;
- `KDOMBuilder` – constructs and parses XML data;
- `KXpathExpr` – supports the XPath expressions;
- `KXquery` – supports the XQuery expressions (only it there is available a libarary supporting XQuery, e.g. Saxon, Basex etc.);

To work with strings, files, and system resources, a utilities are available in the `org.xdef.sys` [Doc3]:

- `SUtils` – includes methods for character encoding of text, support for internationalization, etc;
- `StringParser` – parser of strings or character streams;
- `SPosition` – implementation of the source position (used eg. in the reporter);
- `ReportReader` – reads streams with reports
- `ReportWriter` – writes reports to a stream
- `BNFGrammar` – allows work with the text values described by the enhanced BNF grammar.

see chapter2.3.

The X definition technology comes with a wide range of utilities in the package `org.xdef.util` (see [Doc3])

14.1.1 Launch the validation mode from the command line

Validation mode can be started from the command line of the `org.xdef.util.XValidate` class. If external methods are used in X-Script, all classes in which the appropriate methods are defined must be available in the classpath (see [Doc3]). Command line parameters are in format:

```
[-h] [-d defList] [-x xDefName] [-l logFile] –i xmlFile
```

The parameters are:

- `-h` displays the help message;

- `-d defList` describes the path to an X-definition file or contains a list of files separated by a semicolon. You can use the wildcard characters "*" and "?" in the file name entry, which describes the selection of the entire group of files. This parameter is optional. If not specified, the X-definition location must be described in the XML document using the `xdi:location` attribute (see Chapter 4.7.3.9);

- `-x xDefName` contains the X-definition name to be used for validating input data. Parameter is optional and is only given if parameter -d is specified;

- `-l logFile` is the file path that writes error and data processing information. Parameter is optional. If not defined, the log is written to the standard `System.err` print stream. If the value of the parameter is null, then the file is not created and when an error message was reported, the program terminates with an exception (in this case, warnings and information messages are ignored);

- `-i xmlFile` specifies the path to the XML document file to be validated.

The following example is illustrated by the following example, including the initial setup of paths to libraries and classes with external methods that validates the Vehicles.xml file according to the garage X-definition from the Garage.xdef file:

```
set CLASSPATH=/path/to/mylib;/path/xdef/lib/syntea_xdef3.1.jar
java org.xdef.util.XValidate –d /path/to/Garage.xdef –x garage -i /path/to/Vehicles.xml
```

14.1.2 Lounch the construction mode from the command line

Also, design mode can be run directly from the command line using the command class `org.xdef.util.XCompose`. Command line parameters have the following format (see [Doc3]):

```
[-h] –d defList [-x xDefName] [-r rootName] [-l logFile] [-e encoding] –i xmlFile –o outFile
```

The parameters are:

- `-h` displays the help message;

- `-d defList` describes the path to an X-definition file or contains a list of files separated by a semicolon. You can use the wildcard characters "*" and "?" in the file name entry, which describes the selection of the entire group of files;

- `-x xDefName` contains the X-definition name to be used for validating input data. Parameter is optional and is only given if parameter -d is specified;

- `-r rootName` specifies the name of the root element if it differs from the name of element in the input data (if specified). Parameter is optional. If not specified, the name taken from the root element of the input data;

- `-l logFile` s the file path that writes error and data processing information. Parameter is optional. If not defined, the log is written to the standard `System.err` print stream. If the value of the parameter is null, then the file is not created and when an error message was reported, the program terminates with an exception (in this case, warnings and information messages are ignored);

- -o outFile the pathname of the file into which the created data will be written, ie the constructed XML document;

- -e encoding the name of the code table for the output file. Unless stated, the UTF-8 is set;

- -i xmlFile specifies the path to the XML document file to be validated.

The following example demonstrates how to use the iset of paths to libraries and classes with external methods to construct the AccidentList.html file to be created according to the X-definition AccidentList Html.xdef from the file AccidentList.xml [Doc3]:

```
set CLASSPATH=/path/to/mylib;/path/xdef/lib/syntea_xdef3.1.jar
java org.xdef.util.XCompose –o /path/to/AccidentList.html –x accidentList
 –d /path/to/AccidentListHtml.xdef -i /path/to/AccidentList.xml
```

14.1.3 Checking the accuracy of Xdefinition

Especially when writing large X-definitions, it is advisable to check the accuracy of the X-definition written. For this purpose, the utility org.xdef.util.CheckXdef can be used. The output from the utility is a list of error messages or a listing that indicates the locations where errors were detected (if the -v parameter is specified). The utility output is written to System.out standard print stream. If the external methods are used in the X-definition, the appropriate classes must be available in the classpath when calling the program.

```
[-h] [-v] [-d set of X-definitions] X-definice
```

The parameters are:

- -h displays the help message;

- -v Optional parameter specifying that a detailed listing of errors is to be made. Otherwise, only the error list is displayed;

- X-definice is a required parameter and defines an input file (s) with X-definition;

- -d set of X-definitions is an optional parameter and contains a list of files with X-definitions in a comma separated list. If not defined, a single X-definition is used. The file specification can also contain wildcard characters "*" and "?", And you can specify a set of X-file or X-definition files.

14.1.4 Create indented form of X-definition

The source form of the X-definition can be automatically reformatted to make the result clearer and more readable. For this purpose, use the program org.xdef.util.PrettyXdef, which reads the X-definition and makes the appearance of the element models and their descendants and attributes so that the result is clear and that the offset of the nested elements is standard:

```
[-h] [d outDir] [-i n] [-e encoding] [-p prefix] sourceFile
```

The parameters are:

- -h nepovinný parametr. Pokud je uveden, vypíše se informace o parametrech;

- -d outDir definuje adresář, kam bude uložen přeformátovaný soubor. Není-li parametr uveden, přepíše se vstupní soubor (pokud program během činnosti předčasně skončí, zůstane původní soubor zachován pod původním jménem s příponou bak);

- -i n je nepovinný parametr, který nastavuje počet mezer použitých pro odsazení vnitřních elementů, výchozí hodnota je 3;

- -e encoding nepovinný parametr, který nastavuje znakovou sadu kódování. Hodnota musí být slučitelná s hodnotou parametru *encoding* v zájlaví XML dokumentu (tj. např. *UTF-8*, *windows-1250* apod.). Není-li parametr uveden, použije se původní kódování;

- -p prefix nepovinný parametr, pomocí kterého lze předefinovat prefix pro jmenný prostor X-definic. Není-li tento parametr specifikován, je použit původní jmenný prostor;

- sourceFile určuje vstupní soubor nebo skupinu souborů (opět je možné použít wild card symbol „*").

The following example demonstrates how to use it, including initiating paths to libraries and classes with external methods that reformat the AccidentListHtml.xdef file:

```
set CLASSPATH=/path/xdef/lib/syntea_xdef3.1.jar
java org.xdef.util.PrettyXdef /path/to AccidentListHtml.xdef
```

14.1.5 Conversion of XML schema to X-definition.

Utility for converting from XML schemas to X-definitions. This is the class org.xdef.util.XsdToXdef.

The parameters are:

- -h displays the help message;

- -in path of input main schema location

- -out output file or directory name

- -s separate every schema to a standalone X-definition file

- -p prefix of X-Definition namespace ("xd" if not specified)

- -l path to the log file

14.1.6 Conversion of X-definition to XML schema.

Utility for converting from X-definitions to XML schema. This is the class org.xdef.util.XdefToXsd.

The parameters are:

- -h displays the help message;

- -i list of input sources with X-definitions

- -o output directory

- -m name of root model (optional)

- -x name of X-definition (optional)

- - sp prefix of XML schema namespace (optional, default is "xs")

- - extension of schema file (optional, default is "xsd")

14.2 Types of values in X-script

In the X-script commands and variables with the values of several types can occur (see chapter 4.7.3). Value types that can occur in the X-script are as follows.

14.2.1 int (the integer numbers)

Values are in the range:

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

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

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

Secial predefined constants ($MININT, $MAXINT) are describet in chapter 14.4 Pedefined values (constants).

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(mask)", where the argument is the output format mask, which represents a string of characters that contains control characters, which have the following meaning:

- 0   digits, leading zeros are replaced by a space
- \#   digits, leading zeros are appended to the output
- .   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".
```

14.2.2 float (the floating-point numbers)

Values are in the range:

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

or `0.0` or

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

The specification of numbers with a floating-point 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.

Secial predefined constants ($MINFOLAT, $MAXFLOAT, $PI, $E, $POSITIVEINFINITY, $NEGATIVEINFINITY, $NaN) are describet in chapter 14.4 Pedefined values (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".
```

14.2.3 Decimal (the decimal numbers)

The decimal numbers in the X-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.

14.2.4 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 X-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 X-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 X-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".

14.2.5 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. The recommended format is according to ISO8601, 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

14.2.6 boolean (Boolean values)

The Boolean values may be used in the expressions, parameters of methods and the X-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.

14.2.7 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.

14.2.8 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.

14.2.9 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.

14.2.10   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"

14.2.11   Element (XML elements)

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

14.2.12   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 chapter 14.7.5.

14.2.13   NamedValue (named values)

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

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

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

14.2.14   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, see chapter 14.7.6).

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.

14.2.15   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 X-script with the "throw" command. An object of type "exception" is possible to create in the X-script with the constructor "new Exception(error message)".

14.2.16   Parser (tool used to parse string values)

The objects of this type are generated by the X-definition compiler. The parser is an object, on which it is possible to invoke a validation method. The result of this method is an ParseResult object. The parser object is constructed when a validation method is specified in the X-script.

14.2.17　　　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 ()").

14.2.18　　　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".

14.2.19　　　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 14.9.1.

14.2.20　　　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)".

14.2.21　　　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.

14.2.22　　　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 X-script:

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

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

14.2.23　　　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);
```

14.2.24　　　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.

14.2.25   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 X-script of X-definitions:

```
    XmlOutStream xstream = new XmlOutStream ("c:/data/file.xml", "UTF-8", true);

...
  XStream. writeElementStart();
  XStream.writeElement (); // write the whole child
...
  xstream.writeElement();
  xstream.writeElementEnd(); // write end of started element
...
XStream.close();
```

14.3 Type validation methods

14.3.1 Validation methods of XML schema types

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.

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	w

	validated data. Possible values are: "collapse", "replace" or "preserve"	

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.

V*alidation methods of XML schema datatypes:*

Method name	Description	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	8 bit integer	int	M bempt
Date	date	Datetime	M bempt
dateTime	date and time	Datetime	M bempt
decimal	decimal number	Decimal	T befmpt
double	floating point numbers	double	M befmpt
duration	XML duration.	Duration	M bempt
ENTITY	name of the XML entity	String	L epl
ENTITIES	list of the XML entity names separated by a space	Container	L epl
Float	floating point numbers	float	M bempt
gDate	date	Datetime	M bempt
gDay	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
Int	32 bit integer	int	M bempt

integer	integer number	decimal	M bempt
language	XML schema language specification	String	L belp
List	array of values.	Container	L beilp
Long	integer number	int	M bempt
Name	name (according to the XML name).	String	L
NCName	XML NCName value	String	L
negativeInteger	negative integer number	decimal	M bempt
NMTOKEN	XML NMTOKEN (i.e. letters, digits, "_", "-", ".", ":")	String	L
NMTOKENS	list of NMTOKEN, separated by a space	Container	L
nonNegativeInteger	The positive integer number and zero	Decimal	M bempt
nonPositiveInteger	negative integer number and zero	Decimal	M belpw
normalizedString	character string	String	L bempt
positiveInteger	positive integer number	Decimal	M bempt
QName	XML QName	String	L belp
short	16 bit integer	int	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 more types	Any	-, eip
unsignedByte	unsigned 8 bit integer	int	M bempt
unsignedLong	unsigned 64 bit integer	Decimal	M bempt
unsignedInt	unsigned 32 bit integer	int	M bempt
unsignedShort	unsigned 16 bit integer	int	M bempt

Example of type validation mehods in a X-Script:

```
string(%enumeration=['Hello', 'world'])
string(%pattern=['[A-Z][a-z]{3}'])
string(%whiteSpace='collapse', %length='5')
string(%minLength='3', %maxLength='30')
decimal(%totalDigits='8', %fractionDigits='2')
dateTime(%minInclusive='2000-01-01T00:00:00-01:00')
gMonth(%enumeration=['--01Z','--12'])
list(%item= short)
union(%item=[ boolean, short], enumeration=['true','1'])
```

14.3.2 Other validation methods 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	String	- elp

	the namespace must be defined in the context of the current element		
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	-
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 14.2.5 Datetime (the date and time values)).	Datetime	-
xdatetime (s, t)	date and/or time corresponding to the mask s (see 14.2.5 Datetime (the date and time values)). The resulting value will be reformatted according to the mask t.	Datetime	-
xdtype	Checks if value is valid declaration of the implemented type validation method (i.e. from the table in chapter 14.3.1 Validation methods of XML schema types or from this table)	Parser	-

14.3.3 Data types used in Java external methods

➡ 4.8 External (Java) Methods

The implementing and using external methods, ie methods defined in the Java-classes (not in X-Script), is discussed in Chapter 4.8. Data types of parameters and return values (if they are not void) must match one of the types that are X-definitions or the values returned by a method. For the clarity, the following table lists both the list of types supported by X-Script and its equivalent in Java:

X-Scriptu data type	Java data type	Explanation
int	long	Integer value.
Float	double	Real number value.
Boolean	boolean	Logical hvalue.
Element	org.w3c.dom.Element	XML Element
String	java.lang.String	Characted string

Datetime	`java.util.Calendar`	Date and time.
Duration	`org.xdef.sys.SDuration`	Time iinterval (see ISO 8061).
Container	`org.xdef.XDContainer` `org.xdef.XDValue[]`	The type implenets ArrayList and Map. The array of XDValue is the Contaimer converted to array.
Regex	`java.util.regex.Pattern`[2]	Compiled regular expression.
RegexResult	`java.util.regex.Matcher`[3]	Result of regular expression.
Input	`org.xdef.XDInput`	Output stream.
Output	`org.xdef.XDOutput`	Výstupní datový proud.
Bytes	`byte[]`	Byte array
Report	`org.xdef.sys.Report`	Reporter (see chapter 11.2)
XpathExpr	`org.xdef.xml.KXpathExpr`	XPath expression.
Parser	`org.xdef.XDParser`	Type of validation method.
ParseResult	`org.xdef.XDParseResult`	Result of Parser.
Service	`org.xdef.XDService`	Exteral service (databases) (java.util.sql.Connection, XQuery connection, XML database connection etc.)..
Statement	`org.xdef.XDStatement`	Statement of database. (java.util.sql.Statement, …).
ResultSet	`org.xdef.XDResultSet`	Iterator or a map of values (eg. java.util.sql.ResultSet).
XmlOutStream	`org.xdef.XDXmlOutStream`	Writer of XML data to a stream.
Exception	`org.xdef.XDException`	Rozhraní pro výjimky v X-definicích.
BNFGrammar	`org.xdef.XDBNFGrammar`	Compiled BNF grammar
BNFRule	`org.xdef.XDBNFRule`	BNF rule

14.4 Pedefined values (constants)

X-definition has a set of predefined global variables and constants - their name always starts with the "$" character. In X-Script, you can use the following set of predefined variables and constants:

Jméno	Typ	Význam
`$stdIn`	`Input`	Standard input stream (System.in)
`$stdErr`	`Output`	Standard output stream for error messages (System.err)
`$stdOut`	`Output`	Standard output stream (System.out)
`$PI`	`float`	he constant π, the ratio of the circumference of a circle to its diameter (3.141592653589793)
`$E`	`float`	the constant e, the base of natural logarithms (2.718281828459045)
`$MAXINT`	`int`	constant with the highest whole number (9223372036854775807, ie. 2^{63}-1)
`$MININT`	`int`	constant with the smalest whole number

---

[2] Pro verzi JDK 1.3 je použita knihovna Apache a pak je objekt typu `org.apache.oro.text.regex.Pattern`.

[3] Pro verzi JDK 1.3 je použita knihovna Apache a objekt je typu `org.apache.oro.text.regex.MatchResult`.

		(-9223372036854775808, ie. -2^{63})
$MAXFLOAT	float	the largest possible positive value (1.7976931348623157 E308)
$MINFLOAT	float	the smallest positive nonzero value (4.9 E-324)
$NEGATIVEINFINITY	float	negative infinity
$POSITIVEINFINITY	float	positive infinity
$NaN	float	value is not a valid number (Not a Number)

14.5 Ignored and illegal nodes

X-definition can also handle special cases of a node occurrence. Either the occurrence of described node can be set as illegal, or the occurrence of a node can be ignored and separated from the validation process and from the result object.

The following example lists the case where any text value of the Vehicle element located in front of the Vehicle element will be ignored (omittied) (ie, there may be a text but it is ignored. In addition, in the element Vehicle any possible occurrence of the attribute type will be ignored and any following element will be also ignored:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
     xd:name = "garage"
     xd:root = "Vehicles">

   <Vehicles>
      ignore /* the text here is ignored. */
      <Vehicle xd:script = "occurs 0..*"
         type   = "ignore; enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
         vtn    = "string(7)"
         purchase = "date()"
         manufacturer = "string()"
         model  = "string()" />

      <!— any element here will be ifgnored -->
      <xd:any xd:script="ignore" />
   </Vehicles>
</xd:def>
```

In the following example are declared the illegal nodes from the example above as illegal. The behavior is similar as if those nodes were not described:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
     xd:name = "garage"
     xd:root = "Vehicles">

   <Vehicles>
      illegal /* the text here is illegal. */
      <Vehicle xd:script = "occurs 0..*"
         type   = "ignore; enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
         vtn    = "string(7)"
         purchase = "date()"
         manufacturer = "string()"
         model  = "string" />

      <!— any element here will be illegal -->
      <xd:any xd:script="illegal" />
   </Vehicles>
</xd:def>
```

14.5.1 Spefifification of namespace for X-definition

➡️ 4.7 Sample of Complete X-definition

X-definitions can also be used to describe the X-definition itself. To enable such writing, you must specify the metaNamespace attribute from the X-definition namespace to specify the namespace URI, which will continue to be interpreted as the namespace for X-definition objects. In addition, it is possible to refer to these objects:

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

       <xd:def xd:name = "string()">
           <meta:any meta:script="occurs 1..*; options moreAttributes, moreElements, moreText"/>
       </xd:def>
</meta:def>
```

In the example described above, a document with a root element def and an attribute that contains any sub-elements, attributes, and a text node has been simplified. I.e. a part of the X-definitions has been defined.

14.6 Options

➡ 4.7.1 X-script of X-definition

X-Script makes it possible to customize the defaults of the X-definition processor using both the validation mode and the construction mode. This behavior can be edited using the X-Script, for example, how to process blank values, work with capital or small characters, the processing of white spaces characters, etc.. This which may be different for the different parts of X-definitions. The options section for X-Script is introduced with the options keyword followed by a comma-separated list.

The names of the options are in the following table:

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.
ignoreEntities	the unresolved external entities in DOCTYPE in parsed XML data are ignored. This option is possible to declare only in the x-script of X-definition header.
ignoreTextWhiteSpaces	superfluous spaces in text nodes are removed before further processing.
preserveAttrCase	the low/capital letters in the attribute remain unchanged. This is the default value.
preserveTextCase	the low/capital letters in the text node value 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 X-script of an element, all actions "forget" are ignored for all nested elements and their descendants.
ignoreEntities	the option can be declared only in the X-script of 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 X-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 X-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 X-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 X-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.xdef.org/xdef/3.2" xd:name="a" xd:root="E">

 <E xd:script="options noTrimText">

        <O xd:script="*" />

        optional string();

 </E>

 </xd:def>
```

the following input data will be validated incorrectly:

```
<E>
```

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

The reason for this is the fact that after the initial element <E> is an empty line (before the process of validation the empty lines are not removed due to the option "noTrimText"). The empty line, therefore, is seen as a value of a text node and the engine of X-definition expects the model of a text node 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>
```

14.7 Methods implemented in X-Script

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

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

14.7.1 Implemented general methods

All methods implemented in X-definition are listed in the following table:

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 X-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 X-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 X-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	Container	create attribute

	xquery expression s in the current context. The method can only be used in the "create" action in the X-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.		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 X-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 X-script attributes
getElement()	the result is the current element.	Element	Element, text node attribute
getElementName()	name of the current element.	String	anywhere
getElementLocalName()	name of the current element.	String	attribute, text node
getElementText()	returns a string with the concatenated text content of nodes that are direct descendants of the current element.	String	attribute, text node
getImplProperty(s)	returns the value of the property from the current X-definition that is named s. If the item does not exist, return an empty string.	String	attribute, text node
getImplProperty(s1, s2)	returns the value of the property s1 of X-definition, whose name is s2. If the appropriate X-definition or item does not exist will return an empty string.	String	attribute, text node
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	String	attribute, text

	node n (it can be either an element or attribute). If the namespace URI does not exist, returns an empty string.		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 X-script was read by a validation method). *Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.*	boolean	Only after the method type checking
getParsedBytes()	returns the value of byte array of ParseResult (if the previous X-script was a validation method). *Warning: this method must becalled immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.*	Bytes	Only after the method type checking
getParsedDatetime()	returns the value of Datetime from ParseResult (if the previous X-script was a validation method). *Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.*	Datetime	Only after the method type checking
getParsedDecimal()	returns the value of decimal number of ParseResult (if the previous X-script was a validation method). *Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.*	Decimal	Only after the method type checking
getParsedDuration()	returns the value of Duration of ParseResult (if the previous X-script was a validation method). *Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.*	Duration	Only after the method type checking
getParsedFloat()	returns the float value of ParseResult (if the previous X-script was read by a validation method). *Warning: this method must be called immediately after a call to the validation method, for example, in the section onTrue. If this condition is not met, then the result of the method is not defined.*	float	Only after the method type checking
getParsedInt()	returns the int value of ParseResult (if the previous X-script	int	Only after the

	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.*		method type checking
getParsedValue()	returns an object with the parsed value of ParseResult (if the previous X-script was read by a validation method). *Warning: this method must be called immediately after a 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 text of element
getTextContent()	returns a string with the text content of the current element and its descendants. Returns an empty string if no text is available.	String	text of element and descendants
getTextContent(e)	returns a string with the text content of the element e and its descendants. Returns an empty string if no text is available.	String	text of element and descendants
getText()	returns a string with the value of a current attribute or a text node.	String	attribute text text of element
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 then the method returns null.	Datetime	attribute, text node
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		attribute, text

	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.		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 result 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.	ParsedValue	Only after the

	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.		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 14.2.5.		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 the string s1 begins with the character "<", then the parser uses the value of s converted to UTF-8 byte stream.	Element	attribute, text node
xparse(s, null)	parses an XML document from a string s without an X-definition. Returns the root element. If the s is a string, describing an URL or a path to a file, the parser uses a stream created from s. However, if the string s begins with the character "<", then the parser uses the value of s	Element	attribute, text node

	converted to UTF-8 byte stream.		
XPath(s)	returns a Container created after the execution of the xpath expression s on the current element. If the actual element is null, it returns an empty Container.	Container	Element, attribute, text node
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 X-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

14.7.2 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

14.7.3 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 14.9.	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 14.9.	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

14.7.4 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

14.7.5 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

14.7.6 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

14.7.7 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

14.7.8 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

14.7.9 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

14.7.10 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

14.7.11 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

14.7.12 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.	

14.7.13 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.	

14.7.14 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.	

14.7.15　　　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

14.7.16　　　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

14.7.17　　　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 parameter s1 is set to s2.	Report
x.setType(s)	returns a new Report where the type of report is set to the value s. The value of s must be one of: "E" … error "W" … warning "F" … fatal error "I" … information "M"... message "T" … text	Report

14.7.18　　　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

14.7.19 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

14.7.20 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 with the result.	ResultSet

14.7.21 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	boolean

	case.	
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

14.7.22   Methods of objects of the type uniqueSet

Method name	Description	The result
	the instance of uniqueSet object is created by the declaration statement. See 4.10.1.	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	ParseResult

	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).	
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.	
x.checkUnref)	reports error message if in the table x are items which are not referred in the scope where the method is invoked.	
x size()	Returns number of items in the x.	int
x.toComtainer()	Returns Container created from items in x.	Container

14.7.23   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.	

14.8 Mathematical methods in X-script

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

14.8.1 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

14.8.2 Methods of mathematical functions (taken from java.math.BigDecimal)

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

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

14.9 BNF grammar

➡️ basic knowledge of BNF grammar

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.

In the following text, instead of the EBNF, only the basic abbreviation BNF is used.

14.9.1 BNF 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:

```
ruleName ::= BNF expresion
```

where on the left side of the operator ":: =" is the name of the rule, and on the right side a rule is described using the BNF expression.

Production rules are written into a special element for BNF grammars xd:BNFGrammar, which must be given as a direct descendant of element xd:def. The grammar element has a mandatory attribute name, which contains the name of the BNF grammar object, ie. it matches the BNFGrammar variable name, which has global X-script visibility.

A specific BNF grammar rule can be called from X-script by specifying the name of the BNF grammar as an object, and by calling the rule(String s), where the s parameter contains the name of the symbol to which it is referenced.

The following example demonstrates how BNF grammar can be used, for example, to validate the vehice registration number format, eg. for "1A23456":

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name  = "garage"
      xd:root  = "Vehicles">

 <xd:BNFGrammar name="vrnType">
     Cz ::= [0-9] [A-Z] [0-9]{4}
 </xd:BNFGrammar>

  <Vehicles>
     <Vehicle xd:script = "occurs 0..*"
        type   = " enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
        vrn    = "vrnType.rule('Cz')"
        purchase = "date()"
        manufacturer = "string()"
        model  = "string()" />
  </Vehicles>
</xd:def>
```

14.9.2 BNF 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 |

14.9.3 Set of characters

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

14.9.4 BNF quantifier (repetition of a rule)

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

A?	rule A is optional
A+	rule A may occur once or more times
A*	rule A may not occur or 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

14.9.5 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.

Example:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
      xd:name = "garage"
      xd:root = "Vehicles">

  <Vehicles>
      <Vehicle xd:script = "occurs 0..*"
          type   = " enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
          vrn    = "vrnBNF.rule('Cz')"
          purchase = "date()"
          manufacturer = "string()"
          model  = "string()" />
  </Vehicles>

  <xd:BNFGrammar name="vrnBNF">
      Digit ::= [0-9]
      Letter ::= [A-Z]
      Cz_new ::= Digit Letter Digit{5}
      Cz_old ::= Letter{2,3} Digit{4}
      Cz ::= Cz_new | Cz_old
  </xd:BNFGrammar>
</xd:def>
```

14.9.6 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.

14.9.7 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 org.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 org.xdef.sys.SDatetime value.
$datetime(mask)	parse date and time according to mask in the argument . Parsed text is put to the internal stack as a org.xdef.sys.SDatetime value.
$day	parse day according to ISO specification. Parsed text is put to the internal stack as a org.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 org.xdef.sys.SDuration value.
$error(s)	writes the error message with the parameter to the reporter and returns that the rule fails.
$eos	checks if the end of source was reached.
$find(s)	skips characters from actual source position until it reaches a string from argument (the fails if the string from argument was not found)
$findOneOfChars(s)	skips the position from the actual position to a character from the string from the argument (returns true if the character was found and false if not)
$float	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 org.xdef.sys.SDatetime value.
$monthDay	parse month and day according to ISO specification. Parsed text is put to the internal stack as a org.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 org.xdef.sys.SDatetime value.
$stop	parsing is stopped at the position of the rule.
$stop(s)	parsing is stopped at the position of the rule and the argument is stored to the internal stack.
$whitespace	parse whitespace according to W3C specification
$year	parse year according to ISO specification. Parsed text is put to the internal stack as a org.xdef.sys.SDatetime value.
$yearMonth	parse year and month according to ISO specification. Parsed text is put to the internal stack as a org.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 don't parse any actual text. However, it forces the parsing process fails at the actual position.

14.9.8 Implemented methods for handling the internal stack

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

$clear	clears the internal stack
$info	pushes to the internal stack the information containing name of the actual rule and source position (line, column etc.)
$info(params)	pushes to the internal stack the information containing name of the actual rule and parameter list in parenthesis and source position (line, column etc.)
$pop	removes the item from the top of the internal stack
$push	pushes to the internal stack the text parsed by the actual rule.

$push(arg)	puts a value from the argument to the top of the internal stack (the parameter can be specified in the declaration section). If no argument is specified it is pushed to the internal stack the parsed text.
$rule	pushes to the internal stack the name of the actual rule. After the rule name are positions where parsing of the rule started parsing and where parsing endned (separated by the space).

14.9.9 Externally implemented rules

In BNF grammar, it is possible to refer to the external rules that can be implemented in external Java programs as methods. An external rule can be defined using the `%define` command. Behind it is the symbol $ and a short **alias** name followed by a colon followed by the name of the external method (in a fully qualified form). If the external method has parameters, its values (not types!) can be described in brackets. Parameters are separated by a comma. Values can be int, float, string, datetime, duration. The `%define` commands must be given before the next grammar description.

In the java program, an external method is defined as a static and must retunr a boolean value that indicates to the BNF grammar engine whether the validated (parsed) string is true or does not match the condition. The method must also have a `BNFExtMethod` type parameter through which the `getParsedString()` the method can obtain a string that is validated (parsed).

The following example shows the possibility to verify the format of the vehicle registration mark using an external rule, ie using an external method:

```
<xd:def xmlns:xd = "http://www.xdef.org/xdef/3.2"
     xd:name  = "garage"
     xd:root  = "Vehicles">

  <Vehicles>
     <Vehicle xd:script = "occurs 0..*"
        type   = " enum('SUV', 'MPV', 'personal', 'truck', 'sport', 'other')"
        vrn    = "vrnBNF.rule('anyVrn')"
        purchase = "date()"
        manufacturer = "string()"
        model  = "string()" />
  </Vehicles>

  <xd:BNFGrammar name="vrnBNF">
     %define $Us: $example.MyRules.checkUs("Washington")
     %define $Ru: $example.MyRules.checkRu()

     Digit ::= [0-9]
     Letter ::= [A-Z]
     Cz_new ::= Digit Letter Digit{5}
     Cz_old ::= Letter{2,3} Digit{4}
     Cz ::= Cz_new | Cz_old
     anyVrn ::= Cz | $Us | $Ru
  </xd:BNFGrammar>
</xd:def>
```

Example of Java code with user BNF methods:

```
package example

import org.xdef.sys.BNFExtMethod;

public class MyRules {

    public static boolean checkUs(BNFExtMethod p, String s) {
        StringParser p = p.getParser();
        switch(s) {
            case "Washington":
                ...
                retun true;
                ...
                retun false;
            ...
            default: throw new RuntimeException("Incorrect parameter");
        }
        return false;
    }

    public static boolean checkRu(BNFExtMethod p) {
        ...
    }
}
```

15 Index

Related documents

[1] XML 1.0, W3C Recommendation, 26.11.2008. dostupné z URL http://www.w3c.org/TR/REC-xml

[2] W3C XML Schema, http://www.w3c.org/TR/xmlschema-1, http://www.w3c.org/TR/xmlschema-2, 28.10.2004

[3] X-definition 3.1. The language description (preliminary draft) http://xdef.syntea.cz/tutorial/en/userdoc/xdef3.1_eng.pdf

[4] X-definition 3.1. Introduction to the construction mode http://xdef.syntea.cz/tutorial/en/userdoc/xdef3.1_construction_mode_eng.pdf

[5] X-definition 3.1. Java programming guide http://xdef.syntea.cz/tutorial/en/userdoc/xdef3.1_Programming_eng.pdf

[6] X-components 3.1. User manual

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