Xdefinition 2.1

Václav Trojan

email: vaclav.trojan@xmlprague.cz

Παντα ρει (Everything is in a state of flux)

Herakleitos of Ephesos (535 - 475 B.C.)

Abstract

The paper describes Xdefinition 2.1 as an integral instrument for the design and implementation of projects with XML objects. In our concept of Xdefinitions we respected as much as possible the requirement of application of XML documents in the whole process throughout the life cycle of the information system development. The principal requirements were **comprehensibility** for all participants of the individual stages of IS formation as well as the **binding character** of the description and its **modularity**. By comprehensibility we understand that the description must be short and clear not only for the IT specialists but also for the wider range of other experts participating in the project, particularly in the analytical stages. By the binding character we understand exactness of the description and applicability of the description for the machine processing: description of data may include also the directives for data processing or data generation. The modularity allows decomposition of objects descriptions and configuration of the versions of their structure in the complicated and permanently changing environment of the distributed system. Xdefinitions 2.1 also allow defining bindings of the objects to different situations in the course of their processing.

Xdefinitions itself are XML objects that may be used also as a meta-language to describe generally languages above XML.

1. Why Xdefinitions 2.1?

XML data may be validated with DTD, RELAX NG or XML schema or Schematron. XML objects can be generated by using for example XSLT transformation and, in the data processing; we cannot then avoid working with XML objects also with a universal programming language. In communication with analysts and external participants we must describe the objects structures in a language they understand. Thus the project inevitably involves several different forms of the description of identical objects. The problem is that such a system is difficult to maintain and its integrity might be at risk. It is not easy to make sure that the changes in one form of description are reflected in relevant sites and in different languages. That is why in the version 2.1 of Xdefinitions we emphasised maximum comprehensibility of the description, simple maintenance of objects and possibility of automatic transformation of data to the language of Xdefinitions. We respected also easiness of implementation.

In the continuously changing world of real projects which involves work with a large number of XML objects and their generation, processing and modification or transformation at many often very different sites and in various situations it is useful to minimise the number of the forms of description. Such tool should be intuitively comprehensible and able to:

- Describe the valid structure as much as simple and exactly
- Describe commands for processing of objects in various sites (to minimize dependence of program code on the data structure)
- Describe generation of objects and guarantee validity of the generated objects
- Decompose the object description into more simple segments which then may be used to compose and newly configure more complicated objects
- Ensure easy implementation to different platforms and environments.

To illustrate the philosophy of Xdefinitions concept we will now try to describe the work with a simple XML object. Let us take a simple XML object containing information about a book:

```
<book category="children">
  <title>Harry Potter</title>
  <author>J K. Rowling</author>
  <IBSN>9780439887458</IBSN>
  <price>39.99</price>
</book>
```

First we will try to design a formula for generating such an object. Let us imagine we have a method which for example retrieves from the database a value according to the name of a column in the line of a table. The name of the column will be the parameter of this method. For example invoking the method getItem("cover") will return the string "children". We can simply create the formula for the generation of our object by describing how the values should be filled in the XML object. So instead of values in our object we shall simply specify the methods with relevant parameters that will return the relevant values:

```
<book category="create getItem('category')" >
  <title> create getItem('title') </title>
  <author> create getItem('author') </author>
  <IBSN> create getItem('IBSN') </IBSN>
  <price> create getItem('price') </price>
</book>
```

The processor of this description will process the above formula and compose the subsequent XML objects by creating relevant elements and filling in the values obtained by invoking the functions specified after the key word "create". The result of the invocation will then be XML object composed in accordance with the formula above. Note that the method may be an XPath function or an external method (however, it can be a built-in method of the processor of Xdefinitions). The external program doesn't know anything about the structure of data.

And now let us imagine a different situation when we have an XML object we want to validate. Let us presume we have methods parsing data in the validated object and returning a boolean value "true" if the result is correct or value "false" if the result is incorrect. The formula for validation may then look like:

```
<book category="optional enumeration('children','adult','unknown')">
  <title> required string() </title>
  <author> required string() </author>
  <IBSN> optional numeric(8,15) </IBSN>
  <price> required decimal(4,2) </price>
</book>
```

As we can see instead of values in this case we have described specification whether the value is compulsory or optional and we have described the validation of values by methods verifying the formal correctness of values (type of value). The verification of the type of values may thus be understood again as a specification of a method that will return boolean value depending on whether the value is valid. In this case the processor will proceed in the reverse manner than in our first example. It will process (parse) the input data and it will check up the format of values by invoking relevant methods according to the above description. In case the compulsory value is missing or if the validation method returns false value, the processor will report an error and the object will not be recognised as valid.

Both above mentioned formulas can be merged into one:

```
<book category="optional enumeration('children','adult','unknown');
                create getItem('category')" >
  <title> required string(); create getItem('title') </title>
  <author> optional string(); create getItem('author') </author>
  <IBSN> optional numeric(8,15); create getItem('IBSN') </IBSN>
  <price> required decimal(4,2); create getItem('price') </price>
</book>
```

In the validation regime we shall ignore the part describing the generation of XML object.

When processing XML objects, it may be useful to define instructions determining what should happen in various situations that could occur in the course of processing and insert them in the description. In such a case we may for example require certain actions such as e.g. error notification, deposition of values to the database, etc. So let us now insert in our description instructions for actions to be executed in certain situations. For example:

```
<book category="optional enumeration('children','adult','unknown');
                default setValue('unknown')">
   <title> required string(); onAbsence error('Missing title') </title>
   <author> optional string(); </author>
   <IBSN> optional numeric(8,15); onError error('Incorrect IBSN');
          onAbsence error('Missing IBSN')
   </IBSN>
   <price> required decimal(4,2); onError error('Incorrect price');
           onAbsence error('Unknown price')
   </price>
</book>
```

Bellow we have the formula containing all the above information:

```
<book category="optional enumeration('children','adult','unknown');
                default setValue('unknown');
                create getItem('category')">
   <title> required string();
           onAbsence error('Missing title');
           create getItem('title')
   </title>
   <author> optional string();
            create getItem('author')
   </author>
   <IBSN> optional numeric(8,15);
          onError error('Incorrect IBSN');
          create getItem('category')
   </IBSN>
   <price> required decimal(4,2);
           onError error('Incorrect price');
           onAbsence error('Unknown price');
           create getItem('category')
   </price>
</book>
```

It is clear this formula may be used in the mode for XML object generation (create) or the mode for its processing (and/or validation). In the processing mode we shall ignore the formula parts related to "create" and in the "create" mode we shall ignore the instructions concerning processing.

Such a formula describing methods to be executed in different situations enables us to describe not only the structure but also the way of processing XML data and this way to minimize number of forms of description. Note that it also minimizes the dependence of programming code on the structure of data.

2. Models of elements

As we have demonstrated, the description of an element in Xdefinition contains in the sites of data values the data about their occurrence, generation instructions and potentially also directions for data processing. The complete element description in Xdefinitions is called ***element model***. For the description we use a special language: Xdefinitions ***script***. In an element model the script describes values or text nodes in sites where they may occur. Element model is thus intuitively comprehensible because it has a similar structure as the data it describes.

The validation of the types of values in Xdefinitions is performed by so called *validation methods* the result of which is a boolean value containing information about the validity of the parsed or created object. Since these methods may have parameters which enable further specification of the required features of values, these parameters can be understood as restrictions for a certain type. For example formula "numeric(8, 15)" means sequence of minimum 8 and maximum 15 digits. The whole part of the script describing request for object occurrence and validation method is called ***validation section*** of the script.

It is not sufficient for the description of element features to describe attribute values and text nodes and that is why Xdefinitions include a special attribute "script" (from Xdefinitions' namespace). The script enables us to describe element features e.g. its occurrence, invocation of methods in various situations, etc. It also describes occurrences of child elements in the element model. The following example illustrates the application of the script in case we want to describe that a certain child element can be omitted or that it can occur more than once:

```
<author xd:script="occurs +"> required string </author>
```

```
<IBSN   xd:script="occurs ?"> optional decimal(8,15) </IBSN>
```

The occurrence description may explicitly state the minimum and maximum number (unbound limit we can specify as “*”):

```
<author xd:script="occurs 1,*"> required string </author>
<IBSN   xd:script="occurs 0,1"> optional numeric(8,15) </IBSN>
<price  xd:script="occurs 1"> required decimal(4, 2) </price>
```

Default occurrence if the specification is omitted is equal to "occurs: 1" for elements or “required for attributes”.

To minimize the specification we can omit keywords "occurs", "required" and "optional". Specification of the occurrence may be reduced to characters ‘?’, ‘+” and ‘*’. Also, if a method has the empty parameter list it is possible to omit brackets. The example or reduced form will be:

```
<book category="? enumeration('children','adult')" >
   <title> string </title>
   <author xd:script="+"> string </author>
   <IBSN   xd:script="?"> ? numeric(8,15) </IBSN>
   <price> decimal(4,2) </price>
</book>
```

3. Groups of objects

Description of more complicated structures requires description of the groups of objects and potentially their variations. For this purpose we use auxiliary elements from Xdefinitions’ namespace into which we insert the group items that is being described. The elements inside the group form description of the nodes that belong to the group (e.g. elements, text values, processing instructions, comments or groups again). Xdefinitions enable us to describe the following types of groups:

1) “**xd:sequence**” groups describes the set of items the occurrence of which corresponds with the sequence in the group description.
2) “**xd:choice**” group describes the set of elements from which a certain variation has been selected.
3) “**xd:mixed**” “Mixed” group is useful if we do not care about the sequence of nodes inside the group – all node permutations are valid (an analogy to “interleave” in RELAX NG)

Script can be also used to describe occurrence inside a group:

```
<foo>
   <xd:choice>
      <a/>
      <b xd:script="occurs 3"/>
      optional string()
   </xd:choice>
</foo>
```

The following versions of element “foo” correspond to the above description:

```
<foo><a/></root>

<foo><b/><b/><b/></foo>

<foo>text</foo>

<foo/>
```

Similarly as in case of elements, the attribute “script” may be specified also in the description of groups. For example:

```
<root>
   <xd:sequence script="occurs *">
      <a/>
      <b/>
   </xd:sequence>
</root>
```

The above description corresponds to any number of element sequences of "foo" a "bar".

4. Events and actions

As we have demonstrated, in Xdefinitions we can specify invocation of methods to be executed in various situations or events in course of object processing. For example in course of data processing it may be important to report detail errors (not only formal validity of an object). It may be important also to describe object processing (storing to the database, etc.). Thus in Xdefinitions we enter the names of ***events*** and specification of the method to be executed. This method is called ***action***. The relevant actions are specified after the name of the event.

Xdefinitions recognise a number of different events. In some cases an action connected to an event must return a value of a certain type, in other cases no value needs to be returned. (In the Xdefinition 2.1 the user may also define aliases for the predefined events to be able to specify which actions are executed or ignored on different sites.)

Various events may occur in the course of XML objects processing. Bellow we have listed the principal ones.

1) Events defined for all the objects:

init – occurs at the start of further processing of an object; the action does not return a value

create – occurs only in the "create" mode, action for attributes and text nodes returns value string; for elements it is the iterator and we shall discuss "create" mode later in greater detail

finally – occurs only after the whole object has been processed (allows to execute the action following the prior processing of an object, when the whole object, the errors that occurred, etc. are already known; the action does not return a value

onAbsence – object in data is missing; the action does not return a value

2) Events defined only for elements

match – invoked prior to the model application (before init). The action must return boolean value and if the value is "true", the model is applied, otherwise the model is not applied to the given element (we shall return to this situation in the discussion)

onStart – occurs when processing of element starts (after init and after the attributes were validated, before processing of child nodes starts); the action does not return value

onExcess – occurs when the number of objects exceeds permitted number of occurrences; the action does not return value

forget – invoked after all events, even after finally. Specification of the event does not expect any other action. It just releases all data of the processed element from RAM of the computer. Specification of this event enables to process very large XML objects.

3) **Events defined only for text values (attributes and text nodes):**

passed – the action is executed only if the validation method returns value "true" ; the action does not return a value

onError – the action is executed only if the validation method returns value "false"; the action does not return a value

default – occurs when the value is optional and it is missing; the action must return value of the string type

4.1. Event „match"

Xdefinitions allow - in the element or group script – to describe a situation when the application of an element model depends on yet another condition. This situation can be described by the means of "match" event. In this case the result of the action must be a boolean value. This action is executed prior to the other actions (even prior to "init" action). It is in fact a filter determining whether the relevant model will be applied. If the result of the action is "false", the application of relevant model is skipped (to the given object) and if the result is "true", the model is accepted. With the event "match" we can to describe different variants of models The following example describes various structures of elements that have the same name but whose structure differs depending on the value of the "type" attribute:

```
<xd:choice>
```

```
   <Shape type="string" xd:script="match xpath(@type='circle')">
      <r> float </r>
   </Shape>
   <Shape type="string" xd:script="match xpath(@type='rectangle')">
      <a> float </a>
      <b> float </b>
   </Shape>
</xd:choice>
```

4.2. Event “create”

In the opening section we have mentioned the “create” event. As we have already said, all actions connected to “create” events are invoked only when we are generating new XML documents (this mode is called ***create mode***). Otherwise those actions are ignored. And on the contrary the other actions are ignored in the create mode (except for the validation section the execution of which can be set by a parameter). The actions for create event are used to describe how to generate an XML object. An action for create must return a value which allows creation of the relevant object. In the case of attributes or text values the result of the action must be convertible to a character data. In the case of element or groups the result of the action may be an object convertible to the iterator type which is able to return one or more then one items – child nodes (or none item). This iterator has two methods: hasNext() and getNext() and it is from some types of objects generated automatically:

- element, text node (iterator returns just one item)
- nodeList (iterator returns items from the list).

In the case no “create” action is described, a default action depends on the context in which it is invoked (e.g. the element of a relevant name is automatically created). Context is generally a part of the XML object tree. Two situations may occur: no context is available or a context from the previous action is available.

One possible variant of “create” mode execution is when we add as the input parameter a XML object according to which the result is generated. In such case “create” mode may be perceived as a transformation of the input XML object (similarly as XSLT). In this case create actions can work with a context which may be set by invoking XPath on the current context. For this purpose we use implemented method “from” which executes XPath on the current input context.

Unlike XSLT the Xdefinitions guarantee that the generated object is valid XML (it corresponds to the relevant Xdefinition). To understand following example we should know that if the result of the operation in action create” is a NodeList, the create action creates the iterator, which gradually applies the items of this list to the element description. Bellow we can see the “create” action with XML object on the input. Let us have an input XML object:

```
<monarchs>
   <monarch>
      <name>George I</name>
      <reigned>
         <start>1714</start>
         <end>1727</end>
      </reigned>
   </monarch>
   <monarch>
      <name>George II</name>
      <reigned>
         <start>1727</start>
         <end>1760</end>
      </reigned>
   </monarch>
</monarchs>
```

Xdefinition:

```
<governors>
   <king xd:script="*; create from('//monarchs/monarch')"
         name="string; create from('name/text()')"
         from="int; create from('reigned/start/text()')"
         to  ="? int(); create from('reigned/end/text()')" />
</governors>
```

The result of create action will then be:

```
<governors>
   <king name="George I" from="1714" to="1727"/>
   <king name="George II" from="1727" to="1760"/>
</governors>
```

4.3. Alias events

Users may define for the events in Xdefinitions their own alias names. These user defined events then mean that actions attached to them can be under certain conditions invoked and under different conditions ignored. These conditions can be specified with the activation of Xdefinitions processor. In this way we can modify Xdefinitions behaviour in various situations or at various sites.

5. Declaration of the types of values

In Xdefinitions types of values can be also declared separately from the model. For this purpose we use an auxiliary element “type” from the Xdefinitions namespace:

```
<xd:type name = "currency">
   <validate>enumeration('GBP','USD','EUR')</validate>
   <onError>error('Incorrect currency')</onError>
</xd:type>
```

The reference to the declared type is formally similar to the specification of validation method:

```
 <price> currency </price>
```

5. Namespaces in Xdefinitions

Xdefinitions – similarly as RELAX NG – treat namespaces in the intentions of Namespace specification in XML 1.0 as with strings to distinguish them from local names. In Xdefinitions it is possible to define an arbitrary number of namespaces. The following example shows an Xdefinition describing a model with two different namespaces (prefixes “a” and “b”):

```
<xd:def xmlns:xd="http://www.syntea.cz/xdef/2.1"
        xmlns:a ="http://www.a.com"
        xmlns:b ="http://www.b.com">
   <a:root>
      <b:foo a:id="string" />
   <a:root>
</xd:def>
```

6. References

Within an Xdefinition mutual references to models can be made using “ref” construction. This case is illustrated by the following example:

```
<xd:def xmlns:xd = "http://cz.syntea.xdef/2.1">

  <person>
     <firstName> string </firstName>
     <lastName> string </lastName>
  </person>

  <family>
     <mother      xd:script="ref person"/>
     <father      xd:script="?; ref person"/>
     <xd:mixed>
        <daughter xd:script="*; ref person"/>
        <son      xd:script="*; ref person"/>
     </xd:mixed>
  </family>
```

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

If we want to enable also references to groups, it is necessary to perceive such groups similarly as the element models, i.e. to record them as direct child nodes of Xdefinitions root and to give them a name. We call such kind of groups the ***group models***. Therefore an attribute “name” which serves for a reference must be specified in the group model. Specification to the referred group is written in the script attribute on the site of reference:

```
<xd:def xmlns:xd="http://cz.syntea.xdef/2.1">
   <xd:choice name="foobar">
      <foo/>
      <bar/>
   </xd:choice>

   <root>
      <xd:choice script="ref foobar"/>
   </root>
</xd:def>
```

Xdefinitions also enable making references to the description of attributes:

```
<xd:def xmlns:xd="http://cz.syntea.xdef/2.1">
   <xd:attr name="myAttr"
            script="? int(1, 999)" />

   <root attr1="ref myAttr" />
</xd:def>
```

6.1. Model extension and model modification

We can add to the referred model the attributes and/or we can to add to it the child nodes simply by declaring them:

```
<shape>
   <x> float </x>
   <y> float </y>
</shape>

<rectangle xd:script="ref shape" a="float" b="float" />

<circle xd:script="ref shape">
  <diameter> float </diameter>
</circle>
```

The element „rectangle“ is extended by the attributes „a“ and „b“ and the element „circle“ is extended by the child element „diameter“.

The properties of referred model can be also redefined (including actions). Sometimes it we need to skip or omit some object. For this purpose we can redefine the specification of the occurrence by the keyword „ignore“ (the occurrence of the object is ignored) or illegal“ (the occurrence of the object is forbidden):

```
<square xd:script="ref rectangle" b="illegal" />
```

The attribute „b“ which is declared in the model „rectangle“ is illegal in model „square“.

6.2. Element "any", otherElement, otherAttribute

To enable description of a situation when any element can occur at any site, Xdefinitions introduce a special element “any” from the Xdefinitions namespace. We can describe in the script of this element what should happen. Description of a case when we want to say that also other than specified elements or attributes may occur in the element model will be described in attributes “otherElement” and “otherAttribute” from the Xdefinitions namespace.

7. Processing instructions and XML document

In Xdefinitions it is possible also to describe occurrence of the processing instructions on specified position:

```
<xd:processingInstruction script="onAbsence genProcessingInstruction('xx', 'yy')"
    name="checkName"
    value="checkValue" />
```

When processing XML objects the Xdefinitions processor must be instructed which model is a document root. The document is described in Xdefinitions by means of an auxiliary element from the Xdefinitions namespace.

```
<xd:document>
  <root>
    ...
  </root>
</xd:document>
```

A choice group enables specification of more variants of root elements in the document:

```
<xd:document>
  <xd:choice>
    <foo/>
    <bar/>
  </xd:choice>
</xd:document>
```

8. Recursion in Xdefinitions

References in Xdefinitions can be recursive and we can thus describe complex languages generally above XML. Simple example of recursion in the model:

```
<foo>
   <bar>
      <foo xd:script="?; ref foo" />
   </bar>
</foo>
```

Note that the inner „foo“ element has occurrence „?“, otherwise recursion would be infinite! Valid structures are:

```
<foo><bar/></foo>
<foo><bar><foo><bar/></foo></bar></foo>
<foo><bar><foo><bar><foo><bar/></foo></bar></foo></bar></foo>
...
```

The recursive references we can specify also in the groups:

```
<xd:choice name="command">
   <if>
      <condition xd:script="ref expr"/>
      <xd:choice script="ref command"/>
      <else xd:script="?">
         <xd:choice script="ref command"/>
      </else>
   </if>

   <assign ...>
   ...
</xd:choice>
```

9. Collections of Xdefinitions and mutual references

An Xdefinition can contain more element models and/or groups. However, it is also possible to compose a pool of Xdefinitions, in which in the individual Xdefinitions mutual references can be made to objects they contain. The mutual references are composed from the names of Xdefinitions and it is separated from the names of referred objects with the character “#”. Such a collection of Xdefinitions may be perceived as a ***collection***. In this way it is possible to describe projects working with a large number of XML objects. The collection item can be an Xdefinition or another collection.

For example:

```
<xd:collection xmlns:xd="http://cz.syntea.xdef/2.1">
   <xd:def name="site1">
      <foo xd:script="ref site2#bar" />
   </xd:def>

   <xd:def name="site2">
      <bar> string </bar>
   </xd:def>
</xd:collection>
```

In the Xdefinition which is situated in „site1“ the reference points to the model in the Xdefinition in the site „site2“. Xdefinitions may be stored in different files in different sites and from these building stones the collection can be composed. The parts can be specified by attribute “include” in the collection.

Let us have two Xdefinitions with the description of an object “Person” stored in different files: in the first case describes data with the name and address and the second version contains only the numerical identifier to the database:

1) file at site “http://site1.com/person.xd“:

```
<xd:def xmlns:xd="http://cz.syntea.xdef/2.1" name="person">
  <Person name="string" address="string" />
</xd:def>
```

2) file at site „http://site2.com/person.xd“:

```
<xd:def xmlns:xd="http://cz.syntea.xdef/2.1" name="person">
  <Person id="integer" />
</xd:def>
```

At local file let us have an Xdefinition describing an object “Family” in a file „/myfolder/family.xd“:

```
<xd:def xmlns:xd="http://cz.syntea.xdef/2.1" name="family">
   <family>
      <mother   xd:script="ref person#Person" />
      <father   xd:script="?; person#Person" />
      <xd:mixed>
         <daughter xd:script="*; ref person#Person" />
         <son      xd:script="*; ref person#Person" />
      </xd:mixed>
   </family>
</xd:def>
```

The structure of the individual family members will depend on whether we compose the collection from Xdefinitions from file described in paragraphs 1) or 2). We can specify collection composed with

```
<xd:collection xmlns:xd="http://cz.syntea.xdef/2.1"
    include="file://folder/family.xd, http://site1.com/person.xd" />
```

or

```
<xd:collection xmlns:xd = "http://cz.syntea.xdef/2.1"
    include="file://folder/family.xd, http://site2.com/person.xd" />
```

The fact that the Xdefinitions collection may be composed from different parts makes it possible to describe in detail the variations of objects and their behaviour at different sites of the project.

10. Macros

The flexibility of description is further extended by the possibility of macros declaration and specification. The macro expansion is performed in a special pre-processor prior to further processing of Xdefinitions. Macros specification can be anywhere in the script. Macros can have parameters and they can be nested. The possibility of placing macros in a separate Xdefinition further extends the possibility of object modification.

Finally

In this presentation we have managed to describe only some of the basic features of Xdefinitions. We did not go into details of types descriptions, for example the possibilities of working with values as with keys, of descriptions of canonical forms, declarations of script variables and their applications, etc. We have introduced Xdefinitions as an example of an integrated tool for the work with XML objects in real projects which can be easily implemented into various environments and platforms. The important feature of Xdefinitions for the purpose of specification of the interface with XML objects is their comprehensibility and ease of design. Xdefinitions enable not only the description of the structure of objects and their validation but also programming of these objects including detailed control of error situations. The possibility to describe behaviour of the process on the place where the values occurs enables a special way of programming. The code of such programs is highly independent on the structure of data. These features allow using Xdefinitions as the powerful tool for data validation. Xdefinifions are proved to be able to describe complex processing of XML data in distributed IT systems.

For more information see www.syntea.cz/xdef/2.1/info