Metsäteho Report 131
15.5.2002

Towards StanForD-XML

Juha-Antti Sorsa

Towards StanForD-XML

Juha-Antti Sorsa

Metsäteho Report 131
15.5.2002

Funded by the following shareholders: Koskitukki Oy, Metsähallitus, Metsäliitto Osuuskunta, Pölkky Oy, Stora Enso Oyj, UPM-Kymmene Oyj, Vapo Timber Oy ja Yksityismetsätalouden Työnantajat r.y.

Keywords: StanForD (Standard for Forest Data and Communications), XML (eXtensible Markup Language)

Helsinki 2002

Contents

INTRODUCTION

The Standard for Forest Data and Communications (StanForD) was set up by a group of specialists in Sweden in 1987. This standard was then further developed and adopted for use in Finland. StanForD is now in common usage for data communication in various forest-industry companies.

The time is now right for the development of a new technical version of the existing standard. The new standard will be based on all the work done so far to develop StanForD. However, considering the present state of software development, the technical implementation of the existing standard is a little out-dated, so the new standard will be based on XML technology – the most important new technology for developing standards for data for various types of application.

This document is intended to be the first part of a new standardisation process. The aim is to assess structural and technical differences between the existing standard and the new standard. Design issues related to the new standard will also be examined.

In this document, the name “StanForD” is used to mean the existing standard, while “StanForD-XML” is used as the working name for the new standard. Readers should have some prior knowledge about XML and XML Schema, as well as a good understanding of StanForD. Suitable introductions to XML are recommended in a short bibliography in Appendix 1.

1 DATA

1.1 Data in StanForD

The data in the StanForD file consists of variables. All variables have four components, which have to occur in the following order: variable number, type number, data and end character. For example the following variables are for a tree species name and code in a PRD-file:

```
120 1 Pine~120 3 1~
```

In this example 120 is the variable number. The next numbers are the types (1 and 3) and the data fields are the string "Pine" for the first variable, and the number 1 (the tree species code for pine) for the second variable. The last character ~ is the end character. All variables except checksums (which omit the type number) have this same basic structure. The data may be numeric or alphanumeric and it may comprise one or several elements. There are strict rules on how elements and components are separated, and on which delimiter characters have to be used. These rules are different for numeric and alphanumeric data.

1.2 Data in StanForD-XML

The previous example might be coded in the following way if we use StanForD-XML.

```
<TreeSpecies>
      <TreeSpeciesName>Pine </TreeSpeciesName>
      <TreeSpeciesCode>1</TreeSpeciesCode>
</TreeSpecies>
```

The XML code above declares the TreeSpecies element. That element has two child elements TreeSpeciesName and TreeSpeciesCode which contain the actual data. This kind of element declaration is the basic structure for building up XML files. The actual data item in any element is enclosed using tags. The first start tag in this example is `<TreeSpecies>` and the corresponding end tag is `</TreeSpecies>`. We can define tag names however we wish, but it is natural to choose readable ones. So there is no longer any need for variable and type numbers as there is in StanForD data.

The next subsection covers the naming issues for XML documents in more detail.

1.2.1 Naming issues in StanForD-XML

One design issue when new XML standards are developed is how elements and attributes are named. If names are properly selected, and are long enough to be readable and understandable, the XML file itself is a good

document for the information it presents. Good layout for names also increases the readability of the XML file. One way to form a name for a tag is to combine the words and change all the first letters of the words into uppercase. For example, an element describing the properties of the tree species could have a tag named <TreeSpeciesCode>.

Other important considerations when selecting names are that they should be both accurate enough inside documents, and unambiguous between documents. An example may help to clarify the first issue: If we name the tag for tree species code simply as <Code> , it would also be possible to name both the assortment code and the stem type code as <Code>. This is technically acceptable, and it is always possible to deduce which code it actually is from the context. However this solution is not efficient if we want to find certain elements of an XML document. If names are more specific, finding them is much faster and more straightforward.

Element and attribute names in XML documents can be made globally unambiguous, so that it is always possible to distinguish elements or attributes from different documents that have been given same name. This is achieved by binding the names into a namespace that is unambiguous itself. Technically a namespace is a globally unique string that can be used to distinguish names in different XML documents. Web-page addresses are often used as namespace names, since web-sites have unique names. In addition those addresses are also used as real locations where the schema definitions of the XML documents of the namespace concerned can be found. For example the namespace for the new standard might be something like "www.skoforsk.se/StanForD_XML". Users can always find up to date versions of the StanForD-XML schema files at this address.

1.2.2 Elements vs. attributes

The actual data in XML documents can be located either in elements or in attributes. The tree species data was implemented using the elements described in the beginning of this chapter. If we want to implement it using attributes instead of elements, it can be done in the following way:

```
<TreeSpecies TreeSpeciesName = "Pine" TreeSpeciesCode = "1"/>
```

Now we have only one element TreeSpecies and it has no child elements. Instead it has two attributes: TreeSpeciesName and TreeSpeciesCode. The actual data is nevertheless the same as before.

There are no clear rules about how we should divide our data between elements and attributes. Technically speaking, attributes are a more compact and efficient way to store data but they can be only used for simple, non-structured data. If there is a need to implement hierarchical structures in XML documents, elements have to be used. Additionally if there are a lot of attributes, the readability of the XML document typically decreases. Some documents might provide data that presents meta-information as attributes,

and other data as elements. However this kind of semantic separation of data is quite difficult to achieve in practice. One thing that must be remembered when using attributes is that there are no order relation between them. We can define whatever order for attributes in a schema definition but in actual XML documents they might be in any order. Elements instead must be exactly in that schema definition order.

The same information content for XML documents could be achieved without attributes, by using elements alone. This might simplify application development work, because all the data in XML documents can then be processed in the same way. This design decision has been followed in most the example implementations of StanForD-XML documents. However there is one schema definition where all simple data is implemented using attributes so that it can be compared to pure element based implementations. All examples are described in Appendix 2.

2 DATA TYPES

The crucial difference between StanForD and StanForD-XML concerns type systems. In fact it would be misleading to say that StanForD has a type system, because it only provides some simple types for variables. In contrast, StanForD-XML allows the opportunity to define documents using XML Schema. It provides a very powerful set of predefined data types, and a way to define our own data types. In XML, data types are described as simple when they have no attributes and elements in their content. Other data types are known as complex data types.

2.1 Simple data types

StanForD defines four simple data types: string[80], long string[8], integer[2 bytes], and long integer[4 bytes]. Real numbers are presented by using two variables, one for the integer part, and one for the numerator (decimal places).

In StanForD-XML it is possible to define more extensive sets of simple types. XML Schema has 45 built-in simple types that can be used as such, or by deriving new simple types from the existing types. We can for instance create our own simple type for tree species code using the following schema definition:

```
<xsd:simpleType name = "treeSpeciesCodeType">
      <xsd:restriction base = "xsd:integer">
                <xsd:minInclusive value = "1"/>
                <xsd:maxInclusive value = "4"/>
      </xsd:restriction>
</xsd:simpleType>
```

The definition says that we have data type named "treeSpeciesCodeType" and its base type is an integer. Its value range has also been restricted so that it can contain only the values 1, 2, 3, and 4. This kind of schema definition can be considered as formal documentation for our data. Moreover, the significant advantage is that data can always be checked programmatically against this definition. This kind of program support is not offered by the current standard, and this is one reason why implementations do not obey the standard, as they should.

2.2 Complex data types

In StanForD-XML it is possible to define complex types, i.e. types that have elements in their content. For example the schema on the beginning of the next page defines the complex type "treeSpeciesType". The "TreeSpecies" element declaration in section 2.1 is an example of a valid instance of this type definition.

```
<xsd:complexType name="treeSpeciesType">
  <xsd:sequence>
      <xsd:element name="TreeSpeciesName"
                   type="xsd:string"/>
      <xsd:element name="TreeSpeciesCode"
                   type="treeSpeciesCodeType"/>
  </xsd:sequence>
</xsd:complexType>
```

Using personalised data types like those above offers the following advantages:

1. Complex type definitions can gather all the related information into the same place. This can be viewed as one kind of modelling technique, and type definitions can be easily deduced from the real modelling methods.
2. The maintenance of XML documents becomes easier. If there is any need to insert a new "variable" or modify existing variables, modifications need only be made to the complex type definition where the variable belongs.
3. Type libraries may be developed to provide a set of predefined types that can be used as building blocks when new XML documents are created.

2.3 Type system for StanForD-XML

As has been described above, there are many good reasons to set up such a type system as a backbone for the development and the maintenance of XML documents. This is also relevant to some extent also in the development of StanForD-XML.

The predefined simple type set of XML Schema fulfils most of the requirements for simple types in StanForD-XML. However, it may be necessary to restrict the value ranges for some "variables". For example, we may want to restrict the values applicable to tree species codes. This kind of restriction has been made in the example type definition of the treeSpeciesCode explained in Section 3.1. Value ranges should be restricted whenever possible, to achieve automatic error checking for data.

One design decision is also the need to decide on our own names for simple types that do not have to be restricted in any way. For example one predefined simple type of XML Schema for integers is `xsd:int`. Our own type name for this can be defined in the following way:

```
<xsd:simpleType name = "stanForD_int">
      <xsd:restriction base = "xsd:int"/>
</xsd:simpleType>
```

Here `"stanForD_int"` is our own simple type name for integer values. The advantage of this kind of naming is that if there is any need to change integer presentation some time in future, existing XML documents will not have to be modified at all. The only thing that has to be changed is the simple type definition for the relevant integer. For example, if there is a need to increase the size reserved for integers, the `stanForD_int` type definition can be changed in the following way:

```
<xsd:simpleType name = "stanForD_int">
      <xsd:restriction base = "xsd:long"/>
</xsd:simpleType>
```

Are there then needs to provide complex data types in StanForD-XML? The answer to this question gives another question: Are there complex data structures that we want to use in many places? For example if we use the same structure for tree species all of standard files then that structure should obviously be implemented using complex data type. On the other hand we have data that belongs together but is only used in one place in our XML documents. Then we implement that direct and do not use any predefined data types. There are no complex data types used in example implementations of the PRD files (see Appendix 2). However their need becomes clear when all standard files are studied together and then their similarities can be seen.

2.4 Additional data

It is obvious that there are needs to put information into standard files that is not yet standardised or may not be ever standardised. In current StanForD standard this kind of information is handled by using certain variable numbers.

In StanForD-XML this task is more complicated because standard files are defined by schema definitions. Fortunately XML Schemas provide different kinds of techniques to solve this need. If we look the schema definitions of the example implementations there can be found following element declaration end of every schema file.

```
<xsd:element name="OptionalPart" minOccurs="0">
  <xsd:complexType>
      <xsd:sequence>
        <xsd:any namespace="##any"
                 processContents="skip" minOccurs="0"
                 maxOccurs="unbounded"/>
      </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

This optional element is named "OptionalPart" and it can contain whatever data that is well formed i.e. is syntactically right XML. Because attribute processContent has value "skip" data is not validated against any namespace.

3 STRUCTURE

3.1 Real world modelling

When the real world is modelled, structures are often found within other structures. For example, when modelling bucking results for a tree species, tree species may have the properties of a name, a code, a number of grades and a number of stem types. But within tree species there may also be more than one assortment and stem type structure. An assortment structure itself can have the properties of a name, a code, length, and diameter class structures. Part of the structure for tree species can be seen in Figure 1, which has been produced by the XML Spy schema editor tool.

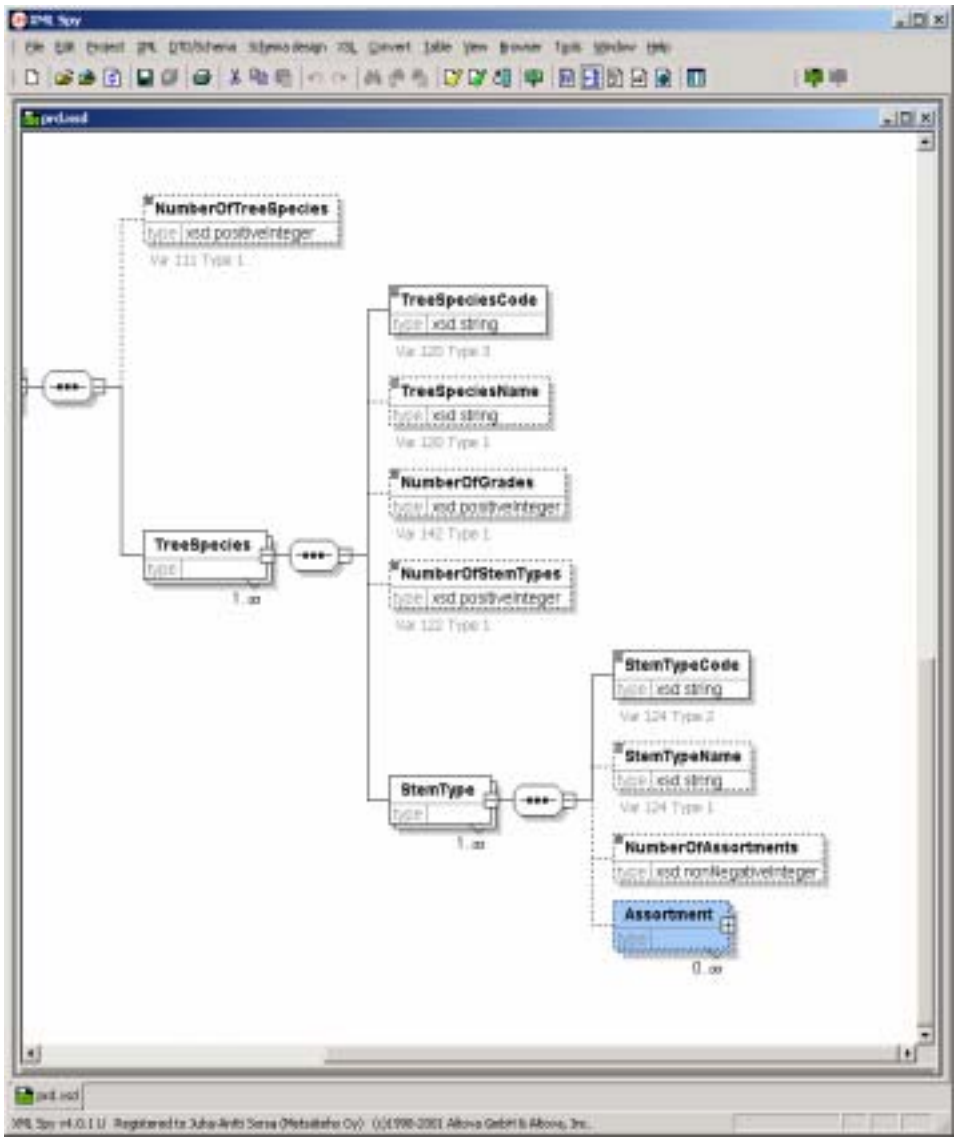

Figure 1. Part of the document structure for tree species

3.2 Structure in StanForD files

The data structure in StanForD files is flat. The natural structure of the data is broken down into pieces, and these pieces are located in different parts of the file. This kind of file structure is very compact, so the sizes of the files remain small. The structured data stored in this way is achieved by multi-indexing a data vector. The data vector is the value of some StanForD variable. For example, if we want to get the volume values from the variable 202 we must use four indexes: tree species, assortment, length class and diameter class. Because of this flat presentation, StanForD files are not very readable and they do not support modern software development work.

3.3 The basic structure of StanForD-XML documents

XML provides a way to present data with its natural structure intact. We can implement the structure exactly as it is seen in Figure 1. With standard files structured in this way, we can easily provide software to manipulate them. The files are also very readable and understandable, as long as we choose proper names for tags.

The most challenging design issue in StanForD-XML is to decide what kind of structure the new standard files will have, especially the structure for the multidimensional matrices so common in StanForD files.

A PRD file includes product information on bucking, as well as classified sum data for logs, such as volumes and numbers. The information content of the PRD file is used as an example when different kinds of solutions for XML document structure are studied.

The most unstructured XML versions of the PRD file could be as flat as current standard files. This would mean that every variable in the old PRD file would have a corresponding element in the new version of the PRD file. The following example XML code illustrates part of a document implemented this way.

```
...
<NumberOfTreeSpecies>3</NumberOfTreeSpecies>
<NumberOfAssortments>4 5 3
</NumberOfAssortments>
<NumberOfDiameterClasses>20 1 2 12 17 2 1 2
11 6 1 9</NumberOfDiameterClasses>
<NumberOfLengthClasses>9 1 1 8 6 3 1 2 12 6 1
8</NumberOfLengthClasses>
<TreeSpeciesName>Pine Spruce Birch</TreeSpeciesName>
<AssortmentCode>111 141 142 143 211 291 241 245 241 321
341 341</AssortmentCode>
...
```

All the elements in this alternative are at the top level of the hierarchy, i.e. there are no elements inside elements. But this kind of standard file implementation is not satisfactory, as it is not yet as readable as it could be, and the natural structure of the production information is still missing. A lot more explicit programming is needed to break up the contents of these data elements.

At the other end the XML document, the PRD file could be implemented so that every piece of information is inside an element and the hierarchy of the elements is maximised. The most multi-layered hierarchy that can be found in production information could for example be the following:

PRD
- tree species
  - stem type
    - assortment
      - length class
        - diameter class
          - number of logs
          - volume of logs

The following piece of code is the XML implementation of this hierarchy:

```
<PRD>
  ...
  <TreeSpecies>
    <TreeSpeciesName>Pine</TreeSpeciesName>
    <TreeSpeciesCode>1</TreeSpeciesCode>
    <StemType>
    <StemTypeCode>11</StemTypeCode>
      <Assortment>
        <AssortmentName>log100</AssortmentName>
        <AssortmentCode>100</AssortmentCode>
        <LengthClass>
          <LengthOfClass>275</LengthOfClass>
          <DiameterClass>
            <LowerLimit>80</LowerLimit>
            <UpperLimit>90</UpperLimit>
            <NumberOfLogs>3</NumberOfLogs>
            <VolumeOfLogs>0.054</VolumeOfLogs>
          </DiameterClass>
           ... other diameter classes
        </LengthClass>
        ... other length classes
      </Assortment>
      ... other assortments
    </StemType>
    ...other stem types
  </TreeSpecies>
   ... other tree species
</PRD>
```

In this implementation alternative, there is no need to store the zero information that such matrices usually frequently contain, as only those log classes that actually have values are given. When this implementation is analysed, the most important issue is to find the best way to implement the two-dimensional matrices every assortment has. In this alternative there is no explicit matrix, rather only those matrix cells that have non-zero values, which are scattered all around the XML document tree. If we wish to present these values as a matrix, we have to gather them together, which is not such a straightforward task.

Hence the most suitable implementation alternative for the production information on the bucking might be something between the two versions presented. It seems that the lowest levels of the hierarchy i.e. the value matrices, should be implemented in some other way. The code in Figure 2 presents one possible solution for doing this.

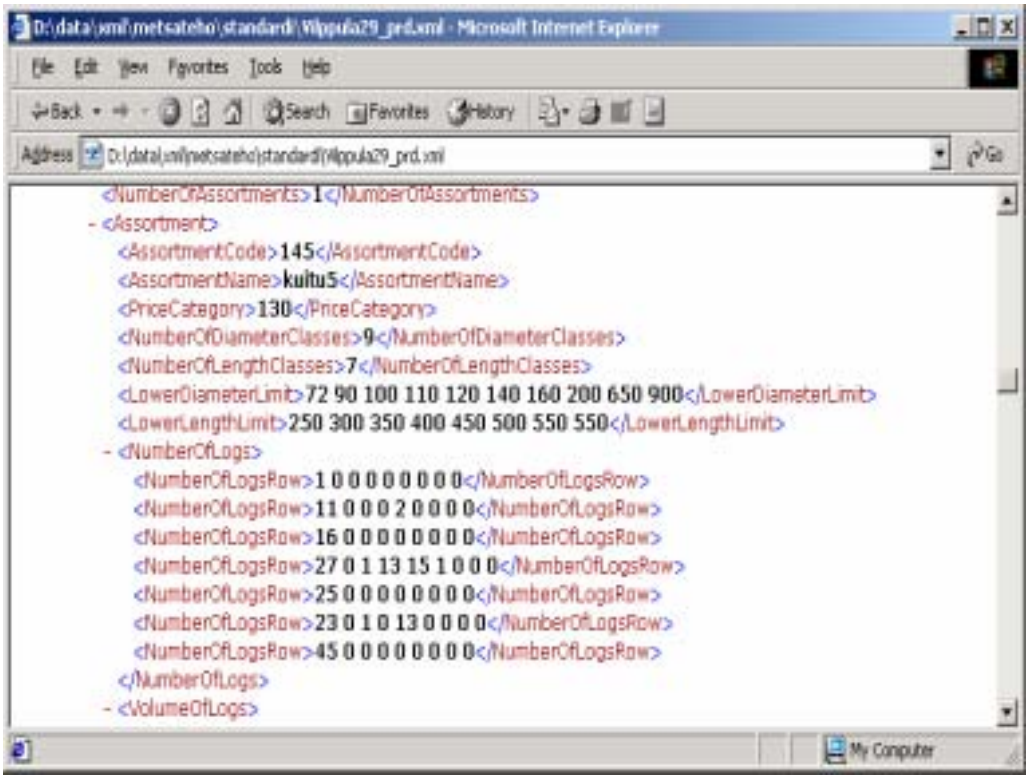

Figure 2. The XML implementation of the product information matrix of the assortment

The matrix implementation in Figure 2 is quite readable, and it is easy to implement software that can read and manipulate the matrix data. One StanForD-XML example implementation of the PRD file has this structure. Example StanForD-XML documents are described in Appendix 2.

3.4 Some problematic structure issues in StanForD-XML PRD document

3.4.1 Stem Type

Stem type information in StanForD files is problematic because it is not used in Sweden at all. Instead in Finland that information has been used and it seems that in short-term it still will be needed. There is also some other same kind of national information.

The stem type level is one mandatory level in the basic hierarchic structure of the StanForD-XML version of the PRD file. This structure was presented in the chapter 3.3. This means that every PRD file in StanForD-XML must have those stem type level elements, whether they are used or not. This may not be so satisfactory solution.

The approach that is presented in "*prd_no_stem_type_level.xsd*" schema file (see Appendix 2) could be one possible solution. In that schema there is no stem type level in the hierarchy. Because we want to preserve the stem type information it must be coded in other way. Every tree species level has the following `StemTypes` element declaration (the diagram notion is from the documentation tool of XML Spy):

element **PRD/LoggingResults/TreeSpecies/StemTypes**

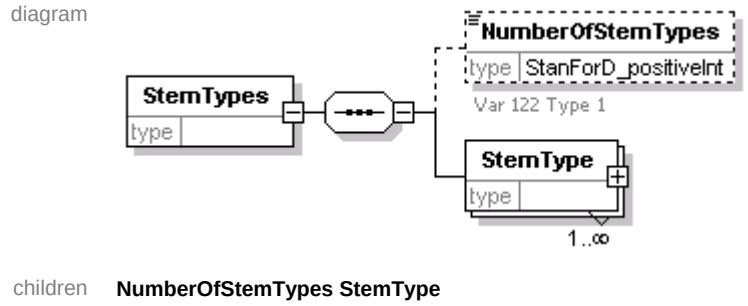

Figure 3. Declaration of StemTypes element

The diagram presents that element `StemTypes` has optional element `NumberOfStemTypes` and one to infinite number of mandatory elements `StemType`. Every `StemType` element has the following structure:

element **PRD/LoggingResults/TreeSpecies/StemTypes/StemType**

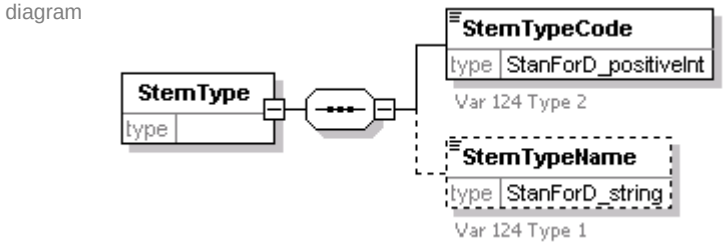

Figure 4. Declaration of StemType element

So the presented `StemTypes` element simple contains the list of stem type codes and names. In addition to this structure we must have one `StemTypeCode` element in every assortment that tells us in what stem type that assortment belongs.

This is the minimum amount of information that must be coded so that it is possible to produce logging results by stem type classification. How much calculated data for stem types we want, is then design issue. For example volumes and stem counts for each stem type could be put under the `StemType` element.

3.4.2 Driver

There is need to provide logging results for several drivers (= operators) in one PRD file. The driver-specific logging results do not fit well into the hierarchic structure of the StanForD-XML document. There are at least two different ways to implement the driver-specific data. The both alternatives have on the top level of the hierarchy the list of Driver elements which has the following structure:

element **PRD/Driver**

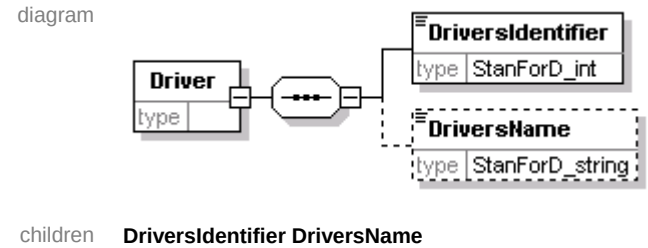

Figure 5. Declaration of Driver element

Each driver element defines the general information of one driver such as name and identification.

In the first alternative the logging results of the drivers are put on the level where they logically belong. For example drivers' logging results at tree species level could be coded in the way that is presented in the Figures 6 and 7.

element **PRD/LoggingResults/TreeSpecies/DriverStatisticsForTreeSpecies**

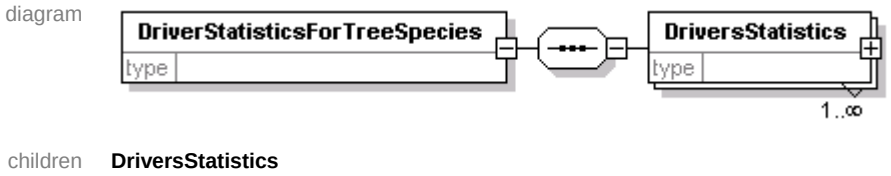

Figure 6. Declaration of DriverStatisticsForTreeSpecies

element **PRD/LoggingResults/TreeSpecies/DriverStatisticsForTreeSpecies/ DriversStatistics**

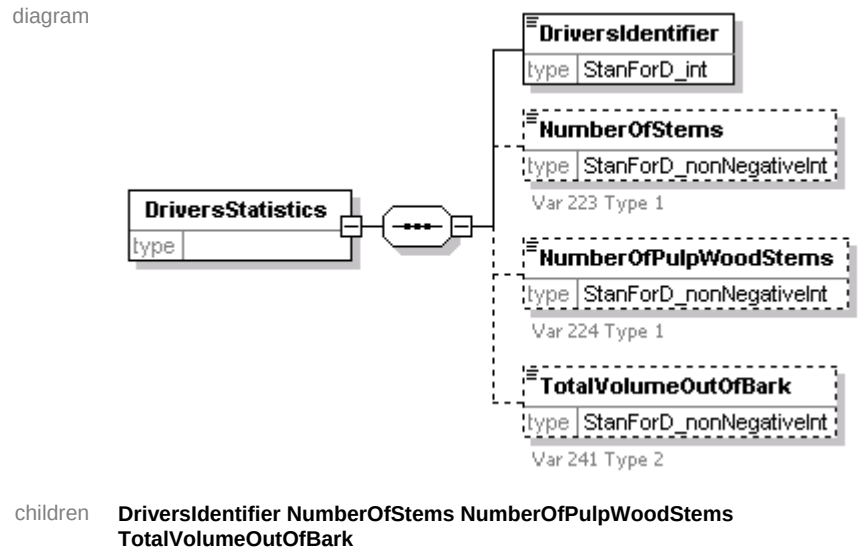

Figure 7. Declaration of DriversStatistics

Same kind of element structure for the driver-specific data is on the assortment level and possible also in stem type level.

The other alternative to implement the logging results of the drivers is to put all the data under the `Driver` element that was presented in Figure 5. In that case we must put there also enough identification data and structures for tree species, assortments and stem types depending on what statistics we want for drivers.

4 OTHER ISSUES

4.1 File names

StanForD requires that every file name consists of a base name and an extension separated by a period. Permissible characters for names are the letters A-Z and integer numbers 0-9. The base name can be up to eight characters long, and the extension three characters long. Different kinds of StanForD files are identified by specific extensions; so all files with the extension ".APT" contain bucking instructions, for instance.

Nowadays, there is no need to restrict the length of file names to eight plus three characters. In the near future all systems will support long filenames, and we should consider whether this kind of restriction is necessary at all in the new standard. We should also consider if there is any need to restrict the characters used in file names. These kinds of system specific restrictions will go out of date so quickly that it may be better not to include them in the standard at all.

In StanForD-XML it is not possible to use specific extensions for different kinds of files, because all xml files must have ".xml" extension. However it would be nice if we are able to see from the file name what its information content is. To achieve this we must put the file identification into the base part of the file name. For example apt file could be named following way

```
vilppula29_apt.xml
```

4.2 File formats

All the data in StanForD files is stored and transferred in ASCII format. A variable defines the used character set.

The 8-bit ISO Latin ASCII character set (ISO-8859-1) fulfils most of the requirements. However there may be a need to use characters that are not in ISO Latin character set. A new character set standard Unicode (UTF-16) was developed in the 1990s. This is a 16-bit code and it can handle the alphabets of many languages. Unicode is the default character set in XML files. There is also a more compact Unicode version called UTF-8, which is only 8 bytes long. XML documents' header rows have an attribute "encoding", which specifies the character set in use in the document. XML processors use this information when they process an XML document.

These days the most typical character set in the XML documents designed to meet the future standard is still ISO-8859-1. If we use the 16-bit Unicode character set, the sizes of our documents will be doubled. Unfortunately the 8-bit version of Unicode does not support the Scandinavian characters (ä,å,ö), so both Unicode versions are problematic choices. If we decide to use UTF-16 in future, however, we can always convert our current files into the relevant character set.

4.3 Sequences of variables

In StanForD, variables may occur in any order in the file, as long as those determining the size of others precede them.

StanForD-XML files are defined using XML Schema. Schema definitions determine the structure of data (i.e. the order of variables). It is possible to provide definitions that allow freedom for the order of variables, but there may be no good reasons for this. Software implementations are easier when there is a precisely determined structure and order for the data. The schema definitions also fix the possible problems that may be produced by dependent variables.

4.4 File termination

StanForD defines exactly how files are terminated by one or more checksums. In the new StanForD-XML standard there is no need to define these kinds of low-level communication details at all.

4.5 Versions of standard

The StanForD standard is evolving all the time. There become needs to include in to the standard new information and therefore new variables are introduced and defined. So time to time there will be published a new version of the standard.

In StanForD-XML all information i.e. the content of the standard files is defined using XML Schema definitions. When a new version of the standard is published this actually means that new XML Schema definitions are published.

It is clear that when the standard is evolving and new versions are published that implementations obey different versions of the standard. For this reason there must be some kind of version management policy in StanForD-XML standardisation process. The policy has to provide at least the following properties:

- all versions of the schema definitions must be stored and they have to be accessed if needed
- the change history of the versions of the schema definitions have to be documented
- all the standard files must have the information what version of the schema definition they obey

5 THE SIZES OF STANFORD-XML FILES

It is clear that when we code the same information using StanForD-XML instead of StanForD, the file sizes will be different. The XML coded bucking files are evidently larger than the corresponding StanForD files. Two factors influence the increase in the size of the files: the length of the tag names in use, and the structure of the XML documents. Issues related to these design solutions were discussed in Sections 3 and 4. It is impossible to say precisely how much bigger XML coded files will be, and while they could be only twice as large as the original StanForD files, in the worst cases they could be dozens of times larger. However, when these files are compressed, the size difference between StanForD and StanForD-XML files becomes insignificant.

We have made some simple tests to show how file sizes differ between StanForD and StanForD-XML files. In all tests a stand from Metsäteho's stem data warehouse was "cut" using a cutting simulator. The stand has approximately 1,200 stems, mostly spruce trees. The stand was cut using two different APT files. In the first APT file only one tree species (spruce) and two assortments were specified. In the second file, four species and nine assortments were specified. Simulations produced StanForD and StanForD-XML versions of the PRD file, as well as XML versions of the PRI files. A more detailed description of the resultant StanForD-XML documents is given in Appendix 2.

Two compression tools were used in the tests. WinZip is a well established and widely used compression tool, whereas XMill is a new tool for compressing XML data efficiently which can exploit the knowledge that a file has XML-coded structure and therefore compress XML files better than traditional compression methods.

Table 1 shows the sizes of PRD files. The first column indicates the original size of the file, and it can be seen that XML-coded files are about twice as large as the current standard files. However when the PRD files are compressed the size difference decreases considerably.

Table 1. The sizes of PRD files

	original size (KB)	WinZip (KB)	compr. rate	XMill (KB)	compr. rate
PRD1 (StanForD)	3.3	1.2	0.36		
PRD1 (XML)	7.5	2.0	0.26	1.9	0.25
PRD2 (StanForD)	7.2	1.6	0.22		
PRD2 (XML)	15.5	2.5	0.16	2.3	0.15

The relative differences between the compressed StanForD files and the compressed StanForD-XML files are much less than those between the uncompressed files. This is obviously because most of the extra space in XML files is redundant, and therefore vanishes when files are compressed. The differences between PRD files compressed by WinZip and XMill are quite small, since the PRD files are already quite compact, and contain no long recurrent structures that could provide extra space for XML files.

The sizes of the StanForD-XML PRI files are shown in Table 2. There are no corresponding original StandForD files, but the resultant files are approximately ten times smaller than the StanForD-XML files. These compression results are quite impressive. XMill especially provides such a good compression rate that the extra space that XML coding produces becomes meaningless. These compression results are understandable if the structure of PRI files is examined, since they simply contain a list of logs, with every log having the same information content. These kind of recurrent XML structures have a lot of redundancy, and can therefore be efficiently compressed.

Table 2. The sizes of PRI files

	Original size (KB)	**Winzip (KB)**	**compr. rate**	**XMill (KB)**	**compr. rate**
PRI1 (XML)	798	31	0.04	14	0.018
PRI2 (XML)	960	40	0.04	17	0.018

Appendix 1. References, web-sites and recommended reading

StanForD

http://www.skogforsk.se/marknad/stanford/estart.htm

- The English language homepage of the StanForD standard.

XML

http://www.w3.org/XML/1999/XML-in-10-points.html

"XML, XLink, Namespace, DTD, Schema, CSS, XHTML... If you are new to XML, it may be hard to know where to begin. This summary in 10 points attempts to capture enough of the basic concepts to enable a beginner to see the forest through the trees. And if you are giving a presentation on XML, why not start with these 10 points? They are hereby offered for your use."

http://www.xml.com/pub/a/98/10/guide0.html

"This introduction to XML presents the Extensible Markup Language at a reasonably technical level for anyone interested in learning more about structured documents. In addition to covering the XML 1.0 Specification, this article outlines related XML specifications, which are evolving. The article is organized in four main sections plus an appendix."

http://www.w3.org/TR/REC-xml

- The base XML specification document.

XML Schema

http://www.xml.com/pub/a/2000/11/29/schemas/part1.html

"A comprehensive introduction to XML Schema, a W3C XML language for describing and constraining the content of XML documents. Includes quick reference tables."

http://www.w3.org/TR/xmlschema-0/

"XML Schema Part 0: Primer is a non-normative document intended to provide an easily readable description of the XML Schema facilities, and is oriented towards quickly understanding how to create schemas using the XML Schema language. XML Schema Part 1: Structures and XML Schema Part 2: Datatypes provide the complete normative description of the XML Schema language. This primer describes the language features through numerous examples which are complemented by extensive references to the normative texts."

Tools

http://www.research.att.com/sw/tools/xmill/xmill.html

"XMill is a new tool for compressing XML data efficiently. It is based on a regrouping strategy that leverages the effect of highly-efficient compression techniques in compressors such as gzip. XMill groups XML text strings with respect to their meaning and exploits similarities between those text strings for compression. Hence, XMill typically achieves much better compression rates than conventional compressors such as gzip."

http://www.xmlspy.com/

"Structured/document editor for editing XML, DTDs, schemas (DCD, XDR, BizTalk, XSD), and XSLT. Provides views for structured editing (grid view, table view) and document editing (WYSIWYG), and a graphical XSLT designer. Supports authoring with XSL-FO. Has full Unicode support. By default, MSXML3 is used, but you can specify an external XSLT processor to be used for XSLT transformations."

Appendix 2. A brief description of example implementations of StanForD-XML documents

The following address in the Skogforsk web site

```
http://www.skogforsk.se/marknad/stanford/XMLproject/
```

contains some example data files of StanForD-XML implementation. There are three different versions of the PRD file and they are the following

```
vilppula29_prd.xml
vilppula29_no_stem_type_level_prd.xml
vilppula29_attr_prd.xml
```

and the corresponding schema files are

```
prd.xsd
prd_no_stem_type_level.xsd
prd_attributes.xsd
```

The first one of these files is the basic version of the StanForD-XML PRD file. In the second one the stem type level has been removed from the hierarchy and in the third one all the primitive data values have been implemented using attributes instead of elements. There are also Word documents for every PRD schema file that are generated by the XML Spy document tool. Those documents include graphical diagrams of the schema definitions.

Then there is one example of the PRI file. It's name is

```
vilppula29_pri.xml
```

and it's schema file is

```
pri.xsd
```

The data for these files was obtained by cutting one stand from Metsäteho's stem data warehouse with a cutting simulator. This stand has about 1,200 stems, most of them spruce trees. The stand was cut using two different APT files. The stand was cut using APT file with four species and nine assortments. Simulations produced StanForD-XML versions of the PRD and PRI files.

In addition there is one schema file "`primitivetypes.xsd`" that has all primitive type definitions that are used in all different PRD schemas.