pure::variants XML Transformation System Documentation

pure-systems GmbH

Version 3.2.19 for pure::variants 3.2

2016

Abstract

This document describes use and extension options of the pure::variants XML transformation system.

Table of Contents

1. Introduction

The XML Transformation System (XMLTS) provides an XML based transformation engine for transforming XML documents. It is based on the following features:

- depth-first and breadth-first traversal algorithms for the XML document tree
- specification of bindings for processing modules to the nodes of XML documents
- flexible integration of arbitrary transformation modules (statically linked, dynamically linked or script based)

In contrast to other XML transformation concepts it is not specified how the XML document tree is traversed and what transformations are possible. In fact you can even use your own tree traversal algorithm and apply any action to the nodes of the XML document you want.

A *processing module* encapsulates the actions to be performed on a node of the XML document. A fixed set of generic modules are part of the standard distribution of the transformation engine, e.g. a module to execute XSLT scripts and a module for collecting and executing transformation actions on file system level. For more complex projects own modules can be integrated.

Several kinds of module integration are supported, i.e. modules statically or dynamically linked to the XMLTS application and script modules. Dynamically linked modules and script modules in particular can be developed independently from the core XMLTS application.

In the following an overview of the transformation process in principle followed by a description of the XMLTS API in detail are presented.

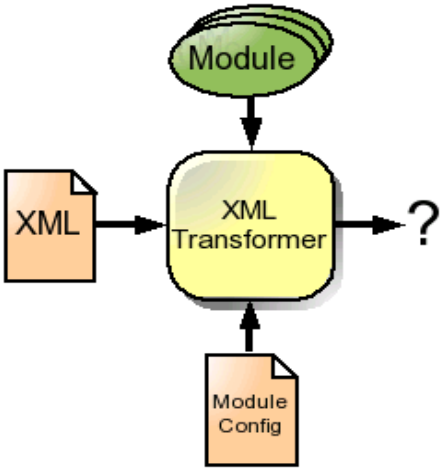
This document is available in the online help as well as in printable PDF format here.

2. The Transformation Process

The basic transformation process implemented in XMLTS works as follows. While traversing the XML document tree it is checked for every visited node whether a module is to be executed on it. If no (more) modules have to be executed the next node is visited as determined by the traversal algorithm. Otherwise the actions of the module(s) are performed on the node. Subsequent module executions on the same node can process not only the node itself but also the results produced by previously invoked modules. The transformation process is finished when the traversal of the XML document tree is finished.

The processing modules to be executed are defined in a *module configuration file*. It lists the available modules including configuration information for each module. In this file it is specified on which nodes a module is to be invoked. This information is evaluated by the transformation engine before the transformation process is started.

Figure 1. XML Transformer



The available modules are initialized by the transformation engine before any module is invoked on a node of the XML document tree. This may give, for instance, a database module the opportunity to connect to a database. Accordingly the transformation engine informs the modules when the traversal of the XML document tree is finished. The database module could now disconnect.

Before a module is invoked on a node it is checked whether the module is ready to run on this node. It is the task of the module to come to a decision depending only on its inner state.

A module may be invoked more than one time on a node. This depends directly on the algorithm used to traverse the XML document tree and if the module is a pre-, post-, or pre- and post-visit module. A pre-visit module is executed before the children of a node are visited. And a post-visit module is executed after the children of a node are visited.

The two standard traversal algorithms depth-first and breadth-first are part of the standard distribution of XMLTS. For most of the transformation tasks they should be sufficient. If they are not, own traversal algorithms can be provided easily, e.g. to let modules be invoked only on the leaves of the XML document tree.

3. Using Processing Modules

During the transformation process modules are the entities that perform the real transformation. Three types of modules are supported, i.e. statically linked modules, dynamically linked modules, and script modules. From the users point of view there is no distinction between these module types. They all are configured and used in the same way.

3.1. Module Configuration

The module configuration file describes the processing modules to be used to transform the input XML document. It is an XML document structured as follows:

```
<?xml version="1.0"?>
<moduleconfig name="config name" version="1.0">

  <input use="true|false">input path</input>

  <output use="true|false" create="true|false|ask" clear="true|false|ask"
          recover="true|false">output path</output>

  <module name="name" tname="name" tversion="version"
          include="XPath" exclude="XPath">
    <parameter name="param name">value</parameter>
    <parameter name="param name">value</parameter>
    ..
  </module>
  ..

</moduleconfig>
```

A processing module is an instance of a so-called template module, i.e. the module implementation. It is created by specifying a name for the processing module and the template module's name and version. Several instances of a template module can be created.

Since a module is executed only on nodes being associated with, every module has to provide the set of associated nodes, i.e. the nodes it will be invoked on. This set is described by two XPath expressions. One expression describing the nodes to be included in the set and a second expression describing the nodes to be excluded from the set. The name and the two XPath expressions are mandatory.

Modules can have parameters. A parameter is identified by its name and must have a value. The required parameters depend on the module.

The optional input and output tags specify local input and output transformation directories. These tags are only considered if attribute "use" contains value "true". The attributes "create" and "clear" control whether the output directory is created resp. cleared before starting the transformation.

A valid module configuration document describing two modules bound to the <variant> node of an XML document could look like this:

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

  <module name="Module1" tname="xslt" tversion="1.0" include="/variant" exclude="">
    <parameter name="in">cc_gen.xsl</parameter>
    <parameter name="out">actionlist_carboncopy.xml</parameter>
    <parameter name="output mode">both</parameter>
    <!-- both means to store on the node and also as file -->
  </module>

  <module name="Module2" tname="actionlist" tversion="1.0" include="/variant" exclude=""/>

</moduleconfig>
```

A module configuration is always evaluated in the directory where it is located. This makes it possible to use relative paths in the module configuration document.

3.2. Available Standard Processing Modules

XSLT Processing Module

A script module for executing XSL templates is the XSLT module. The XSLT module runs in three modes. All require the path to the external XSLT script (parameter `in`). Additionally the path to an output file for the XSLT generated data can be specified (parameter `out`). If no output file is specified or the parameter output mode is set to `both`, the result of executing the XSLT script is returned to the transformation engine as part of the acknowledgement document.

The entry format for the XSLT module in the module configuration document looks like this:

```
<module name="a name" tname="xslt" tversion="1.0" include="/" exclude="">
  <parameter name="in">script.xsl</parameter><!-- required -->
  <parameter name="out">out.data</parameter><!-- optional -->
  <parameter name="output mode">both</parameter><!-- optional -->
  <parameter name="keep doctype">false</parameter><!-- optional -->
  <parameter name="execution time">before</parameter><!-- optional -->
  <parameter name="...">...</parameter><!-- optional -->
</module>
```

The `out` and `output mode` parameters are optional. The `output mode` parameter can only have the value `both`. The `keep doctype` parameter is used to specify whether DOCTYPE definitions from the source document are copied into the result document. The DOCTYPE is copied only if the XSLT script does not define an explicit DOCTYPE for the result document. This parameter defaults to `true`. The `execution time` parameter specifies whether the module is a pre-, post-, or pre- and post-visit module (values `before`, `after`, `both`). This parameter defaults to `before`. Any further parameter is delivered to the XSLT script as string stylesheet parameter.

XMLTS XSLT Extensions

The XSLT module provides a special XSLT extension module. It is defined in the namespace http://www.pure-systems.com/xmlts and is included as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xmlts="http://www.pure-systems.com/xmlts"
                extension-element-prefixes="xmlts">
</xsl:stylesheet>
```

The following extension functions are available:

string os()	Returns the target system type. Currently it's one of win32, macosx, and linux (default).
string version()	Return the transformer version string.
string input-path()	Return the input path depending on the node currently being transformed.
string output-path()	Return the output path depending on the node currently being transformed.
string generate-id()	Return a newly generated unique identifier.
nodeset current()	Return the node currently being transformed.
nodeset entry-points()	Return the transformation entry point list, i.e. a list of node identifiers. Transformation modules can use this list to identify sub-trees of the input document that are to be transformed.
boolean below-entry-point(string)	Return true if the given node identifier specifies a node below a transformation entry point. Transformation modules can use this function to identify sub-trees of the input document that are to be transformed.
nodeset exit-points()	Return the transformation exit point list, i.e. a list of node identifiers. Transformation modules can use this list to identify sub-trees of the input document that are to be ignored.
boolean above-exit-point(string)	Return true if the given node identifier specifies a node above a transformation exit point. Transformation modules can use this function to identify sub-trees of the input document that are to be ignored.
nodeset results-for(nodeset?)	Return the module results for the given nodes. If no argument is given the results for the context node are returned.

nodeset log(string,number?)	Return the empty nodeset. Used for module logging. The first parameter is the log message and the second the logging level (0-9). It is recommend to use a log level between 4 (default, major logs) and 8 (module tracing).
nodeset info(string,string?,nodeset?)	Return the empty nodeset. Used for module info messages. The first parameter is the info message, the second the context of the info (e.g. the ID of a model element), and the last parameter is a set of strings (e.g. a set of IDs of related model elements).
nodeset warning(string,string?,nodeset?)	Return the empty nodeset. Used for module warning messages. The first parameter is the warning message, the second the context of the warning (e.g. the ID of a model element), and the last parameter is a set of strings (e.g. a set of IDs of related model elements).
nodeset error(string,string?,nodeset?)	Return the empty nodeset. Used for module error messages. The first parameter is the error message, the second the context of the error (e.g. the ID of a model element), and the last parameter is a set of strings (e.g. a set of IDs of related model elements). Error messages may abort the XSLT script execution and the current transformation.

The following extension elements are available:

<log level="0-9">message</log>	This is the extension element version of the log() extension function (see above). The level attribute is optional and defaults to 4.
<info context="a string" related="a nodeset">message</info>	This is the extension element version of the info() extension function (see above). The context and related attributes are optional.
<warning context="a string" related="a nodeset">message</warning>	This is the extension element version of the warning() extension function (see above). The context and related attributes are optional.
<error context="a string" related="a nodeset">message</error>	This is the extension element version of the error() extension function (see above). The context and related attributes are optional.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xmlts="http://www.pure-systems.com/xmlts"
                extension-element-prefixes="xmlts">

  <xsl:template match="/">
    <xsl:value-of select="xmlts:info('beginning...')"/>

    <xsl:value-of select="concat('Operating System    : ',xmlts:os())"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="concat('Transformer Version : ',xmlts:version())"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="concat('Input Path          : ',xmlts:input-path())"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="concat('Output Path         : ',xmlts:output-path())"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="concat('Current Node        : ',name(xmlts:current()))"/>
    <xsl:text>&#xA;</xsl:text>
    <xsl:value-of select="concat('Results Count       : ',count(xmlts:results-for()))"/>
    <xsl:text>&#xA;</xsl:text>

    <xmlts:info>finishing...</xmlts:info>
  </xsl:template>

</xsl:stylesheet>
```

The following file system access extension functions are available. They are defined in the namespace http://www.pure-systems.com/path.

string normalize(string)	Return the given path normalized for the target platform.
string dirname(string)	Return the given path without the file part.
string filename(string)	Return the given path without the directory part.
string basename(string)	Return the given path without extension.
string extension(string)	Return the extension of the given file.
string absolute(string)	Return the absolute version of the given path.
string add-part(string,string)	Combine the two paths using the platform specific path delimiter.
number size(string)	Return the size (in bytes) of the given file.
number mtime(string)	Return the modification time of the given file/directory.
string cwd()	Return the current working directory.
string tempdir()	Return the directory for temporary files of the target platform.
string delimiter()	Return the path delimiter of the target platform.
boolean exists(string)	Return true if given file/directory exists.
boolean is-dir(string)	Return true if given path points to a directory.
boolean is-file(string)	Return true if given path points to a file.
boolean is-absolute(string)	Return true if given path is absolute.
string to-uri(string)	Return the file URI build from the given path (file://...).
string read-file(string)	Read a file from a given URI and return its content as string.

The following string manipulation extension functions are available. They are defined in the namespace http://www.pure-systems.com/string.

nodeset parse(string)	Parse the given string as valid XML and return the resulting node set.
boolean matches(string,string)	Match the regular expression pattern in the second string against the first string.
nodeset match(string,string)	Match the regular expression pattern in the second string against the first string and return the set of sub-matches.
string submatch(string,string,number)	Match the regular expression pattern in the second string against the first string and return the n-th sub-match.
string replace(string,string,string,number?)	Replace the matches in the first string with the third string using the regular expression match pattern in the second string. The optional fourth parameter specifies the maximal number of replacements. 0 means unlimited, 1 means to replace only the first, 2 means to replace the first 2 matches etc. Returns the resulting string.
string expand(string)	Expand variables in the given string and return the expanded string. Variables are recognized by the following pattern: `$(VARIABLENAME)`

Action List Processing Module

XMLTS provides the action list processor to be able to perform a fixed set of actions listed in an XML document.

The action list document has the following form:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<actionlist>

  <action type="action type" eid="optional element id">
    <parameter name="param name">value</parameter>
    <parameter name="param name">value</parameter>
    ..
  </action>
  ..

</actionlist>
```

Like the XSLT module the action list module configuration may set parameters changing the behavior of the module. The optional parameter `in` can be used to specify an action list input file to process. If no action list file name is given the action list module searches for action lists in the results of the modules previously called on the same node. With the optional parameter `destroy` the module can be instructed to destroy each action list found in the module results after processing it. The `execution time` parameter specifies whether the module is a pre-, post-, or pre- and post-visit module (values `before`, `after`, `both`). This parameter defaults to `after`.The corresponding entry in the module description document looks like this:

```
<module name="a name" tname="actionlist" tversion="1.0" include="XPath" exclude="XPath">
  <parameter name="in">action list file name</parameter>
  <parameter name="destroy">true or false</parameter>
  <parameter name="execution time">before|after|both</parameter>
</module>
```

An valid module configuration for this module could be the following:

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

  <module name="Create AL" tname="xslt" tversion="1.0" include="/" exclude="">
    <parameter name="in">gen_actionlist.xsl</parameter>
  </module>

  <module name="Exec AL" tname="actionlist" include="/" exclude=""/>

</moduleconfig>
```

Two modules are listed. The first is an XSLT module executing the XSLT script `actionlist.xsl` generating a valid action list. The second is an action list module processing this action list by evaluating the result of the XSLT module.

Supported Actions

Appending text to a file

```
<action type="file.append">
  <parameter name="path">directory part of file name</parameter>
  <parameter name="file">file part of file name</parameter>
  <parameter name="src.path">directory part of file name</parameter>
  <parameter name="src.file">file part of file name</parameter>
  <parameter name="content">content</parameter>
</action>
```

The `file.append` action appends the given content to the file described by `path` and `file` where `path` is the directory part of the file name and `file` is the file part. If the file does not exist it will be created including its full path. The text to be appended can be specified either as content of the parameter `content` or in a separate file using the `src.path` and `src.file` parameters.

Prepending text to a file

```
<action type="file.prepend">
  <parameter name="path">directory part of file name</parameter>
  <parameter name="file">file part of file name</parameter>
  <parameter name="src.path">directory part of source file name</parameter>
  <parameter name="src.file">file part of source file name</parameter>
  <parameter name="content">content</parameter>
</action>
```

The `file.prepend` action prepends the given content to the file described by `path` and `file` where `path` is the directory part of the file name and `file` is the file part. If the file does not exist it will be created including its full path. The text to be prepended can be specified either as content of the parameter `content` or in a separate file using the `src.path` and `src.file` parameters.

Appending a file to another file

```
<action type="file.concat">
  <parameter name="src.path">directory part of source file name</parameter>
  <parameter name="src.file">file part of source file name</parameter>
  <parameter name="dest.path">directory part of target file name</parameter>
  <parameter name="dest.file">file part of target file name</parameter>
</action>
```

The `file.concat` action appends the content of the source file to the target file. If the target file does not exist it will be created including its full path.

Copy a file

```
<action type="file.copy">
  <parameter name="src.path">directory part of source file name</parameter>
  <parameter name="src.file">file part of source file name</parameter>
  <parameter name="dest.path">directory part of target file name</parameter>
  <parameter name="dest.file">file part of target file name</parameter>
</action>
```

The `file.copy` action copies a file to another file. If the target file exists it will be replaced.

Create a symbolic link to a file

```
<action type="file.link">
  <parameter name="src.path">directory part of source file name</parameter>
  <parameter name="src.file">file part of source file name</parameter>
  <parameter name="dest.path">directory part of symbolic link name</parameter>
  <parameter name="dest.file">file part of symbolic link name</parameter>
</action>
```

The `file.link` action creates a symbolic link for a file. It is an error if the symbolic link already exists.

Perform an XSL transformation

```
<action type="xsl.transform">
  <parameter name="in.doc">path to the input document</parameter>
  <parameter name="out.doc">path to the output document</parameter>
  <parameter name="stylesheet">path to the XSL script</parameter>
  <parameter name="keep.doctype">copy DOCTYPE to output document</parameter>
  ...
</action>
```

The `xsl.transform` action executes the given XSLT script on the XML input document. Optionally an output document can be specified used for the transformation results. The `keep.doctype` parameter is used to specify whether DOCTYPE definitions from the input document are copied into the output document. The DOCTYPE is copied only if the XSLT script does not define an explicit DOCTYPE for the output document. This parameter defaults to `true`. Any further parameter is delivered to the XSLT script as string stylesheet parameter. The XSLT script can access the XML document currently transformed by XMLTS using the predefined variable `pv.result`.

Perform a text transformation

```
<action type="xsl.texttransform">
  <parameter name="in.doc">path to the input document</parameter>
  <parameter name="out.doc">path to the output document</parameter>
  <parameter name="stylesheet">path to the XSL script</parameter>
  ...
</action>
```

The `xsl.texttransform` action executes the given XSLT script on the text input document enclosed by a CDATA block. Optionally an output document can be specified used for the transformation results. Any further parameter is delivered to the XSLT script as string stylesheet parameter. The XSLT script can access the document currently transformed by XMLTS using the predefined variable `pv.result`.

Command Execution Module

The command execution module can be used to execute external programs. It depends at least on one parameter, i.e. the program to execute. A second parameter is used to specify the parameters delivered to the program when called. Optionally the module exports the transformation document and the XPath to the current position in the transformation document. Additionally it can be specified when the command shall be executed, i.e. before (the default), after, or before and after (both) the subnodes of the current node are visited. If the executed command returned a value other than 0 the execution is assumed to be failed. This can be avoided by configuring the module to ignore the return value. A last parameter informs the module not to wait for the termination of the command before continuing with the transformation (asynchronous execution), the default is to wait until the command is finished.

The format for the command execution module in the module configuration document looks like this:

```
<module name="a name" tname="exec" tversion="1.0" include="/" exclude="">
  <parameter name="program">app.exe</parameter>
  <parameter name="parameters"></parameter>
  <parameter name="supply document">true</parameter>
  <parameter name="supply location">true</parameter>
  <parameter name="ignore return value">true</parameter>
  <parameter name="no wait">true</parameter>
  <parameter name="execution time">before</parameter>
</module>
```

The following environment variables are set before the program is executed:

Name	Description
XMLTS_INPUT_PATH	Current input path
XMLTS_OUTPUT_PATH	Current output path
XMLTS_DOCUMENT_LOCATION	The exported transformer document (optional)
XMLTS_NODE_LOCATION	The XPath expression for the current node (optional)

In case of synchronous execution the standard and error output of the shell command is returned to the transformation engine as part of the acknowledgement document. This part has the following structure:

```
<exec command="the executed command" exitcode="the program exit code">
  <stdout>the standard output of the command</stdout>
  <stderr>the error output of the command</stderr>
</exec>
```

4. Extending XMLTS

4.1. Statically Linked Modules

Statically linked modules are statically linked to the application. The basic module interface is defined in <ps/xmlt/Module.h>.

```
class Module {
  ...
public:
  virtual void startAction(const Document &doc, xml::SimpleDoc &result) = 0;
  virtual void stopAction(const Document &doc, xml::SimpleDoc &result) = 0;
  virtual void preAction(const Node &node, const Document &doc, xml::SimpleDoc &result) = 0;
  virtual void postAction(const Node &node, const Document &doc, xml::SimpleDoc &result) = 0;
  virtual bool isReady(const Node &node, const Document &doc) = 0;
  virtual void setParameters(const Node &node) = 0;
  virtual void getDescription(xml::SimpleDoc &desc);
  virtual const String getAPIVersion();

  virtual Module *newInstance() = 0;

  bool isPreProcessing() const;
  bool isPostProcessing() const;
  void setPreProcessing(bool state);
  void setPostProcessing(bool state);
  void setProcessingMode(bool prestate, bool poststate);

  bool hasDescription () const;
  void setDescription(const xml::SimpleNode &desc);

  const String &getName() const;
  void setName(const String &name);

  const String &getVersion() const;
  void setVersion(const String &version);
  unsigned getMajor() const;
  unsigned getMinor() const;

  const xml::ID &getID() const;

  void log(const String &message, int level);
};
```

Every module has a unique identifier, a (not necessarily unique) name, and a version. Further module parameters can be delivered to the module by `setParameters()`. The only argument to this method is the root node of the section for this module in the module configuration document. The `getDescription()` method returns a detailed description of the module and its parameters. This description has the following form:

```
<module name="" version="" include="XPath" exclude="XPath">
  <description>module description text</description>
  <parameters>
    <parameter name="" type="ps:string|ps:path|..." optional="true">
      <description>parameter meaning</description>
      <values>
        <value default="yes">a value to choose from</value>
        ...
      </values>
    </parameter>
    ...
  </parameters>
  <addparameters>
    <description>meaning of additional parameters</description>
  </addparameters>
</module>
```

The `include` and `exclude` attributes are optional and shall only indicate default values. Also the attribute `optional` is optional and can only have the value `true`. If there are values given for a parameter then the value of this parameter has to be one of the given values. The optional <addparameters> section indicates that the module accepts additional module parameters with any name, type, and value.

Before any transformation is performed the transformer changes the working directory to the global output directory (if specified). Then `startAction()` is called on every module to signalize that a new transformation process is started. Accordingly `stopAction()` is called on every module when the transformation process is finished. Both methods have two parameters. The first parameter is the XMLT document. The second parameter is an empty document to be filled with an acknowledgement of the following form:

```
<ack state="ok|warning|error|fatal" prune="yes|no">
  <message>a message text</message>
  <result>results</result>
</ack>
```

The state attribute indicates whether problems occurred during the execution of the module. "ok" means that there were no problems. "warning" means that there were problems that did not affect the execution of the module. "error" means that the execution of the module is failed. "fatal" means that the execution of the module is failed and the current transformation should be aborted.

With the prune attribute a module can control whether the children of the current node shall be traversed by the transformation engine or not. In case of using a breadth-first iterator the prune attribute is ignored.

The <message> tag is used to deliver a message to the transformation engine. If a module signals an error it is assumed that an error message is placed in the <message> tag.

The <result> tag is used to deliver the results produced by the module. As part of the acknowledgement structure the results are saved in the result map of the transformation engine and can then be accessed by other modules by asking the result map.

If the XML document tree is traversed using a Node::DepthFirst::Visitor iterator a module is called twice for a node, i.e. before the children of the node are visited and after visiting the children. In the first case `preAction()` is called if `isPreProcessing()` returns true and in the second case `postAction()` is called if `isPostProcessing()` returns true. Using another iterator type causes only `preAction()` to be called. In both cases the first argument of the call is the node the module is invoked on. Like for `startAction()` and `stopAction()` an acknowledgement of the same form must be generated.

Before `preAction()` or `postAction()` are invoked the transformer changes the working directory to the directory specified for the current node (if specified) and calls `isReady()` on the module to check whether the module is ready to run.

To support several instances of a module implementation a module has to provide the `newInstance()` method returning a new instance of itself.

Example

```
#include <ps/xmlt/Document.h>
#include <ps/xmlt/Module.h>
#include <ps/io/Path.h>
using namespace ps;
using namespace ps::xmlt;

// Module for dumping the names of the visited nodes.

class DumpModule : public Module {
public:
  DumpModule();
  virtual ~DumpModule();
  void setParameters(const Node &node);
  void startAction(const Document &doc, xml::SimpleDoc &result);
  void stopAction(const Document &doc, xml::SimpleDoc &result);
  void preAction(const Node &node, const Document &doc, xml::SimpleDoc &result);
  void postAction(const Node &node, const Document &doc, xml::SimpleDoc &result);
  bool isReady(const Node &node, const Document &doc);
  void getDescription(xml::SimpleDoc &desc);
  Module *newInstance();
};

DumpModule::DumpModule() : Module("dump", "1.0") {
  // is a pre-visit module
  setProcessingMode(true, false);
}

DumpModule::~DumpModule() {
}
```

```
Module *DumpModule::newInstance() {
  return new DumpModule;
}

void DumpModule::getDescription(xml::SimpleDoc &desc) {
  String description =
    "<module name='dump' version='1.0'>"
      "<decription>A node name dump module.</description>"
    "</module>";
  desc.parse(description);
}

bool DumpModule::isReady(const Node &node, const Document &doc) {
  return true;
}

void DumpModule::setParameters(const Node &node) {
}

void DumpModule::startAction(const Document &doc, xml::SimpleDoc &result) {
  acknowledge("ok", "no", result);
}

void DumpModule::stopAction(const Document &doc, xml::SimpleDoc &result) {
  acknowledge("ok", "no", result);
}

void DumpModule::preAction(const Node &node, const Document &doc, xml::SimpleDoc &result) {
  // check node
  if (! node.isValid()) {
    acknowledge("error", "no", "node is NULL", result);
    return;
  }
  cout << node.getName() << endl;
  acknowledge("ok", "no", result);
}

void DumpModule::postAction(const Node &node, const Document &doc, xml::SimpleDoc &result) {
  acknowledge("ok", "no", result);
  // no post action defined
}
```

4.2. Dynamically Linked Modules

Dynamically linked modules are dynamically linked to the application. They are realized as shared libraries on Linux or dynamic link libraries on Win32. The corresponding module interface is declared in `<ps/xmlt/DllModuleAPI.h>`.

```
extern "C" {
  // Return the version of the XMLTS API used by the module.
  EXPORT_SYMBOL ps::xml::String getAPIVersion();

  // Return a new instance of the module.
  EXPORT_SYMBOL ps::xmlt::Module *newInstance(const ps::Path &modulehome);
}
```

The `newInstance()` function is called by the transformation system to create a new instance of the module implemented in the library. The module implementation in the library is the same as for statically linked modules. The `getAPIVersion()` function is called by the transformation system to check which module API version this module implements.

Example

```
#include <ps/xmlt/DllModuleAPI.h>
#include <ps/xmlt/XMLTSVersion.h>
#include "DumpModule.h" // see previous example
```

```
extern "C" EXPORT_SYMBOL
ps::xml::String getAPIVersion() {
  return XMLTS_VERSION;
}

extern "C" EXPORT_SYMBOL
Module *newInstance(const ps::Path &modulehome) {
  return new DumpModule;
}
```

Registration of Dynamically Linked Modules

The module registration document describes the modules to be used to transform the input XML document. It is an XML document structured as follows:

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

  <module name="name" version="version" path="path to module"/>
  ...

</moduleregistry>
```

This document is used to tell the transformation system which dynamically linked modules are available and where they are located. If there are several modules having the same name the one with the higher version number is chosen. If one module is a COM module and the other is not then the COM module is chosen. There shall be no two modules listed having the same name and version.

A valid module registration document may look like this:

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

  <module name="dump" version="1.0" path="modules/dump.dll"/>
  <module name="myxslt" version="0.9" path="modules/myxslt.dll"/>

</moduleregistry>
```

4.3. COM Objects

A Windows only kind of module is the COM (Component Object Model) module. COM modules are usual COM objects implementing the transformer module COM interface. It looks like this:

```
interface ITransformerModule {
  HRESULT isReady([in] IDispatch* ixmlts, [out,retval] VARIANT_BOOL* result);
  HRESULT startAction([in] IDispatch* ixmlts, [out,retval] BSTR* result);
  HRESULT stopAction([in] IDispatch* ixmlts, [out,retval] BSTR* result);
  HRESULT preAction([in] IDispatch* ixmlts, [out,retval] BSTR* result);
  HRESULT postAction([in] IDispatch* ixmlts, [out,retval] BSTR* result);
  HRESULT setParameters([in] IDispatch* ixmlts);
  HRESULT getDescription([out,retval] BSTR* result);
};
```

Results are always returned as strings. A module can only access the input data through the interface pointer delivered as the first argument to the functions of this interface. Two concrete interfaces can be queried on this pointer, an OLE dispatch interface and a custom COM interface. The OLE dispatch interface is defined as follows:

```
dispinterface IXMLTS {
  BSTR NodeToString(void);
  BSTR DocumentToString(void);
  BSTR SubtreeToString(void);
  BSTR GetXPath(void);
  BSTR GetVersion(void);
  BSTR ConfigToString(void);
};
```

This is an OLE automation interface and is intended to be used in script modules (referred to by the `com.ps.consul.xmlts` type identifier). `NodeToString()` returns the current node of the XMLT document as string. The same holds for `DocumentToString()`, `SubtreeToString()`, and `ConfigToString()` for the current XMLT document, the subtree of the current node, and the configuration information of the module. `GetVersion()` returns the version of the transformation system and `GetXPath()` an XPath expression to the current node.

If no OLE automation functionallity is needed the faster custom COM interface is recommended. It looks quite the same as the OLE interface providing the same functionallity.

```
interface ICustomXMLTS : IDispatch {
  HRESULT NodeToString([out,retval] BSTR* resstr);
  HRESULT DocumentToString([out,retval] BSTR* resstr);
  HRESULT SubtreeToString([out,retval] BSTR* resstr);
  HRESULT GetXPath([out,retval] BSTR* resstr);
  HRESULT GetVersion([out,retval] BSTR* resstr);
  HRESULT ConfigToString([out,retval] BSTR* resstr);
};
```

4.4. Script Modules

Script modules are a special kind of module that do not need to implement the module interface as shown before. Instead of this they are based on built-in modules being able to execute external scripts. The XSLT and Actionlist modules are script modules.

5. Application development with XMLTS

The XMLTS SDK can be used independently from pure::variants to integrate XMLTS into other applications. The following section briefly shows how to do this.

5.1. XML Transformer Document

The XML transformer document is the main interface to the XML transformer. It is defined in `<ps/xmlt/Document.h>`.

```
class Document {
  ...
public:
  Document(ModuleManager &manager);
  Document(ModuleManager &manager, xmlDocPtr ptr);
  Document(ModuleManager &manager, const xml::SimpleDoc &doc) throw(xml::Exception);
  Document(ModuleManager &manager, const Path &filename) throw(xml::Exception);

  typedef std::list<const xml::SimpleDoc*> ModelList;
  void addModel(const xml::SimpleDoc &model);
  ModelList &getModels() const;

  void addDefaultPaths(const Path &input, const Path &output);
  void addPathPair(xmlNodePtr node, const PathPair &pair);
  void addPathPair(xmlNodePtr node, const Path &input, const Path &output);
  const Path &getInputPath() const;
  const Path &getOutputPath() const;

  void setIgnoreErrors(bool ignoreerrors);
  bool getIgnoreErrors() const;

  void addUserMessage(const UserMessage &message);
  const UserMessageList &getUserMessages() const;

  TransformContext *getTransformContext() const;

  void mapModules(const xml::SimpleDoc &cfg, const xml::Namespace &ns=xml::Namespace())
 throw(xml::Exception);
  void mapModules (const Path &cfgfile, const xml::Namespace &ns=xml::Namespace())
 throw(xml::Exception);

  Module *getCurrentModule() const;
```

```
  ModuleManager &getModuleManager() const;
  ModuleMap &getModuleMap() const;
  ResultMap &getResultMap() const;
  const xml::String &getVersion() const;

  Node setRootNode(Node &node);
  Node getRootNode() const throw(xml::Exception);

  void transform();
  template <class Iterator>
  void transform(Iterator iter=getRootNode());
};
```

The XML document to be transformed is loaded on construction time of a XMLTS document by delivering either the path to the XML file or the ready parsed XML document.

Modules are registered either by using a module registration document processed by `ModuleManager::registerModules()` or by calling `ModuleManager::registerModule()` with the module as argument. The global input and output directories for the modules are set using the `addDefaultPaths()` method. Input and output directories for subtrees of the XML document are specified with the `addPathPair()` methods. `getInputPath()` and `getOutputPath()` return the input and output directories depending on which node of the XML document is currently visited. `getModuleMap()` returns the active modules and its bindings on the nodes of the XMLT document. `getResultMap()` returns the results produced by the modules during the transformation process. The mapping and initialization of the modules is done by `mapModules()`. It expects a module configuration document as input. The built-in modules are registered by calling `ModuleManager::registerBuiltinModules()`. getCurrentModule() returns the currently executed module. `getTransformContext()` returns the current transformation context. With `addUserMessage()` it is possible to collect messages shown to the user when the transformation is finished. To specify that the transformation shall not abort when errors occur, `setIgnoreErrors(true)` can be called.

The transformation described by the transformation description document is performed by calling `transform()`. The default iterator used to traverse the XMLT tree is the Node::DepthFirst::Visitor iterator. There are several iterators available introduced in the next section.

The XMLT document is usually used like this:

```
#include <ps/xmlt/Document.h>
#include <ps/xmlt/XSLTProc.h>
using namespace ps::xmlt;

int main() {
  try {
    // initialize the library
    ps::xml::ID::init();
    XSLTProc::init();

    // create the module manager and register the modules
    ModuleManager man;
    man.registerBuiltinModules();
    ps::Path reg("moduleregistry.xml");
    man.registerModules(reg);

    // parse the XML input file
    ps::Path input("input.xml");
    Document doc(man, input);

    // is document valid and not empty?
    if (doc.isValid() && doc.hasRootNode()) {
      // map the modules to the nodes of the document
      ps::Path config("moduleconfig.xml");
      doc.mapModules(config);

      // transform the document
      doc.transform();
    } else {
      std::cerr << "Document is empty." << std::endl;
    }
```

```
  // an exception occured; print exception error message
  } catch (ps::xml::Exception e) {
    std::cerr << e.getMsg() << std::endl;
  }
  return 0;
}
```

Iterators

Several iterators are available for traversing the XMLT document tree. The two main types are depth-first and breadth-first iterators performing depth-first resp. breadth-first tree traversals. Sub categories are pre-order and post-order, left-to-right and right-to-left, and iterators following the visitor pattern.

Depth-First Iterators

Depth-first iterators perform node-by-node depth-first tree traversals. All depth-first iterators are bidirectional and support prefix and postfix ++/-- operators.

The following iterators are available:

- Node::DepthFirst::iterator
  - is a depth-first pre-order left-to-right iterator
- Node::DepthFirst::PreOrder::iterator
  - is a depth-first pre-order left-to-right iterator
- Node::DepthFirst::PreOrder::LeftRight::iterator
  - is a depth-first pre-order left-to-right iterator
- Node::DepthFirst::PreOrder::RightLeft::iterator
  - is a depth-first pre-order right-to-left iterator
- Node::DepthFirst::PostOrder::iterator
  - is a depth-first post-order left-to-right iterator
- Node::DepthFirst::PostOrder::LeftRight::iterator
  - is a depth-first post-order left-to-right iterator
- Node::DepthFirst::PostOrder::RightLeft::iterator
  - is a depth-first post-order right-to-left iterator
- Node::DepthFirst::Visitor::iterator
  - is a depth-first pre/in-order left-to-right iterator
- Node::DepthFirst::Visitor::LeftRight::iterator
  - is a depth-first pre/in-order left-to-right iterator
- Node::DepthFirst::Visitor::RightLeft::iterator
  - is a depth-first pre/in-order right-to-left iterator

These iterators are used as follows:

```
#include <ps/xmlt/Document.h>
```

```
using namespace ps::xmlt;
..
Node root = doc.getRootNode();
Node::DepthFirst::iterator iter;
for (iter = root; ! iter.done(); ++iter) {
  Node curr = *iter;
  cout << iter->getName() << endl;
}
```

Breadth-First Iterators

Breadth-first iterators perform node-by-node breadth-first tree traversals. All breadth-first iterators are bidirectional and support prefix and postfix ++/-- operators.

The following iterators are available:

- Node::BreadthFirst::iterator
  - is a breadth-first pre-order left-to-right iterator
- Node::BreadthFirst::PreOrder::iterator
  - is a breadth-first pre-order left-to-right iterator
- Node::BreadthFirst::PreOrder::LeftRight::iterator
  - is a breadth-first pre-order left-to-right iterator
- Node::BreadthFirst::PreOrder::RightLeft::iterator
  - is a breadth-first pre-order right-to-left iterator
- Node::BreadthFirst::PostOrder::iterator
  - is a breadth-first post-order left-to-right iterator
- Node::BreadthFirst::PostOrder::LeftRight::iterator
  - is a breadth-first post-order left-to-right iterator
- Node::BreadthFirst::PostOrder::RightLeft::iterator
  - is a breadth-first post-order right-to-left iterator

These iterators are used as follows:

```
#include <ps/xmlt/Document.h>
using namespace ps::xmlt;
..
Node root = doc.getRootNode();
Node::BreadthFirst::iterator iter;
for (iter = root; ! iter.done(); ++iter) {
  Node curr = *iter;
  cout << iter->getName() << endl;
}
```

Base Iterator

All node iterators (depth-first and breadth-first) are compatible to the base iterator. It supports the prefix ++/-- operators only and is used as follows:

```
#include <ps/xmlt/Document.h>
using namespace ps::xmlt;
..
void iterate(Node::BaseIterator &iter) {
```

```
  for (; ! iter.done(); ++iter) {
    Node curr = *iter;
    cout << iter->getName() << endl;
  }
}
..
Node root = doc.getRootNode();
Node::DepthFirst::iterator df_iter = root;
iterate(df_iter);
Node::BreadthFirst::iterator bf_iter = root;
iterate(bf_iter);
```

5.2. Module API Reference

Actionlist Module

It is defined in <ps/xmlt/ALProc.h>.

```
class ALProc {
  ..
public:
  ALProc() {}
  bool process(const Node &node);
  String &errorMsg() const;
};
```

The action list processor is started by calling `process()` with the root node of the action list document as argument. If the actions of the action list could be performed successfully true is returned. In case of an error `process()` returns false and `errorMsg()` returns a corresponding error message.

There is also a built-in module for processing action list. It is defined in <ps/xmlt/ALModule.h>.

```
class ALModule : public Module {
  ..
public:
  ALModule();
  ALModule(const ps::Path &in);
  ..
};
```

XSLT Module

Its main interface is defined in <ps/xmlt/XSLTModule.h>.

```
class XSLTModule : public Module {
  ..
public:
  XSLTModule();
  XSLTModule(const ps::xml::String &name, const ps::xml::String &version);
  XSLTModule(const ps::xml::String &name, const ps::xml::String &version, const ps::Path &in);
  XSLTModule(const ps::xml::String &name, const ps::xml::String &version, const ps::Path &in,
 const ps::Path &out, bool both = false);
  ..
};
```

The XSLT module provides four constructors. All non-default constructors take at least two arguments, i.e. the name and version of the XSLT module. The in parameter is the path to the external XSLT script. The out parameter is used to specify an output file for the XSLT generated data. If no output file is given the result of executing the XSLT script is returned to the transformation engine as part of the acknowledgement document. Delivering true for the both parameter causes the XSLT module to do both.

```
#include <ps/xmlt/Document.h>
#include <ps/xmlt/XSLTProc.h>
#include <ps/xmlt/XSLTModule.h>
using namespace ps::xmlt;
```

```
int main() {
  try {
    // initialize the library
    ps::xml::ID::init();
    XSLTProc::init();

    // create the module manager and add the built-in modules
    ModuleManager man;
    man.registerBuiltinModules();

    // add a new XSLT script module
    XSLTModule mymodule("mymodule", "0.1", "mymodule.xsl", "out.data", true);
    man.registerModule(mymodule);

    // parse the XML input file
    ps::Path input("input.xml");
    Document doc(man, input);

    // is document valid and not empty?
    if (doc.isValid() && doc.hasRootNode()) {
      // map the modules to the nodes of the document
      ps::Path config("moduleconfig.xml");
      doc.mapModules(config);

      // transform the document
      doc.transform();
    } else {
      std::cerr << "Document is empty." << std::endl;
    }
  // an exception occured; print exception error message
  } catch (ps::xml::Exception e) {
    std::cerr << e.getMsg() << std::endl;
  }
  return 0;
}
```