xmerl Application

version 1.1
Chapter 1

xmerl User’s Guide

The xmerl application contains modules with support for processing of xml files compliant to XML 1.0.

1.1 xmerl

1.1.1 Introduction

Features

The xmerl XML parser is able to parse XML documents according to the XML 1.0 standard. As default it performs well-formed parsing, (syntax checks and checks of well-formed constraints). Optionally one can also use xmerl as a validating parser, (validate according to referenced DTD and validating constraints). By means of for example the xmerl xs module it is possible to transform the parsed result to other formats, e.g. text, HTML, XML etc.

Overview

This document does not give an introduction to XML. There are a lot of books available that describe XML from different views. At the www.W3.org site you will find the XML 1.0 specification and other related specs. One site were you can find tutorials on XML and related specs is ZVON.org. However, here you will find some examples of how to use and to what you can use xmerl. A detailed description of the user interface can be found in the reference manual.

There are two known shortcomings in xmerl:

- It cannot retrieve external entities on the Internet by a URL reference, only resources in the local file system.
- xmerl can parse Unicode encoded data. But, it fails on tag names, attribute names and other mark-up names that are encoded Unicode characters not mapping on ASCII.

---

1URL: [http://www.w3.org](http://www.w3.org)
2URL: [http://www.w3.org/TR/REC-xml/](http://www.w3.org/TR/REC-xml/)
3URL: [http://www.zvon.org](http://www.zvon.org)
Chapter 1: xmerl User's Guide

By parsing an XML document you will get a record, displaying the structure of the document, as return value. The record also holds the data of the document. xmerl is convenient to use in for instance the following scenarios:

You need to retrieve data from XML documents. Your Erlang software can handle information from the XML document by extracting data from the data structure received by parsing.

It is also possible to do further processing of parsed XML with xmerl. If you want to change format of the XML document to for instance HTML, text or other XML format you can transform it. There is support for such transformations in xmerl.

One may also convert arbitrary data to XML. So it for instance is easy to make it readable by humans. In this case you first create xmerl data structures out of your data, then transform it to XML.

You can find examples of these three examples of usage below.

1.1.2 xmerl User Interface Data Structure

The following records used by xmerl to save the parsed data are defined in xmerl.hrl. The result of a successful parsing is a tuple \( /\text{CU} \) DataStructure, M \( /\text{CV} \). M is the XML production Misc, which is the mark-up that comes after the element of the document. It is returned "as is". DataStructure is an xmlElement record, that among others have the fields name, parents, attributes and content like:

\[
\text{#xmlElement}\{\text{name=Name,} \\
\text{...parents=Parents,} \\
\text{...attributes=Attrs,} \\
\text{content=Content,} \\
\text{...}\}
\]

The name of the element is found in the name field. In the parents field is the names of the parent elements saved. Parents is a list of tuples where the first element in each tuple is the name of the parent element. The list is in reverse order.

The record xmlAttribute holds the name and value of an attribute in the fields name and value. All attributes of an element is a list of xmlAttribute in the field attributes of the xmlElement record.

The content field of the top element is a list of records that shows the structure and data of the document. If it is a simple document like:

```xml
<?xml version="1.0"?>
<dog>
Grand Danois
</dog>
```

The parse result will be:

\[
\text{#xmlElement}\{\text{name = dog,} \\
\text{...parents = []}, \\
\text{...attributes = []}, \\
\text{content = [{xmlText,[[dog,1]],1,[]},"\nGrand Danois\n",text}],} \\
\text{...}\}
\]
Where the content of the top element is: [{xmlText, [{dog, 1}], 1, [], "\ Grand Danois\ ", text}].

Text will be returned in xmlText records. Though, usually documents are more complex, and the content of the top element will in that case be a nested structure with xmlElement records that in turn may have complex content. All of this reflects the structure of the XML document.

Space characters between mark-up as space, tab and line feed are normalized and returned as xmlText records.

Errors

An unsuccessful parse results in an error, which may be a tuple {error, Reason} or an exit: {'EXIT', Reason}. According to the XML 1.0 standard there are fatal error and error situations. The fatal errors must be detected by a conforming parser while an error may be detected. Both categories of errors are reported as fatal errors by this version of xmerl, most often as an exit.

1.1.3 Getting Started

In the following examples we use the XML file “motorcycles.xml” and the corresponding DTD “motorcycles.dtd”. motorcycles.xml looks like:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE motorcycles SYSTEM "motorcycles.dtd">
<motorcycles>
  <bike year="2000" color="black">
    <name>
      <manufacturer>Suzuki</manufacturer>
      <brandName>Suzuki VL 1500</brandName>
      <additionalName>Intruder</additionalName>
    </name>
    <engine>V-engine, 2-cylinders, 1500 cc</engine>
    <kind>custom</kind>
    <drive>cardan</drive>
    <accessories>Sissy bar, luggage carrier, V&amp;H exhaust pipes</accessories>
  </bike>
  <date>2004.08.25</date>
  <bike year="1983" color="red pearl">
    <name>
      <manufacturer>Yamaha</manufacturer>
      <brandName>XJ 400</brandName>
    </name>
    <engine>4 cylinder, 400 cc</engine>
    <kind>alround</kind>
    <drive>chain</drive>
    <comment>Good shape!</comment>
  </bike>
</motorcycles>
```

and motorcycles.dtd looks like:

```xml
xmerl Application
```
<?xml version="1.0" encoding="utf-8" ?>
<!ELEMENT motorcycles (bike,date?)+ >
<!ELEMENT bike (name,engine,kind,drive,accessories?,comment?) >
<!ELEMENT name (manufacturer,brandName,additionalName?) >
<!ELEMENT manufacturer (#PCDATA)> 
<!ELEMENT brandName (#PCDATA)> 
<!ELEMENT additionalName (#PCDATA)> 
<!ELEMENT engine (#PCDATA)> 
<!ELEMENT kind (#PCDATA)> 
<!ELEMENT drive (#PCDATA)> 
<!ELEMENT comment (#PCDATA)> 
<!ELEMENT accessories (#PCDATA)> 
<!ATTLIST bike year NMTOKEN #REQUIRED 
color NMTOKENS #REQUIRED 
condition (useless | bad | serviceable | moderate | good | excellent | new | outstanding) #IMPLIED >

If you want to parse the XML file motorcycles.xml you run it in the Erlang shell like:

3> {ParsResult, Misc} = xmerl:scan:file("motorcycles.xml").
1.1: xmerl

undeclared],
[])
4>

If you instead receives the XML doc as a string you can parse it by xmerl:scan:string/1. Both file/2 and string/2 exists where the second argument is a list of options to the parser, see the reference manual [page 23].

1.1.4 Example: Extracting Data From XML Content

In this example consider the situation where you want to examine a particular data in the XML file. For instance, you want to check for how long each motorcycle have been recorded.

Take a look at the DTD and observe that the structure of an XML document that is conformant to this DTD must have one motorcycles element (the root element). The motorcycles element must have at least one bike element. After each bike element it may be a date element. The content of the date element is #PCDATA (Parsed Character DATA), i.e. raw text. Observe that if #PCDATA must have a "<" or a "&" character it must be written as "&lt;" and "&amp;" respectively. Also other character entities exists similar to the ones in HTML and SGML.

If you successfully parse the XML file with the validation on as in:

```
xmerl:scan:file('motorcycles.xml',[validation,true])
```

you know that the XML document is valid and has the structure according to the DTD.

Thus, knowing the allowed structure it is easy to write a program that traverses the data structure and picks the information in the xmlElements records with name date.

Observe that white space: each space, tab or line feed, between mark-up results in an xmlText record.

1.1.5 Example: Create XML Out Of Arbitrary Data

For this task there are more than one way to go. The “brute force” method is to create the records you need and feed your data in the content and attribute fields of the appropriate element.

There is support for this in xmerl by the “simple-form” format. You can put your data in a simple-form data structure and feed it into xmerl:export_simple(Content,Callback,RootAttributes).

The Types are:

- Content = [Element]
- Callback = atom()
- RootAttributes = [Attributes]

Element is any of:

- {Tag, Attributes, Content}
- {Tag, Content}
- Tag
- IOString
- #xmlText{}
- #xmlElement{}
- #xmlPI{}

xmerl Application
The simple-form structure is any of \{Tag, Attributes, Content\}, \{Tag, Content\} or Tag where:

- Tag = atom()
- Attributes = \{\{Name, Value\} | xmlAttribute\}
- Name = atom()
- Value = IOString | atom() | integer()

See also reference manual for xmerl [page 20]

If you want to add the information about a black Harley Davidsson 1200 cc Sportster motorcycle from 2003 that is in shape as new in the motorcycles.xml document you can put the data in a simple-form data structure like:

Data =

```
{bike, 
  [[name,  
    [[manufacturer,["Harley Davidsson"]],  
    [brandName,["XL1200C"]],  
    [additionalName,["Sportster"]]]  
  ,  
  [engine,  
    ["V-engine, 2-cylinders, 1200 cc"],  
    [kind,"custom"],  
    [drive,"belt"]]]
```

In order to append this data to the end of the motorcycles.xml document you have to parse the file and add Data to the end of the root element content.

```
{RootEl,Misc}=xmerl:scan:file('motorcycles.xml'),
#xmlElement(content=Content) = RootEl,
NewContent=Content++lists:flatten([Data]),
NewRootEl=RootEl#xmlElement(content=NewContent),
```

Then you can run it through the export_simple/2 function:

```
{ok,IOF}=file:open('new_motorcycles.xml',[write]),
Export=xmerl:export_simple([NewRootEl],xmerl_xml),
io:format(IOF,"-s-n",[lists:flatten(Export)]),
```

The result would be:

```
<?xml version="1.0"?><motorcycles>
  <bike year="2000" color="black">
    <name>
      <manufacturer>Suzuki</manufacturer>
      <brandName>Suzuki VL 1500</brandName>
      <additionalName>Intruder</additionalName>
    </name>
    <engine>V-engine, 2-cylinders, 1500 cc</engine>
  </bike>
</motorcycles>
```
If it is important to get similar indentation and newlines as in the original document you have to add #xmlText() records with space and newline values in appropriate places. It may also be necessary to keep the original prolog where the DTD is referenced. If so, it is possible to pass a RootAttribute {prolog,Value} to export:3. The following example code fixes those changes in the previous example:

```erlang
Data = [
  #xmlText{value=" "},
  {bike,[
    {year,"2003"},
    {color,"black"},
    {condition,"new"}],
    #xmlText{value="\n "},
    {name,[
      #xmlText{value="\n 
"},
      {manufacturer,["Harley Davidsson"],
       #xmlText{value="\n 
"},
      {brandName,["XL1200C"],
       #xmlText{value="\n 
"},
      {additionalName,["Sportster"],
       #xmlText{value="\n "}],
      {engine,["V-engine, 2-cylinders, 1200 cc"],
       #xmlText{value="\n "},
      {kind,["custom"],
       #xmlText{value="\n "},
      {drive,["belt"],
       #xmlText{value="\n "}],
    }},
    #xmlText{value="\n 
"}],
  }},
NewContent=Content++lists:flatten([Data]),
NewRootEl=RootEl#xmlElement{content=NewContent},

Prolog = ["<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE motorcycles SYSTEM "motorcycles.dtd">\n"],
Export=xmerl:export_simple([NewRootEl],xmerl_xml,[{prolog,Prolog}]),
```

The result will be:
1.1.6 Example: Transforming XML To HTML

Assume that you want to transform the motorcycles.xml [page 3] document to HTML. If you want the same structure and tags of the resulting HTML document as of the XML document then you can use the xmerl:export/2 function. The following:

```prolog
2> {Doc, Misc} = xmerl:scan:file('motorcycles.xml').
{xmlElement, motorcycles, [
    {xmlns, []},
    {xmlNamespace, [], []},
    1,
    []],
    [{xmlText, [[motorcycles, i]], 1, [], "\n "},
    {xmlElement, bike, 
```
Will give the result result export.html

Perhaps you want to do something more arranged for human reading. Suppose that you want to list all
different brands in the beginning with links to each group of motorcycles. You also want all motorcycles
sorted by brand, then some flashy colors on top of it. Thus you rearrange the order of the elements and
put in arbitrary HTML tags. This is possible to do by means of the XSL Transformation (XSLT) like
functionality in xmerl.

Even though the following example shows one way to transform data from XML to HTML it also
applies to transformations to other formats

xmerl\_xs does not implement the entire XSLT specification but the basic functionality. For all details
see the reference manual [page 28]

First, some words about the xmerl\_xs functionality:

You need to write template functions to be able to control what kind of output you want. Thus if you
want to encapsulate a bike element in p tags you simply write a function:

```erlang
template(E = #xmlElement{name='bike'}) ->
    ["<p>\n\n" , xslapply(fun template/1,E) , "</p>\n"]
```

With xslapply you tell the XSLT processor in which order it should traverse the XML structure. By
default it goes in preorder traversal, but with the following we make a deliberate choice to break that
order:

```erlang
template(E = #xmlElement{name='bike'}) ->
    ["<p>\n\n" , xslapply(fun template/1,select("bike/name/manufacturer"),E) , "</p>\n"]
```

If you want to output the content of an XML element or an attribute you will get the value as a string
by the value\_of function:

```erlang
template(E = #xmlElement{name='motorcycles'}) ->
    ["<p>\n\n" , value\_of(select("bike/name/manufacturer",E),E) , "</p>\n"]
```

---

4URL: result export.html
5URL: http://www.w3.org/Style/XSL/
In the xmerl xs functions you can provide a select(String) call, which is an XPath\(^6\) functionality. For more details see the xmerl xs tutorial\(^7\).

Now, back to the example where we wanted to make the output more arranged. With the template:

```erlang
template(E = #xmlElement{name='motorcycles'}) ->
    ["<head>
    "<title>motorcycles</title>
    "</head>
    "<body>
    "<ul>
    remove_duplicates(value_of(select("bike/name/manufacturer",E))),
    "<li>
    sort_by_manufacturer(xslapply(fun template/1, E)),
    "</li>
    "</body>
    "</html>"];
```

We match on the top element and embed the inner parts in an HTML body. Then we extract the string values of all motorcycle brands, sort them and removes duplicates by `remove_duplicates(value_of(select("bike/name/manufacturer", E)))`. We also process the substructure of the top element and pass it to a function that sorts all motorcycle information by brand according to the task formulation in the beginning of this example.

The next template matches on the `bike` element:

```erlang
template(E = #xmlElement{name='bike'}) ->
    [value_of(select("name/manufacturer",E)),"<dt>",xslapply(fun template/1,select("name",E)),"</dt>"
    "<dd>
    "<li style="color:green">Manufacturing year: ",xslapply(fun template/1,select("@year",E)),"</li>
    "<li style="color:red">Color: ",xslapply(fun template/1,select("@color",E)),"</li>
    "<li style="color:blue">Shape: ",xslapply(fun template/1,select("@condition",E)),"</li>
    "</dd>
    "</ul>"];
```

This creates a tuple with the brand of the motorcycle and the output format. We use the brand name only for sorting purpose. We have to end the template function with the "built in clause" `template(E) -> built_in_rules(fun template/1, E).` The entire program is motorcycles2html.erl:

```erlang
%%%-------------------------------------------------------------------
%%% File : motorcycles2html.erl
%%% Author : Bertil Karlsson <bertil@localhost.localdomain>
%%% Description :
%%% Created : 2 Sep 2004 by Bertil Karlsson <bertil@localhost.localdomain>
%%%-------------------------------------------------------------------
-module(motorcycles2html).
-include("xmerl.hrl").
-import(xmerl-xs,[xslapply/2, value_of/1, select/2, built_in_rules/2]).
```

---

\(^6\) URL: http://www.w3.org/TR/xpath

\(^7\) URL: xmerl-xs-examples.html
1.1: xmerl

-export([process_xml/1,process_to_file/2,process_to_file/1]).

process_xml(Doc) ->
    template(Doc).

process_to_file(FileName) ->
    process_to_file(FileName,'motorcycles.xml').

process_to_file(FileName,XMLDoc) ->
    case file:open(FileName,[write]) of
      {ok,IOF} ->
        {XMLContent,;} = xmerl_scan:file(XMLDoc),
        TransformedXML=process_xml(XMLContent),
        io:format(IOF,~s,[TransformedXML]),
        file:close(IOF);
      {error,Reason} ->
        io:format("could not open file due to "~p"",[Reason])
    end.

%%% templates

template(E = #xmlElement{name='motorcycles'}) ->
    ["<head>
    "<title>motorcycles</title>
    </head>
    "
    "<body>
    "
    "<h1>Used Motorcycles</h1>
    "
    "<ul>
    "
    remove_duplicates(value_of(select("bike/name/manufacturer",E))),
    "
    "</ul>
    "
    sort_by_manufacturer(xslapply(fun template/1, E)),
    "</body>
    "
    "</html>
    "];

template(E = #xmlElement{name='bike'}) ->
    {value_of(select("name/manufacturer",E)),["<dt>",xslapply(fun template/1,select("name",E)),"</dt>","<dd>"},
    "<li style="color:green">Manufacturing year: ",xslapply(fun template/1,select("@year",E)),"</li>
    "<li style="color:red">Color: ",xslapply(fun template/1,select("@color",E)),"</li>
    "<li style="color:blue">Shape : ",xslapply(fun template/1,select("@condition",E)),"</li>
    "
    "</ul></dd>
    "];

template(E) -> built_in_rules(fun template/1, E).
% helper routines

% sorts on the bike name element, unwraps the bike information and
% inserts a line feed and indentation on each bike element.
sort_by_manufacturer(L) ->
    Tuples=[{H,T} || T <- L],
    SortedTS = lists:sort(Tuples, 1),
    InsertRefName_UnWrap=fun(
        {[Name,V]|Rest},
        fun(
            Name,F)->
                V|F(Rest,Name,F);
            end,
        end),
    SortedRefed=InsertRefName_UnWrap(SortedTS,no_name,InsertRefName_UnWrap),
    % SortedTs=[Y||{X,Y}<-lists:sort(Tuples,1)],
    WS = 
    Fun=fun([H|T],Acc,F)->
        F(T,[H,WS|Acc],F);
        end,
    if length(SortedRefed) > 0 ->
        Fun(SortedRefed,[],Fun);
        true -> []
    end.

% removes all but the first of an element in L and inserts a html
% reference for each list element.
remove_duplicates(L) ->
    remove_duplicates(L,[]).
remove_duplicates([],Acc) ->
    make_ref(lists:sort(lists:reverse(Acc))); remove_duplicates([A|L],Acc) ->
    case lists:delete(A, L) of
    L ->
        remove_duplicates(L,[A|Acc]);
    L1 ->
        remove_duplicates([A|L1],[Acc])
    end.
make_ref([]) -> [];
make_ref([H]) when atom(H) ->
    "<\ul>a href="#"+atom_to_list(H)++"">++atom_to_list(H)++"</a></\ul>";
make_ref([H]) when list(H) ->
    "<\ul>a href="#"+++H++"">++H++"</a></\ul>";
make_ref([H|T]) when atom(H) ->
    ["<\ul>a href="#"+atom_to_list(H)++"">++atom_to_list(H)++",
    "<\ul>a href="#"+atom_to_list(H)++"">++atom_to_list(H)++",}
If we run it like this: `motorcycles2html:process_to_file('result_xs.html', 'motorcycles2.xml').` The result will be `result_xs.html`. When the input file is of the same structure as the previous "motorcycles" XML files but it has a little more 'bike' elements and the 'manufacturer' elements are not in order.

8URL: result_xs.html
xmerl Reference Manual

Short Summaries

- Erlang Module xmerl [page 19] - Functions for exporting XML data to an external format.
- Erlang Module xmerl_eventp [page 22] - Simple event-based front-ends to xmerl_scan for processing of XML documents in streams and for parsing in SAX style.
- Erlang Module xmerl_scan [page 23] - This module is the interface to the XML parser, it handles XML 1.0.
- Erlang Module xmerl_xpath [page 26] - The xmerl_xpath module handles the entire XPath 1.0 spec. XPath expressions typically occur in XML attributes and are used to address parts of an XML document.
- Erlang Module xmerl_xs [page 28] - Erlang has similarities to XSLT since both languages have a functional programming approach.

xmerl

The following functions are exported:

- callbacks(M::atom()) -> [atom()]
  [page 19] Find the list of inherited callback modules for a given module.
- export(Data::Content, Callback) -> ExportedFormat
  [page 19] Equivalent to export(Data, Callback, []).
- export(Data::Content, Callback, RootAttrs::RootAttributes) -> ExportedFormat
  [page 19] Exports normal, well-formed XML content, using the specified callback-module.
- export_content(Es::Content, CBs::Callbacks) -> term()
  [page 20] Exports normal XML content directly, without further context.
- export_element() -> term()
  [page 20] Exports a normal XML element directly, without further context.
- export_element(E, CB::CBs, CBstate::UserState) -> ExportedFormat
  [page 20] For on-the-fly exporting during parsing (SAX style) of the XML document.
• `export_simple(Data::Content, Callback) -> ExportedFormat`
  [page 20] Equivalent to `export_simple(Data, Callback, [])`.
• `export_simple(Data::Content, Callback, RootAttrs::RootAttributes) -> ExportedFormat`
• `export_simple_content() -> term()`
  [page 21] Exports simple XML content directly, without further context.
• `export_simple_element() -> term()`
  [page 21] Exports a simple XML element directly, without further context.

**xmerl_eventp**

The following functions are exported:

• `file_sax(Fname::string(), CallBackModule::atom(), UserState, Options::option_list()) -> NewUserState`
• `stream(Fname::string(), Options::option_list()) -> xmlElement()`
  [page 22] Parse file containing an XML document as a stream, DOM style.
• `stream_sax(Fname, CallBack::CallBackModule, UserState, Options) -> xmlElement()`
  [page 22] Parse file containing an XML document as a stream, SAX style.
• `string_sax(String::list(), CallBackModule::atom(), UserState, Options::option_list()) -> xmlElement()`

**xmerl_scan**

The following functions are exported:

• `accumulate_whitespace(T::string(), S::global_state(), X3::atom(), Acc::string()) -> {Acc, T1, S1}`
  [page 24] Function to accumulate and normalize whitespace.
• `cont_state(S::global_state()) -> global_state()`
  [page 24] Equivalent to `cont_state(ContinuationState, S)`.
• `cont_state(X::ContinuationState, S::global_state()) -> global_state()`
  [page 24] For controlling the ContinuationState, to be used in a continuation function, and called when the parser encounters the end of the byte stream.
• `event_state(S::global_state()) -> global_state()`
  [page 24] Equivalent to `event_state(EventArgs, S)`.
• `event_state(X::EventArgs, S::global_state()) -> global_state()`
  [page 24] For controlling the EventArgs, to be used in an event function, and called at the beginning and at the end of a parsed entity.
• `fetch_state(S::global_state()) -> global_state()`
  [page 24] Equivalent to `fetch_state(FetchState, S)`.
• `fetch_state(X::FetchState, S::global_state()) -> global_state()`  
  [page 24] For controlling the FetchState, to be used in a fetch function, and called when the parser fetch an external resource (eg.
• `file(Filename::string()) -> {xmlElement(), Rest}`  
  [page 25] Equivalent to `file(Filename, []).`
• `file(Filename::string(), Options::option_list()) -> {xmlElement(), Rest}`  
• `hook_state(S::global_state()) -> global_state()`  
  [page 25] Equivalent to `hook_state(HookState, S)`.
• `hook_state(X::HookState, S::global_state()) -> global_state()`  
  [page 25] For controlling the HookState, to be used in a hook function, and called when the parser has parsed a complete entity.
• `rules_state(S::global_state()) -> global_state()`  
  [page 25] Equivalent to `rules_state(RulesState, S)`.
• `rules_state(X::RulesState, S::global_state()) -> global_state()`  
  [page 25] For controlling the RulesState, to be used in a rules function, and called when the parser store scanner information in a rules database.
• `string(Text::list()) -> {xmlElement(), Rest}`  
  [page 25] Equivalent to `string(Test, [])`.
• `string(Text::list(), Options::option_list()) -> {xmlElement(), Rest}`  
• `user_state(S::global_state()) -> global_state()`  
  [page 25] Equivalent to `user_state(UserState, S)`.
• `user_state(X::UserState, S::global_state()) -> global_state()`  
  [page 25] For controlling the UserState, to be used in a user function.

**xmerl xpath**

The following functions are exported:
• `string(Str, Doc) -> docEntity()`  
  [page 27] Equivalent to `string(Str, Doc, [])`.
• `string(Str, Doc, Options) -> docEntity()`  
  [page 27] Equivalent to `string(Str, Doc, [], Doc, Options)`.
• `string(Str, Node, Parents, Doc, Options) -> docEntity()`  
  [page 27] Extracts the nodes from the parsed XML tree according to X Path.

**xmerl xs**

The following functions are exported:
• `built_in_rules(Fun, E) -> List`  
  [page 28] The default fallback behaviour.
• `select(String::string(), E) -> E`  
  [page 28] Extracts the nodes from the xml tree according to X Path.
• `value_of(E) -> List`  
  [page 28] Concatenates all text nodes within the tree.
• `xslapply(Fun::Function, EList::list()) -> List`  
  [page 29] xslapply is a wrapper to make things look similar to xsl:apply-templates.
xmerl Reference Manual

xmerl_xsd

The following functions are exported:

- `file2state(FileName) -> {ok, State} | {error, Reason}`
  
  [page 30] Reads the schema state with all information of the processed schema from a file created with `state2file/1,2`.

- `format_error() -> term()`
  
  [page 30]

- `process_schema(Schema) -> Result`
  
  [page 30] Equivalent to `process_schema(Schema, [])`.

- `process_schema(Schema, Options) -> Result`
  
  [page 31] Reads the referenced XML schema and controls it is valid.

- `process_schemas(Schemas) -> Result`
  
  [page 31] Equivalent to `process_schema(Schemas, [])`.

- `process_schemas(Schemas, Options) -> Result`
  
  [page 31] Reads the referenced XML schemas and controls they are valid.

- `process_validate(Schema, Xml::Element) -> Result`
  
  [page 31] Equivalent to `process_validate(Schema, Xml, [])`.

- `process_validate(Schema, Xml::Element, Opts::Options) -> Result`
  
  [page 31] Validates a parsed well-formed XML element towards an XML schema.

- `state2file(S::State) -> ok | {error, Reason}`
  
  [page 32] Same as `state2file(State, SchemaName)`.

- `state2file(S::State, FileName) -> ok | {error, Reason}`
  
  [page 32] Saves the schema state with all information of the processed schema in a file.

- `validate(Xml::Element, State) -> Result`
  
  [page 32] Equivalent to `validate(Element, State, [])`.

- `validate(Xml::Element, State, Opts::Options) -> Result`
  
  [page 32] Validates a parsed well-formed XML element (Element).
Functions for exporting XML data to an external format.

Exports

callbacks(M::atom()) -> [atom()]

Find the list of inherited callback modules for a given module.

export(Data::Content, Callback) -> ExportedFormat

Equivalent to export(Data, Callback, []) [page 19].

export(Data::Content, Callback, RootAttrs::RootAttributes) -> ExportedFormat

Types:
- Content = [Element]
- Callback = atom()
- RootAttributes = [XmlAttributes]

Exports normal, well-formed XML content, using the specified callback-module. Element is any of:
- #xmlText{}
- #xmlElement{}
- #xmlPI{}
- #xmlComment{}
- #xmlDecl{}

(See xmerl.hrl for the record definitions) Text in #xmlText{} elements can be deep lists of characters and/or binaries.

RootAttributes is a list of #xmlAttribute{} attributes for the #root# element, which implicitly becomes the parent of the given Content. The tag-handler function for #root# is thus called with the complete exported data of Content. Root attributes can be used to specify e.g. encoding or other metadata of an XML or HTML document.

The Callback module should contain hook functions for all tags present in the data structure. Each hook function must have the following format:

Tag(Data, Attributes, Parents, E)
where $E$ is the corresponding #xmlElement{}, $Data$ is the already-exported contents of $E$ and $Attributes$ is the list of #xmlAttribute{} records of $E$. Finally, $Parents$ is the list of parent nodes of $E$, on the form [{ParentTag::atom(), ParentPosition::integer()}].

The hook function should return either the data to be exported, or a tuple 
{'#xml-alias#', NewTag::atom()}, or a tuple {'#xml-redefine#', Content}, 
where $Content$ is a content list (which can be on simple-form; see export_simple/2 for details).

A callback module can inherit definitions from other callback modules, through the 
required function '#xml-inheritance() -> [ModuleName::atom()].

See also: export/2 [page 19], export_simple/3 [page 20].

**export_content**($Es$:Content, $CBs$:Callbacks) -> term()

Types:
- $Content$ = [Element]
- $Callback$ = [atom()]

Exports normal XML content directly, without further context.

**export_element**() -> term()

Exports a normal XML element directly, without further context.

**export_element**($E$, $CB$:CBs, $CBstate$:UserState) -> ExportedFormat

For on-the-fly exporting during parsing (SAX style) of the XML document.

**export_simple**($Data$:Content, $Callback$) -> ExportedFormat

Equivalent to export_simple($Data$, $Callback$, []) [page 20].

**export_simple**($Data$:Content, $Callback$, $RootAttrs$:RootAttributes) -> ExportedFormat

Types:
- $Content$ = [Element]
- $Callback$ = atom()
- $RootAttributes$ = [XmlAttributes]

Exports "simple-form" XML content, using the specified callback-module.

$Element$ is any of:
- {Tag, Attributes, Content}
- {Tag, Content}
- Tag
- IOString
- #xmlText{}
- #xmlElement{}
- #xmlPI{}
- #xmlComment{}
- #xmlDecl{}}}
where

- Tag = atom()
- Attributes = [{Name, Value}]
- Name = atom()
- Value = IOString | atom() | integer()

Normal-form XML elements can thus be included in the simple-form representation. Note that content lists must be flat. An IOString is a (possibly deep) list of characters and/or binaries.

RootAttributes is a list of:

- XmlAttributes = #xmlAttribute{}

See `export/3` for details on the callback module and the root attributes. The XML-data is always converted to normal form before being passed to the callback module.

See also: `export/3` [page 19], `export_simple/2` [page 20].

```prolog
export_simple_content() -> term()
Exports simple XML content directly, without further context.

export_simple_element() -> term()
Exports a simple XML element directly, without further context.
```
xmerl_eventp

Erlang Module

Simple event-based front-ends to xmerl_scan for processing of XML documents in streams and for parsing in SAX style. Each contain more elaborate settings of xmerl_scan that makes usage of the customization functions.

Exports

file_sax(Fname::string(), CallBackModule::atom(), UserState, Options::option_list()) -> NewUserState

Parse file containing an XML document, SAX style. Wrapper for a call to the XML parser xmerl_scan with a hook_fun for using xmerl export functionality directly after an entity is parsed.

stream(Fname::string(), Options::option_list()) -> xmlElement()

Parse file containing an XML document as a stream, DOM style. Wrapper for a call to the XML parser xmerl_scan with a continuation_fun for handling streams of XML data. Note that the continuation_fun, acc_fun, fetch_fun, rules and close_fun options cannot be user defined using this parser.

stream_sax(Fname, CallBack::CallBackModule, UserState, Options) -> xmlElement()

Types:
- Fname = string()
- CallBackModule = atom()
- Options = option_list()

Parse file containing an XML document as a stream, SAX style. Wrapper for a call to the XML parser xmerl_scan with a continuation_fun for handling streams of XML data. Note that the continuation_fun, acc_fun, fetch_fun, rules, hook_fun, close_fun and user_state options cannot be user defined using this parser.

string_sax(String::list(), CallBackModule::atom(), UserState, Options::option_list()) -> xmlElement()

Parse file containing an XML document, SAX style. Wrapper for a call to the XML parser xmerl_scan with a hook_fun for using xmerl export functionality directly after an entity is parsed.
This module is the interface to the XML parser, it handles XML 1.0. The XML parser is activated through xmerl_scan:string/[1,2] or xmerl_scan:file/[1,2]. It returns records of the type defined in xmerl.hrl. See also tutorial\(^1\) on customization functions.

### DATA TYPES

**global_state()** The global state of the scanner, represented by the `#xmerl_scanner/1` record.

**option_list()** Options allow to customize the behaviour of the scanner. See also tutorial\(^2\) on customization functions.

Possible options are:

- `{acc_fun, Fun}` Call back function to accumulate contents of entity.
- `{continuation_fun, Fun}` Call back function to decide what to do if the scanner runs into EOF before the document is complete.
- `{event_fun, Fun}` Call back function to handle scanner events.
- `{fetch_fun, Fun}` Call back function to fetch an external resource.
- `{hook_fun, Fun}` Call back function to process the document entities once identified.
- `{close_fun, Fun}` Called when document has been completely parsed.
- `{rules, ReadFun, WriteFun, RulesState}` Handles storing of scanner information when parsing.
- `{user_state, UserState}` Global state variable accessible from all customization functions.
- `{fetch_path, PathList}` PathList is a list of directories to search when fetching files. If the file in question is not in the fetch path, the URI will be used as a file name.
- `{space, Flag}` 'preserve' (default) to preserve spaces, 'normalize' to accumulate consecutive whitespace and replace it with one space.
- `{line, Line}` To specify starting line for scanning in document which contains fragments of XML.
- `{namespace_conformant, Flag}` Controls whether to behave as a namespace conformant XML parser, 'false' (default) to not otherwise 'true'.

\(^1\) [xmerl examples.html](#)

\(^2\) [xmerl examples.html](#)
{validation, Flag} Controls whether to process as a validating XML parser: 'off' (default) no validation, or validation 'dtd' by DTD or 'schema' by XML Schema. 'false' and 'true' options are obsolete (i.e. they may be removed in a future release), if used 'false' equals 'off' and 'true' equals 'dtd'.

{schemaLocation, [[Namespace, Link]...]} Tells explicitly which XML Schema documents to use to validate the XML document. Used together with the {validation, schema} option.

{quiet, Flag} Set to 'true' if xmerl should behave quietly and not output any information to standard output (default 'false').

{doctype_DTD, DTD} Allows to specify DTD name when it isn't available in the XML document. This option has effect only together with {validation, 'dtd'} option.

{xmlbase, Dir} XML Base directory. If using string/1 default is current directory. If using file/1 default is directory of given file.

{encoding, Enc} Set default character set used (default UTF-8). This character set is used only if not explicitly given by the XML declaration.

Exports

accumulate_whitespace(T::string(), S::global_state(), X3::atom(), Acc::string()) -> {Acc, T1, S1}
   Function to accumulate and normalize whitespace.

cont_state(S::global_state()) -> global_state()
   Equivalent to cont_state(ContinuationState, S) [page 24].

ccont_state(X::ContinuationState, S::global_state()) -> global_state()
   For controlling the ContinuationState, to be used in a continuation function, and called when the parser encounters the end of the byte stream. See tutorial3 on customization functions.

event_state(S::global_state()) -> global_state()
   Equivalent to event_state(EventState, S) [page 24].

event_state(X::EventState, S::global_state()) -> global_state()
   For controlling the EventState, to be used in an event function, and called at the beginning and at the end of a parsed entity. See tutorial4 on customization functions.

fetch_state(S::global_state()) -> global_state()
   Equivalent to fetch_state(FetchState, S) [page 24].

fetch_state(X::FetchState, S::global_state()) -> global_state()
   For controlling the FetchState, to be used in a fetch function, and called when the parser fetch an external resource (eg. a DTD). See tutorial5 on customization functions.

3URL: xmerl_examples.html
4URL: xmerl_examples.html
5URL: xmerl_examples.html
file(Filename::string()) -> {xmlElement(), Rest}
  Types:
  • Rest = list()
  Equivalent to file(Filename, []) [page 25].

file(Filename::string(), Options::option_list()) -> {xmlElement(), Rest}
  Types:
  • Rest = list()
  Parse file containing an XML document

hook_state(S::global_state()) -> global_state()
  Equivalent to hook_state(HookState, S) [page 25].

hook_state(X::HookState, S::global_state()) -> global_state()
  For controlling the HookState, to be used in a hook function, and called when the parser has parsed a complete entity. See tutorial6 on customization functions.

rules_state(S::global_state()) -> global_state()
  Equivalent to rules_state(RulesState, S) [page 25].

rules_state(X::RulesState, S::global_state()) -> global_state()
  For controlling the RulesState, to be used in a rules function, and called when the parser store scanner information in a rules database. See tutorial7 on customization functions.

string(Text::list()) -> {xmlElement(), Rest}
  Types:
  • Rest = list()
  Equivalent to string(Text, []) [page 25].

string(Text::list(), Options::option_list()) -> {xmlElement(), Rest}
  Types:
  • Rest = list()
  Parse string containing an XML document

user_state(S::global_state()) -> global_state()
  Equivalent to user_state(UserState, S) [page 25].

user_state(X::UserState, S::global_state()) -> global_state()
  For controlling the UserState, to be used in a user function. See tutorial8 on customization functions.

6URL:xmerl_examples.html
7URL:xmerl_examples.html
8URL:xmerl_examples.html
xmerl_xpath

Erlang Module

The xmerl_xpath module handles the entire XPath 1.0 spec XPath expressions typically occurs in XML attributes and are used to address parts of an XML document. The grammar is defined in xmerl_xpath_parse.yrl. The core functions are defined in xmerl_xpath_pred.erl.

Some useful shell commands for debugging the XPath parser

c(xmerl_xpath_scan).
yecc:yecc("xmerl_xpath_parse.yrl", "xmerl_xpath_parse", true, []).
c(xmerl_xpath_parse).

xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("position() > -1")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("5 * 6 div 2")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("5 + 6 mod 2")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("5 * 6")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("-------6")) .
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("parent::node()")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("descendant-or-self::node()")).
xmerl_xpath_parse:parse(xmerl_xpath_scan:tokens("parent::processing-instruction('foo')")).

DATA TYPES

docEntity() = xmlElement() | xmlAttribute() | xmlText() | xmlPI() | xmlComment()

nodeEntity() = xmlElement() | xmlAttribute() | xmlText() | xmlPI() | xmlNamespace() | xmlData

option_list() Options allows to customize the behaviour of the XPath scanner.
Possible options are:

\{namespace, #xmlNamespace\} Set namespace nodes, from X miN anspace, in xmlContext

\{namespace, Nodes\} Set namespace nodes in xmlContext.
Exports

\[
\text{string(}\text{Str, Doc)} \rightarrow \text{docEntity()}
\]
Equivalent to \text{string(}\text{Str, Doc, [])} [page 27].

\[
\text{string(}\text{Str, Doc, Options)} \rightarrow \text{docEntity()}
\]
Equivalent to \text{string(}\text{Str, Doc, [], Doc, Options)} [page 27].

\[
\text{string(}\text{Str, Node, Parents, Doc, Options)} \rightarrow \text{docEntity()}
\]
Types:
- \text{Str} = \text{xPathString()}
- \text{Node} = \text{nodeEntity()}
- \text{Parents} = \text{parentList()}
- \text{Doc} = \text{nodeEntity()}
- \text{Options} = \text{option_list()}

Extracts the nodes from the parsed XML tree according to XPath.
Erlang has similarities to XSLT since both languages have a functional programming approach. Using xmerl_xpath it is possible to write XSLT like transforms in Erlang. XSLT stylesheets are often used when transforming XML documents, to other XML documents or (X)HTML for presentation. There are a number of brick-sized books written on the topic. XSLT contains quite many functions and learning them all may take some effort, which could be a reason why the author only has reached a basic level of understanding. This document assumes a basic level of understanding of XSLT. Since XSLT is based on a functional programming approach with pattern matching and recursion it is possible to write similar style sheets in Erlang. At least for basic transforms. This document describes how to use the XPath implementation together with Erlangs pattern matching and a couple of functions to write XSLT like transforms.

This approach is probably easier for an Erlanger but if you need to use real XSLT stylesheets in order to "comply to the standard" there is an adapter available to the Sablotron XSLT package which is written in C++. See also the Tutorial\(^9\).

### Exports

```erlang
built_in_rules(Fun, E) -> List
    The default fallback behaviour. Template funs should end with:
    template(E) -> built_in_rules(fun template/1, E).
```

```erlang
select(String::string(), E) -> E
    Extracts the nodes from the xml tree according to X Path.
    See also: `value_of/1` [page 28].
```

```erlang
value_of(E) -> List
    Types:
    * E = unknown()
    Concatenates all text nodes within the tree.
    Example:
```

\(^9\)URL: xmerl\_xs\_examples.html
<xsl:template match="title">
  <div align="center">
    <h1><xsl:value-of select="." /></h1>
  </div>
</xsl:template>

becomes:

template(E = #xmlElement{name='title'}) ->
  ["<div align="center"><h1>",
    value_of(select(".", E)), "</h1></div>"
]

{xslapply(Fun::Function, EList::list()) -> List}

Types:
- Function = () -> list()

{xslapply} is a wrapper to make things look similar to xsl:apply-templates.

Example, original XSLT:

<xsl:template match="doc/title">
  <h1>
    <xsl:apply-templates/>
  </h1>
</xsl:template>

becomes in Erlang:

template(E = #xmlElement{ parents=[{'doc','.'}], name='title'}) ->
  ["<h1>",
    xslapply(fun template/1, E),
    "</h1>"];
Interface module for XML Schema validation. It handles the W3.org specifications\(^1\) of XML Schema second edition 28 October 2004. For an introduction to XML Schema study part 0.\(^2\) An XML structure is validated by xmerl_xsd:validate/[2,3].

**DATA TYPES**

- `global_state()` The global state of the validator. It is represented by the `#xmerl_xsd{}` record.
- `option_list()` Options allow to customize the behaviour of the validation. Possible options are:
  - `{tab2file,boolean()}` Enables saving of abstract structure on file for debugging purpose.
  - `{xsd_base,filename()}` xsd Base directory.
  - `{fetch_fun,FetchFun}` Call back function to fetch an external resource.
  - `{fetch_path,PathList}` PathList is a list of directories to search when fetching files. If the file in question is not in the fetch_path, the URI will be used as a file name.
  - `{state,State}` It is possible by this option to provide a state with process information from an earlier validation.

**Exports**

- `file2state(FileName) -> {ok, State} | {error, Reason}`
  Types:
  - State = global_state()
  - FileName = filename()
  Reads the schema state with all information of the processed schema from a file created with state2file/[1,2]. The format of this file is internal. The state can then be used validating an XML document.

- `format_error() -> term()`

- `process_schema(Schema) -> Result`
Equivalent to process_schema(Schema, []) [page 31].

process_schema(Schema, Options) -> Result

Types:
- Schema = filename()
- Result = {ok, State} | {error, Reason}
- State = global_state()
- Reason = [ErrorReason] | ErrorReason
- Options = option_list()

Reads the referenced XML schema and controls it is valid. Returns the global_state() with schema info or an error reason. The error reason may be a list of several errors or a single error encountered during the processing.

process_schemas(Schemas) -> Result

Equivalent to process_schema(Schemas, []) [page 31].

process_schemas(Schemas, Options) -> Result

Types:
- Schemas = [{NameSpace, filename()} | Schemas] | []
- Result = {ok, State} | {error, Reason}
- Reason = [ErrorReason] | ErrorReason
- Options = option_list()

Reads the referenced XML schemas and controls they are valid. Returns the global_state() with schema info or an error reason. The error reason may be a list of several errors or a single error encountered during the processing.

process_validate(Schema, Xml::Element) -> Result

Equivalent to process_validate(Schema, Xml, []) [page 31].

process_validate(Schema, Xml::Element, Opts::Options) -> Result

Types:
- Schema = filename()
- Element = XmlElement
- Options = option_list()
- Result = {ValidXmlElement, State} | {error, Reason}
- Reason = [ErrorReason] | ErrorReason

Validates a parsed well-formed XML element towards an XML schema. Validates in two steps. First it processes the schema, saves the type and structure info in an ets table and then validates the element towards the schema.

Usage example:
1> {E, _} = xmerl_scan:file("my_XML_document.xml").

Observe that E2 may differ from E if for instance there are default values defined in my_XML_SCHEMA.xsd.
state2file(S::State) -> ok | {error, Reason}  
    Same as state2file(State,SchemaName)  
    The name of the saved file is the same as the name of the schema, but with .xss extension.

state2file(S::State, FileName) -> ok | {error, Reason}  
    Types:  
    • State = global_state()  
    • FileName = filename()  
    Saves the schema state with all information of the processed schema in a file. You can provide the file name for the saved state. FileName is saved with the .xss extension added.

validate(Xml::Element, State) -> Result  
    Equivalent to validate(Element, State, []) [page 32].

validate(Xml::Element, State, Opts::Options) -> Result  
    Types:  
    • Element = XmlElement  
    • Options = option_list()  
    • Result = [ValidElement, global_state()] | {error, Reasons}  
    • ValidElement = XmlElement  
    • State = global_state()  
    • Reasons = [ErrorReason] | ErrorReason  
    Validates a parsed well-formed XML element (Element).  
    A call to validate/2 or validate/3 must provide a well formed parsed XML element #xmlElement{} and a State, global_state(), which holds necessary information from an already processed schema. Thus validate enables reuse of the schema information and therefore if one shall validate several times towards the same schema it reduces time consumption.  
    The result, ValidElement, is the valid element that conforms to the post-schema-validation infoset. When the validator finds an error it tries to continue and reports a list of all errors found. In those cases an unexpected error is found it may cause a single error reason.  
    Usage example:  
    1> {E, _} = xmerl_scan:file("my_xml_document.xml").  
    2> {ok, S} = xmerl_xsd:process_schema("my_xmlSchema.xsd").  
    3> {E2, _} = xmerl_xsd:validate(E, S).  
    Observe that E2 may differ from E if for instance there are default values defined in my_xmlSchema.xsd.
Index of Modules and Functions

Modules are typed in this way.
Functions are typed in this way.

accumulate_whitespace/1
  xmerl_scan, 24

built_in_rules/2
  xmerl_xs, 28

callbacks/1
  xmerl, 19

cont_state/1
  xmerl_scan, 24

cont_state/2
  xmerl_scan, 24

event_state/1
  xmerl_scan, 24

event_state/2
  xmerl_scan, 24

export/2
  xmerl, 19

export/3
  xmerl, 19

export_content/2
  xmerl, 20

export_element/0
  xmerl, 20

export_element/3
  xmerl, 20

export_simple/2
  xmerl, 20

export_simple/3
  xmerl, 20

export_simple_content/0
  xmerl, 21

export_simple_element/0
  xmerl, 21

fetch_state/1
  xmerl_scan, 24

fetch_state/2
  xmerl_scan, 24

file/1
  xmerl_scan, 25

file2state/1
  xmerl_xsd, 30

file_sax/1
  xmerl_eventp, 22

format_error/0
  xmerl_xsd, 30

hook_state/1
  xmerl_scan, 25

hook_state/2
  xmerl_scan, 25

process_schema/1
  xmerl_xsd, 30

process_schema/2
  xmerl_xsd, 31

process_schemas/1
  xmerl_xsd, 31

process_schemas/2
  xmerl_xsd, 31

process_validate/2
  xmerl_xsd, 31

process_validate/3
  xmerl_xsd, 31

rules_state/1
  xmerl_scan, 25

xmerl Application