xmerl Application

version 1.0

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

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[1] site you will find the XML 1.0 specification[2] and other related specs. One site were you can find tutorials on XML and related specs is ZVON.org[3].

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 markup names that are encoded Unicode characters not mapping on ASCII.

[1] URL: `http://www.w3.org`
[2] URL: `http://www.w3.org/TR/REC-xml/`
[3] URL: `http://www.zvon.org`

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 convienient 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 {`DataStructure,M`}. `M` is the XML production Misc, which is the markup 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:

```
#xmlElement{name=Name,
            ...
            parents=Parents,
            ...
            attributes=Attrs,
            content=Content,
            ...}
```

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

The parse result will be:

```
#xmlElement{name = dog,
            ...
            parents = [],
            ...
            attributes = [],
            content = [{xmlText,[{dog,1}],1,[],"\nGrand Danois\n",text}],
            ...
            }
```

Where the content of the top element is: `[{xmlText,[{dog,1}],1,[],"\nGrand Danois\n",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 charactes between markup as `space`, `tab` and `line feed` are normalized and returned as xmlText records.

Errors

An unsucsessful 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 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;amp;H exhaust pipes</accessories>
  </bike>
  <date>2004.08.25</date>
  <bike year="1983" color="read 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 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)>

<!-- Date of the format yyyy.mm.dd -->
<!ELEMENT date              (#PCDATA)>
<!ATTLIST  bike year NMTOKEN #REQUIRED
               color NMTOKENS #REQUIRED
               condition (useless | bad | serviceable | moderate | good | excellent | new | outstandin
```

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").
{{xmlElement,motorcycles,
             motorcycles,
             [],
             {xmlNamespace,[],[]},
             [],
             1,
             [],
             [{xmlText,[{motorcycles,1}],1,[],"\n  ",text},
              {xmlElement,bike,
                          bike,
                          [],
                          {xmlNamespace,[],[]},
                          [{motorcycles,1}],
                          2,
                          [{xmlAttribute,year,[],[],[],[]|...},
                           {xmlAttribute,color,[],[],[]|...}],
                          [{xmlText,[{bike,2},{motorcycles|...}],
                                    1,
                                    []|...},
                           {xmlElement,name,name,[]|...},
                           {xmlText,[{...}|...],3|...},
                           {xmlElement,engine|...},
                           {xmlText|...},
                           {...}|...],
                          [],
                          ".",
                          undeclared},
              ...
              ],
             [],
             ".",
```

```
             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 ??].

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 `"&"` charcter it must be written as `"<"` and `"&"` 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 markup 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 apropriate 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)`. Content may be a mixture of simple-form and xmerl records as xmlElment and xmlText.

The Types are:

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

Element is any of:

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

- #xmlComment{}
- #xmlDecl{}

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 ??]

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,
     [{year,"2003},{color,"black"},{condition,"new"}],
     [{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>
```

```
    <kind>custom</kind>
    <drive>cardan</drive>
    <accessories>Sissy bar, luggage carrier,V&amp;amp;H exhaust pipes</accessories>
  </bike>
  <date>2004.08.25</date>
  <bike year="1983" color="read 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>
<bike year="2003" color="black" condition="new"><name><manufacturer>Harley Davidsson</manufacturer>
```

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 apropriate 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_simple/3. The following example code fixes those changes in the previous example:

```
    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:

```
<?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;amp;H exhaust pipes</accessories>
  </bike>
  <date>2004.08.25</date>
  <bike year="1983" color="read 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>
  <bike year="2003" color="black" condition="new">
    <name>
      <manufacturer>Harley Davidsson</manufacturer>
      <brandName>XL1200C</brandName>
      <additionalName>Sportster</additionalName>
    </name><engine>V-engine, 2-cylinders, 1200 cc</engine>
    <kind>custom</kind>
    <drive>belt</drive>
  </bike>
</motorcycles>
```

1.1.6 Example: Transforming XML To HTML

Assume that you want to transform the motorcycles.xml [page ??] 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:

```
2> {Doc,Misc}=xmerl_scan:file('motorcycles.xml').
{{xmlElement,motorcycles,
             motorcycles,
             [],
             {xmlNamespace,[],[]},
             [],
             1,
             [],
             [{xmlText,[{motorcycles,1}],1,[],"\n  ",text},
              {xmlElement,bike,
```

```
...
3> DocHtml=xmerl:export([Doc],xmerl_html).
["<!DOCTYPE HTML PUBLIC \"",
 "-//W3C//DTD HTML 4.01 Transitional//EN",
 "\"",
 [],
 ">\n",
 [["<","motorcycles",">"],
   ["\n  ",
    [["<",
      "bike",
      [[" ","year","=\"","2000","\""],[" ","color","=\"","black","\""]],
      ">"],
...
```

Will give the result result_export.html[4]

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 colours 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)[5] 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 ??]

First, some words about the xmerl_xs functionality:

You need to wright 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 wright a function:

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

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:

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

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:

```
template(E = #xmlElement{name='motorcycles'}) ->
    ["<p>",value_of(select("bike/name/manufacturer",E),"</p>"];
```

[4] URL: `result_export.html`
[5] URL: `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:

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

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:

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

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 fuction with the “built in clause” `template(E) -> built_in_rules(fun template/1, E).`

The entire program is motorcycles2html.erl:

```
%%%-------------------------------------------------------------------
%%% 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`

```
-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.~n",[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)),"</
    "<dd><ul>
",
    "<li style=\"color:green\">Manufacturing year: ",xslapply(fun template/1,select("@year",E)),"<
",
    "<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=[X1||X1={H,T} <- L],
    SortedTS = lists:keysort(1,Tuples),
    InsertRefName_UnWrap=
        fun([{[Name],V}|Rest],Name,F)->
                [V|F(Rest,Name,F)];
           ([{[Name],V}|Rest],PreviousName,F) ->
                [["<a name=\"",Name,"\"></>"],V|F(Rest,Name,F)];
           ([],_,_) -> []
        end,
    SortedRefed=InsertRefName_UnWrap(SortedTS,no_name,InsertRefName_UnWrap),
%    SortedTs=[Y||{X,Y}<-lists:keysort(1,Tuples)],
    WS = "
    ",
    Fun=fun([H|T],Acc,F)->
                F(T,[H,WS|Acc],F);
           ([],Acc,F)->
                lists:reverse([WS|Acc])
        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++"\">\s"++H++"</a></ul>";
make_ref([H|T]) when atom(H) ->
    ["<ul><a href=\"#"++atom_to_list(H)++"\">\s"++atom_to_list(H)++",
```

```
</a></ul>"
     |make_ref(T)];
make_ref([H|T]) when list(H) ->
    ["<ul><a href=\"#"++H++"\">\s"++H++",
</a></ul>"|make_ref(T)].
```

If we run it like this: `motorcycles2html:process_to_file('result_xs.html', 'motorcycles2.xml')`. The result will be result_xs.html[8]. 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.

1.2 Xmerl Release Notes

This document describes the changes made to the Xmerl application.

1.2.1 Xmerl 1.0.3

Known Bugs and Problems

- Removed call of undefined function in xmerl_lib.
  Own Id: OTP-5587

1.2.2 Xmerl 1.0.2

Known Bugs and Problems

- Better identification of errors in xml code.
  Own Id: OTP-5498 Aux Id: seq9803
- Some minor bugs fixed.
  Own Id: OTP-5500
- Parser failed on PE reference as EnumeratedType AttType, now corrected.
  Own Id: OTP-5531

1.2.3 Xmerl 1.0.1

Fixed Bugs and Malfunctions

- Fixed bug in xmerl_xpath. Xpath expressions that select nodes of type text() didn't work, like “contexto/text()”, “child::text()”, “descendant::text()”.
  Own Id: OTP-5268 Aux Id: seq9656
- Minor bugs fixed.
  Own Id: OTP-5301

There are also release notes for older versions[9].

[8] URL: `result_xs.html`
[9] URL: `notes_history.html`

1.2.4 Xmerl 1.0

Improvements and New Features

- The OTP release of xmerl 1.0 is mainly the same as xmerl-0.20 of http://sowap.sourceforge.net/. It is capable of parsing XML 1.0. There have only been minor improvements: Some bugs that caused an unexpected crash when parsing bad XML. Failure report that also tells which file that caused an error.
  Own Id: OTP-5174