tDOM – A fast XML/DOM/XPath package for Tcl written in C[1]

Jochen Loewer (loewerj@hotmail.com)

Abstract

The design, implementation and use of the tDOM package for the scripting language Tcl [9] is explained in this article. tDOM extents Tcl with features to handle and manipulate data and documents available in XML efficiently and easily, since tDOM is written in plain C and offers an object−oriented interface for manipulating the document objects in memory using standard DOM[4] methods. Beside the standard XML parser Expat[16], tDOM provides his own simple but fast XML parser, which directly builds up a DOM tree from the information in the parsed XML. In addition to XPointer−like navigation methods, the implementation of a complete XPath query engine in fast C enables the users of tDOM to easily write compact and expressive script code. It is further shown how to use the different tDOM methods/function in the various XML processing tasks like parsing/DOM build, serializing DOM, navigation within the DOM tree and modifying it. At the end some future enhancement are presented like DTD parsing, XPath extension functions, XML namespace support and XSLT.

1. Introduction

tDOM is a package for Tcl for easy and powerful XML/DOM processing with an extra focus on expressiveness and performance. In early 1999 a new upcoming project, which uses structured, hierarchical data serialized in XML, triggered an investigation on which tools for XML/DOM processing are available for Tcl, which was the desired implementation language, since it was well established in our group and proven to be easy to code and at the same time powerful.

At that time frame only TclXML/TclExpat and TclDOM from Steve Ball [1][2] were available. The projects required the use of DOM − just relying on event based XML parsing (SAX) would have been too complex and verbose in coding.

So TclExpat/TclXML alone wouldn't do the job. TclDOM implemented DOM, but beside the fact that it was implemented in plain Tcl and therefore quite slow and memory hungry, it just provided a DOM functionality in very Tcl−like and verbose calling syntax and didn't offer any advanced navigation and query features like XPointer or XPath.

Easy writeable, readable, compact and expressive code which follows the DOM recommendation's syntax closely was one for our major objective. In this way the gained knowledge of the developers about XML/DOM could be re−used in future projects, which might be implemented in other languages. Support for Unicode/UTF−8 wasn't a requirement, since we even want to stay with Tcl8.0.5.

So the decision was made to start the development of a fast DOM implementation for Tcl in C. Expat was initially taken as the XML parser since it was freely available and proven to be fast, reliable and standard conform. We also integrated/kept Steve Balls TclExpat, which provides a good event−based/SAX−like XML parser to Tcl. At that time libxml, a library developed for the GONME desktop,

[1] tDOM can be found at http://sdf.lonestar.org/~loewerj/tdom.cgi

showed up. libxml [19] was quite promising (own XML parser, own DOM, all written in plain C, a good C API, begin of an XPath implementation), but the initial license was to restrictive (note: Currently libxml can also be used under more liberal W3C license).

So the main design goals had been:

- closely follow DOM recommendation
- easy to use, compact and expressive OO−like syntax
- enhanced navigation and manipulation features like XPointer searches, if possible XPath queries).
- fast parsing/DOM build up and fast serializing (DOM−>XML) to allow the information be transmitted between database, application server and client in plain XML and to be stored in simple BLOBs within the database in XML (fast parsing/load leads to a simplified architecture)

During development of the DOM objects special effort was made to ensure that the DOM structure require as few memory as possible, since later there could be many for these DOM node in memory affecting overall performance, if they consume too much memory.

With the later introduced simple XML parser, the load time on an DOM tree into memory could be even improved by factor two compared to the full−featured Expat parser.

The resulting great performance (~2 msec DOM built up time for 5K DOM tree on medium size machines) kept the architecture of first very big project (40k lines of code), which was based on tDOM quite simple:

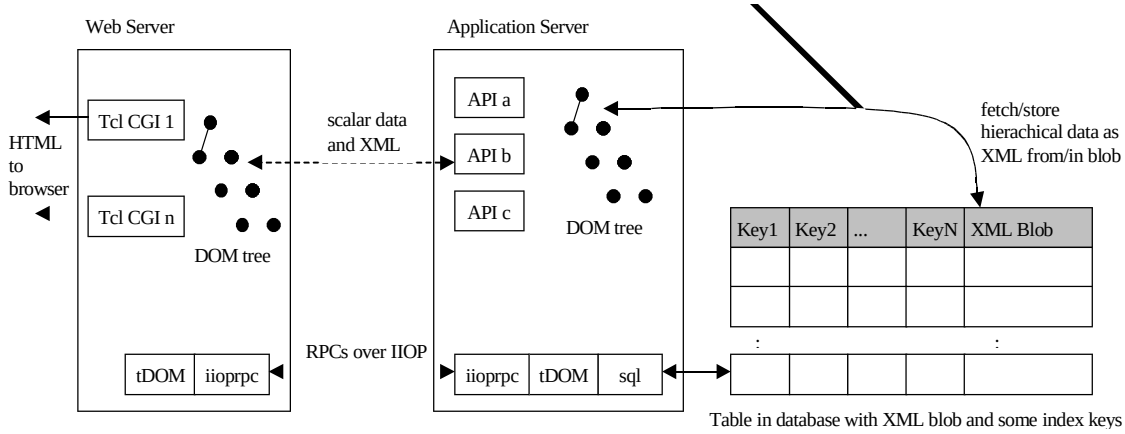

Figure 1: Larger project which uses tDOM

In that project world−wide complex configuration information for systems, kept in XML, is stored in BLOBs together with indexable search fields in a database. An application server or an API library provide a certain number of dedicated APIs, which load the raw XML from the BLOB in the database into the in−memory DOM and processed that DOM or return (part of) the data again as XML. The client side accesses the application server through remote procedure calls over IIOP and retrieve either scalar result values or XMLs. In this architecture there are several points where the DOM tree needs to be created out of the XML blob information. Using tDOM this never turned out to be the bottleneck.

In a different, smaller application, tDOM is used in a more dynamic way. The application acts as a middleware between an SAP R/3 instance and a another business server, which offers its business logic using XML as the request and reply format. Developers on the SAP side want to access this external business logic just using a normal ABAP/4 function call. The Tcl application server therefore connects to the SAP system through the RFC library (Remote Function Calls, basis for SAP BAPIs) and registeres a function within the R/3 system. When ABAP/4 programs then call that function a Tcl function is remotely

executed on the application server side. This Tcl function retrieves the calling parameters/tables, constructs a DOM document in memory according to the XML request specification of the business server and transmits the serialized DOM as XML over a publish/subscribe middleware (ActiveWeb) − this done all in plain Tcl. The response XML is parsed into a result DOM. Then all necessary information is extracted out of that DOM into some ABAP RFC export parameters and the SAP call is finished.

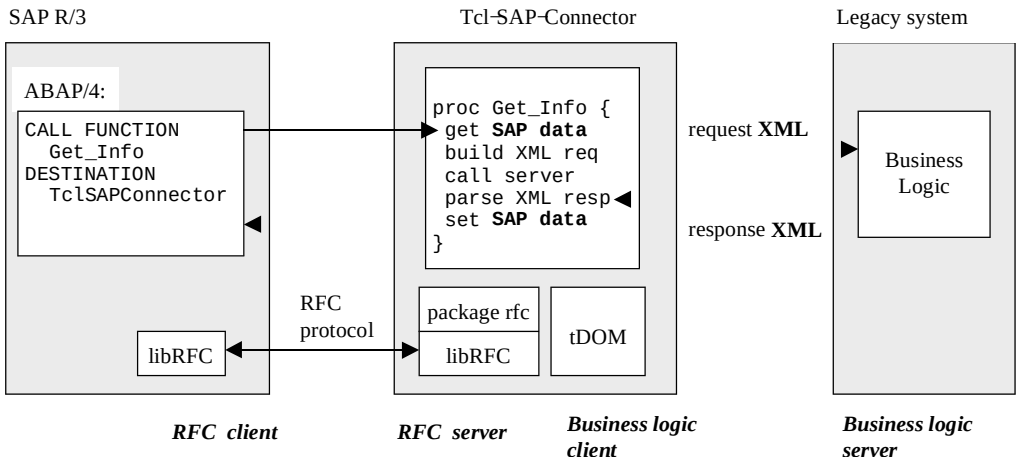

Figure 2: TclSAPConnector which converts RFC call in XML

This middleware application does basically the same thing as SAP BusinessConnector (webMethods) does (beside the nice web administration front end).

2. Achitecture

For a closer look at tDOM architecture look at Figure 3. As mentioned earlier, tDOM comes with a (modified version) of Expat. Expats works as the XML parser, which translates all possible incoming encoding internally into Unicode and outputs always UTF−8 encoded text. Expat interfaces to other components using callback functions, which are invoked based on XML parsing events (~ SAX like interface).

In tDOM two components are based upon Expat and install callback functions in Expat. Steve Balls TclExpat (`tclexpat.c`) is the first one. It provides the event−based XML parsing on Tcl level. The original TclExpat code was improved to gain performance by restricting the callback functions to be just simple Tcl procedure.

The original Tclexpat code allowed Tcl procedures plus some fixed parameters as well. Unfortunately this causes an extra evaluation on Tcl execution level for each call and since this will be done for each single element or text node, this extra work will sum up.

The other component which interfaces to Expat via callbacks is the DOM builder (in `dom.c`). It will create the DOM structures in memory as a side effect during XML parsing.

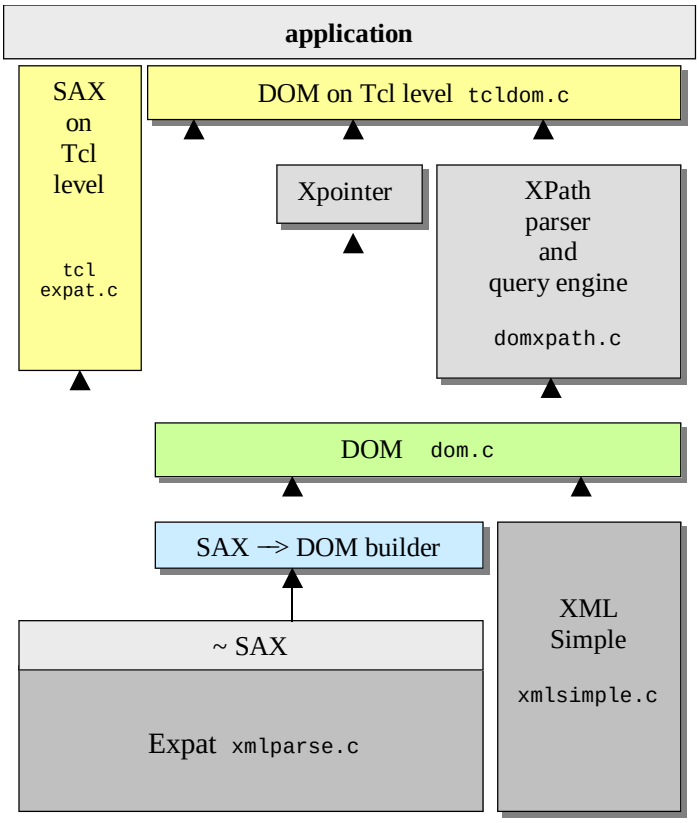

Figure 3: tDOM module architecture

Another way to build a DOM tree in memory is to use the simple XML parser (`xmlsimple.c`). This simple, restricted XML parser, based on an old XML parser from Dr. Richard Hipp, is basically one big C function, which parses the XML input without recursion/need of the C stack and directly builds up DOM structures.

In other words, when compared to Expat, the extra layer with the installable callback functions between the XML parser and the DOM builder functions is completely eliminated. This fact, as well as the non−recursive implementation of the simple XML parser, result in about twice the performance of the Expat+DOM builder bundle. The simple XML parsers handles all important XML features, like elements, attributes, comments, CDATA sections, processing instructions and the default entity replacements (& &quote; < > &27; &#x01f; ...) correctly. It misses complete entity reference replaceements, DTD parsing and encoding conversion. Text is just left untouched.

The `dom.c` module contains all the code to create/delete the DOM structure in memory. Here special effort was made to keep the memory requirements for the DOM structure as low as possible. The less memory is needed for the DOM tree, the greater the performance will be and the more DOM objects can be handle in memory at once. There are special structures defined in `dom.h`, which represent different DOM objects like element nodes, text/comment or CDATA nodes, processing instruction and attribute nodes in the individual minimum way. the definition of element nodes, text/comment or CDATA nodes and PI nodes share a standard header of fields like nodeNumer, nodeType, ownerDocument. After this common header there are the object specific fields. This will allow to apply specific casts in C code in order to access the right fields for individual objects.

Since only some applications like a XML editor would also like to store the line/column information within the original parsed XML document in the DOM structure, there exists special extension structures

for the element and text/comment nodes. A bit in the nodeFlag field indicates the availabilty of line/column information.

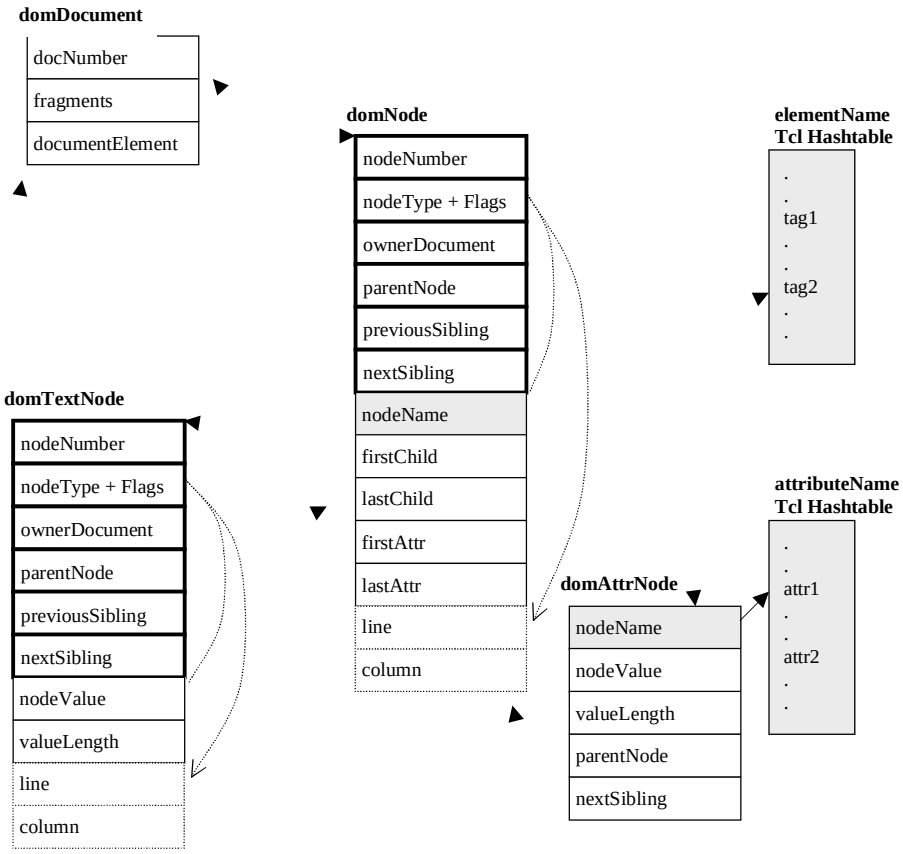

Figure 4: DOM object structures in memory

The element names and the attribute names are not store every time as an individual string together with the element or attribute node. Instead, they are stored once in a Tcl hash table and just the reference to the hash entry is stored in the DOM structure. This will further reduce memory requirements, since quite often the same tag or attribute names occure mutiple times.

Above the plain DOM layer, several XPointer navigation functions like child, psibiling, fsibling and descendant are implemented in dom.c. Since they are coded in C and operate directly on the DOM fields, they provide a very fast way to traverse and search the DOM tree, but offer only a subset of the XPath query features.

Beside the partial XPointer implementation there is an almost complete implementation of the XPath recommendation (`domxpath.c`). It contains a lexer and a parser to the parse the queries in the XPath syntax and a query engine, which uses the query plan based on the abstract syntax tree to retrieve the result of the given query by searching through the given DOM (sub)−tree. The lexer and parser are hand−written, so no compiler tools like (f)lex and yacc/bison have to be used.

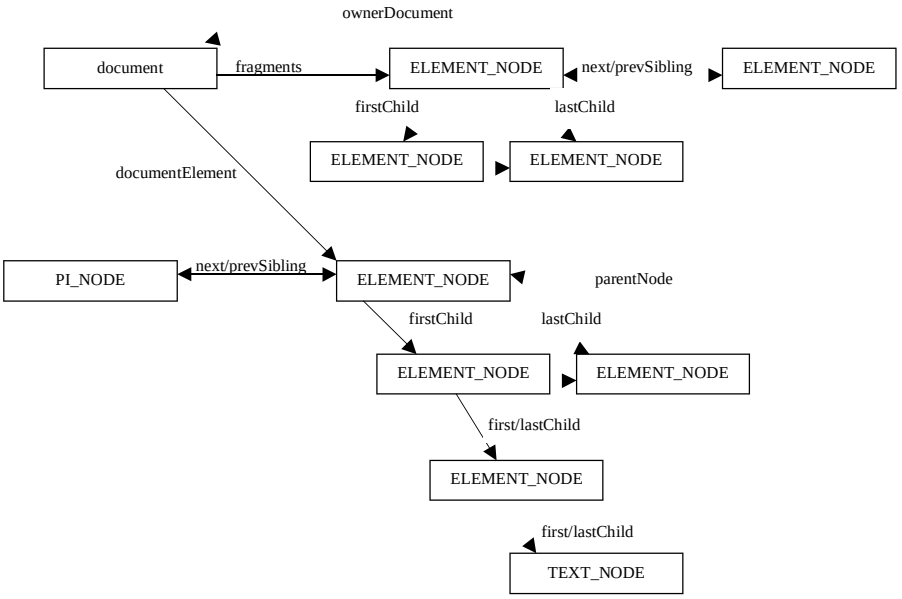

Figure 5: Links between DOM objects for a document

On top of the DOM, XPointer and the XPath layer there is the main Tcl DOM interface layer, which exports all the DOM/XPointer/XPath methods to the Tcl language level. It handles the link between basic memory address of DOM objects and the corresponding Tcl objects (object commands) as well as allocation/free of these object based on Tcl variable scopes. i.e. if an DOM docuemnt is bound to a Tcl variable on Tcl level and this Tcl variable will later automaticallybe freed, because the procedure is left, for example, the underlying DOM document is also freed.

tDOM is a relatively compact package in both C code size and object code size. This makes it well suited for embedded applications. With tDOM−0.5 it will be possible to compile tDOM without Expat support.
A system having the simple XML parser, DOM, XPointer, a full XPath implementation and the Tcl binding layer will just be around 80k binary code.

Code sizes for i386 code:

xmlsimple.o	<= 6 kB
dom.o	<= 11 kB
domxpath.o	<= 32 kB
tcldom.o	<= 32 kB
	<= **81 kB**
xmlparse.o	32 kB
xmlrole.o	12 kB
xmltok.o	100 kB
	145 kB Expat

3. How−to use tDOM

The next sections will focus on the usage of tDOM separated into certain topics. Beside that information the tDOM documentation (the documentation is in XML, precisely TMML, which was converted to HTML using tDOM's tmml2html convert) should be contacted to get syntax and usage of all available methods.

Parsing / XML loading / DOM build up

To access data stored in XML document, parsing the document and building up a DOM tree in memory are the most important things to do. Since this article focuses on DOM manipulation the event−based / SAX−like parsing using just the TclExpat features is not discussed here in detail (consult the tDOM documentation or TclXML [1] if more information is needed).

For parsing a document and building up the in−memory DOM tree multiple ways exists:

```
package require tdom

# read the XML document for a file in one chunk
set size [file size $xmlFile]
set fd   [open $xmlFile ]
set xml  [read $fd $size]
close $fd


1  dom setResultingEncoding iso8859−1
   set doc [dom parse $xml]

2  set doc [dom parse −simple $xml]

3  dom parse −keepEmpties $xml doc
```

In the above examples 1 and 3 uses Expat as underlying XML parser, which handles all the XML features. since internal Expat will always output UTF−8 encoded texts, a back conversion to 8bit character encoding can be made to enable tDOM to be used also in Tcl8.0 environments. With 'setResultEncoding' you can specify the output 8−bit character encoding (possible values are ascii, cp1250, cp1251, cp1252, cp1253, cp1254, cp1255, cp1256, cp437, cp850, en, iso8859−2, iso8859−3, iso8859−4, iso8859−5, iso8859−6, iso8859−7, iso8859−8, iso8859−9, koi8−r). If you leave 'setResultEncoding' or use 'utf−8' you'll get the direct Expat UTF−8 encoded text back. Possibility 2 makes use of the fast simple parser, which should provide double the Expat performance and leave the original text encoding as it is.

Per default text nodes, which contain just white spaces (space, tab, newlines) are not created at DOM build time. This is to able to store structure data in a pretty printed XML version:

```
<tag1>
    <tag2/>
</tag1>
```

This would result in that DOM tree structure:

```
o tag1
|
+−−o tag2
```

A one to one serialization of that DOM tree would output following XML:

```
<tag1><tag2/></tag1>
```

since there is no text between the two tags. If the `-keepEmpties` option is given, the following DOM will be generated:

```
o tag1
|
+--o textNode "\n    "
   |
   o tag2
   |
   o textNde "\n"
```

Memory handling / Automatic Free

In the above parsing examples two different return styles are used. The first two examples (1, 2) build up the DOM tree in memory and return a Tcl command name, which represents that tree in the Tcl world. This command is assigned to a Tcl variable as a string for later use.

The last example (3) specifies the Tcl variable as the last, additional argument. In this case tDOM can set the DOM document command name to that given Tcl variable and registers a deletion callback at the same time. This will trigger a deletion of the whole DOM tree at each time Tcl tries to delete the variable, either because of we left a procedure and all local variable are automatically freed or an explicit 'unset doc' forced a deletion.

```
 proc foo { xml } {
    dom parse -simple $xml localDoc
    set root [$localDoc documentElement]
    ...
    ... (extract data from the DOM)
    ...
    return    <--- here the Tcl var localDoc as well the
                   complete DOM document will be freed
}

proc getDoc { xml } {
    set doc [dom parse -simple $xml]
    return $doc
}
set doc [getDoc $xml]

...
... use $doc here, pass it to functions, etc.
...
$doc delete    <--- forces an explicit free
```

Depending on where and what is later done with the returned document handle choose one of the available methods. If the document handle should be returned to the calling level and is then used later in different calling level, you should you use the `'set doc [dom parse $xml]'` method. But don' forget to do a `$doc delete` later one. In all other cases you could use the other method, which frees you of remembering to delete the document explicitly and therefore avoids memory leaks.

Call Syntax

tDOM provides two different calling syntax for invoking the DOM methods on DOM objects. The first and easiest and natural one is a object−oriented or iTcl/Tk like way. Here tDOM creates Tcl command with standard prefix and increasing number which represent object in the DOM:

```
domDoc<N>   for DOM document objects
domNode<N>  for all nodes (element,text,comment,PI)
```

N is a number 1..

The basic calling syntax is like in iTcl/Tk;

```
$obj method arg1 arg2 ...
```

domNode<N> commands are not created in advance at DOM build time. There are only created on request, for example hen a DOM navigational method return that object.

```
% set doc [dom parse $xml]
domDoc1

% set rootNode [$doc documentElement]
domNode1

 % domNode2 nodeType           <−−− domNode2 command has not been created yet
invalid command name "domNode2"

% set child [$rootNode firstChild]  <−−− return reference to domNode2
                                        will create a command
domNode2

% domNode2 nodeType
ELEMENT_NODE
```

There are no attribute, NamedNodeMap or DocumentFragaments objects as in the original DOM recommendation. All these objects are handled using basic Tcl objects (Tcl lists) or their methods are accessible through some methods in the node object:

```
% set doc [dom parse "<tag a='1' b='2' c='3'/>"]
domDoc2
% set root [$doc documentElement]
domNode5
% $root attributes
a b c
```

The other builtin calling syntax for DOM nodes is more Tcl'ish:

```
domNode $nodeHandle method arg1 arg2 ...
```

Here there is one (class) command, which requires a DOM node object command name or DOM node reference name in form domNode0x<addr> as the first argument. The second argument is the method; all following arguments are method arguments.

```
% dom parse "<tag1><tag2/></tag1>"
domDoc1

% domDoc1 documentElement
domNode1

% domNode domDoc1 firstChild
domNode0x40023098

% domNode domNode0x40023098 parentNode
domNode0x40022fd8
```

```
% domNode0x40022fd8 firstChild
invalid command name "domNode0x40022fd8"
```

Since this dangerous calling method saves just a few percent of the total execution time (by avoiding command creation) the easier to use and to handle first OO−like calling syntax should be used.

tDOM's unknown

tDOM renames the normal Tcl unknown command and provides his own unknown command, which enables the user together with the DOM and XPointer methods to write complex DOM/XPointer expression in direct XPointer notation or use just a more convenient syntax for nested DOM methods, as
some one is used to have in C++ or Java.

```
1  set tagName [[[$node child 1 event type] firstChild] nodeName]

2  set tagName [$node.child(1,event,type).firstChild.nodeName]
```

Serialization

For serializing the in−memory DOM tree again into XML the standard DOM recommendation offers no methods. tDOM extents the node object interface with two fast serialize methods 'asXML' and 'asList' (see Figure 6):

```
% set doc [dom parse "<tag1><tag2/></tag1>"]
domDoc1
% set root [$doc documentElement]
domNode1

% $root asXML
<tag1>
    <tag2/>
</tag1>

% $root asXML −indent 1
<tag1>
 <tag2/>
</tag1>

% $root asList
tag1 {} {{tag2 {} {}}}
```

asXML operates on any (sub)−element and outputs therefore a XML subdocument without the <?xml version="1.0"?> header. This has just to be prepended in order to reproduce the original XML document again.

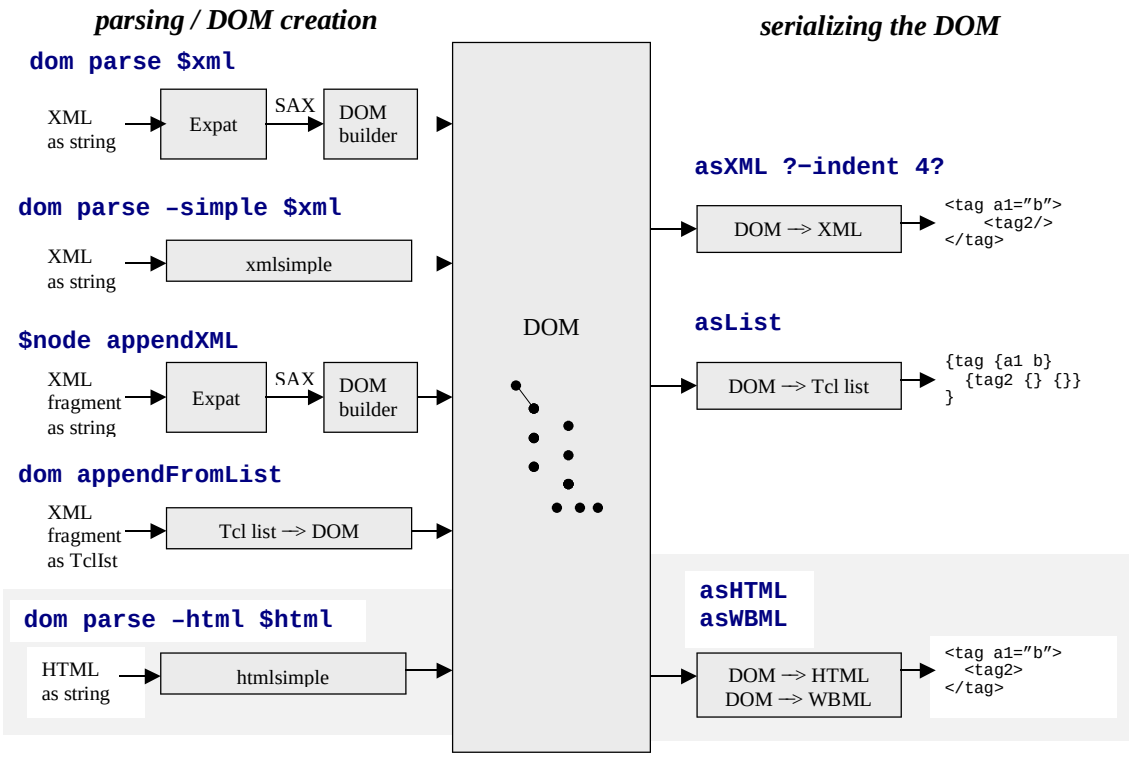

Figure 6: Creation and Serializing of DOM trees

Beside the serialization methods there also exists two helper methods, which provide a convenient way to append a subtree/document fragment to node from a XML string representation or the DOM Tcl list serialization:

```
% $root appendXML "<newTag a='1'/>"
domNode1

% $root asXML
<tag1>
    <tag2/>
    <newTag a="1"/>
</tag1>

% $root appendFromList {secondTag {attr value b 2}
                            {  { thirdtag {} {} }
                            }
                       }
domNode1
% $root asXML
<tag1>
    <tag2/>
    <newTag a="1"/>
    <secondTag attr="value" b="2">
        <thirdtag/>
    </secondTag>
</tag1>
```

In future serialization to HTML or WAP's WBML could be implemented.

DOM Manipulations − Document Fragment List

Methods for creation/removing/replacing/cloning of nodes create nodes, which are not yet part of the document tree, but part of the document. In order to be freed later, when the document is freed, these unlinked nodes are added to a hidden document fragment list. Methods, which append or insert nodes into the tree structure require the nodes to be taken from that document fragment list.

```
    % set doc [dom parse \
      {<tags>
           <first_child/>
           <second_child attr1="value1">
               <grandchild1/>
               <grandchild2 attr2="value2"/>
           </second_child>
           <third_child>
              <grandchild3 attr1="value1−3"/>
           </third_child>
       </tags>}]

    % set root [$doc documentElement]
    domNode1
    % set secondChild [$root descendant all second_child]
    domNode4

 1: % $root removeChild $secondChild
    domNode4

    % $root asXML
    <tags>
        <first_child/>
        <third_child>
            <grandchild3 attr1="value1−3"/>
        </third_child>
    </tags>

    % set thirdChild [$root selectNodes //second_child]
    domNode6

 2: % $thirdChild appendChild $secondChild
    % $root asXML
    <tags>
        <first_child/>
        <third_child>
            <grandchild3 attr1="value1−3"/>
            <second_child attr1="value1">
                <grandchild1/>
                <grandchild2 attr2="value2"/>
            </second_child>
        </third_child>
    </tags>

 3: % $thirdChild appendChild $secondChild
    HIERARCHY_REQUEST_ERR
```

At 1: the complete subtree starting with the node with nodeName ’second_child’ is moved into the hidden document fragments list. The removed subtree doesn’t show up anymore when the DOM tree is serialized to XML afterwards. In 2: the removed subtree is appended to the childs of the node with nodeName ’thirdChild’. A printing of the DOM tree as XML shows, that the complete subtree moved under the ’third_child’ tag. If accidentally the subtree is tried to be appended a second time, the DOM exception ’HIERARCHY_REQUEST_ERR’ will be raised.

Xpointer Navigation

The DOM recommendation provides just the most primitve methods to jump from one node to the next (fristChild,nextSibling,parentNode). The most sophisticated method getElementsByTagName just

retrieves all descendant elements which match the given tag name. In order to provide more advanced select/filter or query functionalities to applications some complex recursive search functions have to be implemented in Tcl. Having a recursive Tcl function visiting all node of an DOM tree could be quite expensive. At least this function won't be used for every simple DOM node lookup. The old XPointer working draft [18] specifies some advanced navigation function, which are implemented in tDOM using very fast recursive C
functions. They should be the faster possibility to navigate through the complete DOM hierachy:

child number|all ?type? ?attrName attrValue?

This will return all child nodes or a specific one, which match a certain node types (#text, #cdata, #element or #all) and optional also have a certain attribute with a certain value. Examples

```
$node child 2 event
```

will return the second event child node

```
$node child -1 event datetime 20000511
```

will return the last event child node, which has an attribute event with a value '20000511'

```
$node child all #text
```

will return the all text child nodes

fsibling number|all ?type? ?attrName attrValue?
psibling number|all ?type? ?attrName attrValue?

similar to child, but they will query upon all following sibling or perceeding siblings.

ancestor number|all ?type? ?attrName attrValue?

will walk up the parentNode hierarchie until a match is found

descendant number|all ?type? ?attrName attrValue?

will do a depth−first search down the subtree applying the same filter/select logic as the child method. When a node is found, which matches the select criterias the tree traversal is stop here, i.e. no deeper node are visited. This is different to the //<tag> XPath expression, which is described later, which returns all matching nodes in any depth.

XPath Queries

While the Xpointer navigation methods provide some enhanced search feature with a somehow fixed query/filter expression, XPath [7] allows very complex and compat queries within of one DOM tree comparable to advanced SQL where clauses. domxpath.c in tDOM, the most complex module within DOM, contains a lexer, a parser for the XPath language and a fast query engine. Currently it provides an almost full XPath implementation, without the namespace and id functions.

To make extensive use of XPath expression in your Tcl code, could greatly ease up and improve code development, while still being a very fast way to extract data form part in the DOM tree. The selectNodes method is the method to evalute a DOM on a subtree:

```
$node selectNodes <XPathString> ?resultType?
```

XPath allows different types for an result of a XPath query depending on the given query:

empty	selectNodes returns a empty list
bool	selectNodes returns 0 or 1
number	selectNodes returns either an integer or a float value
nodes	selectNodes returns a list of nodes (i.e. names of domNode<n> commands)
attrnodes	selectNodes returns a list of attrName attrValue pairs (like Tcl array get)
attrvalues	selectNodes returns a list of attribute values

Here are some examples:

This will loop overall event elements, which have a attribute with the value "server1":

```
foreach event [$node selectNodes { //events[@origin="server1"] } {
}
```

The result was a node list.

This will loop overall event elements which have a attribute with the value "server1" and have a severity subelement, which has an attribute 'type' with value 'critical'

```
foreach event [$node selectNodes { //events[@origin="server1"]
                                          [severity[@type="critical"] }
{
}
```

This will count all the count the number of critical events using the the standard count() function of XPath:

```
set nrCrit [$node selectNodes { count(//events[severity[@type='critical']
           }]
```

The result is a integer.

tDOM ships also with XE. XE is a Tk application, that allows in an easy way the evaluation of XPath queries on local or remote (fetch via HTTP from a server/database for example) XML documents. It formats the result nicely on the result pane and enables to browse/navigate through the sub DOM trees (see Figure 7).

As a big test XML database, tDOM contains a copy of the european XML Mondial database [1111], which contains data about countries/states/mountains/population assembled from many source, mainly the CIA world fact book.
Using the Mondial database the power of XPath queries can be easily demonstrated:

The query

```
//country[name='Germany']/province[population>5000000]/name
```

will return the name tags for all provinces in Germany, which have at least 5 million inhabitants.

The query

```
//mountain[in_country[@ref = string(//country[name='Germany']/@id]]
```

will return the mountain tag of that mountain, which is in Germany − in XML terms has the same id in the ref attribute of this in_country child as the id attribute of a county tag with a child with the text 'Germany'.

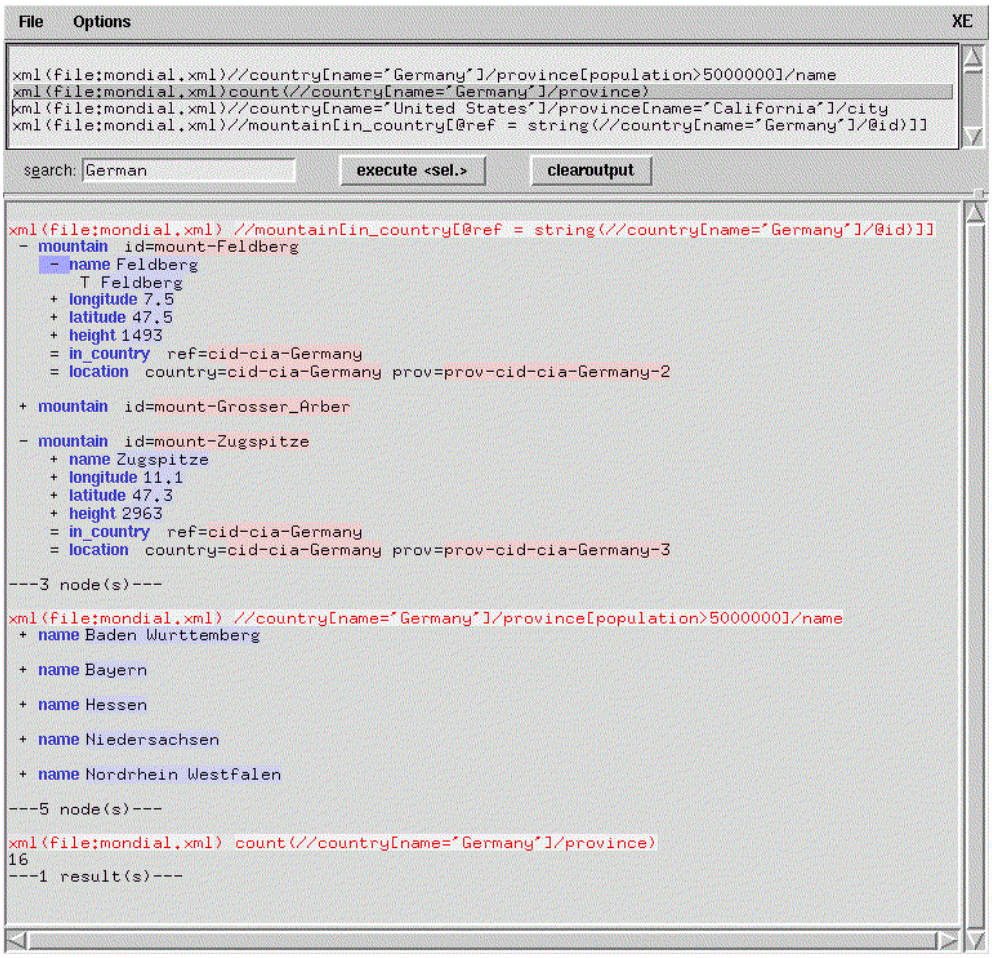

Figure 7: Screenshot of XE showing some complex XPath queries

Extension Namespaces

The method dispatching function in tDOM for the domDoc and domNode objects implement an easy to way to extend these object with new method directly implemented in plain Tcl. If a requested method can't be found in the C code the dispatching logic first tries to look for this kind of Tcl procedure:

```
::dom::domNode::<nodeName>::<method>
```

which is based on the tag name of the element node and the given method.

If this Tcl procedure is found the call

```
$nodeNameObj method arg1 arg2 ...
```

will be rewritten to

```
::dom::domNode::<nodeName>::<method> $nodeNameObj arg1 arg2 ...
```

So the first is always the 'this' object.

This lookup will allow to implement specific element based methods: Element nodes could behave like instanciated objects from a certain class. Imagine we have a HTML select node and would like to have convenient methods to add and delete an option, like addOption / deleteOption. These options require the insertion of a new option tag below the select node.

```
% set doc [dom parse "<select/>"]
% set selectNode [$doc documentElement]
% $selectNode addOption opt1 "First Option"
Usage nodeObj <method> <args>, where method can be:
...
% proc ::dom::domNode::select::addOption { node value text } {

       set doc [$node ownerDocument]
       set optionNode [$doc createElement option]
       set textNode   [$doc createTextNode $text]
       $node appendChild $optionNode
       $node appendChild $textNode
       $optionNode setAttribute value $value
  }
%
% $selectNode addOption opt1 "First Option"
% $selectNode asXML
<select>
    <option value="op1"/>
    First Option
</select>
```

If this procedure isn't found in that namespace the method dispatcher tries to lookup this procedure:

```
::dom::domNode::<method>
```

This allows to implement new DOM methods first on Tcl level. For example this following procedure implements the substringData method for the DOM CharacterData interface:

```
proc ::dom::domNode::substringData { node offset count } {

    set type [$node nodeType]
    if {($type != "TEXT_NODE") && ($type != "CDATA_SECTION_NODE")} {
        return -code error "NOT_SUPPORTED_ERR: node is not a cdata node"
    }
    set endOffset [expr $offset + $count - 1]
    return [string range [$node nodeValue] $offset $endOffset]
}

% set doc [dom parse "<text>Tcl and Tk</text>"]
% set node     [$doc documentElement]
% set textNode [$node firstChild]

% $node substringData 8 2
NOT_SUPPORTED_ERR: node is not a cdata node

% $textNode substringData 8 2
Tk
```

For the document object (domDoc<n>) a different namespace is search for method dispatching:

```
::dom::domDoc::<method>
```

For the DOMImplementation (factory) object, the Tcl function dom, the namespace for method dispatching is:

```
::dom::DOMImplementation::<method>
```

4. Future enhancements

The following is a list of planned or just possible future enhancements, which are roughly ordered by urgency and ease of implementation:

- store compiled XPath query plans (abstract syntax tree) in dual−ported Tcl Object for caching.
- XPath extension functions

  The XPath recommendation describes how specific implementation could extent the XPath query engine by functions, implemented outside the query engine – in the case tDOM in plain Tcl. It is expected that tDOM−0.5 will contain that feature. As an example imagine an XML−enabled application, which has the need the count the distinct number for certain tags.

  ```
  set criticalEvents [$node selectNodes //event[@datetime='20000615']
  ```

  This will not work , if the attribute datetime contains dates or datetimes in different formats like '`2000−06−15 12:31`', '`20000615`', '`20000702 02:27:33`' or even '`06/15/2000 12:31`'.

  To use the simple XPath query from above the datetime has to be converted/unified first. So if we would have a XPath function `date()` this query would work:

  ```
  set criticalEvents [$node selectNodes                          \
                              //event[date(@datetime)='20000615']
  ```

  This could be implemented with the XPath extension namespace in tDOM−0.5 in the following way:

  ```
  proc ::dom::xpathFunc::date { ctxNode pos nodeListType nodeList args } {
       if {[llength $args == 2]} {
           foreach {type value} $args break
           switch $type {
               string –
               attrvalues {
                   # for these type date() is defined
                   break
               }
               default {
                   # for the types empty/bool/number/nodes/attrnodes
                   # date() is not defined
                   return –code error "bad argument type"
               }
           }
           # now actually convert the date string
           set convertedDate [UnifyDate $value]

           # return a string object back to the XPath query engine
           return [list string $convertedDate]

       } else {
           return –code error  "wrong number of args: date(x)"
       }
  }
  ```

  The XPath has a generic C callback interface for pluging in new functions. tDOM−0.5 will provide a Tcl mapping using that callback interface.

- XML−Namespace support in the DOM

  As the use of XML documents with XML namespace continues to grow and since newer standards like XSLT completely rely on namespaced tags, namespace support in tDOM and its underlying C structures become more and more important. tDOM−0.5/0.6 will hopefully only slightly extend the DOM memory structures in order to hold the XML namespace information.

The idea is to keep the memory requirements as low as possible. On the Tcl level the DOM level 2 method should be available then.

- make DTD information available

  Scriptics has already modify the original Expat parser to invoke callback when DTD information is processed. This included element definition and restricted attribute definition. To have the DTD information available enables quite a lot of further capabilities, like XML validation or the generation of dedicated load/serialize functions (a DTD −> Tcl compiler, which outputs tailored Tcl code for certain XML−aware business objects). Before tDOM−0.5 some effort was made the further extent the Expat parser based on the Scriptics changes. In the mean time Perls XML−Parser−2.29 comes with an extended Expat parser as well, which has all the needed features. Therefore tDOM−0.5/0.6 might use the Expat code from XML−Parse−2.29.

- HTML parser / Tidy integration

  To be able to parse existing HTML code and build up a DOM tree, enables a rich set of application which extract /condense/mediate/index information on the internet, when combined with Tcl scripting and XPath queries. Things like webMethods WIDL [11] could be easily implemented. The HTML parser could be taken from Tidy[12], which is implemented in C, but also provide some warning/rewriting tips while parsing bogus HTML. A Tcl−fied Tidy would provide the basis for tools like a 'TkTidy'.

- HTML serializer ((`$node asHTML`) for XML −> DOM −> docProcessor −> HTML converters
- WBML serializer (`$node asWBML`) for WAP devices
- XPath query engine performance improvements (keep already evaluated sub−expressions as in `//mountain[in_country[@ref = string(//country[name='Germany']/@id)]]`, use tag/attribute occurence list/index)
- provide TclDOM compatibility mode

  In order to re−use all the existing code developed using TclDOM and to gain execution speed, all compatibility layer above tDOM written in Tcl would be desirable. tDOM has already a simple Tcl like (non OO like) calling interface, which should provide an easy way to simulate the TclDOM calling interface.

- persistent DOM / XML database/store

  In order to avoid the sometimes time consuming part of parsing serialized DOM tree from a externally stored XML document and building the DOM tree in memory again and again, the DOM could be made persistent once. All the tDOM DOM operations as well the XPath queries will then be evualated on top that persistent DOM tree later on. Some experiments with memmory mapped files (mapped a fixed locations) and malloc/free replacement functions, which operated on that shared memory, showed already good results for Tcl array like persistent data structures, even in a multi−user/process environment. The fixed memory mapped files make the interface simple and shift the burden of memory/disc optimization to the operating system. With 64−bit operating systems with a large virtual address space (HP/UX 11, ...) the fixed location shouldn't be a problem anymore.

- XML−RPC / SOAP /XIOP implementation
- XSLT written in Tcl or even better in plain C
- implement a MOM, a message oriented middleware, re−using the www.xmlBlaster.org ideas and the tDOM DOM and XPath features.
- make virtual DOM trees avialable for content/database/legacy system intregation (like Tamino's X−Node machine[13], TSIMMIS/Garlic's mediators [14][15])

  If DOM nodes are not just fixed structures, but evaluted/generated on the fly using adaptor or mediator functions contacting different (remote) sources like databases/legacy systems, the tDOM navigation functions (firstChild, getElementsByName) and XPath queries could be applied on top these uniformed DOM tree.

5. References

[1] Steve Ball; TclXML; http://www.zveno.com/zm.cgi/in−tclxml
[2] Steve Ball; TclDOM, http://www.zveno.com/zm.cgi/in−tclxml/in−tcldom
[3] Carsten Zerbst; XML−Programmierung; iX 2/2000, page 132;
http://www.heise.de/ix/artikel/2000/02/132/
[4] W3C; Document Object Model (DOM); http://www.w3.org/DOM/
[5] W3C; Document Object Model (Core) Level 1;
http://www.w3.org/TR/REC−DOM−Level−1/level−one−core.html
[6] Megginson; SAX 1.0: The Simple API for XML;
http://www.megginson.com/SAX/SAX1/index.html
[7] W3C; XML Path Language Version 1.0; http://www.w3.org/TR/xpath
[8] Brent B. Welch; Practical Programming in Tcl and Tk; Prentice Hall
[9] Mozilla; Introduction to a XUL Document; http://www.mozilla.org/xpfe/xptoolkit/xulintro.html
[10] Comanche, a GUI for Apache; http://www.covalent.net/projects/comanche
[11] webMethods; WIDL http://www.webmethods.com/ ; http://www.w3.org/TR/widl
[12] Tidy; http://www.w3.org/tidy
[13] Software AG; Tamino XML database; http://www.softwareag.com/tamino/
[14] TSIMMIS project; Stanford University; http://www−db.stanford.edu/tsimmis
[15] The Garlic project; IBM Research Almaden;
http://www.almaden.ibm.com/cs/garlic/homepage.html
[16] James Clark; Expat; http://www.jclark.com/xml/expat.html
[17] Wolfgang May; Mondial database in XML;
http://www.informatik.uni−freiburg.de/~may/Mondial/#XML
[18] W3C; XPointer 97; http://www.w3.org/TR/WD−xml−link−970731