Avoiding Problems Reading DOM XML Nodes

By David Adams

Technical Note 06-44

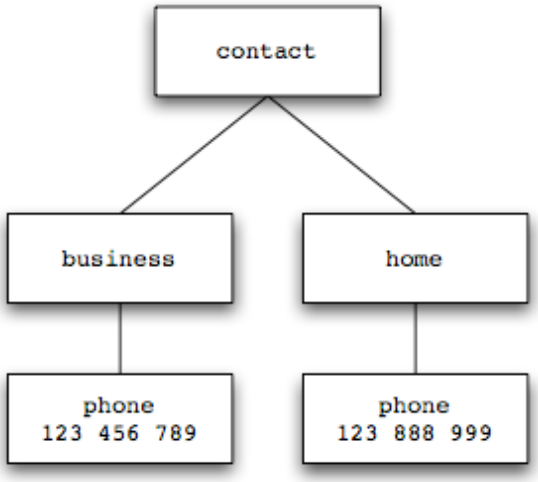
Overview

4th Dimension's DOM (Document Object Model) XML commands render a source XML document or variable as a tree of linked nodes. The DOM command set includes tools for navigating through the tree and reading information from nodes. The tree navigation commands, such as **DOM Get Parent XML element** and **DOM Get first child XML element**, use the OK system variable to indicate when the command has left the tree and reached a non-existent node. This feature makes it possible to navigate through the tree without knowing in advance exactly how many ancestors, siblings, or descendants a particular node has. However, it also leads to the navigation commands returning references to invalid nodes. This creates problem when reading an element's name, value, or attributes. This technical note explains some simple strategies for managing bad nodes and avoiding problems.

Review of DOM Features and Behavior

4th Dimension 2004 includes two suites of commands for reading XML, the DOM commands and the SAX commands. This technical note deals only with the DOM commands. As mentioned, the DOM commands treat parsed XML as a tree of linked nodes. As an example, consider some simple XML and a diagram of the related DOM tree:

```
<?xml
     version="1.0"
     encoding="UTF-8"
     standalone="no" ?>

<contact>

  <business>
    <phone>123 456 789</phone>
  </business>

   <home>
    <phone>123 888 999</phone>
  </home>

</contact>
```



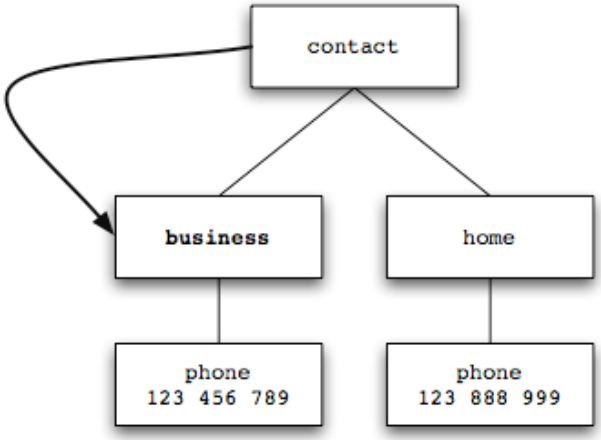
There are many ways to produce the XML shown above, including the lines of code listed below:

```
C_STRING(16;$root_xmlref)
C_STRING(16;$businessPhone_xmlref)
C_STRING(16;$homePhone_xmlref)
$root_xmlref:=DOM Create XML Ref("contact")
$businessPhone_xmlref:=DOM Create XML element($root_xmlref;"/contact/business/phone")
$homePhone_xmlref:=DOM Create XML element($root_xmlref;"/contact/home/phone")
DOM SET XML ELEMENT VALUE($businessPhone_xmlref;"123 456 789")
DOM SET XML ELEMENT VALUE($homePhone_xmlref;"123 888 999")
```

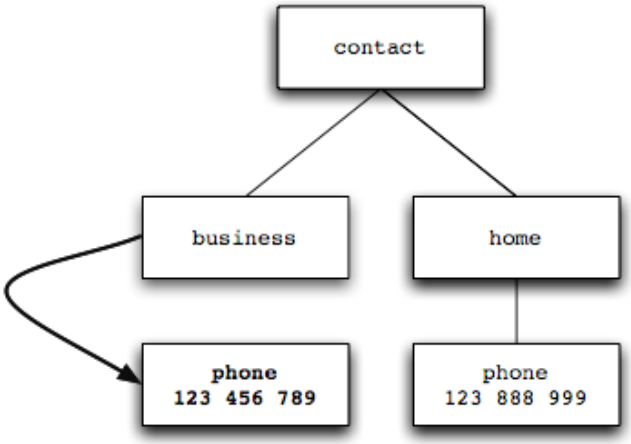
Now consider some code that steps through the elements in this XML reading each name and value:

```
C_STRING(16;$working_xmlref)
C_TEXT($elementName_t)
C_TEXT($elementValue_t)
$working_xmlref:=DOM Get first child XML element($root_xmlref)
DOM GET XML ELEMENT NAME($working_xmlref;$elementName_t)   ` business
DOM GET XML ELEMENT VALUE($working_xmlref;$elementValue_t)

$working_xmlref:=DOM Get first child XML element($working_xmlref)
DOM GET XML ELEMENT NAME($working_xmlref;$elementName_t)   ` phone
DOM GET XML ELEMENT VALUE($working_xmlref;$elementValue_t)  `123 456 789

` The next node is off the tree and invalid.
$working_xmlref:=DOM Get first child XML element($working_xmlref)
DOM GET XML ELEMENT NAME($working_xmlref;$elementName_t)
DOM GET XML ELEMENT VALUE($working_xmlref;$elementValue_t)

DOM CLOSE XML($root_xmlref)
```

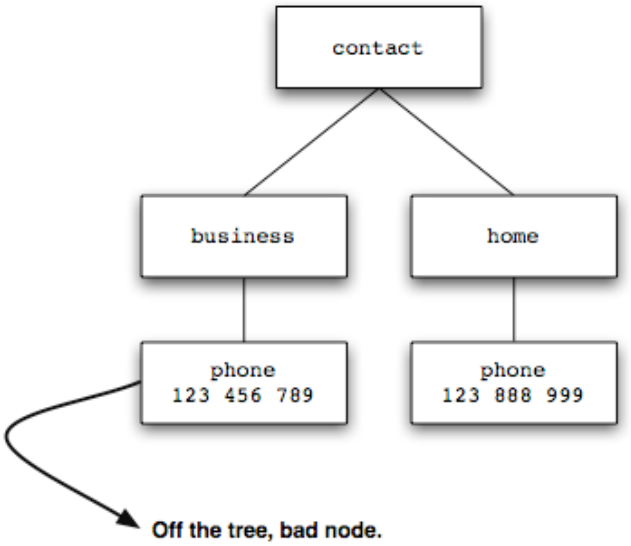
The $root_xmlref in the first line of code contains a reference to the XML tree produced in the earlier code fragment. The root reference could have also been obtained by parsing stored XML using **DOM Parse XML source** or **DOM Parse XML variable**. Within the tree, the first call to **DOM Get first child XML element** moves from the `custom` root element to the `business` element, as illustrated below:



The second call to **DOM Get first child XML element** moves from the `business` element to the `phone` element, as illustrated below:



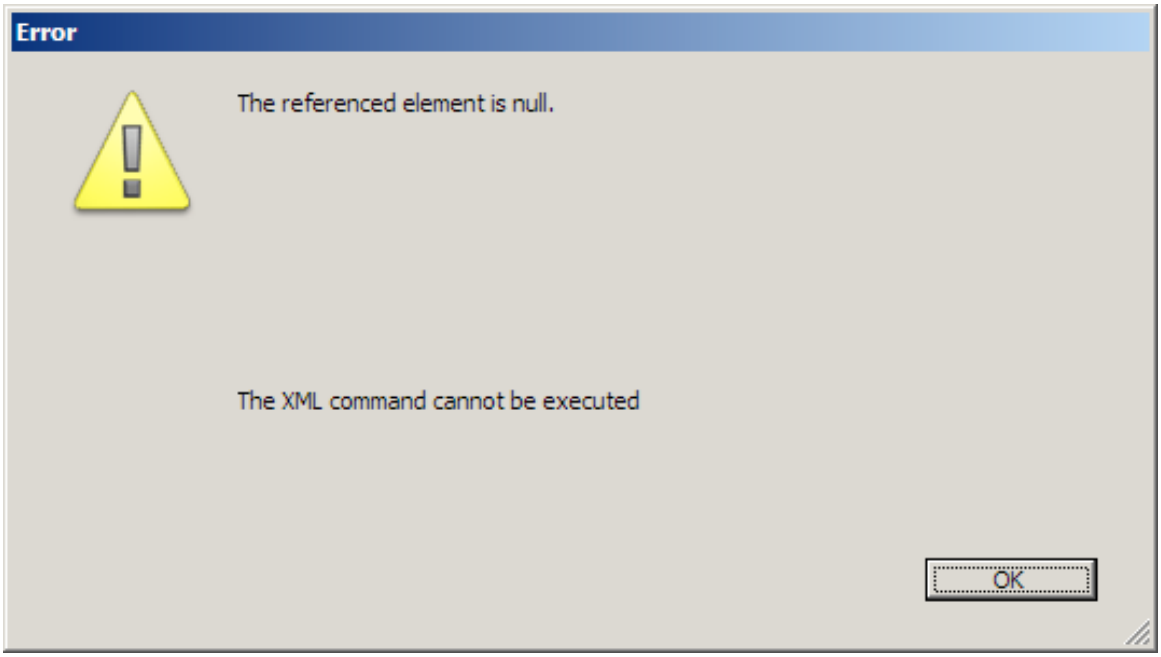
The third call to **DOM Get first child XML element** moves from the `phone` element off the tree, as illustrated below:



At this point, 4th Dimension sets the OK system variable to 0. Again, this is all normal behavior and does not represent a bug. However, what happens if code attempts to read an invalid node reference? Consider the line of code shown below when $working_xmlref holds an invalid reference:

```
DOM GET XML ELEMENT NAME($working_xmlref;$elementName_t)
```

In some versions of 4th Dimension, some commands quit 4th Dimension unexpectedly when passed a bad node reference. The correct and expected behavior is for an error to be thrown. If there is no error handler in place, then 4th Dimension displays an error dialog, like the following:



Avoiding Problems While Reading Nodes

Overview

The relevant information from the preceding discussion can be summarized as follows: It's normal to navigate to invalid nodes using the DOM commands, but some of the DOM information functions have problems with invalid node references. Fortunately, there are several simple strategies that help you avoid problems with the DOM commands:

Test the OK Variable

The DOM navigation commands set the OK system variable to 0 when moving to an invalid node. In this manner, for example, generic tree navigation routines can detect when the last ancestor, sibling, or child node has been reached. Likewise, any DOM navigation code can test the OK system variable

before calling one of the DOM information reading commands. The problem code shown above is shown again below with tests on the OK variable:

```
C_STRING(16;$working_xmlref)
C_TEXT($elementName_t)
C_TEXT($elementValue_t)
$working_xmlref:=DOM Get first child XML element($root_xmlref)
If (OK=1)
   DOM GET XML ELEMENT NAME($working_xmlref;$elementName_t)   ` business
   DOM GET XML ELEMENT VALUE($working_xmlref;$elementValue_t)
End if

$working_xmlref:=DOM Get first child XML element($working_xmlref)
If (OK=1)
   DOM GET XML ELEMENT NAME($working_xmlref;$elementName_t)   ` phone
   DOM GET XML ELEMENT VALUE($working_xmlref;$elementValue_t)   `123 456 789
End if

$working_xmlref:=DOM Get first child XML element($working_xmlref)
If (OK=1)
   DOM GET XML ELEMENT NAME($working_xmlref;$elementName_t)
   DOM GET XML ELEMENT VALUE($working_xmlref;$elementValue_t)
End if
```

While there are easier to maintain and more flexible ways to approach reading XML than the code shown above, the simple example illustrates that the built-in OK system variable is enough to prevent problems, in most cases.

Use the Optional Parameters While Navigating Nodes

XML nodes have a name, a value, and may have attributes. As a convenience, the DOM navigation commands support optional parameters to hold the name and value of the selected node. This feature is very helpful as it simplifies the code and entirely avoids the problem of passing a bad reference to **DOM GET XML ELEMENT NAME** or **DOM GET XML ELEMENT VALUE**. The fragment below illustrates how much simpler code is when using this strategy:

```
C_STRING(16;$working_xmlref)
C_TEXT($name_t)
C_TEXT($value_t)
$working_xmlref:=DOM Get first child XML element($root_xmlref;$name_t;$value_t)   ` business
$working_xmlref:=DOM Get first child XML element($working_xmlref;$name_t;$value_t)   ` phone
$working_xmlref:=DOM Get first child XML element($working_xmlref;$name_t;$value_t)   ` <-- Bad
node
DOM CLOSE XML($root_xmlref)
```

The last call to **DOM Get first child XML element** returns an invalid node reference, as in the earlier examples. However, no error is generated. The name and value variables are empty and the OK system variable is set to 0. The main limitations of this technique are that it provides no support for reading node attributes and often doesn't help with generic tree-walking code.

Use an ON ERR CALL Method

Installing a custom error handler with **ON ERR CALL** is an effective technique for preventing invalid node reference errors from interfering with program operation. This approach is necessary if code attempts to count or read node attributes without testing the OK system variable.

Write a Node Validation Routine

It's possible to build a general node reference validation routine by installing a custom error handler and then calling a routine, such as **DOM GET XML ELEMENT NAME**, that provokes an error on invalid node references. The code below illustrates this technique:

```
C_BOOLEAN($0;$nodeRefIsOkay_b)
C_STRING(16;$1;$noderef)

$noderef:=$1
$nodeRefIsOkay_b:=True

Error:=0
ON ERR CALL("DOM_ReferenceIsValidOnError")

C_TEXT($elementName_t)  ` Line below should throw an error if the element isn't valid.
DOM GET XML ELEMENT NAME($noderef;$elementName_t)

ON ERR CALL("")

If ($elementName_t="")
   $nodeRefIsOkay_b:=False
Else
   $nodeRefIsOkay_b:=True
End if

$0:=$nodeRefIsOkay_b
```

The sample database includes a slightly more advanced version of this code in a routine named *DOM_ReferenceIsValid*.

Special Case: XML Attributes and the #document Node

One more special case deserves mention. An XML node may have zero or more attributes, such as the id attribute in the element shown below:

```
<contact id="1">
```

Within the DOM command suite, the number of attributes associated with a node can be counted with **DOM Count XML attributes**. Unfortunately, calling this function in one very special situation causes some versions of 4th Dimension to quit unexpectedly. Using **DOM Get parent XML element**, it is possible to navigate above the root of the tree to an artificial node that holds the XML document information, such as XML version and encoding type. The simple node validation test code listed earlier fails to see anything wrong with this node because it has a name and, in many respects, is a valid node. However, the **DOM Count XML attributes** function may not handle this node safely. Fortunately, the node is easy to recognize because it is

called `#document`. The code below shows how to count the attributes on any XML node without generating errors on invalid nodes or the `#document` node:

```
C_BOOLEAN($0;$attributes_count)
C_STRING(16;$1;$noderef)

$noderef:=$1

$attributes_count:=0

Error:=0
ON ERR CALL("DOM_ReferenceIsValidOnError")

C_TEXT($elementName_t)` Line below should throw an error if the element isn't valid.
DOM GET XML ELEMENT NAME($noderef;$elementName_t)

ON ERR CALL("")

Case of
   : ($elementName_t="")` Not a valid node, do not count attributes.
     $attributes_count:=0

   : ($elementName_t="#document")` Special info node, do not count attributes.
     $attributes_count:=0

   Else ` A valid node, count attributes.
     $attributes_count:=DOM Count XML attributes($noderef)
End case

$0:=$attributes_count
```

The sample database includes a slightly more advanced version of this code in a routine named *DOM_CountAttributes*.

Note *For more details on handling XML attributes, see Technical Note xx-xx,* ***Enhanced Tools for Reading XML Attributes****.*

Summary

The DOM parsing commands convert XML into a tree of linked nodes. The DOM XML commands provide complete freedom of movement within the tree but can navigate off the tree altogether. While this situation is normal, it can cause problems when developers try to read element names, values, or attributes. Using the built-in features and simple techniques explained in this technical note, 4th Dimension developers can avoid all problems related to reading XML nodes through DOM.