Testing if an XML Node Has Relatives

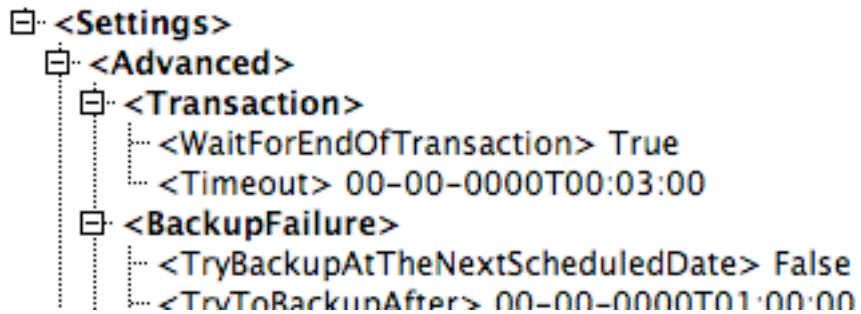
By David Adams
Technical Note 07-04

Abstract

This technical note explains how to test a node for ancestors, children, or siblings and includes a sample database with the method implemented using the built-in 4th Dimension DOM commands.

Overview

4th Dimension's native XML commands provide detailed information about an XML element, including the element's name, attributes, value, and CDATA section contents. However, it is often useful or necessary to determine if the current node has parent, child, or sibling nodes. As an example, imagine a database that displays XML settings in a hierarchical list. To make the contents of the list easier to understand visually, elements with children might be in bold, as in the screenshot below:



The code fragment below illustrates how to achieve this visual effect:

```
If (XmlNode_HasChildren ($xml_node))
  SET LIST ITEM PROPERTIES($list;$list_item;False;Bold;0)
End if
```

Some other examples of reasons to test for related nodes are listed below:

- Various generic XML tree-walking algorithms require testing for children and parents to determine when to stop.
- Custom XML import code may need to know if a node has more siblings or any children to determine when to save and create records and related records.
- When the same element name is used at different levels within an XML tree, testing for ancestors, children, or siblings is sometimes enough to determine which instance of an element name is current.

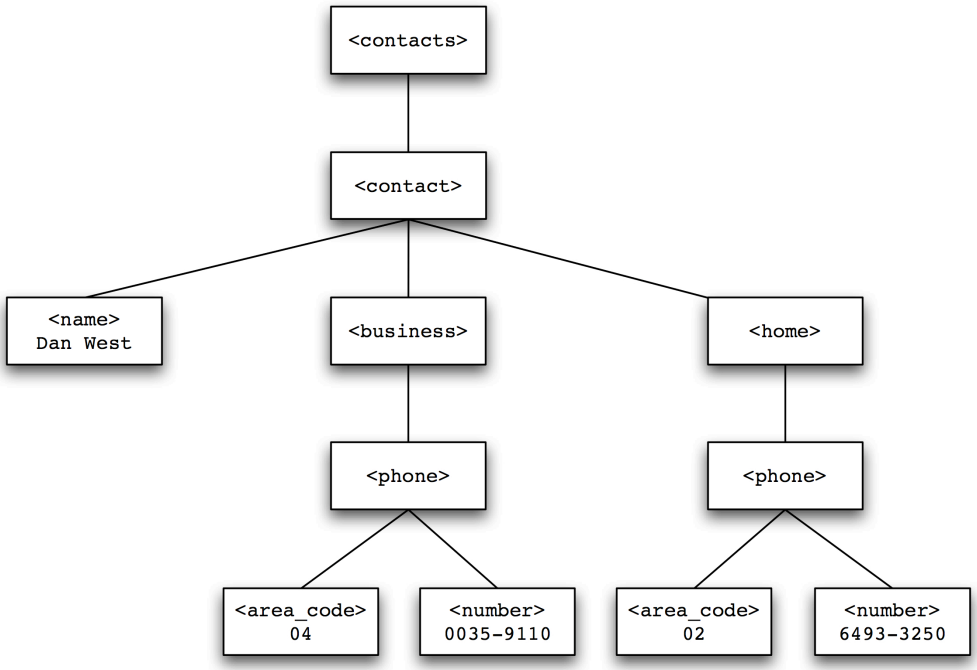
This technical note describes routines to test a node for ancestors, children, or siblings and includes a sample database with the method implemented.

Background: DOM, SAX and Trees

Before looking at the code, it is worth reviewing a few points about XML processing. The 4th Dimension language supports two different parsers, DOM (Document Object Model) parser and the SAX (Simple API for XML) parser. Both tools provide access to the same information about a node but approach XML processing very differently. Using the SAX model, an XML document is treated as a stream of text. In this environment, there is not really a concept of related nodes. Using the DOM model, an XML document is rendered internally in a tree of linked nodes. For example, consider the short XML sample below:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<contacts>
   <contact>
      <name>Dan West</name>
      <business>
         <phone>
            <area_code>04</area_code>
            <number>0035-9110</number>
         </phone>
      </business>
      <home>
         <phone>
            <area_code>02</area_code>
            <number>6493-3250</number>
         </phone>
      </home>
   </contact>
</contacts>
```

Rendered as a tree, the nodes are linked as diagrammed below:



In a DOM environment, it is meaningful to talk about a node's relatives. For example, the <business> element has <contact> and <contacts> as ancestors, <name> and <home> as siblings, and <phone>, <area_code> and <number> as children.

Note *XML paths, element names, and attribute names are all case-sensitive. See 4D Technical Note 05-41,* Case-Sensitive Operations in 4th Dimension*, for code to handle case-sensitive comparisons.*

Using the Routines

Using the code listed in this technical note is straightforward, as illustrated in the code fragment below:

```
C_STRING(16;$contacts_xmlref)
$contacts_xmlref:=DOM Find XML element($root_xmlref;"/contacts/")  ` Find a node.
$hasAncestors_b:=DOM_NodeHasAncestors ($contacts_xmlref)
$hasChildren_b:=DOM_NodeHasChildren ($contacts_xmlref)
$hasSiblings_b:=DOM_NodeHasSiblings ($contacts_xmlref)
```

Calling code needs to do is find an XML node using any of the DOM commands. It does not matter how the node reference is found or generated, as the relative-checking routines are entirely generic. Additionally, it is safe to pass an invalid node referenced to the routines. Internally, the node relative testing routines use a custom error handler to suppress error displays. If a null or invalid XML reference is passed into the routines they return a result of **False**.

The Routines

The code of the relative testing routine and their error management routines are listed below.

DOM_NodeHasChildren

The *DOM_NodeHasChildren* routine tests if a node has one or more children. The result is **False** if the node reference is invalid.

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

$nodeHasChildren_b:=False` Default to False in case there are errors.

$noderef:=$1

DOM_StartCustomErrorHandling

C_STRING(16;$child_xmlref)
$child_xmlref:=DOM Get first child XML element($noderef)` Try to get a child.
` If the resulting xmlref is valid, there is at least one child.
$nodeHasChildren_b:=DOM_ReferenceIsValid ($child_xmlref)
```

```
DOM_StopCustomErrorHandling

$0:=$nodeHasChildren_b
```

DOM_NodeHasAncestors

The *DOM_NodeHasAncestors* routine tests if a node has one or more ancestors. The result is **False** if the node reference is invalid or points to the `#document` node. The `#document` element is a synthetic node above the root of the tree and should not be treated as a true ancestor.

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

$nodeHasAncestors_b:=False` Default to False in case there are errors.
$noderef:=$1

DOM_StartCustomErrorHandling

C_STRING(16;$parent_xmlref)
C_TEXT($name_text)
$parent_xmlref:=""
$name_text:=""
` Try to get a parent.
$parent_xmlref:=DOM Get parent XML element($noderef;$name_text)
$nodeHasAncestors_b:=DOM_ReferenceIsValid ($parent_xmlref)
` If the resulting xmlref is valid, there is at least one ancestor.
If ($nodeHasAncestors_b)` There appear to be ancestors.
  If ($name_text="#document")` #document is an artificial node above the tree.
    $nodeHasAncestors_b:=False
  End if
End if

DOM_StopCustomErrorHandling

$0:=$nodeHasAncestors_b
```

DOM_NodeHasSiblings

The *DOM_NodeHasSiblings* routine tests if a node has one or more siblings. The result is **False** if the node reference is invalid.

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

$noderef:=$1
$nodeHasSiblings_b:=False` Default to False in case there are errors.

DOM_StartCustomErrorHandling

C_STRING(16;$sibling_xmlref)
` Try to get an earlier sibling.
$sibling_xmlref:=DOM Get previous sibling XML element($noderef)
```

```
  ` If the resulting xmlref is valid, there is at least one sibling.
  $nodeHasSiblings_b:=DOM_ReferenceIsValid ($sibling_xmlref)
  If (Not($nodeHasSiblings_b))` There was no previous sibling, is there a next sibling?
    ` Try to get an earlier sibling.
    $sibling_xmlref:=DOM Get Next sibling XML element($noderef)
    ` If the resulting xmlref is valid, there is at least one sibling.
    $nodeHasSiblings_b:=DOM_ReferenceIsValid ($sibling_xmlref)
  End if

  DOM_StopCustomErrorHandling

  $0:=$nodeHasSiblings_b
```

DOM_ReferenceIsValid

The DOM_ReferenceIsValid routine is adapted from 4D Technical Note #06-40, *Enhancing the DOM XML Reading Functions*. The modified code is listed below:

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

  $noderef:=$1

  $nodrefIsValid_b:=True

  DOM_StartCustomErrorHandling

  C_TEXT($elementName_t)` Line below should throw an error if the element is not valid.
  $elementName_t:=""
  DOM GET XML ELEMENT NAME($noderef;$elementName_t)

  Case of
    : (DOM_Error#0)` There was an error of some kind.
      $nodrefIsValid_b:=False

    : ($elementName_t="")` Elements must have names to be operated on safely.
      $nodrefIsValid_b:=False

    Else
      $nodrefIsValid_b:=True
  End case

  DOM_StopCustomErrorHandling

  $0:=$nodrefIsValid_b
```

DOM_StartCustomErrorHandling

The *DOM_StartCustomErrorHandling* routine stores existing error state information and installs a custom error handler.

```
  If (Undefined(Error))
    Error:=0
  End if
```

```
C_STRING(80;DOM_PreviousErrorHandler_s)
C_LONGINT(DOM_PreviousValueOfError_l)
C_LONGINT(DOM_Error)
DOM_PreviousErrorHandler_s:=Method called on error
DOM_PreviousValueOfError_l:=Error
DOM_Error:=0` Assign a value in error method, if run.
Error:=0

ON ERR CALL("DOM_ErrorTrappingRoutine")
```

DOM_StopCustomErrorHandling

The *DOM_StopCustomErrorHandling* routine clears the custom error handler and restores the error state information saved by *DOM_StartCustomErrorHandling*.

```
ON ERR CALL(DOM_PreviousErrorHandler_s)` Restore original error handler.
Error:=DOM_PreviousValueOfError_l` Restore original value of Error.
```

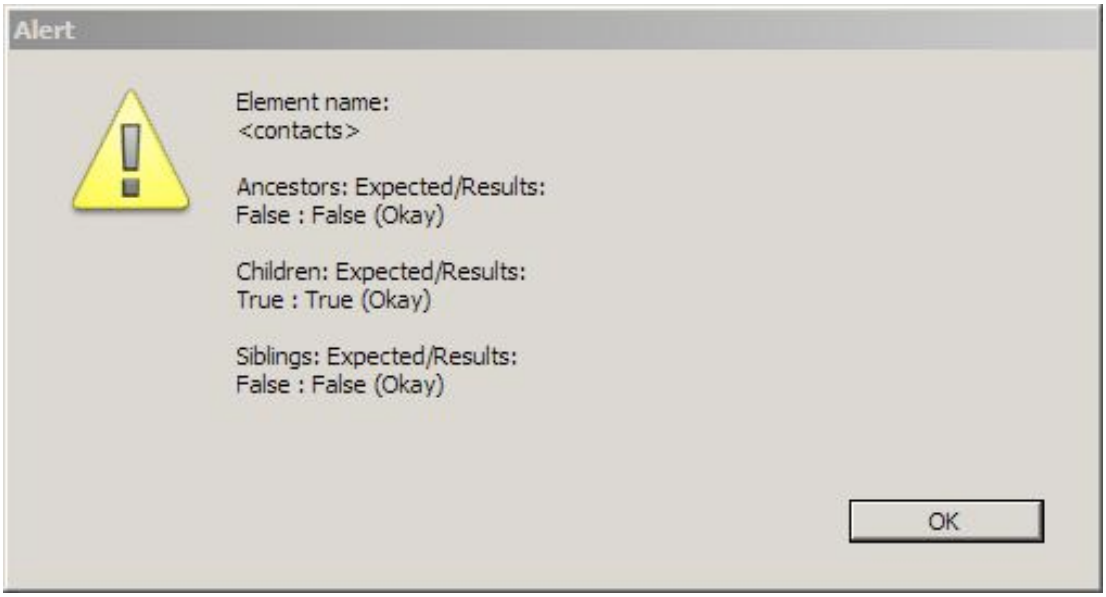
DOM_ErrorTrappingRoutine

The *DOM_ErrorTrappingRoutine* method is installed by the *DOM_ReferenceIsValid* and *DOM_StartCustomErrorHandling* routines to trap errors arising from attempting to read invalid node references. The error handler includes a single line of code, listed below:

```
DOM_Error:=Error
```

The Sample Database

The sample database includes the code listed above and a simple test routine named *Test_NodeHasRelatives*. The test code is designed to exercise the *DOM_NodeHasAncestors*, *DOM_NodeHasChildren*, and *DOM_NodeHasSiblings* methods in various conditions, including passing in good nodes, bad nodes, and the synthetic `#document` node. For each test, the code displays a simple status alert reporting if the test conditions were met, such as the screen shown below:



Summary

It is often helpful during XML processing to test if a node has ancestors, children, or siblings. While 4th Dimension's XML reading commands do not include functions to return information about a node's relations, it is easy to add using existing commands. This technical note documents such code and includes a sample database with full implementations.