Be Careful Writing XPath Expressions Against XML Documents that may have Non-Existent Elements

Roger L. Costello
July 29, 2014

Rule: The result of evaluating an XPath expression that compares a non-existent element against anything is always false.

Failure to remember this rule will give you countless headaches and hard-to-detect bugs.

Example: this Book element has a child Genre element:

```
<Book>
    <Title>How to Read a Book</Title>
    <Author>Mortimer J. Adler</Author>
    <Date>1940</Date>
    <ISBN>0-671-21209-5</ISBN>
    <Publisher>Simon and Schuster</Publisher>
    <Genre>non-fiction</Genre>
</Book>
```

The following XPath expression selects the Book if the value of the child Genre element is 'non-fiction':

```
Book[Genre eq 'non-fiction']
```

The result of evaluating the XPath against the XML is the Book.

Suppose that 'non-fiction' is the default genre. If a book is non-fiction, it is okay to omit the Genre element. The following Book element is non-fiction so it omits the Genre element:

```
<Book>
    <Title>Six Great Ideas</Title>
    <Author>Mortimer J. Adler</Author>
    <Date>1981</Date>
    <ISBN>0-02-072020-3</ISBN>
    <Publisher>Macmillan Publishing Company</Publisher>
</Book>
```

For that XML, the result is empty when the exact same XPath expression is evaluated. This is because of this expression:

```
Genre eq 'non-fiction'
```

There is no Genre element in the XML, so the expression is comparing a non-existent element against a string. Remember: *the result of comparing a non-existent element against anything is always false*.

Example of Failing to Heed the Rule in Schematron

Now let's consider some Schematron code that behaves incorrectly: it accepts invalid input. The cause of the erroneous behavior is not taking into account the XPath rule.

Below is the input XML document, containing a list of Books of various genres. The default genre is non-fiction, so it's okay to omit the Genre element if the Book is non-fiction.

The Schematron is coded to enforce this business rule:

> The Author of each Book within a genre must be unique.

Collect all the non-fiction books and there may not be multiple books by the same author. Ditto for the other genres.

The Schematron code establishes a Book as the context. It then creates two variables:

- $currentBook: holds the Book currently being assessed
- $BooksOfSameGenre: holds all the preceding and following Books that have the same genre as $currentBook

The Schematron assert statement asserts that $currentBook/Author does not match any Author element in $BooksOfSameGenre:

```
<sch:rule context="Book">

    <sch:let name="currentBook" value="."/>
    <sch:let name="BooksOfSameGenre"
             value="(preceding-sibling::Book[Genre eq $currentBook/Genre],
                     following-sibling::Book[Genre eq $currentBook/Genre])"/>

    <sch:assert test="not($currentBook/Author = $BooksOfSameGenre/Author)">
        The Author of each Book within a genre must be unique.
    </sch:assert>

</sch:rule>
```

That's pretty straightforward, right?

Actually, it has an insidious bug that may or may not manifest itself, depending on the input. If all the non-fiction Books explicitly specify a Genre element, then the Schematron code works correctly. If some of the non-fiction books omit the Genre element, then the Schematron code will allow invalid data to pass as valid. Ouch!

Why does the Schematron code have this behavior ? It's because of this XPath expression:

```
Genre eq $currentBook/Genre
```

Suppose the Schematron processor is currently positioned at a Book that doesn't have a Genre element. Then $currentBook/Genre is a non-existent element and … sounds like a record that's stuck, doesn't it? … *the result of comparing a non-existent element against anything is always false*.

In the following input data all Genre's are explicitly specified and the Schematron code detects the error (there are two non-fiction books with the same author):

```
<BookCatalogue>
   <Book>
      <Title>How to Read a Book</Title>
      <Author>Mortimer J. Adler</Author>
      <Date>1940</Date>
      <ISBN>0-671-21209-5</ISBN>
      <Publisher>Simon and Schuster</Publisher>
      <Genre>non-fiction</Genre>
   </Book>
   <Book>
      <Title>Introduction to Formal Languages</Title>
      <Author>Gyorgy Revesz</Author>
      <Date>1983</Date>
      <ISBN>0-486-66697-2</ISBN>
      <Publisher>McGraw-Hill</Publisher>
      <Genre>non-fiction</Genre>
   </Book>
   <Book>
      <Title>Illusions The Adventures of a Reluctant Messiah</Title>
      <Author>Richard Bach</Author>
      <Date>1977</Date>
      <ISBN>0-440-34319-4</ISBN>
      <Publisher>Dell Publishing Co.</Publisher>
      <Genre>fiction</Genre>
   </Book>
   <Book>
      <Title>Six Great Ideas</Title>
      <Author>Mortimer J. Adler</Author>
      <Date>1981</Date>
      <ISBN>0-02-072020-3</ISBN>
```

```
    <Publisher>Macmillan Publishing Company</Publisher>
    <Genre>non-fiction</Genre>
  </Book>
  <Book>
    <Title>The First and Last Freedom</Title>
    <Author>J. Krishnamurti</Author>
    <Date>1954</Date>
    <ISBN>0-06-064831-7</ISBN>
    <Publisher>Harper &amp;amp; Row</Publisher>
    <Genre>philosophy</Genre>
  </Book>
</BookCatalogue>
```

The following input data is exactly the same, except some books omit Genre, relying on the default value. The Schematron code fails to detect the error:

```
<BookCatalogue>
  <Book>
    <Title>How to Read a Book</Title>
    <Author>Mortimer J. Adler</Author>
    <Date>1940</Date>
    <ISBN>0-671-21209-5</ISBN>
    <Publisher>Simon and Schuster</Publisher>
    <Genre>non-fiction</Genre>
  </Book>
  <Book>
    <Title>Introduction to Formal Languages</Title>
    <Author>Gyorgy Revesz</Author>
    <Date>1983</Date>
    <ISBN>0-486-66697-2</ISBN>
    <Publisher>McGraw-Hill</Publisher>
    <Genre>non-fiction</Genre>
  </Book>
  <Book>
    <Title>Illusions The Adventures of a Reluctant Messiah</Title>
    <Author>Richard Bach</Author>
    <Date>1977</Date>
    <ISBN>0-440-34319-4</ISBN>
    <Publisher>Dell Publishing Co.</Publisher>
    <Genre>fiction</Genre>
  </Book>
  <Book>
    <Title>Six Great Ideas</Title>
    <Author>Mortimer J. Adler</Author>
    <Date>1981</Date>
    <ISBN>0-02-072020-3</ISBN>
```

```
      <Publisher>Macmillan Publishing Company</Publisher>
   </Book>
   <Book>
      <Title>The First and Last Freedom</Title>
      <Author>J. Krishnamurti</Author>
      <Date>1954</Date>
      <ISBN>0-06-064831-7</ISBN>
      <Publisher>Harper &amp;amp; Row</Publisher>
      <Genre>philosophy</Genre>
   </Book>
</BookCatalogue>
```

That's bad. That's a hard-to-detect bug since it manifests for some input data but not for others.

Lesson Learned: When your XPath expressions need to deal with XML that has potentially non-existent elements, write the XPath very carefully.

Testing XPath Expressions on Non-Existent Elements

```
<Test>
   <A>10</A>
</Test>
```

Compare an existing element (A) against a non-existing element (B):

Test	Result	Lesson learned
(A lt B) or false()	false	comparison against a non-existent element returns false
(A le B) or false()	false	comparison against a non-existent element returns false
(A gt B) or false()	false	comparison against a non-existent element returns false
(A ge B) or false()	false	comparison against a non-existent element returns false
(A ne B) or false()	false	comparison against a non-existent element returns false
(A < B) or false()	false	comparison against a non-existent element returns false
(A <= B) or false()	false	comparison against a non-existent element returns false
(A > B) or false()	false	comparison against a non-existent element returns false
(A >= B) or false()	false	comparison against a non-existent element returns false
(A != B) or false()	false	comparison against a non-existent element returns false

Compare a non-existing element (B) against an existing element (A):

Test	Result	Lesson learned
(B lt A) or false()	false	comparison against a non-existent element returns false
(B le A) or false()	false	comparison against a non-existent element returns false
(B gt A) or false()	false	comparison against a non-existent element returns false
(B ge A) or false()	false	comparison against a non-existent element returns false
(B ne A) or false()	false	comparison against a non-existent element returns false
(B< A) or false()	false	comparison against a non-existent element returns false
(B <= A) or false()	false	comparison against a non-existent element returns false
(B > A) or false()	false	comparison against a non-existent element returns false
(B >= A) or false()	false	comparison against a non-existent element returns false
(B != A) or false()	false	comparison against a non-existent element returns false

Compare a non-existing element (B) against a non-existing element (C):

Test	Result	Lesson learned
(B lt C) or false()	false	comparison against a non-existent element returns false
(B le C) or false()	false	comparison against a non-existent element returns false
(B gt C) or false()	false	comparison against a non-existent element returns false
(B ge C) or false()	false	comparison against a non-existent element returns false
(B ne C) or false()	false	comparison against a non-existent element returns false
(B< C) or false()	false	comparison against a non-existent element returns false

(B <= C) or false()	false	comparison against a non-existent element returns false
(B > C) or false()	false	comparison against a non-existent element returns false
(B >= C) or false()	false	comparison against a non-existent element returns false
(B != C) or false()	false	comparison against a non-existent element returns false

Conclusion

The Boolean comparison of a non-existent element against anything yields false.

Test a non-existing element (B) to see if it exists:

Test	Result	Lesson learned
(B) or false()	false	a non-existing element returns false

Test the negation of a non-existing element (B) to see if it exists:

Test	Result	Lesson learned
not(B) or false()	true	a non-existing element negated returns true

Conclusion

The negation of a non-existent element is true: the not of nothing is something.

Select the element Test if it has a non-existing child element (B):

Test	Result	Lesson learned
/Test[B]	query returned no results	do not use a predicate with a non-existent element

Select the element Test if it does not have a non-existing child element (B):

Test	Result	Lesson learned
/Test[not(B)]	returns Test	okay to use a predicate with the negation of a non-existent element

Select the element Test if it's not the case that it does not have a non-existing child element (B):

Test	Result	Lesson learned
/Test[not(not(B))]	query returned no results	do not use a predicate with a non-existent element

Conclusion

Predicates must only contain existing elements or the negation of non-existing elements.

Select the element Test if it has a non-existing child element (B):

Test	Result	Lesson learned
exists(B)	false	okay to test for the existence of a non-existent element
not(exists(B))	true	okay to test for the negation of the existence of a non-existent element