XQuery Testing from XML Schema Based Random Test Cases

Jesús M. Almendros-Jiménez(✉) and Antonio Becerra-Terón

Department of Informatics, University of Almería, 04120 Almería, Spain
{jalmen,abecerra}@ual.es

Abstract. In this paper we present the elements of an XQuery testing tool which makes possible to automatically test XQuery programs. The tool is able to systematically generate XML instances (i.e., test cases) from a given XML schema. The number and type of instances is defined by the human tester. These instances are used to execute the given XQuery program. In addition, the tool makes possible to provide an user defined property to be tested against the output of the XQuery program. The property can be specified with a Boolean XQuery function. The tool is implemented as an oracle able to report whether the XQuery program passes the test, that is, all the test cases satisfy the property, as well as the number of test cases used for testing. In the case of the XQuery program fails the testing, the tool shows counterexamples found in the test cases. The tool has been implemented as an XQuery library which makes possible to be used from any XQuery interpreter.

1 Introduction

Testing [21] is essential for ensuring software quality. The automation of testing enables the programmer to reduce time of testing and also makes possible to repeat testing after each modification to a program. A testing tool should determine whether a test is passed or failed. When failed, the testing tool should provide evidences of failures, that is, counterexamples of the properties to be checked. Additionally, a testing tool should generate test cases automatically [1]. Fully random generation could not be suitable for an effective and efficient tool. Distribution of test data should be controlled, by providing user-defined test cases, that is, data distribution should be put under the human tester's control. For testing XML based applications some benchmarks datasets are available (for instance, *XMark* [20], *Michigan Benchmark* [19] and *XBench* [22]). However, they are not always suitable for testing applications. There are some cases of automatic data generators for XML: *ToXgene* [2] using XML Schemas with annotations of data distribution functions, *VeXGene* [16] using DTDs, and *XBeGene* [14] based on examples. XML test case generation can find an application field in *Web Services* [3,12] and *Access Control Policies* [7].

This work was supported by the EU (FEDER) and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grant TIN2013-44742-C4-4-R, as well as by the Andalusian Regional Government under Project P10-TIC-6114.

© Springer International Publishing Switzerland 2015
Q. Chen et al. (Eds.): DEXA 2015, Part II, LNCS 9262, pp. 268–282, 2015.
DOI: 10.1007/978-3-319-22852-5_23

XQuery has evolved into a widely accepted query language for XML processing and many XQuery engines have been developed. Even though some tools provide mechanisms for debugging (for instance, XMLSpy[1], Oxigen[2] and Stylus Studio[3], among others), users should be equipped with a large number of mechanisms for detecting failures in their applications. Among them testing would facilitate the detection of bugs due to mistakes when using XQuery expressions. Most of programming errors come from wrong XPath expressions (i.e., requesting paths/nodes of XML trees that do not exist), unsatisfiable Boolean conditions and incompatible XPath expressions. Validation of XML documents against XML Schemas mitigates some of these drawbacks. When an XQuery expression against a given document returns an empty value/wrong answer, stepwise/breakpoint/trace based debuggers can help to detect failures but it is only useful for ensuring the correct execution for a single XML input instance. In order to have a stronger confidence, XQuery programs should be tested against a test suite covering a large range of test cases.

In this paper we present the elements of an XQuery testing tool which makes possible to automatically test XQuery programs. The tool is able to systematically generate XML instances (i.e., test cases) from a given XML schema. The number and type of instances is defined by the human tester. These instances are used to execute the given XQuery program. In addition, the tool makes possible to provide an user defined property to be tested against the output of the XQuery program. The property can be specified with a Boolean XQuery function. Our proposal can be seen as a black-block approach to XQuery testing. The tool takes as input an XML Schema, an XQuery program and a property to be checked against the output of the XQuery program. The tool automatically generates a test suite from the XML Schema, it executes the XQuery program and for each output of the program it checks the given property.

The tool is implemented as an oracle able to report whether the XQuery expression passes the test, that is, all the test cases satisfy the property, as well as the number of test cases used for testing. In case of success (i.e., all the outputs of the XQuery program satisfy the property), the tool reports "Ok". Otherwise the tool reports counterexamples (i.e., inputs of the XQuery program for which the output does not satisfy the given property). The tool is customizable in the following sense. The human tester can define the structure and content of the test cases from the XML schema. Additionally, the human tester can control the number of test cases. He or she can tune the number and size of XML trees generated from the XML schema. Thus, although the tool generates random test cases, the tester can control the size of the test suite. The property to be checked is defined by a Boolean XQuery function. Usually, the property expresses a constraint in terms of XPath expressions and logical connectors: for all, exists, and, or, etc., possibly making use of XQuery functions. It makes possible to express a rich repertory of properties against output documents: the occurrence

[1] http://www.altova.com/xmlspy/xquery-debugger.html.
[2] http://www.oxygenxml.com/xml_editor/xquery_debugger.html.
[3] http://www.stylusstudio.com/xquery_debugger.html.

of a certain value, the range of a certain attribute, the number of nodes, etc. The code of the XQuery program is not used to generate the test suite. The adequacy of the test suite to a given XQuery program is determined by the human tester, that is, the human tester has to select an XML Schema suitable to generate a test suite covering as many cases as possible. In essence he or she gives with the schema the required paths as well as the values of tags and attributes of the input documents. The tool randomly generates test cases as combinations of paths and values. In summary, the tool enables a partial validation of the XQuery program: for a subset of the input documents and a given property. Since the testing is automatic and customizable, the human tester can easily change values and paths as well as he or she can increase the number of test cases and play with properties to have an stronger confidence about the soundness of the program.

Our approach is inspired by similar tools in functional languages. This is the case of the *Quickcheck* tool [9] for Haskell, and the *PropEr* tool [18] for Erlang. Properties in these approaches are specified by functions, and are automatically tested from random test cases. Since XQuery is also a functional language, there seems natural to provide a similar tool for XQuery programs. Nevertheless, the context is different. XQuery is a functional language handling XML documents, which are in essence ordered labeled trees, and XQuery has a main element XPath expressions. Thus, here random test case generation is focused on trees and thus specific algorithms can be defined to automatically generate trees of a certain size. Properties in Quickcheck and PropEr are also defined by functions. Thanks to the Higher Order capabilities of XQuery, the property can be passed as argument to the tester, as well as the XQuery program, enabling the implementation of the tool in a similar way to Haskell and Erlang, that is, the tool is an XQuery program implemented as an XQuery library and thus can be used from any interpreter.

Our work is also inspired by some previous works about testing of XML applications. In [4] the *XPT (XML-based Partition Testing)* approach is presented which makes possible the automatic generation of XML instances from a given XML Schema. The *TAXI* tool[4] has been developed in this framework [5]. XPT is an adaptation of the well-known *Category Partition Method* [17], used to generate instances with all the possible combinations of elements. In this approach, a test selection strategy is studied including user defined weight assignments for *choice* statements in XML Schemas and derivation of XML Schemas from an initial XML Schema according to choice statements. XPT is able to generate XML instances according to a fixed number of instances, a fixed functional coverage (in percentage terms) and also a mixed criterium (fixed number of instances and functional coverage). TAXI populates instances by specifying a source (i.e., a URL), by manual insertion of values, or by taking values from the schema (in the *enumeration* section). In [6] they use TAXI to test *XSLT* stylesheets. The tool is able to report the result of the testing, using XML Schemas as model to which output XML instances must conform. In our approach, we take the XML Schema as input for test case generation similarly to TAXI. But there

[4] http://labsewiki.isti.cnr.it/labsedc/tools/taxi/public/main.

are some differences. In our case, the XML Schema rather than representing all the possible inputs of the program, is specifically used to generate test cases. Thus, even when an XML Schema for the program exists, the human tester has to select from the XML Schema those relevant elements for the program: paths and values to be tested. It does not mean that the original schema does not serve for test case generation, but usually the number of instances can be greater than necessary, affecting the performance of the tool. In particular, the human tester has to select relevant values and to incorporate them to the XML Schema in the *enumeration* section, and *choice* statements have to explicitly be selected by the human tester. It does not mean that the human tester can test more than one combination of them. Additionally, our work can be seen as an extension of the TAXI approach enabling property-based testing of XQuery programs.

In [10] they propose the automatic generation of XML documents from a DTD and an example. The framework called *GxBE (Generate-XML-By-Example)* uses a declarative syntax based on XPath to describe properties of the output documents. They claim that datasets are useful for testing when they conform a certain schema (an DTD schema extended with cardinality constraints), when they have some specific characteristics (i.e., datasets returning empty answers are not very useful) and the data values match an expected data distribution. They are able to express global properties on the document, in particular, to express the so-called count constraints making use of XPath and the count function. With regard to GxBE, our approach randomly generates test cases from the XML Schema in which values have been incorporated, while properties are not considered to test case generation. Rather than properties are used for testing after test case generation and query execution. In GxBE they propose to use properties in test case generation as pre-conditions, in order to generate a more suitable test suite. We believe that this is an interesting idea and we will incorporate it to our tool in the future. In GxBE they are concerned about test case generation efficiency. In our approach, test case generation efficiency cannot be ensured but it is under control. Firstly, the human tester can customize the number of test cases by modifying the input XML Schema, as well as by selecting the size of the trees. Additionally, test case generation is dynamic. It means that when testing a given program, test cases are incrementally generated and the tool stops when a failure (i.e., the property is not satisfied) has been found. It drastically reduces the time required for testing.

In [15] they describe how to test output documents of XML queries. They permit the specification of properties on XML documents via an XML template in which expected nodes, unexpected nodes, expected ordering and expected cardinality can be specified. In this work rather than test case generation, they propose the specifications of some properties on the output (i.e., post-conditions) to be checked against the output document. We adopt a similar approach but enabling the specification of a richer repertoire of properties. In fact, we do not restrict the type of property, making possible to use any XQuery expression to test the program. Nevertheless, we implicitly assume that the property to be checked is considerably simple. Testing should be focused on simple properties,

otherwise the testing process would be as complex as the programming process. Moreover, complex properties affect the performance of the tool.

There are some works about white-box testing in this context. A partition-based approach for XPath testing has been proposed in [11]. They propose the construction of constraints from categorization and choice selection in order to generate test cases for XPath expressions. In [8] they study test case generation based on the category partition method and make use of *SMT* and *Z3* solvers for input data generation. Test cases are specified in XML. Our approach is still a black-box technique, but we will consider as future work to include white-box test case generation. In particular, we believe that the XML schema input of the test case generation can be filtered/generated from the program.

1.1 Example

Now, we would like to show an example to illustrate the approach. Let us suppose the following XQuery program:

```
for $book in $file/book return
    if ($book/title="UML" and $book/price<100) then
        <book_UML>
          {$book/@year}{$book/author}{$book/price}
        </book_UML>
    else if ($book/title="XML" and $book/@year>2000 and $book/price<100) then
        <book_XML>
          {$book/@year}{$book/author}{$book/price}
        </book_XML>
    else ()
```

And let us suppose that we would like to know whether the output of the program satisfies the following properties: *“Prices of books are smaller than 100”* and *“Book are after 2000”*. Our tool works as follows. Firstly, we have to define an XML Schema for generating test cases. For instance, the schema shown in Fig. 1. There, we intentionally select values (defined in the enumeration statement) for titles, authors, prices and years, oriented to the selected properties (i.e., years after and before 2000, and prices greater and smaller than 100). Titles are selected according to the intended behavior of the program (i.e., selection of UML and XML books), and one author is provided for completing book information. Next, we convert the query into a function as follows:

```
declare function tc:books_query($file)
{for $book in $file/book return
    if ($book/title="UML" and $book/price<100) then
        <book_UML>
        {$book/@year}{$book/author}{$book/price}
        </book_UML>
    else if ($book/title="XML" and $book/@year>2000 and $book/price<100) then
      <book_XML>
      {$book/@year}{$book/author}{$book/price}
      </book_XML>
    else ()
};
```

After, we can define the following Boolean XQuery functions that act on the output document.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:simpleType name="authorType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Buneman"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="yearType">
  <xs:restriction base="xs:integer">
    <xs:enumeration value="1995"/>
    <xs:enumeration value="2005"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="priceType">
  <xs:restriction base="xs:integer">
    <xs:enumeration value="80"/>
    <xs:enumeration value="150"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="titleType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="UML"/>
    <xs:enumeration value="XML"/>
  </xs:restriction>
</xs:simpleType>
<xs:element name="bib">
<xs:complexType>
<xs:sequence>
<xs:element name="book" minOccurs="1" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="author" type="authorType" minOccurs="1"
                maxOccurs="unbounded"/>
      <xs:element name="title" type="titleType"/>
      <xs:element name="price" type="priceType"/>
    </xs:sequence>
     <xs:attribute name="year" type="yearType" use="required"/>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Fig. 1. XML Schema example

```
declare function tc:books_price_100($book){
  $book/price<100
};

declare function tc:books_year_2000($book){
  $book/@year>2000
};
```

Now, we can call the tool with the first property as follows, reporting the answer below:

```
tc:tester("schema.xsd","tc:books_query", "tc:books_price_100",1)
Ok: passed 80 tests.
Trivial: 35 tests.
```

and also, with the second property, reporting the answer below:

```
tc:tester("schema.xsd","tc:books_query", "tc:books_year_2000",1)
Falsifiable after 8 tests.
Counterexamples:
<bib>
  <book year="1995">
    <author>Buneman</author>
    <title>UML</title>
```

```
    <price>80</price>
  </book>
</bib>
....
```

The first reported answer means that all the books of the output have price smaller than 100 for the given 80 test cases. From them, 35 produce an empty answer, and thus the property cannot be checked. Empty answers come from test cases in which prices are greater than 100. The second reported answer means that the tool finds at least one counterexample (shown bellow) from eight test cases. Thus, the second property is not satisfied. We can inspect now the program revealing that UML books are not filtered by year. In the first case, we are not sure whether the property is satisfied, but 80 test cases passed it. We have incorporated a parameter to the tool (the "1" occurring in the example) to limit the number and size of generated test cases. Increasing this number, we can have a stronger confidence about this property. More details will be given in Sect. 2. Let us remark that for the examples shown the tool spends 101 and 20 ms, respectively. We have evaluated our tool with several examples (see Sect. 4).

The rest of the paper is structured as follows. Section 2 will describe the algorithm for test case generation. Section 3 will present how XQuery programs are tested by our tool. Section 4 will show examples of evaluation. Finally Sect. 5 will conclude and present future work.

2 Test Case Generation

In this section we will describe how test cases are randomly generated from an XML Schema. As was commented in the introduction, the XML schema used for test case generation is not necessarily the XML Schema given to the input program. Rather than, the human tester has to define, in most cases, a new XML Schema based on the original one, in which he or she selects the elements that are sufficient to test the program. The original schema leads to a too wide range of test cases that could not be required to test the program, affecting the performance of testing. Although our approach is a black box testing mechanism, the human tester has to select from the space of input data, those relevant to the program. Basically, the selected number of tags/paths of the document as well as values should be relevant.

The XML Schema offers a wide range of mechanisms to express input data structure. However, from testing process point of view, only a small subset of mechanisms is actually required. For instance, *choice* and *all* statements for declaring content of complex types are used in the XML Schema to produce variants of XML trees in which some tags can occur or not in the trees. We only consider the case of *sequence*, in which the human tester can play with *minOccurs* and *maxOccurs* elements, to force the occurrences of tags. Allowing zero in *minOccurs*, *choice* is a particular case of *sequence* in the test case generation. The current implementation does not consider the case of *all* (which permits any ordering of tags) given that it produces a huge number of cases. There are other mechanisms explicitly avoided like *any*, *anyAttribute* and *list*. Additionally,

attributeGroup, *Group*, *redefine*, *extension*, *union*, *import* and *include* enabling reuse of definitions are not considered. Keys are also not considered. *ref* can be used to define recursive definitions in which a recursive *complexType* has to be defined. Finally, *enumeration* is used for defining the values of a certain tag. When a certain tag does not have the corresponding enumeration statement, the tool assigns a default value. In summary, an XML Schema for test case generation includes: *attribute* (*use*), *element* (*minOccurs/maxOccurs*), *sequence* (*minOccurs/maxOccurs*), *complexType*, *ref* and *enumeration*. Neither *use* nor *minOccurs/maxOccurs* are required assuming default values: optional and 1, respectively.

Test case generation can be controlled by the human tester. XML trees are generated in a certain order of increasing size. Basically, the random test case generator starts from the smallest XML tree conforming the given schema, and in each step it increases the size of previously generated trees by adding a new branch up to *maxOccurs* value. Recursive definitions are also unfolded in each step. In case *maxOccurs* is unbounded or with recursive definitions, the random test case generator is able to produce an infinite number of trees, but it never happens because the number of steps is a parameter of the test case generator. When the testing process is carried out, the test cases are dynamically generated up to the required number of steps, but whenever the program fails to satisfy the given Boolean property the tester stops and no more cases are generated. Thus, there will happen that only the required cases to fail the program are computed. In the case of success, they will be fully computed. From a practical point of view, the human tester should request a small number of steps in the beginning and increase in case of success. In Sect. 4 we will show experiments made for different XML Schemas. Figure 2 shows an example of test case generation for an XML Schema. Starting from the XML Schema shown in the left corner (schematically represented), in each step, the test case generator produces new schemas, and schemas are populated with values. In Fig. 2, step 1 generates 1, step 2 generates 1.1, step 3 generates 1.1.1, 1.1.2, and 1.1.3, and step 4 generates 1.1.1.2,

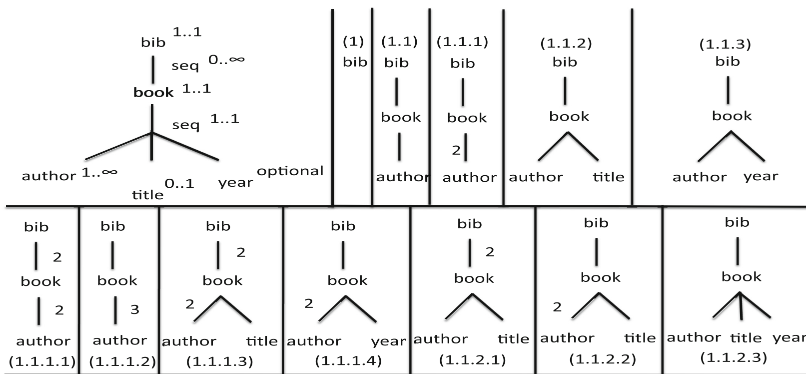

Fig. 2. Test Case Generation Example

1.1.1.3, 1.1.1.3, 1.1.2.1,1.1.2.2,1.1.2.3, and so on. In each step the schemas previously generated are used to compute new ones. Basically, the idea is to increase minOccurs values, to add optional attributes and to unfold recursive definitions in each step. The order in which the trees increase is top-down.

3 Property-Based Testing

Now, we would like to explain how testing process is carried out. Firstly, we will define the meaning of a certain program passing a test, and secondly we will describe the implementation of the tool. We only consider the case of unary Boolean properties, and programs with single input. The following definitions can be generalized to the case of n-ary properties and multiple inputs.

Assuming a finite set of test cases $t_1, \ldots, t_n$ of an XML Schema Σ, a program $\mathcal{P}$ and a Boolean property p, we say that $\mathcal{P}$ passes the test p in $t_1, \ldots, t_n$ whenever for each t_i, $i = 1, \ldots, n$ such that $\mathcal{P}(t_i) \neq \emptyset$ then $p(\mathcal{P}(t_i))$. We say that $\mathcal{P}$ fails the test p whenever there exists t_i, $i = 1, \ldots, n$ such that $\mathcal{P}(t_i) \neq \emptyset$ and $\neg p(\mathcal{P}(t_i))$. Finally, we say that p cannot be checked for $\mathcal{P}$ in $t_1, \ldots, t_n$, whenever for all t_i, $i = 1, \ldots, n$, $\mathcal{P}(t_i) = \emptyset$.

Given a program $\mathcal{P}$ and an XML Schema Σ, we say that $\mathcal{P}$ conforms to Σ whenever there exists t which conforms to Σ such that $\mathcal{P}(t) \neq \emptyset$. In other words, $\mathcal{P}$ conforms to Σ whenever at least an answer of $\mathcal{P}$ is not empty, and thus it means that Σ is correct for $\mathcal{P}$.

Obviously, if p cannot be checked for $\mathcal{P}$ then the test cases $t_1, \ldots, t_n$ are not sufficient and thus we cannot say anything about the program $\mathcal{P}$. It can happen in the following cases: (a) $\mathcal{P}$ does not conform to Σ and (b) $t_1, \ldots, t_n$ are not enough. To solve (a) the human tester has to modify Σ. In case (b), a more complete set of test cases has to be used. The implemented tester is able to detect (for a finite set of test cases) when a program $\mathcal{P}$ passes or fails a test t_i as well as when p cannot be checked for $\mathcal{P}$.

Now, we present the main elements of the implementation. The testing tool has been implemented as an XQuery function that takes as arguments the schema, the program, the property and the number of steps of test case generation. The program and property have to be represented as functions. It does not mean that only functions can be tested. In case of large XQuery programs the main query has to be represented as a function taking as argument an input XML document. In the current implementation only unary functions are allowed. We will consider as future work the extension of the tester to n-ary functions. The code of the tester is as follows:

```
declare function tc:tester($schema as node()*,$query as xs:string, $property
    as xs:string, $i as xs:integer){
  tc:tester_loop($schema,$query,$property,0,$i,0,0)};
```

Basically, the tester is a loop of i steps (according to the provided argument), in which in each step a set of schemas is generated. The schema instances are populated with values, and they are taken as input of the given program, and the given property is checked for each output. Whenever all the instances pass the

test the loop continues. Otherwise, no more schemas and instances are generated and the tester reports "Falsifiable", showing the number of tested cases and the counterexamples. In case of loop continues, it ends when the number of steps is reached. In such case, the tester reports "Ok" whenever all the instances pass the test, showing the number of test cases and empty answers. When the number of empty answers is equal to the number of test cases, the tool reports "Unable to check the property". The tester loop is as follows:

```
declare function tc:tester_loop($schema as node()*,$query as xs:string,
     $property as xs:string,$k as xs:integer, $i as xs:integer,$tests as xs:
     integer,$empties as xs:integer){
if ($k>$i) then if ($tests=$empties) then tc:show_unable()
                         else tc:show_passed($tests,$empties)
else tc:tester_schema($schema,$schema,$query,$property,$k,$i,$tests,$empties)
};
```

and each step is implemented as follows:

```
declare function tc:tester_schema($schemas as node()*,$all as node()*,
      $query as xs:string,$property as xs:string,$k as xs:integer,
      $i as xs:integer,$tests as xs:integer,$empties as xs:integer){
if (empty($schemas))
          then let $new := tc:new_schemas($all)
              return tc:tester_loop($new,$query,$property,$k + 1,$i,
                               $tests,$empties)
          else
          let $sc := head($schemas)
          let $structure := tc:skeleton($sc/xs:schema/xs:element)
          let $examples := tc:populate($structure,tc:getTypes($structure),
                               tc:getVal($sc/xs:schema,
                               tc:getTypesName($structure)))
          let $total := count($examples) return
          if (not($total=0))
            then
              let $fquery := function-lookup(xs:QName($query),1)
              let $fproperty:=function-lookup(xs:QName($property),1)
              let $no := (for $example in $examples
                  let $result := $fquery($example)
                  where not($fproperty($result)) return
                  if (empty($result)) then <empty/> else $example)
                  let $noempty := $no[not(name(.)="empty")]
                  let $falsifiable := count($noempty)
                  let $newempties :=  count($no[name(.)="empty"])+$empties
                  let $newtests := $tests + count($examples)
                  return
                  if ($falsifiable=0)
                  then tc:tester_schema(tail($schemas),$all,$query,
                            $property,$k,$i,$newtests,$newempties)
                  else tc:show_falsifiable($newtests,$noempty)
                  else if ($tests=$empties) then tc:show_unable()
                            else tc:show_passed($tests,$empties)
};
```

where `tc:new_schemas` is responsible to the generation of schemas, `sc:skeleton` produces the skeleton of the instance and `tc:populate` fills the skeleton with elements.

3.1 Examples

Now, we would like to show a batch of examples, in order to prove the benefits of the tester.

Example 1. Assuming the schema of Fig. 1, we can test the following program:

```
declare function tc:yearofUMLbooks($file){
for $x in $file/book
where $x/title="UML" and $x/@year>2000
return $x/@year};
```

with regard to the following property:

```
declare function tc:after2000($year){
  $year>2000};
```

obtaining the following answer:

```
Ok: passed 80 tests.
Trivial: 48 tests.
```

48 trivial cases (i.e., empty answers) are checked, corresponding to the case of non "UML" and before 2000 books, which are generated according to the given schema. In the case of the property:

```
declare function tc:before2000($year){
  $year<2000};
```

the following answer is reported:

```
Falsifiable after 8 tests.
Counterexamples:
<bib>
  <book year="2005">
    <author>Buneman</author>
    <title>UML</title>
    <price>80</price>
  </book>
</bib>
<bib>
...
```

The counterexamples shown are non-empty solutions which do not satisfy the given property. The human tester has to include enough values for obtaining a suitable answer. In this example, the schema should include at least a value after 2000 (i.e., 2005), and "UML" as value for title. In case the human tester omits the following line in the XML Schema:

```
<xs:attribute name="year" type="yearType" use="required"/>
```

Now, the tester reports *"Unable to test the property"*. The same happens when the title is omitted. In both cases, it is due to empty answers. In the opposite case, author and price are not required to test this program. The human tester can remove from the schema unnecessary tags and attributes for improving performance.

Example 2. In the case of the following program:

```
declare function tc:books_query($file){
for $book in $file/book return
if ($book/title="UML" and $book/price<100) then
    <book_UML>
    {$book/@year}{$book/author}{$book/price}
    </book_UML>
else if ($book/title="XML" and $book/@year>2000 and $book/price<100) then
    <book_XML>
    {$book/@year}{$book/author}{$book/price}
    </book_XML> else ()
};
```

we can ask about:

```
declare function tc:books_price_100($book){
  $book/price<100};
```

obtaining the following answer with respect to the schema of Fig. 1:

```
Ok: passed 80 tests.
Trivial: 35 tests.
```

35 trivial cases corresponding to tests in which prices are greater than 100, as well as XML books published before 2000. But we can express more complex properties like:

```
declare function tc:allbooksofBuneman($book){
  every $b in $book satisfies $b/author="Buneman"};
```

In this case the tester also answers “Ok” because “Buneman” is the only value for author. Also we can check a more complex property like the following, obtaining the same answer:

```
declare function tc:price_and_year($book){
  every $b in $book satisfies
  if (name($b)="book_UML") then $b/price<100
  else $b/@year>2000 and $b/price<100};
```

Example 3. The adequacy of the selected XML Schema does not only depend on the selected values for each tag and attribute, but also on the required number of branches. Here the minOccurs and maxOccurs values play a key role. For instance, let us suppose we test the following program:

```
declare function tc:third_book($bib){
  let $third := $bib/book[3]
  where $third/title="UML"
  return <third>{$third/title}{$third/author[3]}</third>};
```

with regard to the following property:

```
declare function tc:UML($book){
  $book/title="UML"};
```

In this case, if we call the tester with:

```
tc:tester(.,"tc:second_book", "tc:UML",1)
```

then we obtain *“Unable to test the property”*. It happens because the value one has been selected as number of steps for test case generation. It means that according to the XML Schema only bibliographies with at most one book have been created. The same happens with two steps. Only for three steps the following answer is reported:

```
Ok: passed 656 tests.
Trivial: 400 tests.
```

In this example the human tester should modify the XML Schema and declare minOccurs as three for the sequence of books. It forces the generation of useful test cases in the first step. It also happens for optional attributes. The same happens for branches. For instance, let us suppose that in the schema of Figure 1, minOccurs is set to zero in all the cases. The same previous query and property cannot be checked after eight steps. In the case of long paths and recursive definitions, the number of steps has to be larger, otherwise useful test cases are not generated in early stages.

Table 1. Benchmarks of testing

Query	*Time*	*Passed*	*Tests Passed*	*Falsifiable*	*Counterexamples*	*Property*
Q1	419 ms	✓	584	✗	✗	`every $book in $bib/bib satisfies $book/@year>1991`
Q1	121 ms	✗	✗	✓	8	`some $book in $bib/bib satisfies $book/@year<1991`
Q2	1.401 ms	✓	306	✗	✗	`every $result in $results/result satisfies $result/title and $result/author`
Q2	31 ms	✗	✗	✓	2	`some $result in $results/result satisfies not($result/title) or not($result/author)`
Q3	1.428 ms	✓	306	✗	✗	`every $result in $results/result satisfies $result/title and $result/author`
Q3	24 ms	✗	✗	✓	2	`some $result in $results/result satisfies not($result/title) or not($result/author)`
Q6	1.424 ms	✓	306	✗	✗	`every $book in $bib/book satisfies count($book/author)<=2`
Q6	23 ms	✗	✗	✓	2	`some $book in $bib/book satisfies count($book/author)>2`
Q7	1.348 ms	✓	1.884	✗	✗	`let $count := count($bib/book) return every $i in 1 to $count - 1 satisfies $bib/book[$i]/title<=$bib/book[$i+1]/title) satisfies $b=true()`
Q7	27 ms	✗	✗	✓	12	`let $count := count($bib/book) return some $i in 1 to $count - 1 satisfies $bib/book[$i]/title>$bib/book[$i+1]/title`
Q8	438 ms	✓	696	✗	✗	`every $book in $books satisfies every $item in $book/* satisfies contains(string($item),"Suciu") or name($item)="title"`
Q8	26 ms	✗	✗	✓	24	`some $book in $books satisfies some $item in $book/* satisfies not(contains(string($item),"Suciu")) and not(name($item)="title")`
Q9	1.500 ms	✓	1.518	✗	✗	`every $result in $results/results satisfies contains($result/text(),"XML")`
Q9	17 ms	✗	✗	✓	2	`some $result in $results/results satisfies not(contains($result/text(),"XML"))`
Q10	1.989 ms	✓	1.365	✗	✗	`count($results/minprice) = count(distinct-values($results/minprice/@title))`
Q10	25 ms	✗	✗	✓	4	`not(count($results/minprice) = count(distinct-values($results/minprice/@title)))`

4 Evaluation

We have evaluated our tool with the XQuery use cases available in the W3C page[5]. We have checked properties on the queries of Sect. 1.1.9.1 of the cited repository (only those queries with a single input parameter). We have analyzed the time required for each query and the number of tests. The properties, response times (in milliseconds) as well as number of tests are shown in Table 1. Benchmarks have been made on a 2.66 GHz Inter Core 2 Duo MAC OS machine, with 4 GB of memory. We have used the BaseX XQuery interpreter [13]. Properties have been selected to be representative to the computation of each query. We have tested each property and its negation in order to measure time required to pass and fail the property. From the table, we can conclude that for a reasonable number of tests (from three hundred to thousand test cases in some examples), the tester is able to answer in a short time. The selected schema is crucial to have a short time. When the number of values for each type, as well the number of selected paths is high the performance is worst. The tester implementation as well as the examples shown in the paper can be downloaded from http://indalog.ual.es/TEXTUAL.

[5] http://www.w3.org/TR/xquery-use-cases/.

5 Conclusions and Future Work

In this paper we have presented a tool for testing XQuery programs. We have shown how the proposed tool is able to automatically generate a test suite from the XML Schema of the input documents of the program in order to check a given property on the output of the program. It makes possible to have an stronger confidence about the correct behavior of the program. As future work, we would like to extend our work as follows. Firstly, there is a natural extension to cover programs with more than one input document. The current implementation of the tool works with any kind of XQuery expression but having a single input document. Secondly, we would like to extend the tool by adding more information about test cases. In the current implementation, only the number of empty solutions are shown, but other than empty solutions are relevant. For instance, non-empty solutions returning empty values for parts of the query (i.e., paths, subexpressions, etc.) can be useful for the human tester. Thirdly, we will consider how to filter test case generation by considering properties on the input document. The XML Schema imposes restrictions in terms of cardinality, but input programs can require to satisfy more complex properties. Input-output properties (i.e., properties relating input and output) will be also subject of study in the future. Fourthly, we would like to extend our work to the case of white box testing. In particular, XML Schemas can be automatically filtered/generated from the program in order to generate useful test cases. Finally, a richer repertoire of XML Schema statements will be included in the implementation.

References

1. Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P., et al.: An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. **86**(8), 1978–2001 (2013)
2. Barbosa, D., Mendelzon, A., Keenleyside, J., Lyons, K.: ToXgene: a template-based data generator for XML. In: Proceedings of the 2002 ACM SIGMOD, pp. 616–616. ACM (2002)
3. Bartolini, C., Bertolino, A., Marchetti, E., Polini, A.: WS-TAXI: a WSDL-based testing tool for web services. In: International Conference on Software Testing Verification and Validation, 2009, pp. 326–335. IEEE (2009)
4. Bertolino, A., Gao, J., Marchetti, E., Polini, A.: Automatic test data generation for XML schema-based partition testing. In: Proceedings of the Second International Workshop on Automation of Software Test (AST), p. 4. IEEE Computer Society (2007)
5. Bertolino, A., Gao, J., Marchetti, E., Polini, A.: TAXI-a tool for XML-based testing. In: Companion to the Proceedings of the 29th International Conference on Software Engineering, pp. 53–54. IEEE Computer Society (2007)
6. Bertolino, A., Gao, J., Marchetti, E., Polini, A.: XModel-based testing of XSLT applications. WEBIST **2**, 282–288 (2007)
7. Bertolino, A., Lonetti, F., Marchetti, E.: Systematic XACML request generation for testing purposes. In: 36th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 3–11. IEEE (2010)

8. Chimisliu, V., Wotawa, F.: Category partition method and satisfiability modulo theories for test case generation. In: 2012 7th International Workshop on Automation of Software Test (AST), pp. 64–70. IEEE (2012)
9. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM SIGPLAN Not. **46**(4), 53–64 (2011)
10. Cohen, S.: Generating XML structure using examples and constraints. Proc. VLDB Endow. **1**(1), 490–501 (2008)
11. De La Riva, C., Garcia-Fanjul, J., Tuya, J.: A partition-based approach for XPath testing. In: International Conference on Software Engineering Advances, p. 17. IEEE (2006)
12. Fisher, M., Elbaum, S., Rothermel, G.: An automated analysis methodology to detect inconsistencies in web services with WSDL interfaces. Softw. Test. Verification Reliab. **23**(1), 27–51 (2013)
13. Grün, C.: BaseX. The XML Database (2015). http://basex.org
14. Harazaki, M., Tekli, J., Yokoyama, S., Fukuta, N., Chbeir, R., Ishikawa, H.: XBeGene: scalable XML documents generator by example based on real data. In: Gaol, F.L. (ed.) Recent Progress in DEIT, Vol. 1. LNEE, vol. 156, pp. 449–460. Springer, Heidelberg (2013)
15. Kim-Park, D.S., de la Riva, C., Tuya, J.: An automated test oracle for XML processing programs. In: Proceedings of the First International Workshop on Software Test Output Validation, pp. 5–12. ACM (2010)
16. Jeong, H.J., Lee, S.H.: A versatile XML data generator. Int. J. Softw. Effectiveness Effi. **1**, 21–24 (2006)
17. Ostrand, T.J., Balcer, M.J.: The category-partition method for specifying and generating functional tests. Commun. ACM **31**(6), 676–686 (1988)
18. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Proceedings of the 2011 ACM SIGPLAN Erlang Workshop, pp. 39–50. ACM Press, New York, September 2011
19. Runapongsa, K., Patel, J.M., Jagadish, H.V., Chen, Y., Al-Khalifa, S.: The Michigan benchmark: towards XML query performance diagnostics. Inf. Syst. **31**(2), 73–97 (2006)
20. Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I., Busse, R.: XMark: a benchmark for XML data management. In: Proceedings of the 28th International Conference on Very Large Data Bases, VLDB Endowment, pp. 974–985 (2002)
21. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verification Reliab. **22**(5), 297–312 (2012)
22. Yao, B.B., Ozsu, M.T., Khandelwal, N.: XBench benchmark and performance testing of XML DBMSs. In: Proceedings of the 20th International Conference on Data Engineering, 2004, pp. 621–632. IEEE (2004)