A Partial Test Oracle for XML Query Testing

Dae S. Kim-Park, Claudio de la Riva, Javier Tuya
Department of Computing, University of Oviedo, Campus of Viesques, s/n, 33204 (SPAIN)
kim_park@lsi.uniovi.es, claudio@uniovi.es, tuya@uniovi.es

Abstract

A partial test oracle is proposed to verify the actual outputs in access testing on XML data. The considered software under test is a query program which receives as input an XML document obtained from an XML repository of any kind, and produces XML data as output. To deal with the actual outputs from this testing process, the partial oracle evaluates the correctness of the test executions according to: (1) a loose specification provided by the tester, and (2) a set of predefined constraints that describe invariant properties of the expected outputs. By means of the loose specification, the oracle can particularize the constraints to the concrete behaviour of the query program to test. This approach enables the oracle to give automatically a response about the correctness of the program under test with a certain precision at feasible cost. To illustrate the usefulness of the approach a case study is presented.

1. Introduction

One of the main challenges in software testing is the test oracle problem [1]. An oracle is known as a mechanism able to determine whether or not a software under test has behaved as expected during execution. In most cases the oracle is manual, which means that a human directly provides the expected output for each test case. But the derived costs of this approach make test oracle automation an important matter to be considered in software testing research.

Oracle automation is a relevant issue, especially when testing software that interacts with huge volumes of data located on external systems, as occurs, for example, in applications that need to retrieve data from database management systems. For this type of software manual oracles may be unfeasible when the amount of data to handle during testing is considerably large or complex.

Recently, with the advent of the Web and its underlying technologies based on the eXtensible Markup Language (XML) [9], data access operations against XML data sources (XML documents located on repositories) have become commonplace. The widespread use of XML-based formats for data representation and interchange between heterogeneous systems has drastically increased the dependency on XML data access operations in a variety of systems, especially in those concerned with Web services (for example, e-commerce applications or XML data services). These operations are carried out by means of XML queries that may be error-prone, and thus, they need to be tested, but the amount and the complexity of the data that may be involved in the tests require a feasible oracle.

In this work, we tackle the oracle problem in the particular case of access testing on XML data. The target programs to test are query processes treated as black-box programs that receive as input an XML document, and output an XML document fragment, which is a set of XML nodes (as defined in [12]) that does not need to be a well-formed XML document. The input XML document represents the external data required to be processed by the query, while the output XML document fragment, or *output fragment* for simplicity, is the actual output of the test and the result of the querying process. The target program can be represented by a diversity of XML query languages, such as XPath [10] and XQuery [11], as well as programmatically using XML manipulation libraries (for example a Java program using the SAX/DOM API). This diversity is possible since the black-box approach abstracts the details of the querying process to test. Bearing this testing scheme in mind, we concentrate solely on the oracle problem, leaving aside the techniques concerning test case generation [5].

The proposed oracle for XML query testing is a so-called *partial oracle* [7], which is known as an oracle able to determine whether an actual output of a test is incorrect without knowing the correct output. As such, a partial oracle in query testing does not need to infer the expected output to detect faults in the target

program. Hence, the internal processing of the partial oracle is easier to deal with, and the volume of data implicated in the tests has less impact on the oracle efficiency.

In general terms, oracle effectiveness depends on a specification derived from requirements which describes the expected behaviour of the target program. The more detailed the specification, the more precise the oracle is, but a precise oracle is inherently complex and costly to obtain. To balance the relationship between the level of detail supplied by the specification and the precision needed by the oracle to check the correctness of the target program, our partial oracle specification is composed by the following two elements: (1) *Behavioural requirements*, which are provided by the tester and comprise a set of loose requirements about a concrete query program; and (2) *Oracle constraints*, which describe invariant properties of the expected outputs. Oracle constraints are inherent to the oracle and independent from the program under test. They can supply a diagnostic for the test relying on given behavioural requirements. Because the oracle constraints are invariant, they can be embedded in the oracle; thus, from the point of view of the tester, only the behavioural requirements need to be provided to make the partial oracle operate.

The most remarkable contributions of this approach are:

- The use of a partial oracle to alleviate the oracle problem when testing query programs. This type of oracle is aimed to detect faults in test case executions even when the correct output is unknown. This facilitates the evaluation of the executions when there are large data structures in the test input and/or output, such as XML data.
- The definition of the oracle based on loose requirements, and constraints intended to check properties of the program outcomes. With this approach the tester does not need to provide accurate data to the oracle in order to specify the expected behaviour of the query, but the oracle can detect some types of faults despite the lack of precision.
- An initial step towards the oracle automation. The proposed oracle checks the test executions based on a set of constraints whose evaluation may be automated.

The rest of the paper explains in-depth the details of the approach. Section 2 presents an overview of the partial oracle integration in the testing environment. Section 3 details the behavioural requirements that need to be provided by the tester. Section 4 proposes a set of oracle constraints for query programs. Section 5 includes a sample case study for a better understanding of the partial oracle operation and its usefulness. Section 6 outlines some related work. Finally, Section 7 includes some conclusions and ideas for future work.

2. Overview of the partial test oracle

Figure 1 shows a diagram of the partial oracle integrated in the testing environment. The diagram represents a test case execution being evaluated against the partial oracle.

The partial oracle has two sources of information to use in the evaluation: the oracle specification, and the test data. The oracle specification is composed by the behavioural requirements provided by the tester, and the oracle constraints defined during the oracle design. On the other hand, the test data includes the test input and the resulting actual output.

To evaluate the execution, the oracle makes use of the behavioural requirements to parameterize the oracle constraints in order to check the presence of different types of errors.

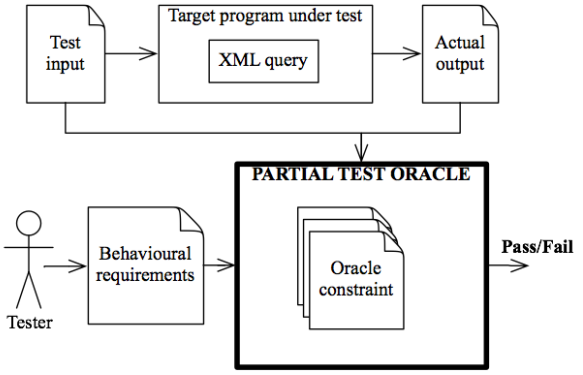

Figure 1. Partial oracle in the testing environment

During the constraint evaluation, if any constraint is violated, the partial oracle indicates the presence of a fault in the target program with a "Fail" response. A fail message may be provided by the oracle according to the violated constraint. Otherwise, if no faults have been found, the oracle gives a "Pass" response.

In the following sections, the partial oracle specification is detailed to give a view of its inner working.

3. Behavioural requirements

In this section the behavioural requirements are detailed. The requirements are represented by functions and sets comprising a loose specification about the expected behaviour of the target program.

Before presenting the behavioural requirements, the following definitions are needed.

Definition 1: A *query* q is a function defined as $q: D \rightarrow F$, where D is the infinite set of all XML documents, and F is the infinite set of all document fragments, satisfying $D \subset F$. As a convention, the input data of test cases is named $I \in D$. Thus, $q(I) \in F$ is the resulting output fragment (the actual output of the test).

Definition 2: A *specification-compliant query*, q_s, is a query function that complies with specific requirements imposed by the domain of application. In other words, the q_s function could act as an oracle for a query under test q expected to follow the same requirements (being $q_s(I)$ the correct expected output of the test cases associated with the query), but obviously, there is no easy way to obtain q_s. As such, q_s is only used for notation purposes to support other definitions further on.

Given these definitions, the elements that describe the behavioural requirements are presented below.

Definition 3: The *relaxed specification-compliant query*, t_s, is a query function that returns a superset of the correct expected output ($q_s(I)$). In other words, $q_s(I) \subseteq t_s(I)$, $\subseteq$ being the XML hierarchy-independent inclusion. The query t_s is likely to violate some requirements of the domain of application. Nevertheless, a query complying with this definition is not hard to obtain and can act as an approximation of q_s.

As an *ad hoc* example, consider the specification-compliant query (q_s) given by the XPath expression

```
doc("doc.xml")/a/b[d < 1.2]/c[e = 5][f = "12"].
```

The query retrieves data from the XML document `doc.xml` consisting of element nodes **`c`** which contains child elements `e` with value `5` and child elements `f` with value `"12"`. The elements **`c`** should be children of elements **`b`** containing elements `d` with a value less than `1.2`. In turn, the selected elements **`b`** must be under the root node (the document node) **`a`**. Note that this query is not the query under test; it is the query that should be in place of the query under test in order to obtain a correct output. This query is unavailable to the tester, but is shown here to clarify the example. A relaxed specification-compliant query, t_s, could be constructed from the specification of q_s by ignoring some predicate expressions and steps, obtaining, for example, the query

```
doc("doc.xml")//c[e = 5].
```

This query specifies that the output must contain every element node **`c`** in the XML document `doc.xml` whose child element `e` equals 5. Intuitively, it can be seen that the relaxed queries (t_s) are easier to obtain than specification-compliant queries (q_s) and hence, they are less error-prone. The example becomes meaningful when the query under test is far more complex.

Definition 4: The *inserted nodes*, A_s, is a finite set of XML nodes that are known to be always added to the output fragment returned by the query function q_s; thus, it establishes an inclusion property. A_s is a disjoint set of I ($A_s \cap I = \emptyset$). A_s is said to be *complete* if it contains all the nodes specified in the requirements, or *incomplete* if contains only a representative part of them.

Definition 5: The *excluded nodes*, D_s, is a set which contains the XML nodes that are not expected to appear in the actual output $q(I)$; thus, it establishes an exclusion property.

Definition 6: The *ordering specification*, R_s , is defined as $R_s = \{p_1, p_2, \ldots, p_N\}$ where p_i is an ordering pair. Each ordering pair is defined as $p_i = (s_i, op_i)$, in which s_i is a reference to a node in the output fragment and *op* is one of the operations in $\{\uparrow, \downarrow\}$, where $\uparrow$ is the ascending order and $\downarrow$ is the descending order.

At this point, the behavioural requirements are defined as follows.

Definition 7: The *behavioural requirements* for a query is defined as the tuple $S =< t_S, A_s, D_s, R_s >$, where t_s is a relaxed specification-compliant query, A_s is a set of inserted nodes, D_s is a set of excluded nodes, and R_s is an ordering specification. The s subscript indicates that the elements are specification-based. The level of detail given to the tuple elements is directly related to the precision of the partial oracle. It is not obligatory to fill every element exhaustively; for example, A_s does not need to be complete in order to obtain a useful result from the oracle, but as a consequence, the oracle may not be able to detect certain errors. The tester should bear in mind that, although each element of the requirement tuple contains data related with a specific feature or the query, they can be used by the oracle constraints to check the presence of varied types of errors. For example, the element of the requirement tuple A_s can be used to determine insertion errors as well as selection errors, though the set only contains insertion related information.

In the following sections it will be noticeable that the contents of the tuple *S* depend on the data required by the proposed oracle constraints in Section 4. Hence, if new oracle constraints are specified, the *S* tuple may be redefined with more elements.

4. Oracle constraints

The oracle constraints establish necessary conditions for the correct behaviour of the target query programs. These constraints are expected to be satisfied

in every correct test case execution, so that they are embedded in the oracle.

The criterion we have used to guide the definition of the oracle constraints is oriented to cover the main operations involved in the XML querying processes. For this purpose, we have modelled (black-boxed) the XML query behaviour as a combination of the three basic operations below, exemplified with the XPath and XQuery capabilities. Alternative methods to query XML data (e.g., Java with SAX/DOM) could also be represented by these operations, but for the sake of simplicity they have been omitted.

- *Input data selection*. This can be considered the most important operation on XML data retrieving. It takes a selected subset of nodes from the input XML document and passes it to the output fragment. This operation is present in every query that accesses data, and absent in queries that only perform data-independent computations such as arithmetic calculations. The operation includes the navigational functionality provided by XPath.
- *Data insertion*. This consists in adding new nodes in the output fragment that are not present in the input document. In terms of XPath and XQuery, it is equivalent to a direct or procedural insertion of nodes, by means of node constructors or the `fn:insert-before()` function [13], respectively.
- *Data ordering*. The nodes in the output fragment can be ordered by zero or more fields (XML element or attribute values). While the XPath language lacks functionalities to cover this operation, the XQuery language provides data ordering capabilities by means of the ORDER BY construction within FLWOR expressions. FLWOR stands for a very common construction in XQuery (composed by FOR, LET, WHERE, ORDER BY and RETURN expressions) similar to the SELECT statement from SQL. The usage of FLWOR expressions will be demonstrated further on in the case study (Section 5).

In the following subsections, a number of constraints are proposed for each operation, but note that further constraints could be specified in order to check the test case executions with more precision.

4.1. Constraints for input data selection

Constraints for input data selection are intended to check whether the data from the test case input is selected appropriately by the program. Some of them overlap with the same purpose as constraints for data insertion (see Section 4.2) checking both data selection and insertion operations. Constraints for input data selection use the insertion (A_s) and exclusion (D_s) sets of the behavioural requirements to perform the evaluation on selection operations, as well as the specification-compliant query t_s.

In order to characterize the constraints for input data selection, the following definitions are needed.

Definition 8: the root nodes of an XML document fragment, $roots(E)$, with $E \in F$, is the set of nodes in E without parent in the XML hierarchy.

Definition 9: $descendant(n_1, n_2, E)$ is a predicate which is true when the node n_1 is a descendant of the node n_2 in the document fragment $E \in F$. Otherwise it is false.

The constraints for input data selection are presented below.

- ***Constraint 1 (Inclusion in relaxed output)***: $q(I) \subseteq t_s(I)$ must be satisfied. Also, if the query does not carry out data insertion operations, the constraint can be refined as $q(I) \subseteq t_s(I) \subseteq I$. The constraint can detect general selection errors derived from mistaken predicates or bad node references. Also, it can detect unexpected insertion of nodes when $q(I) \nsubseteq I$ and $A_s = \emptyset$ with A_s complete.
- ***Constraint 2 (Inclusion of root nodes in relaxed output)***: $roots(q(I)) \subseteq roots(t_s(I))$ must be satisfied. This constraint states that the actual output contains a part of the expected root nodes. If it is not satisfied, we can infer that the query does not retrieve all the required data.
- **Constraint 3 (Hierarchy consistency)**: For each pair (n_1, n_2) where $n_1, n_2 \in t_s(I)$, if the condition $descendant(n_1, n_2, q(I))$ is true, then the predicate $descendant(n_1, n_2, t_s(I))$ must be satisfied. The constraint checks the hierarchy consistency among the XML nodes. If it is not satisfied, we determine that the query has applied an incorrect transformation to the input data.
- **Constraint 4 (Absence of unexpected nodes)**: $D_s \cap q(I) = \emptyset$ must be satisfied. In other words, the excluded nodes, D_s, must not be in the actual output.

4.2. Constraints for data insertion

The constraints of this section are oriented to check the correctness of data insertion operations only. They depend on the inclusion (A_s) set of the behavioural requirements.

- ***Constraint 5 (Presence of expected nodes)***: If the query carries out insertion of data, then $q(I) \cap A_s = A_s$. This constraint checks whether every node specified in the set of inserted nodes, A_s, is included at least once in the output fragment.
- ***Constraint 6 (Inserted nodes contained in the complete set)***: If the query carries out insertion of data and A_s is complete, then $(q(I) - A_s) \subseteq I$.

When it is not satisfied, it means that an unexpected insertion has been detected.

4.3. Constraints for data ordering

Only one constraint is proposed to check the data ordering. It depends on the ordering specification set, R_s, included in the behavioural requirements (defined in Section 3). Before showing the constraint for data ordering, the definitions below are required.

Definition 10: The order predicate, $order(E,p)$, determines whether the document fragment $E \in F$ complies with the ordering defined by the ordering pair p (as in definition 6 in Section 3).

Definition 11: The ordered groups of a document fragment $E \in F$, $Gr(E,p)$, is a function that returns a sequence containing sets of nodes from E grouped by the ordering pair p.

— ***Constraint 7 (Correct ordering)***: Given an ordering specification R_s, the predicate $Order(q(I), R_s)$ is satisfied. This predicate is defined recursively as

$$Order(E, R_s) = order\big(E, first(R_s)\big) \wedge \bigwedge_{g \in Gr(E, first(R))} Order\big(g, R_s - first(R_s)\big)$$

where E is a document fragment, and the $first(R_s)$ function returns the first ordering pair of R_s. The predicate $Order(g, \emptyset)$ is always true. The constraint checks that the ordering in the output file is as expected. If it is not satisfied, the ordering operations may be missing or wrong in the target program.

5. Application

In this section a case study is presented to show the usefulness of the partial oracle. First, a program specification and a test input are presented. Then, the specification is translated into a tuple of behavioural requirements which are finally used by the oracle constraints to detect faults.

5.1. Specification and test input

The specification for the target program in this example states that *it is required to retrieve the books from the file* `library.xml` *whose price does not exceed the limit amount of 70$. The output must be enclosed between* `cheapBooks` *element tags, shown in ascending order by year of publication and in descending order by price. In addition, each book element must contain its title, publisher and price, and the publication year as the element* `publicationYear`.

The considered input for the test case of this sample is shown in Figure 2. It is supposed to be contained in a file named `library.xml`, and includes information about books in a library.

```
<library>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last><first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price>45.95</price>
  </book>
  <book year="1994">
    <title>Advanced Programming in the Unix
           environment</title>
    <author>
      <last>Stevens</last><first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author>
      <last>Abiteboul</last>
      <first>Serge</first>
    </author>
    <author>
      <last>Buneman</last><first>Peter</first>
    </author>
    <author>
      <last>Suciu</last><first>Dan</first>
    </author>
    <publisher>Morgan Kaufmann
               Publishers</publisher>
    <price>89.95</price>
  </book>
</library>
```

Figure 2. Sample input XML data (`library.xml`)

5.2. Sample behavioural requirements

Based on the specification given in Section 5.1, a behavioural requirement tuple, S, is provided manually by the tester. The behavioural requirements in this case study will be composed by the following elements:

— The relaxed specification-compliant query, t_s, in Figure 3 represented in the XQuery language (other languages could be used instead). The query specifies an output composed by the books from the file `library.xml`, enclosed between `cheapBooks` element tags. Each book contains its title, publisher and price, and the publication year as the element `publicationYear`. This query has been obtained by omitting the limit amount of 70$ and the ordering by the year and price. The relaxed specification-compliant query could be constructed in many other ways; for example omitting the ordering by one of the fields or the limit amount only.

- The inserted nodes set A_s = {cheapBooks, publicationYear}. A_s is complete according to the specification because it contains every node in the expected output that is not in the input (Figure 2). A_s could also be specified as an incomplete set if we ignore any of its current members (cheapBooks and/or publicationYear).
- The excluded nodes set D_s = {library, author, @year}. In the expected output, the node library is replaced by the element cheapBooks, the element author is not required, and the attribute node year is substituted by the element publicationYear.
- The ordering specification represented by the set R_s ={(/cheapBooks/book/publicationYear, ↑), (/cheapBooks/book/price, ↓)}.

  The ordering is the same as the specification; however, it could alternatively establish an ordering by publicationYear or by price only, or by none of them. The node references have been represented as XPath expressions, but it is not mandatory.

As seen, alternative behavioural requirement tuples with different levels of detail could be constructed. Nonetheless, in further sections we are going to consider only the given behavioural requirements.

```
<cheapBooks>{
  for $b in doc("library.xml")/library/book
    return
      <book>
        <title>{$b/title/text()}</title>
        <publisher>
          {$b/publisher/text()}
        </publisher>
        <price>{$b/price/text()}</price>
        <publicationYear>
          {fn:data($b/@year)}
        </publicationYear>
      </book>
}</cheapBooks>
```

Figure 3. Relaxed specification-compliant query (t_s).

5.3. Fault detection

Suppose that an incorrect target program is implemented as the XQuery expression shown in Figure 4. The expression has some injected faults that will be detailed later on in this section. Hence, due to the presence of faults, the partial oracle verdict should be a "Fail" response.

The actual output of the target program q (Figure 4) is represented in Figure 5. It has been obtained from the input shown in Figure 2.

```
for $b in doc("library.xml")//book
  let $maxPrice := 70.0
  where $b/price <= $maxPrice
  order by $b/@year ascending,
           $b/price ascending
  return
    <book>{$b/@year}
        <author>{$b/title/text()}</author>
        <publisher>
           <price>{$b/price/text()}</price>
          {$b/publisher/text()}
         </publisher>
        <publicationYear>
          {fn:data($b/@year)}
         </publicationYear>
    </book>
```

Figure 4. Target program (q) with faults.

```
<book year="1994">
  <author>TCP/IP Illustrated</author>
  <publisher><price>45.95</price>
    Addison-Wesley</publisher>
  <publicationYear>1994</publicationYear>
</book>
<book year="1994">
  <author>Advanced Programming in the Unix
       environment</author>
  <publisher><price>65.95</price>
    Addison-Wesley</publisher>
  <publicationYear>1994</publicationYear>
</book>
```

Figure 5. Actual output ($q(I)$).

The partial oracle takes the requirements provided by the tester (Section 5.2), the input in Figure 2, and the actual output in Figure 5 to evaluate the fulfilment of the oracle constraints proposed in Section 3. The results of the evaluation are presented below, showing each constraint result individually. Note that the partial oracle must decide when to give a response and stop the evaluation when faults are encountered.

- *Constraint 1 (Inclusion in relaxed output)*: The constraint is not satisfied, because the actual output $q(I)$ contains the author node, which is not included in $t_s(I)$. The constraint 4 will detect a fault due to the presence of the author node in the actual output. This overlapped detection is a consequence of treating each constraint independently from the others.
- *Constraint 2 (Inclusion of root nodes in relaxed output)*: Being $roots(q(I))$ = {book} and $roots(t_s(I))$ = {cheapBooks}, we have evidence that the target program q retrieves books, but the cheap books are missing. This shows that the constraint can detect the fault despite the fact that the books in the actual output $q(I)$ comply with the definition of a cheap book (with price less than or equal to 70$) given by the specification in Section 5.1.
- *Constraint 3 (Hierarchy consistency)*: In the actual output $q(I)$, $descendant$(price, publisher, $q(I)$) is true, while $descendant$(price, publisher, $t_s(I)$)

is false. Then, the target program q contains a faulty transformation involving the node `price`.

- *Constraint 4 (Absence of unexpected nodes)*: The constraint detects the wrong selection of the `author` node in the actual output $q(I)$. In fact, it is a bad node definition in the target program, in which the `title` node has been mistaken as `author`.
- *Constraint 5 (Presence of expected nodes)*: A fault is detected, because there is a missing insertion of the node `cheapBooks`.
- *Constraint 6 (Inserted nodes contained in the complete set)*: The A_s set given in Section 5.2 is complete. Then, the constraint detects an unexpected inclusion of the attribute node `@year`, because $(q(I) - A_s) - I = \{$`@year`$\}$, which implies $(q(I) - A_s) \nsubseteq I$.
- *Constraint 7 (Correct ordering)*: The actual output of the example in $q(I)$ does not hold the ordering pair (`/cheapBooks/book/price`, ↓). Then, this constraint detects an ordering fault.

Each of the detected faults could be shown to the tester as human-readable messages in order to ease the debugging of the target program. For example, the fault message derived from the constraint 7 could be: *"The element node* `/cheapBooks/book/price` *is not sorted in descendant order."*

To end this case study, a possible specification-compliant query (q_s) is shown in Figure 6. Note that the specification-compliant query has been determined manually from the textual specification. Here it is presented for illustrative purposes only, but actually it would not be available.

```
<cheapBooks>{
  for $b in doc("library.xml")/library/book
  let $maxPrice := 70.0
  where $b/price <= $maxPrice
  order by $b/@year ascending,
           $b/price descending
    return
      <book>
        <title>{$b/title/text()}</title>
        <publisher>{$b/publisher/text()}
        </publisher>
        <price>{$b/price/text()}</price>
        <publicationYear>
           {fn:data($b/@year)}
        </publicationYear>
      </book>
}</cheapBooks>
```

Figure 6. Specification-compliant XML query (q_s).

The correct expected output of the query in Figure 3 is shown in Figure 7. As with the specification-compliant query, it is presented for illustrative purposes.

```
<cheapBooks>
  <book>
    <title>Advanced Programming in the Unix
         environment</title>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
    <publicationYear>1994</publicationYear>
  </book>
  <book>
    <title>TCP/IP Illustrated</title>
    <publisher>Addison-Wesley</publisher>
    <price>45.95</price>
    <publicationYear>1994</publicationYear>
  </book>
</cheapBooks>
```

Figure 7. Expected output ($q_s(I)$).

If the specification-compliant query in Figure 6 is provided as the target program and the behavioural requirements of Section 5.2 are supplied, the oracle will give a "Pass" response.

6. Related work

Weyuker [7] presents the non-testable programs as those for which it is not possible to obtain a test oracle, either because it does not exist, or because it is difficult to obtain in practice. The latter is the case of XML query testing due to the complex structure and volume of the output data, and the huge expressiveness provided by the query languages involved in the tests. Weyuker states that partial oracles are suited to check the execution of non-testable programs. Also, Bertolino [1] states that partial oracles may be the only viable solution to oracle automation.

Several works also apply partial oracles to particular types of target programs. Hunter and Strooper [4] propose a method to obtain partial oracles from event models to test concurrent programs. Ran et al [6] provide a testing framework for Web database applications based on partial oracles, addressing the evaluation of the database status, but not the queries.

Chen et al [3] develop the theory of metamorphic testing, which presents similarities with partial oracles in the sense that metamorphic relations can be understood as oracle constraints. However, metamorphic testing is more oriented to the test case generation, and the correctness concluded by metamorphic relations is based on multiple executions of the target program and not on expected results inferred from requirements.

Willmor and Embury [8] propose an approach to specify intensional test cases for relational database applications. The intensional database test cases are formed by preconditions that specify the initial state of the database, and postconditons that must hold after execution of the target program. While the preconditions are outside the scope of the present work, the postconditions have an equivalent purpose to the

partial oracle we presented, but the major differences are (1) the target programs we addressed (XML queries), which provide more complex capabilities such as for data transformation, and (2) the oracle constraints we proposed, which set invariant conditions that are known independently from the test cases details.

7. Conclusions and future work

The evaluation carried out by the proposed partial oracle is based on a set of constraints that describe invariant properties of the expected outputs. To check the correctness of each test case execution, the constraints are particularized using a tuple of behavioural requirements provided by the tester. The behavioural requirements for each test case are loose, thus simpler than a full specification. Furthermore, the requirements are represented in a familiar notation for the tester (XML queries and sets of node names).

It is expected that once the tester provides the behavioural requirements, the oracle will be able to perform the evaluation without human intervention, thus giving a response automatically. It should be clarified that oracle automation has not been developed in this work, since the approach does not detail how the oracle constraints should be evaluated.

Because the constraints set only necessary conditions to verify the test executions, the verdict of the partial oracle cannot conclude the correctness with total certainty, as it can only guarantee whether some types of faults are present. For example, the proposed constraints are unable to detect wrong values on atomic types [12] (for example, integers or strings) located in the actual output of the test.

The immediate future work includes definition of more oracle constraints to improve the intrinsic precision of the partial oracle.

A suitable paradigm to implement an automated tool based on the proposed partial oracle could be a rule-based system [2], in which the oracle constraints could be represented by rules, and the behavioural requirements could be treated as rule variables. The order in which the constraints are evaluated is important to improve the performance of the partial oracle, or to help in the debugging process since the correction of an error may fix other derived errors. In a rule-based system this could be achieved by establishing rule priorities.

Another possible line of work could be devoted to adapt the oracle to cover other target programs besides XML queries, such as SQL for relational data.

8. Acknowledgements

This work was partially funded by the Department of Education and Science (Spain) and ERDF funds within the National Program for Research, Development and Innovation, project Test4SOA (TIN2007-67843-C06-01) and the RePRIS Software Testing Network (TIN2007-30391-E).

9. References

[1] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams", *International Conference on Software Engineering 2007: Future of Software Engineering*, IEEE Computer Society, 85-103, 2007.

[2] B.G. Buchanan, R.O. Duda, "Principles of Rule-Based Expert Systems", Stanford University, CA, USA, 1982.

[3] T.Y. Chen, F.-C. Kuo, T.H. Tse, Z.Q. Zhou, "Metamorphic Testing and Beyond", *Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice*, 00: 94:100, 2003.

[4] C. Hunter, P. Strooper, "Systematically deriving partial oracles for testing concurrent programs", *ACM International Conference Proceeding Series, In Proceedings of the 24th Australasian conference on Computer science*, 11: 83-91, 2001.

[5] D.S. Kim-Park, C. de la Riva, J. Tuya, J. García-Fanjul, "Generating Input Documents for Testing XML Queries with ToXgene". Testing: Academic and Industrial Conference – Practice and Research Techniques, Fast Abstract Track, 2008

[6] L. Ran, C. Dyreson, A. Andrews, R. Bryce, C. Mallery, "Building test cases and oracles to automate the testing of web database applications", *Information and Sofwtare Technology*, Butterworth-Heinemann, Newton, MA, USA, 51 (2): 460-477, 2009.

[7] E.J. Weyuker, "On Testing Non-testable Programs", *The Computer Journal*, 25(4): 465-470, 1982.

[8] D. Willmor, S.M. Embury, "An intensional approach to the specification of test cases for database applications", *In Proceedings of the 28th International Conference on Sofwtare Engineering*, Shangai, China, 102-111, 2006.

[9] World Wide Web Consortium, "Extensible Markup Language (XML)", http://www.w3.org/TR/REC-xml/, 2008.

[10] World Wide Web Consortium, "XML path language 2.0 (XPath 2.0)", http://www.w3.org/TR/xpath20/, 2007.

[11] World Wide Web Consortium, "XQuery 1.0. An XML query language", http://www.w3.org/TR/xquery/, 2007

[12] World Wide Web Consortium, "XQuery 1.0 and XPath 2.0 Data Model (XDM)", http://www.w3.org/TR/xpath-datamodel/, 2007.

[13] World Wide Web Consortium, "XQuery 1.0 and XPath 2.0 Functions and Operators", http://www.w3.org/TR/xpath-functions/, 2007.