Type Checking and Testing of SPARQL Queries*

Jesús M. Almendros-Jiménez and Antonio Becerra-Terón
Dept. of Informatics. University of Almería. 04120-SPAIN. {jalmen,abecerra}@ual.es

In this paper we describe a property-based testing tool for SPARQL. The tool randomly generates test cases in the form of instances of an ontology. The tool checks the well typed-ness of the SPARQL query as well as the consistency of the test cases with the ontology axioms. With this aim, a type system has been defined for SPARQL. Test cases are later used to execute queries. The output of the queries is tested with a Boolean property which is defined in terms of membership of ontology individuals to classes. The testing tool reports counterexamples when the Boolean property is not satisfied.

1 Introduction

Property-based testing (PBT) is a well-known technique of program testing [25] involving the specification of *properties/assertions* on the output of a program to be tested. Properties on the output describe the required relationships between output data, which should be ensured by *bug-free programs*. Normally, *test cases* are generated as input of the program and the properties/assertions are checked on the output of the test cases. When a counterexample is found, that is, an input test case in which the property is not satisfied by the result, the program has a bug. PBT can use either *black-box* techniques (i.e., *randomly generated test cases*) or *white-box* techniques (i.e., *test cases generated from code analysis*) [4]. Among others, PBT has been studied in Java [19, 7], functional languages [11, 23] model transformation languages [22], and relational databases [10]. PBT has as main goal to detect program bugs, using specifications (i.e., properties, constraints, etc.,) to describe the expected behavior of programs. PBT is mainly a debugging tool, and programmers use PBT to detect his/her own mistakes when manual testing (i.e., user generated test cases) are not enough to analyze all the possible inputs of his/her programs. Automatic and randomly generated test cases are helpful by providing exhaustive case analysis, revealing more failures than user generated test cases.

This paper presents an implemented testing tool, whose purpose is the debugging of a SPARQL query Q, given an underlying OWL ontology **TBox** T, as well as additional explicit properties (i.e., constraints) on admissible valuations for the query. A constraint takes the form of a (possibly complex) OWL concept C, built out of the signature of T (and Q), which must be verified by admissible valuations of a given variable of the query.

The debugging strategy consists on finding instances A (i.e., OWL ontology **ABox**es of T) which provides at least one answer to Q, and such that at least one of these answers violates one of the constraints. If such an **ABox** exists, then the constraint is not verified by all answers to the query in all ABoxes verifying T, which is considered as a bug (called a *"type 1 bug"* in what follows). Alternatively, if no instance of T providing some answer to Q can be found, a different type of bug (*"type 2 bug"* in what follows) is identified, indicating that Q is incompatible with T (regardless of the additional constraints).

*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.

Submitted to:
PROLE 2017

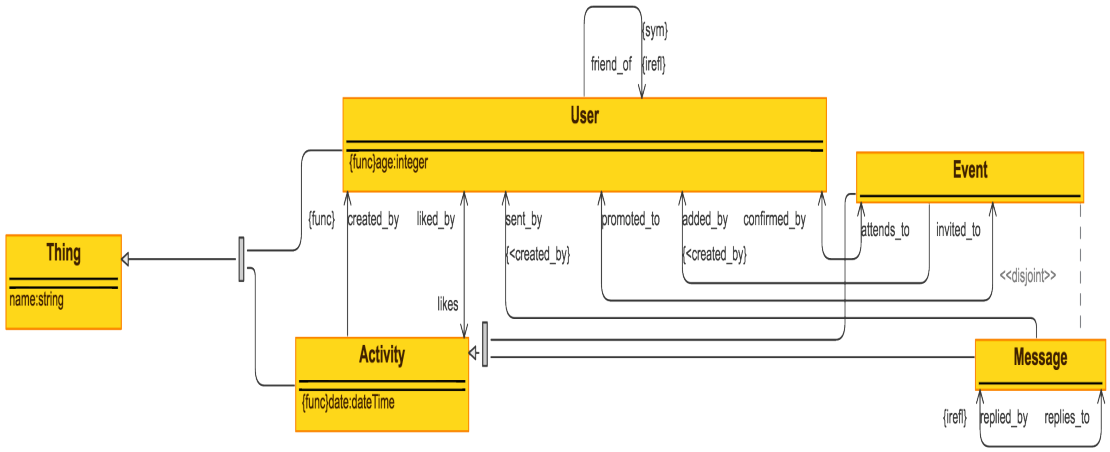

Figure 1: Ontology Example

Debugging is performed in two successive phases. The first phase is a preliminary type checking, which allows spotting some cases of type 2 bugs. The second phase consists on randomly generating (with parameters) ABoxes out of the signature of T, until either:

- an **ABox** verifying T and providing an answer to Q which violates some constraint is found: this is a type 1 bug.
- no **ABox** verifying T and providing an answer to Q can be found: this is likely to indicate a type 2 bug (although not certain).
- **ABox**es verifying T and providing answers to Q have been found, but none of these answers violate a constraints: this is likely to indicate the absence of type 1 or type 2 bugs.

A *Description Logic (DL)* reasoner (namely *Hermit* [13]) is used for the preliminary type-checking phase, as well as checking satisfiability of $A \cup T$ for each generated **ABox** A. Finally, in order to check whether $\{C(v)\} \cup T$ is satisfiable for a constraint C and valuation v of a given variable the DL reasoner is also user. The random generation of **ABox**es relies on an existing tool developed for XML/XQuery [3], granting the tester the possibility to customize multiple parameters. Thus, test cases (i.e., randomly generated **ABox**es) are used to test a SPARQL query with constraints given on output variables. When the program passes the test (i.e., the constraint on output variables) the tool reports *"Ok: passed"*; otherwise the program fails the test and the tool reports *"Output Property Falsifiable"*. In the case of fail, the tool is able to show counterexamples; that is, examples of input data (i.e., test cases –randomly generated **ABox**es–) not verifying the required constraints on the output. Each counterexample serves as witness of the program bug.

In order to illustrate the work, we will consider an example of ontology (graphically depicted in Figure 1) used in the rest of the paper as a running example. The ontology example defines a *social network* in which there are two main classes *User* and *Activity* in order to represent users and activities of users in the social network, respectively. *Activity* has as subclasses *Message* and *Event* which can be *created_by* users. *created_by* is an object property that maps *Activity* to *User* elements. *created_by* property has as sub-properties *added_by* and *sent_by*: *sent_by* and *added_by* are used for *Message* and *Event* elements, respectively. A user can be *friend_of* another one. In addition, a given user *attends_to* an event, and he/she can be *invited_to* an event. Users *like* activities, activities are *liked_by* users (thus, *likes* is the inverse of *liked_by*), and events are *promoted_to* users (thus, *promoted_to* is the inverse of

(1)
```
SELECT ?event ?user2
WHERE {
  ?event sn:created_by ?user1 . ?event  sn:likes ?user2 .
  ?user2 sn:invited_to ?event
  }
```

(2)
```
SELECT ?msg1
WHERE {
  ?msg1 sn:sent_by ?user . ?msg1 sn:replied_by ?msg1
  }
```

(3)
```
SELECT ?user1 ?event ?user2
WHERE {
  ?event sn:added_by ?user1 . ?event sn:date ?date .
  ?user1 sn:friend_of ?user2 . ?user2 sn:age ?age .
  FILTER (?age >= 40 &&
      ?date >= '2017-01-01T00:00:00Z'^^xsd:dateTime)
  }
```

(4)
```
SELECT ?event ?date
WHERE {
  ?event sn:date ?date . sn:athleticswch sn:date ?date2
  FILTER(?date >= ?date2 &&
  ?date < '2017-01-01T00:00:00Z'^^xsd:dateTime)
  }
```

(5)
```
SELECT ?user
WHERE {
  ?user sn:name  ?name1 ;
    sn:age  ?age1 .
  sn:antonio sn:name  ?name2 ;
    sn:age  ?age2 .
  FILTER (?age1 > ?age2 && ?name1 != ?name2)
  }
```

(6)
```
SELECT ?event ?likes ?attendees WHERE {
    ?event sn:created_by ?user
  { SELECT ?event (COUNT(?person) AS ?likes) WHERE {
      ?person sn:likes ?event }
    GROUP BY ?event }
  UNION
  { SELECT ?event (COUNT(?person) AS ?attendees) WHERE {
      ?person sn:attends_to ?event }
    GROUP BY ?event }
}
```

Figure 2: Examples of Buggy SPARQL Queries

invited_to). Events are *confirmed_by* users. Finally, a message *replies_to* another message, and a message is *replied_by* another message (thus, *replies_to* and *replied_by* are inverse of each other). Additionally, we express some constraints on properties: *friend_of* and *replied_by* are irreflexive properties (and thus the corresponding inverses), and *created_by* is functional one. With regard to data properties, we consider *name*, *age* for users and *date* for activities. This is a simple **TBox**, but enough to illustrate our examples. Let us suppose now the queries shown in Figure 2:

Case (1). Here there is a *"type 2 bug"*. More concretely, it is due to the signature of properties (i.e., domain and range), and thus it is considered as a typing error. The triple pattern *?event sn:likes ?user2* is wrong, because users like activities (and, in particular, events) and thus, the order of the triple components is wrong. Here the bug can be solved by changing the triple pattern by *?event sn:liked_by ?user2*, or by changing the order *?user2 sn:likes ?event*. From the programmer's point of view, this query returns an empty answer when the bug is present, but some hint could be given to the programmer to detect such mistake. Our testing tool includes type checking (i.e., domain and range conformance checking, among

others), and thus an error message will be reported. Type checking is crucial for carrying out testing because property-based testing cannot be used on empty answers.

Case (2). Now there is also a *"type 2 bug"*, and the bug is due to ontology constraints. The triple pattern *?message1 sn:replied_by ?message1* is wrong, because *replied_by* is an irreflexive property: a message cannot be answered by itself. Thus, again the answer will be empty. In this case the bug can be solved by replacing the triple pattern *?message1 sn:replied_by ?message1* by *?message1 sn:replied_by ?message2*, that is, using distinct variables names. Here the testing tool is not able to find test cases because all of the possible test cases (i.e., **ABox** instances) are incompatible with the ontology axioms. Consistency of test cases is also crucial because only consistent inputs can be used for property-based testing.

Case (3). The query is well-typed and does not contradict ontology constraints. However, let us suppose the answer is empty or the query returns wrong results. Here, the programmer can use the testing tool to discover the bug, but he/she additionally needs to specify the expected behavior of the query. In cases (1) and (2) the expected behavior of the query can be also specified but *"type 2 bugs"* arise regardless the imposed constraints. More concretely, in this case he/she has to define a (possibly complex) concept describing the semantics of output variables. Let us suppose that the expected behavior of the query is *"Retrieve events before this year added by users older than 40, and friends of these users"*, and wrong answers, that is, events added by users younger than 40, are retrieved. Here, there is a non-obvious mistake: the triple pattern *?user2 sn:age ?age* is wrong because *user1* should be older than 40 (instead of *user2*) to fit with the expected behavior. By specifying the complex concept *"user older than 40"*, and requiring the membership of variable *user1* to this concept, the testing tool reveals a *"type 1 bug"*, when test cases (i.e., **ABox** instances) are randomly generated. The testing tool reports a counterexample, that is, a test case for which the property is not verified on the output. For instance, a counterexample including a user younger than 40 who added an event.

Case (4). Here we must consider again the expected behavior (i.e., the query is well-typed and constraints are not violated). The expected behavior of the query is *"Retrieve date of events of this year after Athletics Word Championship"*. Here the bug can be found on the *FILTER condition*: *?date* $<$ *'2017-01-01T00:00:00Z"^^xsd:dateTime*. The answer will be empty, because *Athletics Word Championship* is celebrated this year, and thus "after" (i.e., *?date* $>=$ *?date2*) contradicts *?date* $<$ *'2017-01-01T00:00:00Z"^^xsd:dateTime*. In this case, there is a *"type 2 bug"* but the error is due to the incompatibility of the FILTER condition with the knowledge base. Here, the testing tool can be used to test the concept *"event of this year"* on the variable *event*. Test cases with non empty answers cannot be found in this case. This case can be considered similar to case (2), but here test cases are consistent with regard to the ontology axioms.

Case (5). We must consider again the expected behavior. The expected behavior is *"Retrieve users younger than Antonio but with distinct names"*. The query returns people older than 40, even though *Antonio* is 40 years old, due to the wrong filter condition *?age1* $>$ *?age2*. Here the opposite case of (4) happens: FILTER condition is compatible with the knowledge base. The testing tool can be used to test the concept *"user older than 40"* on the variable *user1*. In this case, the testing tool will report a counterexample (i.e., a test case for which the concept *"user older than 40"* on the variable *user1* is not verified). For instance, an user younger than *Antonio*, with distinct name. This case is again a *"type 2 bug"* similar to case (3), but the bug is due to a wrong FILTER condition.

Case (6). Finally, an example of nested SPARQL expression is shown. The expected behavior of the query is: *"Retrieve events with likes and attendees, and number of them for each one"*. Here, the UNION operator is used to join likes and attendees of the events, where each one is computed with an inner SPARQL expression. Let us suppose the programmer uses the testing tool with the concepts *"Event with at least one like"* and *"Event with at least one attendee"* on the variable *event*. Testing tool will

report counterexamples: events with only attendees and events with only likes, respectively. The error in this example is due to the wrong use of the UNION operator, which also retrieves events with only likes and events with only attendees (i.e., no join is accomplished). Thus *"type 2 bugs"* on SPARQL queries can also occur on nested SPARQL queries.

While testing has been studied in other contexts (e.g., SQL [27, 10, 9, 12, 18] and XML-based databases [1, 6]) and programming bugs are well established for SQL (e.g., [8, 16]), as far as we know SPARQL debugging and testing have not been studied yet. In fact, we have evaluated the best state-of-the-art SPARQL implementations and we have found that they are not equipped with testing facilities. Additionally, type checking is not currently supported by SPARQL implementations. This has as consequence empty answers for wrongly typed queries. Only benchmarking datasets [24, 15, 5] are publicly available to analyze performance of SPARQL implementations. Also there are works about analysis of data [21, 20], but less attention has been paid to SPARQL analysis.

This tool has been successfully used in two previous contexts [3, 1, 2]: the database query language *XQuery* [26] and the model transformation language *ATL* [17]. Both ones share a common element: input and output data are XML documents. This is also the case of SPARQL which uses the XML serialization of RDF/OWL ontologies. Test cases are generated from an XML Schema with the tool developed for XML/XQuery. In the current approach, a **TBox** to XML Schema has been defined in order to reuse the previously developed test case generator. A Web tool available at `http://minerva.ual.es:8080/SPARQL/` has been developed for on-line debugging of SPARQL queries, and also a batch of examples have been used to evaluate the proposal. The tool has been implemented in the BaseX XQuery interpreter [14], mainly responsible of random test case generation. SPARQL (*Apache Jena ARQ engine*[1]) has been embedded into XQuery thanks to *Java binding* capabilities of BaseX. Also, the reasoner HermiT has been embedded into XQuery for type checking of SPARQL queries, consistency checking of test cases, as well as for property-based testing.

The structure of the paper is as follows. Section 2 describes the RDF/OWL to XML Schema and the random generation of ontology instances. Section 3 describes the type checking algorithm. Section 4 presents property-based testing. Section 5 shows a batch of examples. Section 6 presents benchmarks, evaluation remarks and the Web tool. Finally, Section 7 concludes and presents future work.

2 Random Generation of Ontology Instances

In this section, we will explain how ontology instances are generated by the tool, in particular, firstly, how ontologies (i.e., ontology classes and properties) are mapped into XML Schemas; secondly, how XML Schemas are used to generate test cases (i.e., ontology instances), and, finally, how the human tester can customize the test case generator.

In [3] a XML test case generator has been designed. This test case generator takes as input an *XML Schema* and it is able to automatically generate XML documents conforming the XML Schema. The test case generator is customizable in the sense that the human tester provides an XML Schema to the test case generator, in which he/she specifies the type and size of documents to be generated, i.e. XML *labels* and *attributes*, and *number* and *values* of them. The number is specified by *minOccurs* and *maxOccurs* attributes of the labels in the XML Schema, as well as by *use* attribute of XML Schema attributes. The values are specified in the *simpleType* section of the XML Schema using an *enumeration* of values. The tool randomly generates XML documents as *combinations* of labels and attributes values, iterating from an *initial set* of XML documents of minimal size conforming the XML Schema and *increasing size in*

[1] `https://jena.apache.org/documentation/query/`

```
<xs:schema>
<xs:element name="rdf:RDF">
<xs:complexType>
<xs:sequence>
<xs:element name="sn:User" minOccurs="0" maxOccurs="unbounded">
<xs:complexType>
<xs:attribute name="rdf:about" type="UserType" use="required"/>
<xs:sequence>
<xs:element name="sn:age" type="ageType" minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="sn:attends_to" minOccurs="0" maxOccurs="unbounded">
<xs:complexType>
<xs:attribute name="rdf:resource" type="EventType" use="required"/>
</xs:complexType>
</xs:element>
</xs:schema>
```

Figure 3: XML Schema of the Ontology Example

```
<xs:simpleType name="UserType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="default"/>
    </xs:restriction>
  </xs:simpleType>
<xs:simpleType name="ageType">
    <xs:restriction base="xs:integer">
      <xs:enumeration value="0"/>
    </xs:restriction>
</xs:simpleType>
<xs:simpleType name="integer">
    <xs:restriction base="xs:string">
      <xs:enumeration value="http://www.w3.org/2001/XMLSchema#integer"/>
    </xs:restriction>
</xs:simpleType>
```

Figure 4: *SimpleType* of the Ontology Example

each step. The random test case generation can be potentially infinite when *maxOccurs* attributes are set to *"unbounded"*. However, the test case generator can be also customized by the number of iterations and thus producing a *finite number* of test cases. The input of the test case generator in the current approach is a **TBox** like the running example, while the output of the test case generator is an **ABox**. Thus, the test case generator has to produce users and activities, as well as *friend_of*, *created_by*, etc., instances. Additionally, the test case generator has to set values for *name*, *age* and *date*. In order to use the XML test case generator we have defined a **TBox** to XML Schema mapping. With this transformation the test case generator can be used to produce RDF/OWL XML documents containing **ABox**es.

Figure 3 shows (a piece) of the XML Schema generated from the **TBox** of the ontology example. The mapping is as follows: classes are mapped to XML labels in which `rdf:about` is considered as an XML attribute. Object properties are mapped to XML labels, and the XML attribute `rdf:resource` is used to fill relationships. Additionally, data properties are mapped to XML labels and the range of them to *type* attributes. Each property is added as sub-label of its domain. Attributes `rdf:about` have type *CType* whenever they belongs to class *C*. For instance, *UserType* for class *User*. The same can be said for the XML attribute `rdf:resource`, depending, in this case, on the range of the object property. In the case of data properties, the type is *DType* whenever the range of the data property is *D*. The elements of each type (i.e., *CType* and *DType*) are defined as a *simpleType* definition of the XML Schema (see Figure 4). In the running example, we have six types, i.e., *nameType*, *ageType*, *dateTimeType*, *userType*, *activityType*, *messageType* and *eventType*. There is an additional and fixed *simpleType* for each XML datatype. Attributes *minOccurs* and *maxOccurs* of the XML Schema have been set (by default) to zero and unbounded, respectively. Also each *simpleType* definition has an enumeration, in the running

example: *nameType*, *ageType*, *dateTimeType*, *userType*, *activityType*, *messageType* and *eventType* which is initially set to default values.

Next, the human tester can edit the XML Schema in order to provide values for *minOccurs* and *maxOccurs*, as well as can fill *simpleType* with values: literals for datatypes and individuals for classes, in order to generate useful instances for each query. This process is facilitated by the developed Web tool, in which the list of classes and properties is shown, and *minOccurs* and *maxOccurs* as well as *simpleType* statements can be easily modified (see Section 5). In terms of ontologies, the human tester edits the XML Schema in order to select classes and properties of the ontology **TBox**. The selection depends on the SPARQL query to be tested: classes and properties occurring on the *triple patterns*. It is required for improving the performance of testing; when more classes and properties than the required ones are used, testing is possible but performance is probably lower. The human tester has also to decide the number and value of individuals, the number of relationships between individuals, as well as the number and value of datatypes, according to the SPARQL query to be tested. The choice of number of individuals and their relationships depends on the number and form of triples patterns. The choice of value of datatypes depends on relevant and extreme values on Boolean conditions.

With this transformation and instantiation of parameters, the test case generator designed in [3] can be used to produce ontology instances of increasing size. Basically, starting from an initial step (step *0*) with minimal ontology instances according to *minOccurs* values, it generates new elements in each step (step *n+1*) to ontologies of step *n*, up to *maxOccurs* value is reached. In other words, the test case generator adds new individuals and property values in each step, increasing the size of ontology instances. Additionally, it randomly takes values for individuals and datatypes from the *simpleType* section. The number of steps *n* is a parameter of the test case generator, in order to limit the size of ontology instances. The human tester can play with this parameter generating larger ontology instances. Also, the human tester can play with *minOccurs* (and *maxOccurs*) values in order to generate a larger number of individuals for each class and relationships between individuals as well as a larger number of values for properties.

Since the human tester can edit the XML Schema for customizing test cases, unfortunately he/she can generate inconsistent test cases as well as insufficient test cases. Let us suppose that *age* is defined as functional property and the *minOccurs* value for *age* is set to two in the XML Schema. In this case only test cases violating the functional property of *age* would be generated. Fortunately, the testing tool will report an error when only inconsistent test cases are generated: *"It was not possible to find consistent tests"*. Moreover insufficient test cases can be generated when *minOccurs* of *age* is set to zero, the number of iteration steps is zero, and *age* is used in a triple pattern of the query. Fortunately, in this case the tool reports an error of empty test cases: *"It was not possible to find non trivial tests"*. When it happens, it means that test cases are not enough for the query (i.e., a *coverage* problem arises) and the human tester can increase the number of iterations (and thus, the next step will generate values for *age*), or equivalently he/she can increase *minOccurs* to one.

3 Type Checking

In this section, we define a type system for SPARQL. A type checking algorithm has been implemented for this type system which is based on consistency checking of an ontology. The rules of the type system are shown in Figure 5, and it works with triple patterns and equalities and inequalities of FILTER conditions of a SPARQL query. The type system is focused on two main elements: (1) signatures of properties and (2) category of variables. In (1) domains and ranges of properties are considered (those

(OP1) $s\ p\ o \vdash$	*o:D*, *s:E*	if $p \in op(O) \cup op(RDF) \cup op(RDFS) \cup op(OWL)$ and o is a variable % for each $D \in RangesOP(p)$, for each $E \in DomainsOP(p)$
(OP2) $s\ p\ o \vdash$	*fail*	if $p \in op(O) \cup op(RDF) \cup op(RDFS) \cup op(OWL)$ and o is a literal
(DP1) $s\ p\ o \vdash$	*o:D*, *s:E*	if $p \in dp(O) \cup dp(RDF) \cup dp(RDFS) \cup dp(OWL)$ and o is a variable % for each $D \in RangesDP(p)$, for each $E \in DomainsDP(p)$
(DP2) $s\ p\ o \vdash$	*s:D*	if $p \in dp(O) \cup dp(RDF) \cup dp(RDFS) \cup dp(OWL)$, o is a literal and $datatype(o) \in RangesDP(p)$ % for each $D \in DomainsDP(p)$
(DP3) $s\ p\ o \vdash$	*fail*	if o is a literal and $p \in dp(O) \cup dp(RDF) \cup dp(RDFS) \cup dp(OWL)$ and $datatype(o) \notin RangesDP(p)$
(VAR) $s\ p\ o \vdash$	*s,p:* **rdfs:Resource**	if p is a variable
(VOC) $s\ p\ o \vdash$	*fail*	if ns(t) is ns(O), rdf, rdfs or owl and name(t) $\notin$ voc(O) $\cup$ voc(RDF) $\cup$ voc(RDFS) $\cup$ voc(OWL), t is s, p or o
(FIL1) $l \diamond r \vdash$	*l:datatype(r)*	if r is a literal
(FIL2) $l \diamond r \vdash$	*l,r:* **rdfs:Literal**	if l and r are variables
(FIL3) $l \diamond r \vdash$	*fail*	if l and r are literals and datatype(l) $\neq$ datatype (r)

Figure 5: Type System for SPARQL

explicitly defined for object and data properties), and domains and ranges are used to type variables. In (2) *rdfs:Literal* and *rdfs:Resource* are used to type variables. Thus, assuming an ontology O, the type system infers type assertions $\varepsilon : \tau$ or *fail* where ε is a variable or an individual, and τ are datatypes (i.e., xsd:integer, xsd:string, etc.,), and classes of the ontologies O, *RDF*, *RDFS* and *OWL*. The rules analyses triple patterns $s\ p\ o$, as well as equality and inequality conditions $l \diamond r$, where $\diamond$ can be one of $=,!=,>,<,>=$ and $<=$. Let us remark that the type system is limited to equality and inequality conditions. A further extension of the type system to cover with SPARQL functions is considered as future work.

We denote by *op(O)* (respectively, *dp(O)*) the set of object (respectively, data) properties of an ontology O, and by *DomainsOP(p)* (respectively, *DomainsDP(p)*) the (explicitly declared) domains of an object (respectively, data) property p. Similarly, *RangesOP(p)* (respectively, *RangesDP(p)*) denotes the (explicitly declared) ranges of an object (respectively, data) property p. Let us remark that even though the type system works with explicitly declared domains (ranges), domains (ranges) inferred from explicitly declared domains (ranges) are later taken into account by the type checking algorithm since it used an OWL reasoner as inference engine. *voc(O)* denotes the vocabulary of an ontology O, and similarly *voc(RDF)* and *voc(OWL)* represent the vocabulary of RDF and OWL, respectively. *datatype(l)* denotes d, where $l = v\hat{}\hat{}d$ is a typed literal. Finally, *ns(i)* (respectively, *name(i)*) denotes the namespace (respectively, name) of i.

Rule *(OP1)* works with triple patterns in which the object is a variable and an object property. It covers the case *?x sn:likes ?y*, inferring *?x : User* as well as *?y : Activity*. It covers also the case of *?x rdf:type ?C*, inferring *?x : rdfs:Resource* and *?C : rdfs:Class*, and the case *sn:Antonio sn:likes ?y*, inferring *sn:Antonio : User* and *?y : Activity*. Rule *(OP2)* covers the case of an object property and literal, returning fail. Rule *(DP1)* covers the case of a data property and a variable in the object position. It covers the case *?x sn:age ?y*, inferring the type *?x : User* and *?y: xsd:integer*. Rule *(DP2)* considers the case of data property and a literal in the object position, whenever the type of the literal is a range of the data property. It covers the case *?x sn:date '2017-01-01T00:00:00Z'^^xsd:dateTime* inferring *?x : Activity*. Rule *(DP3)* is the opposite case of *(DP2)*: the type of the literal is not a range of the data property. It covers the case *?x sn:age '2017-01-01T00:00:00Z'^^xsd:dateTime*, returning fail. Rule *(VAR)* covers the case in which the property position is a variable. It covers the case *?x ?p ?y* inferring *?x: rdfs:Resource ?p: rdfs:Resource*. Let us remark that *?y* cannot be typed since the range of *?p* is unknown, and *?y* can be a resource or a literal. Rule *(VOC)* covers the case of wrong namespaces. For instance, *?x sn:wrong ?y*, *?x rdf:wrong ?y*, *?x rdfs:wrong ?y* and *owl:wrong ?y*, returning fail. Finally, rules *(FIL1)*, *(FIL2)* and

(FIL3) cover the case of equalities and inequalities. For instance, *?x = '10'xsd:integer*, *?x=?y* and *'2017-01-01T00:00:00Z'^^xsd:dateTime = '10'^^xsd:integer*, respectively. In the first case the type system infers *?x : xsd:integer*, in the second one *?x: rdfs:Literal* and *?y: rdfs:Literal*, and the third one fails.

The type checking algorithm proceeds as follows. The rules of Figure 5 are applied for each triple pattern and equality and inequality condition of the query. When the rules infer *fail* the SPARQL query is wrongly typed. Otherwise, it collects all the $\varepsilon : \tau$ assertions, and calls the ontology reasoner with the assertions. When $\phi : \tau$ has the form $?x : \tau$, it replaces the variable *?x* by a fresh individual called *:x* (distinct from the individuals of O). All the occurrences of the same variable are replaced by the same individual. Additionally, datatypes (i.e., xsd:integer, xsd:string, etc.,) are defined as subclasses of *rdfs:Literal*, and pairwise disjoint, and *rdfs:Literal* and *rdfs:Resource* are also declared as disjoint. Finally, all the classes of O are declared as subclasses of *rdfs:Resource*. In order to get a more accurate type checking, disjointness of O classes is required when it is appropiate. For instance, *Activity* and *User* should be declared as disjoint in the running example. The ontology reasoner (*HermiT* in our implementation) is used to check consistency of the assertions (class assertions) with the axioms of O. When the ontology reasoner detects an inconsistency the SPARQL query is wrongly typed in O, otherwise is well-typed in O.

For instance, let us suppose the triple patterns *?x sn:name ?y*, *?z sn:age ?y*, *?z sn:sent_by ?u* and the FILTER condition *?u = '10'^^xsd:integer*. The following assertions are inferred applying the rules of Figure 5: *?x : sn:User*, *?y : xsd:string*, *?z : sn:User*, *?y : xsd:integer*, *?z : sn:Message*, *?u : sn:User* and *?u : xsd:integer*. This set of assertions is inconsistent because *?y*, *?z* and *?u* have incompatible types whenever *User* and *Activity* are disjoint. Let us suppose now the case of *?x rdf:type sn:User*, *?y sn:age ?x*. In this case, the following assertions are deduced: *?x: rdfs:Resource*, *sn:User : rdfs:Class*, *?y : sn:User* and *?x : rdfs:Literal*. Thus, *?x: rdfs:Resource* and *?x : rdfs:Literal* are inconsistent because *rdfs:Resource* and *rdfs:Literal* are disjoint.

4 Property-based Testing of SPARQL

Now, we formally define the testing process of SPARQL queries. Given a **TBox** T, a test case (i.e., an **ABox**) t, where $T \cup \{t\}$ is consistent, and a SPARQL query Q, let $i_1, \ldots, i_n$ be individuals on the output of the query Q for the input T $\cup \{t\}$, then given a **TBox** T_0 containing a finite set of concepts $C_1, \ldots, C_m$ we say that Q passes the test $C_1, \ldots, C_m$ in t if for every $j = 1, \ldots, n$, $k = 1, \ldots, m$ then $C_k(i_j)$ is satisfied in $T \cup T_0 \cup \{t\}$, where $C_k(i_j)$ denotes i_j belongs to C_k. Analogously, we say that Q fails the test $C_1, \ldots, C_m$ in t, if for some $j \in \{1, \ldots, n\}$, $k \in \{1, \ldots, m\}$, $C_k(i_j)$ is not satisfied in $T \cup T_0 \cup \{t\}$. There exists a special case in which the set $\{i_1, \ldots, i_n\}$ is empty, and thus, we cannot say anything about Q in t. Also in the case $T \cup \{t\}$ is not consistent, t is not useful to test the query. Our tool is able to check whether a query Q (a) passes or (b) fails $C_1, \ldots, C_m$ in a finite set of test cases $t_1, \ldots, t_l$. Additionally, the tool is able to check (c) $T \cup \{t_i\}$ is inconsistent for every $i = 1, \ldots, l$ and (d) $i_1, \ldots, i_n$, for every t_i, is empty. In case (a), the tool reports the number n of used test cases. In case (b) (i.e., failing the test), the tool reports counterexamples, that is, the subset of $t_1, \ldots, t_l$ for which some of $C_1, \ldots, C_m$ is false. In case (c) the tool reports *"Unable to test the property. It was not possible to find consistent tests."*, and in the case (d) the tool reports *"Unable to check the property. It was not possible to find non trivial tests."*

```
<xs:element name="rdf:RDF">
<xs:element name="sn:Event" minOccurs="1" maxOccurs="unbounded">
<xs:sequence>
<xs:element name="sn:date" type="dateType" minOccurs="1" maxOccurs="1">
<xs:element name="sn:added_by" minOccurs="1" maxOccurs="1">
</xs:sequence>
</xs:element>
<xs:element name="sn:User" minOccurs="2" maxOccurs="unbounded">
<xs:sequence>
<xs:element name="sn:age" type="ageType" minOccurs="1" maxOccurs="1">
<xs:element name="sn:friend_of" minOccurs="1" maxOccurs="unbounded">
</xs:sequence>
</xs:element>
</xs:element>
```

Figure 6: XML Schema of Example 3

5 Examples

Now, we show how property-based testing works on examples. The examples are the same as shown in Figure 2.

Example (1) The first example is a *"type 2 bug"* in which type checking is carried out. Applying the rules of Figure 5, the following assertions are inferred: *?event : sn:Activity*, *?user1 : sn:User*, *?event : sn:User*, *?user2 : sn:Activity*, *?user2: sn:User* and *?event: sn:Event*, according to Figure 1. Now, the testing tool calls to the ontology reasoner with the inferred assertions and the ontology of Figure 1, and reports the following diagnosis:

```
Test cases cannot be generated:
DisjointClasses(#Activity #User)
ClassAssertion(#Activity #event)
ClassAssertion(#User #event)
```

It means that *?event* has type incompatible types *Activity* and *User*, and thus the query is wrongly typed.

Example (2) This example shows *"type 2 bug"* in which **ABox** instances with non empty answers cannot be found without violating ontology constraints. This is due to the irreflexivity of *replied_by*. In this case, the testing tool tries to generate test cases but it reports the following message:

```
Unable to test the property.
It was not possible to find consistent tests.
```

Example (3). In this example, the query is well-typed and does not violate ontology constraints. Let us suppose that the testing tool is called with the following class assertion *Mature(?user1)* on variable *?user1*, where *Mature* is a complex concept defined as follows: $Mature \equiv age\ some\ integer[>= 40]$.

Now, the selection of the XML Schema is crucial to generate useful test cases. Firstly, *minOccurs* of *added_by*, *date*, *friend_of* and *age* should be set to one. Additionally, at least one event and two users should be generated, and thus *minOccurs* should also be set for them to one and two, respectively. Other classes and properties can be pruned from the XML Schema. With this choice, just 0 iterations of the testing tool. By increasing the number of iterations to one, two, etc., more users and events will be generated. *maxOccurs* should be set to one for *age* whenever *age* is a functional property, and the same can be said for *added_by* and *date*. It avoids the generation of trivially inconsistent test cases, but it is not actually required. Also increasing the number of iterations, more *friend_of* relationships will be generated. The selected schema for this query is shown in Figure 6.

```
<xs:simpleType name="EventType">
<xs:restriction base="xs:string">
<xs:enumeration value="#tennis"/>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name="UserType">
<xs:restriction base="xs:string">
<xs:enumeration value="#luis"/>
<xs:enumeration value="#jesus"/>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name="dateType">
<xs:restriction base="xs:dateTime">
<xs:enumeration value="2016-01-01T00:00:00Z"/>
<xs:enumeration value="2018-01-01T00:00:00Z"/>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name="ageType">
<xs:restriction base="xs:integer">
<xs:enumeration value="30"/>
<xs:enumeration value="50"/>
</xs:restriction>
</xs:simpleType>
```

Figure 7: Values for Example 3

Additionally, values for individuals and datatypes have to be given. With this aim, the XML Schema *simpleType* definitions are modified as shown in Figure 7, including the values *tennis* for events, *luis* and *jesus* for users, 30 and 50 for ages, and 2016-01-01 and 2018-01-01 for dates. This selection is made according to the query: at least two users, one event, and a couple of values for ages and dates are needed in order to test the FILTER condition. The testing tool reports the following message:

```
 Output Property Falsifiable after 256 tests.
 ---------------
 Counterexample:
 ---------------
<rdf:RDF >
  <sn:Event  rdf:about="#tennis">
    <sn:date rdf:datatype="#dateTime">2016-01-01T00:00:00Z</sn:date>
    <sn:added_by rdf:resource="#jesus"/>
  </sn:Event>
  <sn:User  rdf:about="#luis">
    <sn:age rdf:datatype="#integer">50</sn:age>
    <sn:friend_of rdf:resource="#jesus"/>
  </sn:User>
  <sn:User  rdf:about="#jesus">
    <sn:age rdf:datatype="#integer">30</sn:age>
    <sn:friend_of rdf:resource="#luis"/>
  </sn:User>
</rdf:RDF>
```

It means that after 256 test cases, the testing tool has found a counterexample for *Mature(?user1)*. Thus the testing tool reveals a *"type 1 bug"*. The counterexample shows an event *tennis* which has been added by *jesus*, *luis* is a friend of *jesus* of 50 years old, and *jesus* is 30 year old. Thus, *?user1* which is bound to *jesus* is not *Mature*. This counterexample serves as witness of the bug, which can be found in *?user2 sn:age ?age*. Replacing this triple pattern by *?user1 sn:age ?age*, the testing tool answers as follows:

```
Ok: passed 256 tests.
```

In the case of the XML Schema has not been correctly selected, for instance, setting *age* to zero, and requiring a number of iterations equal to zero, the testing tool answers as follows:

```
Unable to test the property.
It was not possible to find non trivial tests.
```

Here, instead of inconsistent tests (like in Example (2)), the testing tool is not able to find non trivial test cases, that is, the result of the query is empty for all the generated test cases.

Example (4). This example shows that even though the XML Schema is correctly selected the testing tool can report:

```
Unable to test the property.
It was not possible to find non trivial tests.
```

This is due to incompatibility of the FILTER condition with the knowledge base, because *sn:athleticswch* has a date on the ontology later than *2017-01-01*. Let us remark that test cases, in this example, have to include the assertions of *sn:athleticswch*.

Example (5). In this example, the FILTER condition is compatible with the knowledge base, but *sn:antonio* is 40 years old on the ontology, and when the property to be tested on *?user* is $Young \equiv age\ some\ integer[< 40]$, the testing tool reports the following message:

```
Output Property Falsifiable after 18 tests.
 ---------------
 Counterexample:
 ---------------
<rdf:RDF>
  <sn:User rdf:about="#luis">
    <sn:age rdf:datatype="#integer">50</sn:age>
    <sn:name>luis</sn:name>
  </sn:User>
</rdf:RDF>
```

which means that the *?user* variable is not *Young*. Let us remark that test cases, in this example, have to include the assertions of *sn:antonio*.

Example (6). Finally, the testing tool on this query, with regard to the property $Popular\_event \equiv Event\ and\ (confirmed\_by\ some\ User)$ on *?event* reports the following message:

```
 Output Property Falsifiable after 32 tests.
 ---------------
 Counterexample:
 ---------------
<rdf:RDF>
  <socialnetwork:Event  rdf:about="#beach">
    <socialnetwork:created_by rdf:resource="#antonio"/>
  </socialnetwork:Event>
  <socialnetwork:User rdf:about="#antonio">
    <socialnetwork:likes rdf:resource="#tennis"/>
    <socialnetwork:attends_to rdf:resource="#tennis"/>
  </socialnetwork:User>
</rdf:RDF>
```

showing a counterexample of an event, *beach*, created by *antonio*, but *antonio* likes *tennis* and attends to *tennis*, so *?event* cannot be considered a *Popular_event*.

6 Benchmarks, Evaluation and Web Tool

Now, some benchmarks with the testing tool are shown. We have analyzed the time (in milliseconds) and number of test cases for each example of Section 5, increasing the number of iterations: zero and

Query	Test Cases	Steps	Answer	Time
Example 1	_	_	Wrongly Typed	1,655
Example 2	0	0	Unable	2,975
Example 2	0	1	Unable	15,973
Example 3 (Mature(?user1))	512	0	Falsifiable	8,355
Example 3 (Mature(?user1))	512	1	Falsifiable	9,378
Example 3 (This_Year(?event))	512	0	Passed	9,039
Example 3 (This_Year(?event))	10,752	1	Passed	90,101
Example 4	0	0	Unable	1,643
Example 4	0	1	Unable	2,330
Example 5	18	0	Falsifiable	2,385
Example 5	18	1	Falsifiable	2,616
Example 6	32	0	Falsifiable	2,288
Example 6	32	1	Falsifiable	3,439

Table 1: Benchmarks of the Testing Tool: Iteration Steps

one. Additionally we have analyzed the time (in milliseconds) for each example of Section 5, increasing *MinOccurs* values for classes and properties (thus, increasing also the number of test cases). Benchmarks have been made with a 1.7HHZ Intel Core i7 machine under MAC OS 10.9.5, with 8GB 1600 MHz DDR3 of RAM memory.

As we can see in Table 1, in Example 3 (for property *Mature(?user1)*) when the number of steps (i.e., iterations) increases, the number of test cases increases as well as the execution time. This happens only when the query passes the test or the tool is unable to find test cases. When the query fails the test and a counterexample is found, the testing tool stops (see Example 3 (for property *Mature(?user1)*), 5 and 6). The best strategy is to test the query with zero steps, and in the case the testing succeeds, to test the query with more steps in order to get better confidence. When the testing fails, increasing number of steps does not have consequences in execution time. Execution times range from 1 seconds to around 1 minute and a half for ten thousand test cases. The execution time depends on the selected classes and properties (i.e., *minOccurs* values) as well as the number of individual and datatype values for each one (i.e., *simpleType* enumeration), as the Table 2 shows (the number of individuals and datatypes is the same as the *minOccurs* values in each case). For instance, Example 3 uses four properties: *added_by*, *date*, *age* and *friend_of*, as well as two classes: *Event* and *User*, while Example 2 only uses one class: *Message* and two properties: *sent_by* and *replied_by*. Thus Example 3 testing takes more time than Example 2 testing. Increasing the number of individuals in Example 3 (i.e., two events and two users) as well as two instances of *friend_of* for each user, the time required is considerably greater. In Examples 5 and 6, even though the result is "Falsifiable", when the values of *minOccurs* increase, the answering time is greater because the test case generator builds bigger test cases even though smaller test cases can be enough to refute the constraint. Thus, the best strategy is to test the query with small *minOccurs* values, in order to get falsifiable constraints as soon as possible, and when testing successes to increase *minOccurs* values in order to get better confidence. Execution times range from 2 seconds to around 6 minutes.

The question now is what are the limitations of the testing tool? Firstly, PBT reveals the presence of bugs but it cannot ensure their absence. PBT is a technique for detecting errors based on the generation of a finite number of test cases. A bigger set of test cases provides more confidence, but it cannot ensure the absence of errors. Additionally, since the set of test cases is generated from the XML Schema, a wrong selection of the XML Schema can lead to insufficient test cases. The human tester can play with the XML Schema in order to get better results of testing. Also increasing the number of iterations, the confidence is improved. Secondly, type checking is used to detect wrongly typed queries. Type checking is limited to domain and range of properties and categories of variables. Ontology reasoning is used to detect wrongly typed queries, using consistence checking on type assertions. The procedure of type checking does not depend on a set of test cases. The type checking procedure could be extended to

Query	Item	MinOccurs	Time
Example 1	Event	1	
	User	1	
	Created_by	1	
	Likes	1	
	Invited_to	1	1,655
Example 1	Event	2	
	User	2	
	Created_by	1	
	Likes	1	
	Invited_to	1	1,451
Example 2	Message	2	
	Sent_by	1	
	Replied_by	1	2,975
Example 2	Message	2	
	Sent_by	1	
	Replied_by	2	6,610
Example 3	Event	1	
	User	2	
	Date	1	
	Added_by	1	
	Age	1	
	Friend_of	1	5,113
Example 3	Event	2	
	User	2	
	Date	1	
	Added_by	1	
	Age	1	
	Friend_of	2	154,509
Example 4	Event	1	
	Date	1	1,643
Example 4	Event	2	
	Date	1	1,944
Example 4	Event	3	
	Date	1	5,038

Query	Item	Minoccurs	Time
Example 5	User	1	
	Age	1	
	Name	1	2,385
Example 5	User	2	
	Age	1	
	Name	1	11,843
Example 5	User	3	
	Age	1	
	Name	1	191,480
Example 6	Event	1	
	User	1	
	Created_by	1	
	Likes	1	
	Attends_to	1	2,288
Example 6	Event	2	
	User	1	
	Created_by	1	
	Likes	1	
	Attends_to	1	2,953
Example 6	Event	1	
	User	2	
	Created_by	1	
	Likes	1	
	Attends_to	1	2,502
Example 6	Event	2	
	User	2	
	Created_by	1	
	Likes	1	
	Attends_to	2	113,732
Example 6	Event	2	
	User	2	
	Created_by	1	
	Likes	2	
	Attends_to	2	383,405

Table 2: Benchmarks of the Testing Tool: MinOccurs

detect some cases covered by the PBT procedure. For instance, adopting the same mechanism of type checking, in Example 2, *?msg1 sn:replied_by ?msg1* can be sent to the ontology reasoner with a fresh individual *:msg1*, that is, adding the following object property assertion *:msg1 sn:replied_by :msg1* to the ontology. Thus, object property assertions can be also checked by the ontology reasoner. The ontology reasoner would reveal that the individual *:msg1* violates the constraint irreflexivity on the property *replied_by*. However, this mechanism cannot be used in other cases. For instance, *?event sn:date ?date*, *sn:athleticswch sn:date ?date2*, *?date >= ?date2* and *?date < '2017-01-01T00:00:00Z"^^xsd:dateTime*. It is due to the limitations of the ontology reasoner which cannot work with *?date >= ?date2*. In general terms, the testing tool is limited to the reasoning capabilities of the ontology reasoner.

Finally, the developed Web tool is available at: `http://minerva.ual.es:8080/SPARQL/` enabling (1) the transformation of an ontology to an XML Schema (2) facilitating the customization of XML Schemas by automatically pruning the XML Schemas from the selection classes and properties by the user and (3) allowing to set *minOccurs* and *maxOccurs* values as well as adding individuals and values by the user. The tool also permits on-line editing and testing of SPARQL queries, showing on-line type-checking and testing results: *"Ok: passed"* or *"Output Property Falsifiable"* as well as the number of test cases, and counterexamples when the query fails the test. The Web tool also incorporates the examples shown in the paper, as well as the used XML Schemas.

7 Conclusions and Future Work

A limitation of our approach is that constraints on output variables can be only specified with memberships to concepts. This is a limitation of the current implementation but we plan to extend constraint definition on classes (for instance, disjointness of output classes) and properties (for instance, sub-property relationships for output properties). A drawback of our tool is that in some cases too many inconsistent test cases are generated. Some constraints expressed in the input ontology are not explicitly used by the test case generator, but inconsistent test cases are later discarded using the ontology reasoner. In our experience using the tool, generation of inconsistent test cases has no dramatic consequences of performance, but we will study how to improve it. With regard to SPARQL coverage, the testing tool is able to test any SELECT query. ASK, DESCRIBE and CONSTRUCT queries are out of the scope of the testing tool. Finally, our testing tool generates test cases without taking into account the code, and the human tester's intervention is required. We plan to extend our work to white box testing which means to automatically generate/prune the XML Schema from the code. It would produce a completely automatic testing tool without human tester's intervention.

References

[1] Jesús M. Almendros-Jiménez & Antonio Becerra-Terón (2015): *XQuery Testing from XML Schema Based Random Test Cases*. In: *Database and Expert Systems Applications,DEXA'2015*, LNCS 9262, Springer, pp. 268–282.

[2] Jesús M. Almendros-Jiménez & Antonio Becerra-Terón (2016): *Automatic Generation of Ecore Models for Testing ATL Transformations*. In: *Model and Data Engineering, MEDI'2016*, LNCS 9893, Springer, pp. 16–30.

[3] Jesús M Almendros-Jiménez & Antonio Becerra-Terón (2017): *Automatic property-based testing and path validation of XQuery programs*. *Software Testing, Verification and Reliability* 27(1-2).

[4] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn et al. (2013): *An orchestrated survey of methodologies for automated software test case generation*. *Journal of Systems and Software* 86(8), pp. 1978–2001.

[5] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George HL Fletcher, Aurélien Lemay & Nicky Advokaat (2017): *gMark: Schema-driven Generation of Graphs and Queries*. *IEEE Transactions on Knowledge and Data Engineering* 29(4), pp. 856–869.

[6] Antonia Bertolino, Jinghua Gao, Eda Marchetti & Andrea Polini (2007): *Systematic generation of XML instances to test complex software applications*. In: *Rapid Integration of Software Engineering Techniques*, Springer, pp. 114–129.

[7] Chandrasekhar Boyapati, Sarfraz Khurshid & Darko Marinov (2002): *Korat: Automated testing based on Java predicates*. In: *ACM SIGSOFT Software Engineering Notes*, 27, ACM, pp. 123–133.

[8] Stefan Brass & Christian Goldberg (2006): *Semantic errors in SQL queries: A quite complete list*. *Journal of Systems and Software* 79(5), pp. 630–644.

[9] Rafael Caballero, Yolanda García-Ruiz & Fernando Sáenz-Pérez (2010): *Applying constraint logic programming to SQL test case generation*. In: *International Symposium on Functional and Logic Programming*, Springer, pp. 191–206.

[10] David Chays, Yuetang Deng, Phyllis G Frankl, Saikat Dan, Filippos I Vokolos & Elaine J Weyuker (2004): *An AGENDA for testing relational database applications*. *Software Testing, verification and reliability* 14(1), pp. 17–44.

[11] Koen Claessen & John Hughes (2011): *QuickCheck: a lightweight tool for random testing of Haskell programs*. *ACM SIGPLAN notices* 46(4), pp. 53–64.

[12] Claudio De La Riva, María José Suárez-Cabal & Javier Tuya (2010): *Constraint-based test database generation for SQL queries*. In: *Proceedings of the 5th Workshop on Automation of Software Test*, ACM, pp. 67–74.

[13] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos & Zhe Wang (2014): *HermiT: an OWL 2 reasoner*. *Journal of Automated Reasoning* 53(3), pp. 245–269.

[14] Christian Grun (2016): *BaseX. The XML Database*. `http://basex.org`.

[15] Yuanbo Guo, Zhengxiang Pan & Jeff Heflin (2005): *LUBM: A benchmark for OWL knowledge base systems*. *Web Semantics: Science, Services and Agents on the World Wide Web* 3(2), pp. 158–182.

[16] Muhammad Akhter Javid & Suzanne M. Embury (2012): *Diagnosing faults in embedded queries in database applications*. In: *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, ACM, pp. 239–244.

[17] Frédéric Jouault, Freddy Allilaire, Jean Bézivin & Ivan Kurtev (2008): *ATL: A model transformation tool*. *Science of computer programming* 72(1), pp. 31–39.

[18] Gregory M Kapfhammer, Phil McMinn & Chris J Wright (2013): *Search-based testing of relational schema integrity constraints across multiple database management systems*. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, IEEE, pp. 31–40.

[19] Sarfraz Khurshid & Darko Marinov (2004): *TestEra: Specification-based testing of Java programs using SAT*. *Automated Software Engineering* 11(4), pp. 403–434.

[20] Dimitris Kontokostas, Patrick Westphal, Sören Auer, Sebastian Hellmann, Jens Lehmann & Roland Cornelissen (2014): *Databugger: a test-driven framework for debugging the web of data*. In: *Proceedings of the 23rd International Conference on World Wide Web*, ACM, pp. 115–118.

[21] Dimitris Kontokostas, Patrick Westphal, Sören Auer, Sebastian Hellmann, Jens Lehmann, Roland Cornelissen & Amrapali Zaveri (2014): *Test-driven evaluation of linked data quality*. In: *Proceedings of the 23rd international conference on World Wide Web*, ACM, pp. 747–758.

[22] Levi Lúcio, Bruno Barroca & Vasco Amaral (2010): *A technique for automatic validation of model transformations*. In: *Model Driven Engineering Languages and Systems*, Springer, pp. 136–150.

[23] Manolis Papadakis & Konstantinos Sagonas (2011): *A PropEr Integration of Types and Function Specifications with Property-Based Testing*. In: *Proceedings of the 2011 ACM SIGPLAN Erlang Workshop*, ACM Press, New York, NY, pp. 39–50.

[24] Michael Schmidt, Thomas Hornung, Georg Lausen & Christoph Pinkel (2009): *SP2Bench: a SPARQL performance benchmark*. In: *2009 IEEE 25th International Conference on Data Engineering*, IEEE, pp. 222–233.

[25] Mark Utting, Alexander Pretschner & Bruno Legeard (2012): *A taxonomy of model-based testing approaches*. *Software Testing, Verification and Reliability* 22(5), pp. 297–312.

[26] Priscilla Walmsley (2007): *XQuery – search across a variety of XML data*. O'Reilly. Available at `http://www.oreilly.com/catalog/9780596006341/index.html||http://www.oreilly.com/catalog/9780596006341/index.html`.

[27] Jian Zhang, Chen Xu & S-C Cheung (2001): *Automatic generation of database instances for white-box testing*. In: *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, IEEE, pp. 161–165.