Link: Exploiting the Web of Data to Generate Test Inputs

Leonardo Mariani[§] Mauro Pezzè[†§] Oliviero Riganelli[§] Mauro Santoro[§]

[§] Department of Informatics, Systems and Communications
University of Milano Bicocca - Milano, Italy
{mariani,pezze,riganelli,santoro}@disco.unimib.it

[†] Faculty of Informatics
University of Lugano - Lugano, Switzerland
mauro.pezze@usi.ch

ABSTRACT

Applications that process complex data, such as maps, personal data, book information, travel data, etc., are becoming extremely common. Testing such applications is hard, because they require realistic and coherent test inputs that are expensive to generate manually and difficult to synthesize automatically. So far the research on test case generation techniques has focused mostly on generating test sequences and synthetic test inputs, and has payed little attention to the generation of complex test inputs.

This paper presents Link, a technique to automatically generate test cases for applications that process complex data. The novel idea of Link is to exploit the Web of Data to generate test data that match the semantics of the related fields, and satisfy the semantic constraints that arise among interrelated fields. Link automatically analyzes the GUI of the application under test, generates a model of the required inputs, queries DBPedia to extract the data that can be used in the tests, and uses the extracted data to generate complex system test inputs.

The experimental results show that Link can generate realistic and coherent test inputs that can exercise behaviors difficult to exercise with currently available techniques.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools (e.g., data generators, coverage testing)*

General Terms

Verification

Keywords

System testing, realistic test input, Web of data.

1. INTRODUCTION

Many applications require realistic and coherent data to be thoroughly tested. *Realistic data* are semantically meaningful data that correspond to precise sets of elements, such as the names of Swiss cities, the names of car manufacturers and the symbols of chemical elements. *Coherent data* are data composed of multiple semantically correlated fields, such as the fields that compose a full address that includes a city name, a corresponding zip code and a street name compatible with the city and the zip code. Examples of applications that require realistic and coherent data range from services available on the Web, for instance, services for booking or searching, to desktop applications, for instance, applications to archive and classify books, to apps that interact with the environment, for instance, maps and shopping apps.

Manually generating a thorough test suite for a GUI with many semantically interrelated fields can be very expensive because the test designers shall find semantically correct and coherent data that cover the many aspects of the GUI. Generating test cases automatically may largely reduce the effort required to generate test cases. Unfortunately, so far research in automatic generation of system-level test cases has focused mostly on deriving test sequences, with approaches based on models [36, 37], learning [34], search-based algorithms [29], and reuse [28, 24]. These strategies do not tackle the problem of generating realistic and coherent test inputs, and can hardly cope with applications that extensively exploit the semantic of the input data.

Recently Bozkurt and Harman have proposed an approach to generate realistic and complex test data based on Web service composition [17], while McMinn, Shabaz and Stevenson have defined an approach to generate complex data based on the integration with Web searches [35, 42].

The approach by Bozkurt and Harman generates complex data by searching for Web services that produce the required type of data, and uses other Web services to feed the inputs of the selected Web service. This approach produces useful results when it can discover the right composition of Web services, but in several cases, a perfect composition might not exist and significant manual effort could be necessary to activate the chain of dependencies among Web services. Moreover, the early results show that this composition works well for simple inputs, mostly single field data, but is much less effective when dealing with complex inputs [17].

The approaches based on Web searches extract keywords from the artifact under test and exploit these keywords to search the Web. The text in the pages returned by the search is parsed and filtered to be used as test inputs [35]. The approach has a low precision and is well suited for single

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07...$15.00
http://dx.doi.org/10.1145/2610384.2610397

field data, but cannot address the frequent case of forms with multiple correlated input fields.

In this paper, we present *Link*, a new automated approach to generate realistic and coherent data that can produce a large volume of complex test inputs composed of multiple semantically correlated fields. In this way, the approach overcomes the limitations of both manual generation and of the current automatic tools. The novel idea of Link is to take advantage of the largest structured source of information available on the Web: the Web of Data [14]. The Web of Data is a freely available global data space containing billions of interconnected statements, usually represented with RDF triples [18]. The interconnections between statements capture relevant semantic relations. This space can be accessed, queried and browsed using different endpoints and protocols [16]. Link analyzes the graphical interface of an application, produces a model that captures the semantic of the data required to test the application, and automatically extracts the test inputs necessary to feed the interface using the DBPedia SPARQL (Sparql Protocol And Rdf Query Language) endpoint [46]. The generated test inputs have the following unique characteristics:

- they are *syntactically correct*, that is, the inputs are legal according to the syntax required by the fields,
- they are *semantically valid*, that is, the inputs are realistic and meaningful according to the semantic of the fields,
- they are *semantically coherent*, that is, the inputs are mutually meaningful, sound and consistent when considered as a whole.

While the existing approaches can generate inputs that are syntactically correct and, to some extent, semantically valid, Link is the only technique that can automatically generate a relevant amount of data that satisfy all the three key characteristics outlined above, since none of the existing techniques can generate test inputs that are semantically coherent.

The paper is organized as follows. Section 2 provides some background information about the Web of data and DBPedia. Section 3 overviews the Link approach. Sections 4 and 5 describe how Link associates GUI labels with RDF triples and how Link builds a model that represents the test inputs required by the application, respectively. Section 6 describes the process to extract relevant data from the Web of Data, and to turn the data into test inputs. Section 7 presents the empirical results obtained applying Link to a number of applications. Section 8 discusses related work. Section 9 summarizes contributions and provides final remarks.

2. THE WEB OF DATA

The Web is quickly evolving into a platform for data integration and information management. This evolution is led by the Web of Data, a huge data space of interlinked data that can be accessed and populated through various protocols and techniques. The creation and evolution of the Web of Data is driven by a set of best practices for publishing and connecting structured data on the Web, known as the Linked Data principle [14].

The Linked Data principle specifies how to define and publish machine-readable typed links between arbitrary items in the world. The items are identified with Uniform Resource Identifiers (URIs) and the links are represented with the Resource Description Framework (RDF) language. An URI is a compact sequence of characters that identifies either an abstract or physical resource [13]. For instance, the URI `http://dbpedia.org/resource/Berlin` identifies the city of Berlin.

RDF provides a way of encoding typed statements in the form of triples ⟨subject, predicate, object⟩ [33]. The subject is a URI, the object is either a URI or a literal, and the predicate is a URI that specifies a relation among the subject and the object. For instance, the RDF triple ⟨`<http://dbpedia.org/resource/Berlin>` `<http://dbpedia.org/ontology/country>` `<http://dbpedia.org/resource/Germany>`⟩ indicates that the city of Berlin is part of Germany.

Everybody can publish new data sources including statements and links to statements published by others. We generically refer to such data source as a *Knowledge Base*. In few years the application of the Linked Data principle led to a huge amount of typed and interlinked statements available and freely accessible on the Web. In 2011, the Web of Data included about 31.5 billion of RDF triples, with about 500 million of RDF links connecting different data sources [15].

DBpedia, the knowledge base that we used in our experiments, is one of the largest knowledge bases available on the Web. DBpedia is obtained by extracting structured information from Wikipedia, and turning this information into an accessible form that includes entities associated with URIs, rich RDF descriptions of each entity, classification of entities into hierarchies, and links among entities [16]. DBpedia currently includes more than one billion RDF triples.

Knowledge bases such as DBpedia can be conveniently accessed online through SPARQL endpoints, which are interfaces that support the execution of SPARQL queries. SPARQL is a query language for data represented as RDF triples [46]. SPARQL supports select statements that can return results in various forms, such as tables, RDF triples, and RDF graphs (an RDF graph is a graph where nodes represent subjects and objects of RDF triples, and edges represent predicates). For instance, the following SPARQL query extracts all the cities of Germany from DBpedia:

```
SELECT ?subject WHERE {
 ?subject <http://www.w3.org/2000/01/rdf-schema#type>
          <http://dbpedia.org/ontology/City>.
 ?subject <http://dbpedia.org/ontology/country>
          <http://dbpedia.org/resource/Germany>.
}
```

Link interacts with DBPedia only using SPARQL endpoints, and thus can extract data from other knowledge bases that implement a SPARQL endpoint, i.e., almost all the non-trivial knowledge bases currently available.

3. AUTOMATIC TESTING WITH SEMANTICALLY CORRELATED INPUTS

Link aims to automatically generate tests for complex inputs that include semantically interrelated values. This is the case for example of an application that searches for books given an author, a title of a book written by that author and the isbn number corresponding to that book. A valid test case is composed of three semantically related values, like $\langle Umberto\ Eco, The\ name\ of\ the\ rose, 978-0156001311\rangle$, which are difficult to generate without proper semantic information. Link succeeds in generating valid test cases by exploting the Web of Data. As shown in Figure 1 Link works in two main stages, *generate the model* and *test the application*.

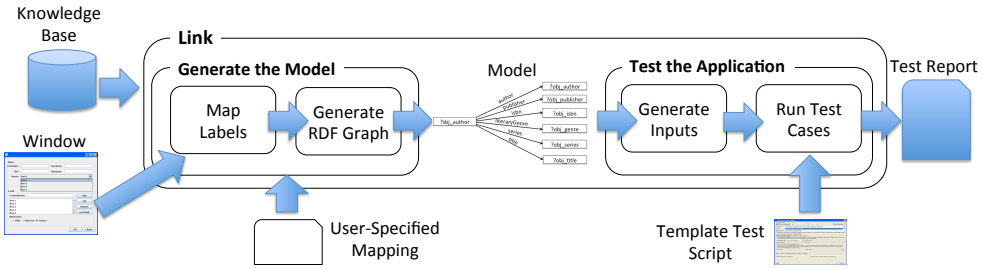

Figure 1: The Link approach.

In the first stage (*Generate the Model*), Link examines the fields in the window under test and uses the Web of Data (Knowledge Base) to generate a model of the input values. The model represents the semantics of the inputs required by the windows. Link builds the model in two phases: It first maps labels that characterize the input values in the windows under test with types in the Web of Data (*Map Labels*), and then clusters the values in a graph that represent the semantic relations among the values (*Generate RDF Graph)*.

In the Map labels phase, Link parses the windows under test and extracts the labels associated with the input widgets that must be filled in, looks for classes and predicates corresponding to the extracted labels in the knowledge base, and returns a set of RDF triples that represent the semantics of the single input fields of the application.

In the Generate RDF graph phase Link discovers correlations among RDF triples and produces an RDF graph [33] that captures the semantics of the inputs required by the window under test. The RDF graph produced in this phase captures both the semantics of the single input fields and the semantics of the correlations among fields.

In this stage, the test designers can provide additional information to improve the precision of the mapping. The additional information is provided in the form of a User-Specified Mapping and may for example add information to generic labels, as we will see in the next sections. This information is the only manual activity required from the test designers and, although useful, is not required.

In the second stage (*Test the Application*), Link generates and executes the test inputs. It first instantiates the relations among input values in concrete inputs (*Generate Inputs*) and then instantiates the test script with the generated concrete inputs and executes it (*Run Test Cases*).

Link generates inputs by querying the knowledge base and extracting a set of values that both satisfies all the relations represented in the model and are semantically different from the previously extracted values. A single test input consists of multiple data values, one for each input widget, that are both realistic and semantically correlated according to the relations represented in the model.

Link enters the inputs into the application by running a template test script, which consists of a set of statements that bring the windows under test in the target state, enter the input data generated in the previous phase, execute the functionality, check the result, close the application, and reset the state, if necessary. Test scripts can be conveniently recorded with capture and replay tools. In our experiments we used IBM Rational Functional tester [31] and Robotium [9].

In the next sections we illustrate the phases of the technique using a realistic running example. We consider the generation of test inputs for an application that can search for books given the following fields: the `author`, the `publisher`, the `isbn`, the `genre`, the `serial` and the `title`.

4. MAP LABELS

The Map Labels phase produces a set of RDF triples that capture the semantic of the inputs in the window under test in three steps: *identify descriptors*, *map descriptors to classes and predicates*, and *discover alternative mappings*. The *identify descriptors* step identifies the GUI labels associated with the input fields. The *map descriptors to classes and predicates* step checks if the descriptors occur as either predicate or class names in the knowledge base. The *discover alternative mappings* step finds a mapping for the descriptors that do not occur in the knowledge base as is, by exploiting various strategies.

If the test designer has provided a predefined mapping between some of the labels in the GUI and the concepts in the knowledge base, Link uses this mapping for those labels. Thus the analysis steps described below apply to the labels that do not occur in the user-specified mapping.

4.1 Identify Descriptors

For each widget Link identifies the corresponding textual descriptor in the GUI, which is usually a label that describes the data that can be entered in the input widget.

Link identifies the descriptor associated with each widget by looking for the label closest to the input widget. Link takes into account the best practices in GUI design, such as the position that the labels usually have with respect to the input widgets and the hierarchical organization of the GUI elements. Link has been demonstrated to produce the right associations with high precision and recall, as reported in details in a former paper, where we present the approach to locate labels in a GUI [12].

Link implements simple cleaning strategies to remove the noise that might occur in labels, such as removing special characters and separate words when labels are expressed by means of the Camel notation.

In the running example, the descriptors associated with the input widgets are `author`, `publisher`, `isbn`, `genre`, `serial`, and `title`.

4.2 Map Descriptors to Classes and Predicates

In the second step, Link maps the retrieved descriptors to the data in the knowledge base. Link creates this mapping with a SPARQL query that looks for predicates and classes whose names match the descriptors. Since predicates are

used more often than classes to represent attributes, Link tries first to map a descriptor to a predicate, and then to a class only if it does not find a matching predicate.

Using either a predicate or a class name makes sense only if it is used frequently enough in the knowledge base. Link finalizes a mapping only if the selected predicate or class occurs more than in a given number of RDF triples, where the threshold is a parameter that can be fixed depending on the knowledge base. In our experiments with DBPedia, we set the threshold to 100.

The search for a suitable mapping of the descriptors can be restricted to the namespaces that correspond to the best structured and organized ontologies, among the many namespaces supported by a knowledge base. In the case of DBPedia we initially limit the search to predicates and classes defined in the namespace dbpedia-owl.

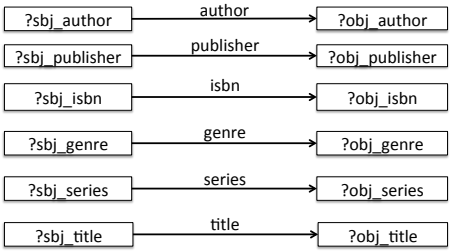

Figure 2: The RDF triples extracted after the first phase for the running example.

4.3 Discover Alternative Mappings

If Link can find neither a predicate nor a class that matches a descriptor, it attempts to work around this case by iteratively considering synonyms, alternative namespaces and synonyms with alternative namespaces.

Synonyms. Link searches for synonyms using WordNet, a freely available online lexical database with more than 118K terms classified [25]. WordNet groups terms in sets of cognitive synonyms, called synsets. Each synonym in a synset has a sensenumber, which is a positive integer that indicates how well that synonym can be used to express the meaning of the synset. A sensenumber equal to 1 indicates a perfect matching. The greater the sensenumber is the less precise the matching is. Given a word, the WordNet Web service can automatically find its synsets and the synonyms in the synsets.

Link uses WordNet to retrieve a set of possible synonyms for a given descriptor by selecting the words with the best sensenumber from each synset. Link searches the synonyms in the knowledge base and computes their support, where the support is the number of instances (i.e., RDF triples) that use the predicate or the class. Link uses the synonym with the best support to map the descriptor.

In the running example, the descriptor `serial` cannot be mapped with any predicate or class in DBPedia. However using WordNet, Link automatically extracts the two synonyms `series` and `serialPublication`, which are the best representatives of their synsets. Since `series` has a support equal to 19.482 when used as a predicate and `serialPublication` has a support equal to 0, Link successfully replaces `serial` with `series`.

Alternative Namespaces. When Link fails to map a descriptor even using synonyms, it extends the search of a mapping for the descriptor to all the namespaces supported by the knowledge base. For example, DBPedia uses terms from more than 200 namespaces. Some of these namespaces are quite general, but several are domain specific, such as the audio, geo and radio ontologies, and can be extremely useful for mapping descriptors of domain-specific applications.

Synonyms with Alternative Namespaces. When Link fails with synonyms and namespaces, it combines the two strategies and tries to map the synonyms of the descriptor with the predicates and the classes that occur in every namespace used in the target knowledge base.

The descriptors that are not mapped to the knowledge base with any of the above strategies are ignored and no test inputs are generated for the corresponding fields.

This phase produces a set of RDF triples with classes and predicates corresponding to the input widgets to be used for generating test cases. Figure 2 shows the resulting RDF triples for the running example. The question marks indicate free variables. We conventionally use ?sbj_* for variables that represent the subjects of the relations, and ?obj_* for variables that represent the objects of the relations. In the graph we display labels instead of URIs, and removed the name of the namespaces to improve readability. The model produced in this phase indicates that there exist resources in the knowledge base (the subjects of the RDF triples) that have specific attributes (represented with predicates), but does not give information about their relations yet.

In a nutshell, the figure indicates the predicates that occur both in the windows under test and in DBPedia (author, publisher, etc). The variables that occur in the left and right side of the triples indicate that we are not interested in the specific elements that the predicates connect in DBPedia.

5. GENERATE RDF GRAPH

The set of triples generated in the previous phase represents the initial (disconnected) RDF graph. In this phase, Link refines the RDF graph to capture the semantics of both the inputs and the relations among fields. It starts from the initial (disconnected) RDF graph and queries DBPedia to identify and add new relations among predicates.

This phase is articulated in two steps. The first step, *Initialize the RDF Graph*, produces a model that represents the semantic relations among the inputs. Sometime the model produced in this step might be unsuitable for extracting test data from the selected knowledge base. When this happens, the second step, *Refine the Model*, modifies the model until it creates a version suitable for generating test data.

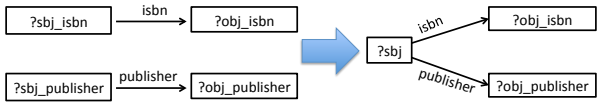

Figure 3: A simplified transformation of the RDF graph to include relations between two predicates.

5.1 Initialize the RDF Graph

This step finds the relations between the RDF triples retrieved in the previous phase (i.e., the relations between the input fields of the windows under test) and augments the RDF graph with these relations. Link systematically

searches for relations between the classes and predicates extracted in the previous phase using the knowledge base.

We first present the three classes of relations that Link can discover, and then show how Link can deal with the RDF triples that might remain isolated.

Relations Between Predicates. Two predicates are related if the selected knowledge base includes at least a resource that occurs as subject of both predicates. For instance, DBPedia includes resources, like `Books`, that occur as subject of both the predicates `isbn` and `publisher`. Thus the two predicates are semantically related.

Link discovers the relations between pairs of predicates by executing SPARQL queries that look for resources that occur as subject of pairs of predicates. If the SPARQL query returns one or more resources, the graph is modified according to the newly discovered information. For example, Figure 3 shows how the graph is transformed to represent the fact that predicates refer to a same subject: the two subjects in the left part of the graph in Figure 3 are merged into a single subject.

When several pairs of predicates are related, the model may include a single subject associated with multiple predicates. For instance, the model in Figure 4 shows the case of predicates `isbn`, `publisher` and `genre`. This model is a correct representation of the information in the knowledge base as long as the predicates are not only related pairwise but they are applicable all together at the same time on a same subject. Since the graph is built working on pairs of relations and not considering all the relations at once, it might be the case that there is no resource in the knowledge base that can occur as subject of all the predicates at the same time. This happens in the running example. In DBPedia there are resources that occur as subject of both `isbn` and `publisher`, and as subject of both `publisher` and `genre`, but there is no resource that can occur as subject of both `isbn` and `genre`. Thus the model in Figure 4 does not capture correctly the relation among the three predicates.

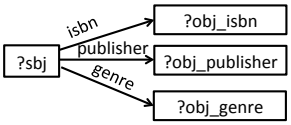

Figure 4: An example RDF graph with a subject sharing three predicates.

To produce a correct graph, when Link discovers relations between predicates it does not merge the subjects as shown in Figure 3, but keeps the subjects and adds new predicates that show that both predicates can be applied at the same time to both subjects, as shown in Figure 5.

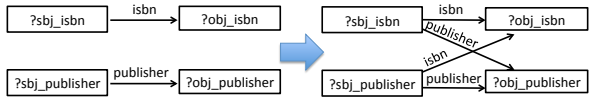

Figure 5: The transformation of the RDF graph to include relations between two predicates.

Considering that the subjects are free variables the graph in the right part of Figure 5 shows twice that there are resources that can be the subject of both the `isbn` and the `publisher` predicates. In a sense this graph represents the same information of the graph of Figure 4 with some redundancy. The benefit of using this transformation instead of the one shown in Figure 3 is clear when considering the case of the three labels `isbn`, `publisher` and `genre`. In fact the resulting RDF graph is the one shown in Figure 6 instead of the one shown in Figure 4.

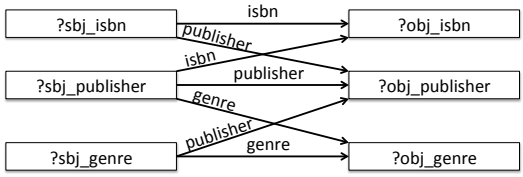

Figure 6: An excerpt of the initial RDF graph generated for the running example.

The RDF graph in Figure 6 carries important information that will be exploited by Link when refining the model. It is easy to see that the model in Figure 6 has three subjects, two of the subjects (`sbj_isbn` and `sbj_genre`) have two predicates while one of the subjects (`sbj_publisher`) has three predicates. Since in DBPedia there is no resource that can satisfy the three relations at the same time, the model refinement will be able to modify the part of the model related to the subject `sbj_publisher`, which is the problematic part of the graph, preserving the other relations between predicates. This cannot be done by simply referring to Figure 4 where there is a single subject and all the relations play the same role in the graph.

Relations Between Classes and Predicates. Link represents a label mapped to a class as an RDF triple where the subject is a free variable, the predicate is `<http://www.w3.org/2000/01/rdf-schema#type>`, which is the predicate that represents the type relation in RDF, and the object is the label (i.e., the name of the type). For instance if the label `song` is mapped to the class `Song`, Link adds the triple ⟨`?sbj_song`, `<http://www.w3.org/2000/01/rdf-schema#type>`, `Song`⟩ to the model. The semantics of the RDF triple is that there exist resources whose type is `Song` in the selected knowledge base.

Discovering relations between a predicate and a class is a special case of discovering relations between two predicates. Figure 7 shows the same transformation shown in Figure 5 applied to the case of a class `Song` and a predicate `artist`. The resulting graph shows that there are resources of type `Song` that have `artist` as an attribute.

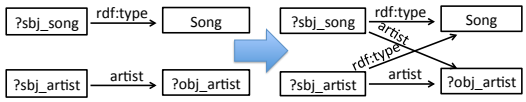

Figure 7: The transformation of the RDF graph to include relations among a class and a predicate.

Relations Between Classes. Given two classes (i.e., two RDF triples) whose predicate is `<http://www.w3.org/2000/01/rdf-schema#type>`, Link searches for relations between these two classes (i.e., it looks for predicates that relate one

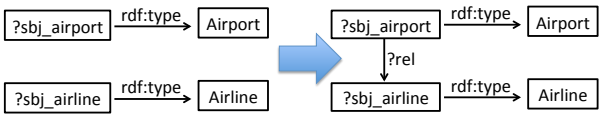

Figure 8: The transformation of the RDF graph to include relations between two classes.

subject to the other). If it finds at least a relation, it extends the model accordingly.

For instance, the left part of Figure 8 shows the case of two resources of type `Airport` and `Airline`. Since the knowledge base includes multiple relations that relate resources with these types, for instance, the `hub` relation that indicates that an airport can serve as a hub for a given airline, and the `operator` relation that indicates that a given airline can operate in a given airport, Link refines the graph adding these relations as shown in the right part of Figure 8.

The label on the newly added edge does not indicate a specific predicate but is a free variable (represented with the symbol ?rel). This is because there is no reason to bound the relation between the two resources to a specific predicate already in this phase, when there are multiple predicates that can be used to relate them. The model preserves its generality without constraining the value of the predicate, but simply representing the fact that the two resources can be directly related according to some predicates.

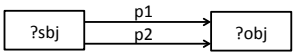

Figure 9: The template of a practical synonym.

Relating Unrelated Triples. Once Link has extended the RDF model with the relations discovered between RDF triples, there may still be some isolated triples, that is, tripes not related with any other one. The lack of relations may depend for example on imprecise mapping between terms in the GUI and concepts in the knowledge base, as in the case of the `title` predicate in the running example. Isolated RDF triples may represent a problem in generating test cases, because it may be difficult finding meaningful values for the related fields.

To identify useful relations for isolated RDF triples, we exploit the knowledge about the input domain discovered so far and coded in the form of relations between triples. For instance when dealing with the `title` predicate in the running example, Link has identified that the application needs to be tested with resources that relate to the predicates `author`, `publisher`, `isbn`, `genre`, `serial` and `title`. Link uses this information to modify the predicates in the isolated triples with semantically equivalent predicates that better relate with the model (i.e., better relate with the knowledge of the input domain discovered so far).

Link identifies the candidate synonyms of a predicate in an isolated triple by first applying the strategies described in Section 4.3 (synonyms, alternative namespaces, and the combination of the two), and then using a new strategy that we call discovery of the *practical synonyms* and that can discover good replacements of a predicates even if the new predicate is not an official synonym.

Intuitively, the practical synonym strategy infers synonyms from the knowledge base, and is based on the intuition that if two predicates `p1` and `p2` are consistently used with a same subject and a same object, as shown in Figure 9, they likely represent the same information. More in general, we say that `p2` is a practical synonym of `p1` if the ratio between the number of resources that are subject of both `p1` and `p2` according to the schema shown in Figure 9 and the number of resources that are subject of `p1` is greater than 0.5.

To make a choice semantically coherent with the application domain, Link computes the practical synonyms by selecting only the resources of the most specific type that use both the predicates in the model and the predicate in the isolated triple.

When multiple candidate replacements are returned by any of the query strategies, Link selects the predicate that relates with the highest number of predicates that are not isolated in the model, that is the one that better integrates with the input domain of the application. In the running example, the isolated RDF triple `title` cannot be related with other predicates in the model using just synonyms and namespaces, but Link successfully identified that `name` is a practical synonym of `title` and automatically replaced `dbpedia:title` with `foaf:name`.

5.2 Refine the Model

The model initialization may produce either a connected or a non-connected RDF graph. A connected RDF graph indicates that all the labels extracted from the GUI of the application have been related within a single semantic context, and can be used all together to generate test inputs as shown in the next section. An RDF graph composed of more than one connected component indicates that different subsets of labels refer to different contexts, and each component is exploited independently to generate test cases: Different subsets of the input widgets are filled in using the data obtained with different connected components.

Each connected component can be used to extract data from the knowledge base as follows: The free variables that occur as objects of predicates can be used to retrieve meaningful values from the knowledge base. For instance the free variables `?obj_isbn`, `?obj_publisher`, and `?obj_genre` shown in Figure 6 can be used to retrieve ISBN numbers, publisher names, and genre names. The free variables that occur as subjects of the predicate `type` that prescribes that a given input in the GUI must be of a given semantic type can be also used to retrieve meaningful values.

In practice Link extracts data from the knowledge base by executing a SPARQL query where each RDF triple in the graph is translated into a constraint of the query. For instance the two RDF triples with subject `?sbj_isbn` shown in Figure 6 require that the ISBN number and the publisher name relate to a same resource. This is the standard way of translating an RDF graph into a SPARQL query [46]. The complete SPARQL query generated from the RDF graph in Figure 6 is:

```
SELECT DISTINCT ?obj_isbn ?obj_publisher ?obj_genre WHERE {
?sbj_isbn dbpedia-owl:isbn ?obj_isbn .
?sbj_isbn dbpedia-owl:publisher ?obj_publisher .
?sbj_publisher dbpedia-owl:isbn ?obj_isbn .
?sbj_publisher dbpedia-owl:publisher ?obj_publisher .
?sbj_publisher dbpedia-owl:genre ?obj_genre .
?sbj_genre dbpedia-owl:publisher ?obj_publisher .
?sbj_genre dbpedia-owl:genre ?obj_genre .
}
```

Although each connected component represents a semantic context, some components may not be suitable for generating data, that is, the SPARQL query generated from the model returns no data. This may occur because components are identified looking at relations between pairs of triples, and, when considering these relations all together, the knowledge base might not include resources that satisfy all of them at the same time. For instance, an application that requires in input both the login data and some book information may generate a component in the graph that incidentally relates these two aspects, but such relation does exist in the knowledge base. We call such components "dead components".

Link addresses this problem by refining dead components into smaller components that generate useful data. It weights the edges in the graph according to their contribution to generate data from the knowledge base, and uses the minimum cut algorithm [43] to identify the minimum weighted number of edges that must be removed from the graph to obtain two connected components from the current one. The resulting components are then checked against their capability of extracting tuples from the knowledge base. The procedure is applied iteratively until all components generate data from the knowledge base. Intuitively, the refinement step eliminates casual relations introduced in the graph by the incremental algorithm that works on pairs of relations without considering all the relations at the same time.

Since the minimum cut algorithm eliminates edges with small weights and preserves the ones with big weights, Link weights edges assigning small weights to the problematic edges and high weight to relevant edges of the graph. Link weights an edge e as follow:

1. it considers the RDF triple $t = \langle s, e, o \rangle$ associated with e
2. it identifies the portion of the graph close to this triple, that is all the RDF triples $t' = \langle s', e', o' \rangle$ that share with t either the subject ($s = s'$) or the object ($o = o'$). We indicate with G_e the graph with the triple t and all the triples t'.
3. it computes the support of e as the number of instances that satisfy G_e, intuitively the support indicates how many tuples that satisfy the constraints in the subgraph exists in the knowledge base. Edges with high support do not cause problems, while edges with small support strongly constraint the results.
4. it computes the connectivity of e as the ratio between the total number of edges in the whole graph and the edges in G_e. Intuitively the connectivity indicates how many dependencies exist between e and the other predicates. The more dependencies exist, the less the value of the connectivity is, and the more likely e is over constraining the extraction of data.
5. it computes the weight of e as the sum of the support and the connectivity. Since the support is usually significantly bigger than the connectivity, the support drives the minimum cut algorithm (having many instances is a priority of Link), and when a portion of the graph has the same support, the connectivity becomes the auxiliary cutting criterion.

For instance, in the running example all the triples are connected within a single component, but when the component is used to query the knowledge base no data is returned from DBPedia. Thus Link assigns weights to edges and identifies the most problematic set of edges. The algorithm identifies the edges with the predicate `genre` as the most problematic ones and removes them, thus disconnecting `?obj_genre` from the rest of the graph. This is the right action, because the attribute `genre` is not well supported in DBPedia and only few resources use it. Note that the graph in Figure 6 well supports this refinement step, while the graph in Figure 4, where all the edges have the same role, cannot be refined according to this strategy.

Once Link has identified the problematic edges, it removes the ones that do not cause the loss of a descriptor associated with an input widget that must be used in the test. If removing a triple causes the loss of a descriptor from the graph, Link first tries to turn the dead component into a component that can return data by replacing the descriptor. Link searches for a replacement that enables the component to return data by sequentially considering synonyms, namespace, synonyms and namespaces and practical synonyms as described in Section 5.1. If Link finds multiple alternatives, it considers the ones that cause the component to return the biggest amount of data. Otherwise, it removes the edge and adds an isolated triple to the model. In the running example, Link successfully replaces the predicate `genre` with the alternative predicate `literatureGenre` without disconnecting the graph.

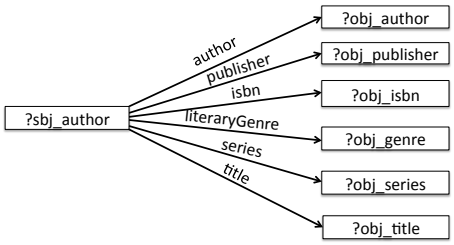

Figure 10: The final RDF graph

Finally, Link applies the hitting set algorithm [40] to each connected component of the RDF graph to produce the minimal graph (i.e., the graph with the minimum number of subjects) that returns the same data as the original one. Figure 10 shows the minimal graph for the running example.

6. GENERATE INPUTS AND RUN TESTS

In this phase, Link uses the connected components of the RDF graph produced in the former phases to generate test cases by extracting meaningful data from the knowledge base.

Link can generate test inputs by querying DBpedia with the SPARQL queries straightforwardly derived from the RDF connected components. Simply querying DBpedia can generate enormous amounts of data that result in unmanageable test suites. Link generates manageable test suites by incrementally producing test inputs, alternating between inputs semantically close and inputs semantically distant from the previously generated inputs. Semantically close inputs validate the behaviour of the application for small variations of the inputs. Semantically distant inputs sample different areas of the input domain.

At each iteration step, Link considers the n connected components $C_1 \dots C_n$ of the RDF graph and generates $n+1$ positive test inputs (1 test inputs if the RDF graph is connected). Link extracts a tuple t_i for each component C_i by

Table 1: Test Inputs

Iteration	Test Input
1 (initial)	Author="C. S. Lewis", Title="The Last Battle", Series="The Chronicles of Narnia", Publisher="The Bodley Head", literaryGenre="Children's literature", isbn="ISBN 978-0-00-720232-4"
2 (distant)	Author="Libba Bray", Title="A Great and Terrible Beauty", Series="Gemma Doyle Trilogy", Publisher="Random House", literaryGenre="Fantasy literature", isbn="ISBN 0-385-73028-4"
3 (close)	Author="Mario Puzo", Title="The Sicilian", Series="The Godfather", Publisher="Random House", literaryGenre="Crime", isbn="ISBN 0-671-43564-7 (Hardback edition) & ISBN 0-345-44170-2 (Paperback edition)"
4 (distant)	Author="Larry Niven", Title="Convergent Series", Series="The Draco Tavern", Publisher="Del Rey Books", literaryGenre="Science fiction", isbn="0-345-27740-6"

querying DBpedia with the SPARQL query corresponding to the component. The n test inputs are obtained by using the data values in each individual tuple independently. Note that each tuple (i.e., each test input) consists of a set of semantically coherent inputs. When the RDF graph contains more than one connected component, Link generates an extra test input as the union of the tuples $t_1, \ldots t_n$ generated for all the components of the RDF graph.

To extract tuples either semantically close or distant from the tuples already extracted, Link exploits the notion of semantic distance between resources. In particular, it uses the Weighted Combined Distance defined by Passant [41].

The Weighted Combined (semantic) Distance is a metric designed to measure the distance between semantic concepts in knowledge bases and takes into account the direct links between the resources (the more direct links exist, the closer two resources are), the indirect links between resources (the more links to shared resources exist, the closer the two resources are), the popularity of resources (highly connected resources have a small impact on the computation of the distance). The process terminates with a timeout defined by the testers.

In the running example, Link generated one component. Table 1 shows the tuples extracted from the knowledge base and the corresponding tests for the fist four iterations of the algorithm. Link automatically extracted meaningful and coherent data about books. It extracted both similar tuples, for instance books with the same publisher at iterations 2 and 3, and distant tuples, for example, books of different nature or genre and with one or more ISBNs.

7. EMPIRICAL EVALUATION

To evaluate Link, we considered two key aspects, the quality of the models that Link generates with respect to the corresponding GUI, and the effectiveness of Link in generating test cases with respect to state-of-art techniques. We evaluated the quality of the models by generating models for GUIs from different domains and measuring the number of fields mapped on consistent and coherent sets of data. We evaluated the effectiveness of the model with respect to other approaches by comparing the test cases generated with Link to the test cases generated with a grammar-based approach to see if and when considering the semantic aspects can improve the test suites. The results confirm that the models are accurate, and that Link generates good test suites.

Table 2: Quality of Models.

Domain	Form	Mapped	Syn/NS	PS	Tot	CC
books	33	51.6%	5.6%	19.1%	76.3%	1.7
music	33	62.5%	0%	17.3%	79.8%	1.2
movies	39	58.9%	5.9%	20.6%	85.4%	2.1
cars	42	38.7%	38.6%	11.2%	88.5%	3.5
all	147	52.3%	13.9%	16.8%	83%	2.2

7.1 Quality of Models

To evaluate the quality of the Link models we considered the Metaquerier repository [5], a repository of 147 real-life Web forms of applications from four different domains: books, music, movies, and cars [21]. We applied Link in a fully automated way (we did not provide any user-specified mapping) to the 147 available forms and measured the number of input fields that Link successfully mapped on coherent and consistent values using DBPedia and the number of components in the resulting RDF graph that measures the semantic consistency among the fields of the form.

Table 2 shows the results obtained for each domain. Column *Domain* identifies the Metaquerier category. Column *Form* indicates the amount of available forms. Column *Mapped* indicates the percentage of labels mapped directly from the GUI to classes or predicates in the knowledge base (the data in this and in the next columns are averages over the forms). Columns *Syn/NS* and *PS* indicate the percentage of labels mapped using synonyms combined with alternative namespaces and the practical synonyms, respectively. Column *Tot* indicates the total percentage of labels automatically mapped with Link. Column *CC* indicates the average number of connected components in the RDF graph returned by Link. The value 1 represents the optimal case in which all the fields of the form are semantically related.

Link has been able to automatically map 83% of the input fields on average. This result indicates that Link can effectively handle a relevant portion of the input space of an application in a fully automatic way, and little manual intervention is necessary to achieve full support. We manually inspected the models and verified that most of the labels that Link has not been able to map are generic words with little correlation to the considered domain, such as the labels `keyword` and `label` that occur in many forms, and that typically require the human intervention to be disambiguated and properly mapped to concepts of the knowledge base.

Column *Mapped* indicates that the Link baseline approach that maps labels to classes and predicates in the knowledge base directly from the GUI is effective (Link has been able to map more than half of the input fields directly from the GUI on average). The complementary approaches based on synonyms, namespaces and practical synonymous are extremely important to achieve high coverage (synonyms and namespaces contributed to map up to 38.6% additional input fields, and practical synonyms contributed to map up to 20.6% additional input fields).

Link has been also particularly effective in semantically relating the input fields producing an average number of 2.2 connected components. We manually inspected the models and verified that most of the RDF graphs with more than one connected component correspond to models with isolated triples that includes a generic label, such as `category` and `price`, with no clear correlation with the rest of the triples. Few simple user-specified mappings may easily disambiguate such labels and lead to much better results.

Link has been less effective in the car domain than in the other domains. In the car domain, Link mapped directly the smallest number of labels, and compensated this lack of efficiency with an extensive use of synonyms, namespaces and practical synonyms. Although in the car domain Link ended up mapping 88.5% of the labels, the extensive use of indirect mapping reduced the quality of the resulting RDF graphs which fail in capturing the semantic relation between some relevant inputs, as witnessed by the relatively high average number of connected components. This is due to DBPedia that does not cover well the car domain. The data obtained for the car domain witness a good performance of Link also when the used knowledge base is not well suitable for that domain.

7.2 Link vs Regular Expressions

Table 3: Effectiveness of generated test suites.

Application	Total		Normal		Branches	
	L	R	L	R	L	R
aMetro	266	623	220	0	133	0
DataCrow	285	449	70	8	17	0
LDAP	284	465	50	465	1	0
LyricFinder	167	181	158	0	1	0
MyFlights	68	203	60	0	52	6
OsmAnd	139	253	63	4	104	1

Most of the techniques to automatically generate system test cases, such as GUITAR [36] and AutoBlackTest [34], focus on generating event sequences that use user-defined inputs and are not designed to generate semantically meaningful test inputs for complex forms. The only techniques that consider semantic information are the ones proposed by Bozkurt and Harman, McMinn et al. and Shahbaz et al. [17, 35, 42], but unfortunately no prototype tools were publicly available at the time of writing.

Thus, we investigated the advantages of using semantic information to generate complex inputs over a classic syntactic approach by comparing the results of Link with the results obtained with test cases generated using regular expressions. While Link is fully automated and application independent, except for the user specified mapping, we defined sets of regular expressions tailored for each application, and thus particularly effective. Writing a user specified mapping amounts to indicate that for example the field *from* of the flight search window maps to class *Airport* and is easily and quickly done by a tester, while writing a complete set of regular expressions requires the right expertise and can be time consuming.

We selected six applications that consider real and coherent data with different levels of semantic correlations among fields: aMetro [1], an Android application for managing maps and routes that needs real and coherent inputs to find paths between stations, DataCrow [2], a Web application for cataloging resources that needs real and coherent inputs only for a small subset of the fields, Ldap Address Book [3], a desktop Ldap client that only checks the syntactic correctness of the input fields without caring about coherency and semantics, Lyrics Finder [4], an Android application for finding the text of songs that needs real and coherent values about the data of the song, like title and artist. MyFlights [6], an Android application for monitoring flights that requires coherent inputs for the departing and arriving airports, and for the other information about flights. OsmAnd [7], an Android application to find addresses that needs coherent inputs to locate the address on the map.

We tested the main functionality of each application (finding paths between metro stations in aMetro, searching books in DataCrow, adding accounts in Ldap, retrieving lyrics in Lyrics Finder, monitoring flights in MyFlights and locating addresses in OsmAnd) using Link for 3 hours and using regular expressions for the same amount of time for each application. To generate test inputs with regular expressions, we queried RegExLib [8] to select a suitable regular expression for each input field in the application, and, when necessary, we manually improved the expression to better fit the required input. We then generated positive test inputs by producing values that satisfy the expressions.

We compare the two approaches using the number of branches uniquely covered by each technique, which intuitively represents cases that only one of the two techniques can cover, and the number of faults revealed by each technique.

Table 3 reports the results. Columns *Application* indicates the tested application. Columns *Total* and *Normal* report the number of total and normal test cases generated with Link (column L) and regular expressions (column R). The normal test cases identify those test cases that fill out the forms with meaningful data, and thus exercise the normal usage of the application. Such test cases check the correctness of the target functionality, while the others check the behaviour of the form in the presence of wrong inputs. While all test cases are important, the normal ones match most of the test cases generated by test experts and thus are expected to occur in good amount in useful test suites. Column *Branches* reports the amount of branches uniquely covered by one of the two techniques, and thus represents the amount of code explored by only one of the two techniques.

The data reported in column *Normal* indicate that Link generates many normal test cases, while regular expressions generate few if any semantically meaningful test cases. The only exception is LDAP that stores the fields required to specify an account regardless of the semantics of the inputs, and thus the inputs are independent from the semantics. We identified normal executions, that is test cases with semantically meaningful data, by designing oracles that check when the test cases exercise the normal usage of the application.

Similarly the data reported in column *Branches* indicate that Link performs better than regular expressions, since it executes many branches that are not executed otherwise. In particular, Link executes a large amount of branches not executed otherwise for more than half of the case studies (aMetro, DataCrow, MyFlight and OsmAnd). Unsurprisingly the effectiveness of the two approaches is similar when the validity of the inputs is loosely related to the semantics of the fields (LDAP and Lyrics Finder).

We also found that Link revealed three faults not revealed by regular expressions: regular expressions revealed no faults, while Link revealed three severe faults in aMetro, Lyrics Finder and OsmAnd, respectively, that cause program crashes and that require complex normal inputs to be exercised. The faults detected in aMetro and in Lyrics Finder are unknown faults.

The fault in aMetro causes a crash of the application when an input asking for a path between two non-connected stations follows an input asking for an existing path between stations. These inputs can be hardly generated with syntax-based approaches.

The fault in Lyrics Finder causes a crash when a lyric of an existing song that is not present in the Musix Match online

catalog is asked. This fault can be revealed only by specifying a complex and coherent input consisting of a proper song title and the corresponding artist, which is almost impossible to generate with regular expressions.

The fault in OsmAnd is a memory leak that causes a crash of the application when many normal inputs are executed. Link can easily generate several normal inputs even when complex and coherent inputs are required by the application, and thus can easily reveal this fault, while this fault can be hardly revealed with syntax-based approaches, due to the limited number of normal values that they generate.

In summary, the results confirm the intuition that when the target application exploits the semantic and coherence of the inputs, Link works better than approaches based on the syntax only. The readers should notice that Link is inexpensive. The only manual step is the definition of a user-specified mapping, which is not required, and that we used only in three of the case studies to add a maximum of two mappings to disambiguate generic terms.

7.3 Threats to Validity

The main internal threats to validity concern the knowledge base and the input parameters. In the empirical evaluation we used DBPedia, which is a general purpose knowledge base. The use of domain specific knowledge bases can only improve the results, thus the reported results are a pessimistic approximation of the effectiveness of the technique.

We selected the values of the few parameters of the technique, like the threshold that determines if a label is a practical synonym of another label, based on our experience with DBPedia. We checked the stability of the parameters and we observed that the results do not change significantly for small changes of their values. We do not know to what extend the values of the parameter and the observation about their stability depend on the chosen knowledge base.

Another internal threat to validity concerns the correctness of the prototype implementation. We did carefully test the implementation, inspected sample executions, and compared the results of multiple executions of the experiments.

The external threats to validity concern mainly the generalizability of the results. We worked with a limited number of applications, but we alleviated the problem of generalizability by choosing applications from different domains. The preliminary results indicate that Link is more effective when the application exploits the semantic and coherence of the inputs and performs as grammar based approaches when generating positive cases for applications that only care about the syntax of the input fields.

8. RELATED WORK

The problem of automatically generating test cases has been addressed in many ways [10]. The most relevant classes of approaches are random testing, coverage-driven testing, specification-based testing, and, more recently, the usage of realistic data. In the following we discuss these strategies.

Techniques based on *random testing* [39, 11] and its variants [23, 32, 22], such as adaptive random testing, can effectively test large portions of the execution space with no manual intervention, but they are totally ineffective when the testing of the application requires realistic and coherent test inputs. Link exploits the semantic data available on the Web to provide a solution that is largely automatic and that can be used with applications that require complex inputs.

Coverage-based test input generation techniques aim to generate test inputs that can cover the code elements that have not been covered by the already generated tests. There are many techniques that address this problem in different ways, for instance using concolic testing (e.g., PEX [44], CREST [19], KLEE [20], DART [27]), and search based testing (e.g., EXYST [29], EvoSuite [26]). The effectiveness of these techniques largely depend on both the complexity of the application code and the scale of the system under test. When effective, they can indeed generate test inputs that cover several relevant corner cases. Link complements these techniques since (1) it does not analyze the source code of the program and thus can be applied to large scale programs, (2) it generates complex and meaningful test inputs, and (3) it can generate test inputs that reveal missing logic errors that cannot be revealed with coverage-based techniques.

Model-based testing techniques generate test inputs from specifications, when suitable specifications are available [45, 30]. The effectiveness and sophistication of the test inputs depend on the completeness and accuracy of the specification. Abstract specification models produce few important tests but may overlook important corner cases, while detailed specification models support thorough testing activities, but require relevant effort. Complete and accurate specification models are often unavailable. Link provides an automatic approach for the generation of complex test inputs that can be applied even in absence of a specification.

Few recent approaches target the *generation of realistic test inputs* either using Web services [17] or combining Web Searches with regular expressions [35, 42]. These approaches demonstrated to be effective when testing functionalities that require test inputs that are independent, but are not designed to generate test inputs for functionalities that require realistic and coherent inputs. Link provides an automatic way to generate these inputs.

The generation of realistic test inputs has been also taken into considered in the domain of testing interactions within multi-agent systems. In particular, eCat [38] is a technique for the generation of test inputs from ontologies. Different from Link, eCat uses the ontology as a a specification model, while Link does not require any specification but automatically identifies the domain of the application and exploits the Web of Data to generate tests.

Finally, GUI testing techniques [36, 34] share with Link the high-level objective of testing an application using the GUI. However, while Link specifically addresses the generation of test inputs, these techniques focus on the generation of test sequences relying on predefined data-sets as test data. Link could be perspectively integrated with these techniques to improve the effectiveness of test generation.

9. CONCLUSIONS

Effective testing of many applications requires complex and coherent inputs that must be *syntactically correct*, *semantically valid* and *semantically coherent*. None of the techniques proposed so far can automatically produce test inputs that satisfy all these characteristics.

Link *explores for the first time the possibility of using the Web of data to generate complex and coherent inputs*. The preliminary empirical results confirm that our approach produces useful test suites that include syntactically correct, semantically valid and semantically coherent inputs.

10. REFERENCES

[1] aMetro. subway on android. http://www.ametro.org, 2013.

[2] Data crow. the ultimate cataloguer. http://sourceforge.net/projects/datacrow/, 2013.

[3] Ldap address book. http://sourceforge.net/projects/ldapaddrbook/, 2013.

[4] Lyrics finder. https://github.com/Danguilherme/lyrics-finder, 2013.

[5] MetaQuerier: Exploring and integrating the deep web. http://metaquerier.cs.uiuc.edu, 2013.

[6] Myflights. https://github.com/markrebhan/MyFlights, 2013.

[7] Osmand. http://osmand.net, 2013.

[8] Regexlib. http://regexlib.com, 2013.

[9] Robotium. https://code.google.com/p/robotium/, 2013.

[10] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978 – 2001, 2013.

[11] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *proceedings of the International Symposium on Software Testing and Analysis*, 2010.

[12] G. Becce, L. Mariani, O. Riganelli, and M. Santoro. Extracting widget descriptions from GUIs. In *proceedings of the International Conference on Fundamental Approaches to Software Engineering*, 2012.

[13] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (uri): Generic syntax. Request for comments: 3986, The Internet Society, 2005.

[14] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–2, 2009.

[15] C. Bizer, A. Jentzsch, and R. Cyganiak. State of the lod cloud. http://lod-cloud.net/state/, 2011.

[16] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. Dbpedia - a crystallization point for the web of data. *Journal of Web Semantics*, 7(3):154–165, 2009.

[17] M. Bozkurt and M. Harman. Automatically generating realistic test input from web services. In *proceedings of the International Symposium on Service Oriented System Engineering*, 2011.

[18] D. Brickley and R. V. Guha. Rdf vocabulary description language 1.0: Rdf schema. W3c recommendation, W3C, 2004.

[19] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *proceedings of the International Conference on Automated Software Engineering*, 2008.

[20] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *proceedings of the Symposium on Operating Systems Design and Implementation*, 2008.

[21] K. C.-C. Chang, B. He, and Z. Zhang. Toward large scale integration: Building a MetaQuerier over databases on the web. In *proceedings of the Conference on Innovative Data Systems Research*, 2005.

[22] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83(1):60–66, 2010.

[23] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. In *proceedings of the International Conference on Software engineering*, 2008.

[24] P. Devaki, S. Thummalapenta, N. Singhania, and S. Sinha. Efficient and flexible gui test execution via test merging. In *proceedings of the International Symposium on Software Testing and Analysis*, 2013.

[25] C. Fellbaum. Wordnet and wordnets. In K. Brown, editor, *Encyclopedia of Language and Linguistics*, pages 665–670. Oxford, Elsevier, second edition edition, 2005.

[26] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *proceedings of the International Conference on Quality Software*, 2011.

[27] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *proceedings of the International Conference on Programming Language, Design and Implementation*, 2005.

[28] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *proceedings of the International Conference on Software Engineering*, 2009.

[29] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: High coverage, no false alarms. In *proceedings of the International Symposium on Software Testing and Analysis*, 2012.

[30] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *Transactions on Software Engineering and Methodologies*, 22(1):6:1–6:42, 2013.

[31] IBM. IBM rational functional tester. http://www-01.ibm.com/software/awdtools/tester/functional/, 2013.

[32] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: object capture-based automated testing. In *proceedings of the International Symposium on Software Testing and Analysis*, 2010.

[33] G. Klyne and J. J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. W3c recommendation, W3C, 2004.

[34] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. AutoBlackTest: Automatic black-box testing of interactive applications. In *proceedings of the International Conerence on Software Testing, Verification and Validation*, 2012.

[35] P. McMinn, M. Shahbaz, and M. Stevenson. Search-based test input generation for string data types using the results of web queries. In *proceedings of the International Conference on Software Testing, Verification and Validation*, 2012.

[36] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *Transactions on Software Engineering*, 31(10):884–896, 2005.

[37] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web

applications. *Transactions on Software Engineering*, 38(1):35–53, 2012.

[38] D. C. Nguyen, A. Perini, and P. Tonella. Ontology-based test generation for multiagent systems. In *proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, 2008.

[39] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *proceedings of the International Conference Companion on Object-Oriented Programming, Systems, and Applications*, 2007.

[40] T. V. Paschos. A survey of approximately optimal solutions to some covering and packing problems. *ACM Computing Surveys*, 29(2):171–209, June 1997.

[41] A. Passant. Measuring semantic distance on linking data and using it for resources recommendations. In *proceedings of the AAAI Spring Symposium: Linked Data Meets Artificial Intelligence*, 2010.

[42] M. Shahbaz, P. McMinn, and M. Stevenson. Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing. In *proceedings of the International Conference on Quality Software*, 2012.

[43] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997.

[44] N. Tillmann and J. D. Halleux. Pex: white box test generation for .NET. In *proceedings of the International Conference on Tests and Proofs*, 2008.

[45] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2006.

[46] W3C SPARQL Working Group. Sparql 1.1 overview. W3c recommendation, W3C, 2013.