Verification of Semantic Web Service Annotations Using Ontology-Based Partitioning

Khalid Belhajjame, Suzanne M. Embury, and Norman W. Paton
School of Computer Science, University of Manchester

Abstract—Semantic annotation of web services has been proposed as a solution to the problem of discovering services to fit a particular need, and reusing them appropriately. While there exist tools that assist human users in the annotation task, e.g., Radiant and Meteor-S, no semantic annotation proposal considers the problem of verifying the accuracy of the resulting annotations. Early evidence from workflow compatibility checking suggests that the proportion of annotations that contain some form of inaccuracy is high, and yet no tools exist to help annotators to test the results of their work systematically before they are deployed for public use. In this paper, we adapt techniques from conventional software testing to the verification of semantic annotations for web service input and output parameters. We present an algorithm for the testing process, and discuss ways in which manual effort from the annotator during testing can be reduced. We also present two adequacy criteria for specifying test cases used as input for the testing process. These criteria are based on structural coverage of the domain ontology used for annotation. The results of an evaluation exercise, based on a collection of annotations for bioinformatics web services, show that defects can be successfully detected by the technique.

Index Terms—Semantic web services, semantic annotations, test adequacy, specification-based testing, model-based test generation.

1 Introduction

Semantic annotations of web services have been proposed as a (partial) solution to web service discovery and composition [5], [4]. These annotations relate the various service elements (i.e. operations, inputs and outputs) to concepts in ontologies describing their semantics, form and role. For example, an operation which takes a protein sequence as input and performs a search for the most similar protein in some given database might be annotated with the task concept *SimilaritySearch*. Its single (string-valued) input parameter might be annotated with the concept *ProteinSequence*, and its output with the concept *ProteinAccessionNumber*.

Such semantic annotations can be used by tools to e.g., assist in service discovery and composition [16], [17]. Such tools can only be of value if semantic annotations accurately reflect web services' semantics. Our experience using data flow connections between services in workflows to check compatibility of annotations has revealed that this is frequently not the case [2]. Manual annotation of services is a time-consuming task that requires expertise in the application domain, in its representation in the specific ontology being used, and in the task supported by the web service. Where the annotations are supplied by the web service providers, one would reasonably expect them to be of good quality, in the sense that the chances that they suffer from errors can be low. However, the annotation task is generally not integrated within the web service development lifecycle, and, as a result, web services are often annotated by third parties. For example, web service annotations in registries, such as Biocatalogue[1], are provided in the majority of cases by the users of web services or curators who are hired to perform the annotation task, and not by the service providers. In the typical case, the annotator will have been involved with neither the design of the domain ontology nor the implementation of the web service, and there is therefore considerable scope for mistaken interpretations of concepts and behaviours in selecting annotations. Despite this, there is a dearth of tools and methods that can assist in the verification of annotation accuracy. Based on discussions with annotators working on the ^{my}Grid project[2], peer inspection of annotations is the main means of verification in practical use, supplemented with feedback from users after the annotations have been published.

In this paper, we explore how techniques from software testing can be applied to the problem of verifying semantic annotations of web service parameters. Software testing is a well established discipline that has been a major research topic in software engineering during the last three decades, techniques from which have proved to be effective in practice for verifying that the behaviour of a software program conforms with its specification [18]. A software program, e.g., a web service, is tested by using a test suite composed of a collection of test cases, each of which specifies data values for feeding the software execution and the outputs expected as a result according to the specification. The software is executed using the input values specified by the test cases. A defect is found if the outputs delivered as a result of the software execution are different from those expected by a test case.

1. `http://www.biocatalogue.org`
2. `http://www.mygrid.org.uk`

We adapt the above technique for verifying the annotations of web services. We begin by examining the semantics of annotations on service parameters, and derive from this the form of a test case for parameter annotations (Section 2). These semantics provide us with the basics of an oracle for automatic test case generation (Section 3). An exhaustive test that uses every possible test case can be expensive, and most of time is impossible as the domains of the inputs can be infinite. Because of this, only a subset of possible input values is used to construct test cases. To ensure that this subset is *representative* of the range of possible input values, thereby increasing its defect-detecting power, it is constructed carefully based on test adequacy criteria. In this respect, we propose two new adequacy measures, based on coverage of concepts in the ontology used for annotation, that can be used to automatically select the inputs needed to form representative test suites (Section 4). We assess the effectiveness of the adequacy criteria by applying them to the verification of real-world annotations produced by a domain expert, and show that both approaches result in much higher defect detection rates than when the same number of test cases are selected at random from the input pool (Section 5). We compare our method to existing proposals (in Section 6), and conclude by highlighting our main contributions (in Section 7).

2 Verifying Semantic Annotations

Semantic annotations for web service parameters are effectively high-level descriptions of the domains of the parameters that go beyond the usual data types found in interface description languages (such as WSDL). As such, it should be possible to generate test cases based on the information they contain, in much the same way as is done in other forms of model-driven/specification-based testing (e.g. [9], [10], [24]). In these approaches, however, the specification is regarded as being the correct description of the software's behaviour, and the test cases created are intended to reveal defects in the software. In the case of semantic annotations, however, which are typically generated after the code has been publicly deployed, and by some third party not involved in its creation, the software is considered to be *de facto* correct and the test cases derived are intended to verify that the specification accurately reflects its behaviour. In addition, we are not attempting to verify the complete behaviour of the web service. Instead, we are attempting to verify a somewhat weaker proposition, namely, whether the concepts used to annotate the parameters are accurate descriptions of the sets of values that can legally be input to or output by the service.

In the standard model-based test case generation approaches, for a web service with n input parameters, we would use the specification model to generate the following pieces of information for each test case:

- An n-tuple containing the values that should be used as inputs to the web service in one test run.
- A statement of the expected outcome of the test when these input values are given.

In order to generate test cases for semantic parameter annotations, therefore, we must show how to create both these test case components based only on the ontology concepts that have been assigned to the web service parameters under test.

2.1 Generation of Inputs for Parameter Tests

Since web services are black-box software components, we must find a way to verify the input parameter semantics by executing the services and observing the result. In conventional testing, we would be interested in the actual data value returned by the web service for some given input values, but in this context, as we have said, we are testing only whether the input is legal for the service and not whether it leads to a correct result. How, then, can we determine whether an input is legal based on observation of the execution of the service?

In order to provide one possible answer to this question, we introduce the notion of *acceptance* of an input parameter by a web service operation. The web service interface standard allows programmers to clearly distinguish normal termination of an operation (in which a result message is returned, for example) from abnormal termination (in which an exception is thrown). Therefore, we can say that an input (or collection of inputs) is accepted by an operation only if, when supplied with the input(s) as parameters, the operation terminates normally. This gives us a very simple operational means of distinguishing web service behaviour with valid inputs from that with invalid inputs. Note, however, that in practice, unfortunately, it is more difficult to interpret both normal and abnormal terminations of web services. In the case of abnormal terminations, there are several other factors that might cause a web service to exit with an exception, besides being supplied with an invalid input. For example, an e-bookstore may report that the book is out of order. Such a termination should not be interpreted as abnormal. Also, the server hosting the web service might be temporarily inaccessible, or the service may be attempting to manage its own load by refusing requests when its load gets too high. In some cases, problems will go away if the tests are run again, and the user need not be bothered with them. However, in others, the only safe way to interpret the results of the test is to ask a human user to investigate. Similarly, a service execution which seems normal, i.e., does not throw any exception, may be unsuccessful. Although the web services standards provide proper exception mechanisms for flagging errors to the caller, many web service implementations make no use of them. This is commonly the case when a web service is rapidly implemented by wrapping existing software implemented in a programming language that did not support exception handling. Such programs typically report errors to the user by returning an empty string, or a string value that contains an error message. For example, the *getUniprotEntry* web service provided by the DDBJ[3], returns an empty message when the value used as input is invalid. Where the forms of these error messages are known, we can write string matching functions that distinguish between a normal and an abnormal termination even when no exception has been thrown.

We can see that to test a semantic annotation for an input parameter, we should generate a collection of legal and illegal values for the parameter, and create test cases which look for normal and abnormal termination respectively. For a given parameter *ip* annotated with concept c_{ip}, the set of legal values are those which are in the extension of c_{ip} and the set of illegal values are those in the extension of $\neg c_{ip}$ (i.e., any value in the extension of any concept that is not equivalent to c_{ip} and which is not in the extension of c_{ip}). Thus, to generate candidate input values, we need a pool of data values that are annotated with the ontology concepts to which they belong.

Since semantic types are typically more complex than standard data types, the task of generating values for them is correspondingly more challenging. Although some finite numeric ranges or simple enumerated types may be present as parameters for some operations, in general we cannot expect to be able to provide a set of built-in data generators that will cover the majority of cases, as we can in software testing. For certain semantic types, it may be possible for the user to indicate a computational procedure by which candidate instances can be obtained. For example, a large and representative collection of *ProteinSequence* instances can be created by querying one of the major public protein databases (such as Uniprot[4]). This may be a satisfactory solution for semantic types that occur frequently in operation parameters, but it demands too much effort from the annotator to be a good general solution.

An alternative source of annotated instances does exist, however, namely, workflow provenance logs [6]. A provenance log is effectively an execution trace that is recorded by the workflow management system during workflow enactment. Most importantly for our purposes, these logs contain copies of the intermediate data values that were produced by the execution of each component service in the workflow. Every data item that was passed as an input parameter to some component service, or returned as an output, is recorded in the log. Where those services have been semantically annotated, the logged data values are also tagged with the semantic type of the parameter with which they are associated.

By trawling these workflow logs, we can collect annotated instances for use in test case generation. Note, however, that the annotations found in the provenance logs may not be entirely accurate. As in object-oriented programming, if an input parameter has a semantic type of c then it may legitimately take instances of *c or any of its subconcepts* as values. Thus, the annotations given to the data values in the provenance logs may often be superconcepts of the correct annotations, and therefore somewhat inaccurate. Fortunately, however, this does not affect the validity of their use in test case generation. Consider, for example, an operation which takes a *Sequence* but which is supplied with an instance of *ProteinSequence* (a subconcept of *Sequence*) as an input at run-time. This instance will then be mistakenly tagged as a *Sequence* in the provenance logs, and may be supplied as a possible test input whenever *Sequence* instances are required. But, since the instance actually *is* an instance of its tagged class, no serious consequences arise.

It is worth stressing that only workflows with services that have annotations that are known to be correct are used. Moreover, the instances that are collected by trawling provenance logs are checked by a human user before they used in the generation of test cases.

So far, we have focused on the values to be used for constructing the test cases for input parameters. Regarding the test cases for output parameters, we exploit an obvious source, namely the results generated by the service operation in question when invoking them using the (legal) input values from the pool of annotated instances.

2.2 Determining Expected Outcome for Parameter Tests Under the Weak Semantics

To complete the test case for a given operation parameter, we must be able to determine what the expected outcome of the case should be, if its annotation is an accurate description of the legal values for that parameter. For this, we need a clear statement of the intended semantics of parameter annotations. Unfortunately, the literature on semantic web services has not reached a consensus on exactly what the meaning of a semantic annotation for an operation parameter is. Two alternative semantics can be gleaned from the literature. Unsurprisingly, the clearest statement of intended semantics can be found in the proposals for languages for specifying semantic annotations (such as SAWSDL[5], WSML[6] and OWL-S[7]). The specifications of these languages state that all valid inputs for an annotated parameter must belong to the domain concept with which it has been annotated. However, it is not necessary for the operation to accept *all* such instances as valid (or, at least, such a condition is not explicitly stated by the authors of these proposals).

To illustrate this point with an example, let us return to the *SimilaritySearch* operation mentioned earlier, the sole input parameter of which is annotated with the *ProteinSequence* concept. This annotation should be read as indicating that no instances of any concepts that are disjoint with *ProteinSequence* should be accepted by the operation; so no *NucleotideSequence* instances should be accepted, nor any *Organism* instances. But, there may

3. `http://xml.nig.ac.jp`
4. `http://www.uniprot.org`
5. `http://www.w3.org/TR/sawsdl/`
6. `http://www.wsmo.org/wsml/`
7. `http://www.daml.org/services/owl-s/1.0/`

also legitimately be instances of *ProteinSequence* that cannot be accepted by the operation.

We call this interpretation of the annotations the *weak semantics*, since it does not place very stringent requirements on which instances the operation can accept. Similarly, under the weak semantics, the instances that are generated by an operation output belong to the concept used for annotating that output. However, the output is not required to produce every possible instance of the concept used for annotation. With the following notation, we can define this semantics formally. An operation *op* has a set of input parameters given by the function *inputs(op)* and a set of outputs given by *outputs(op)*. A parameter is denoted by a pair $\langle op, p \rangle$ where p is the name of the parameter and *op* is the operation to which the parameter belongs. The set of all input parameters is *INS* and the set of all outputs is *OUTS*. If θ is the set of concepts $\{c_1, \ldots, c_n\}$ in our domain ontology, then the function: $domain : INS \cup OUTS \rightarrow \theta$ returns the domain annotation for any given parameter. Given a function *hasInstances()*, which takes a concept and returns the set of all instances of that concept, we can define a correct parameter annotation under the weak semantics as follows.

Definition 2.1: An input (resp. output) parameter $\langle op,p \rangle$ is annotated correctly under the weak semantics iff:

- there exists an instance $i \in hasInstances(domain(\langle op, p \rangle))$ that is accepted (resp. generated) by *op*, and
- no instance $j \in hasInstances(\neg domain(\langle op, p \rangle))$ exists that is accepted (resp. generated) by *op*.

Based on this definition, we can see that if an operation accepts (resp. generates) a particular instance of a concept that is disjoint with the annotation c for the relevant input (resp. output) parameter, then the annotation is incorrect. However, if an instance of c is not accepted (resp. generated) by the operation, then we cannot conclude that the annotation is incorrect.

As explained earlier, in the case of input parameters, we use operation acceptance as a means for determining validity based on the execution of the web service operation in question. We need a similar operational means for output parameters. By selecting legal input values from our pool of annotated instances, we can create a set of output values produced by the operation. However, to determine the result of the test case, we need to determine to which semantic concept the instance produced belongs. This is a much harder task than we faced when generating input parameter test cases, when we were given a concept and had to locate instances in its extension. In the case of output parameters, we have the reverse task. For example, given a string value, we must determine whether it is an instance of *DNASequence* or *GeneIdentifier* or *LabName*.

Annotating an instance, that is discovering the semantic type of an instance, based solely on its content is, in general, a difficult task. One (last resort) option would be to rely on the domain expertise of the annotator, and to ask him or her to classify the output values produced by the web services. However, it would obviously be preferable if some automatic means of classifying output instances could be found.

In fact, an automatic mechanism does exist. Our discussion of the requirements for testing input parameters points to a resource of software components able to perform this classification task: namely, the annotated web services themselves. Consider, for example, a web service with an output that is annotated with the *GeneOntologyID* concept. If we assume that the annotations of service inputs have a strong semantics (see Section 2.3), then any web service with an *input* parameter annotated with this same concept can, in theory, act as a means of testing whether the output value belongs to the concept or not. For instance, we can potentially use the *GetGOTerm*[8] web service to verify the semantic type of any values claiming to be instances of the *GeneOntologyID* concept. If *GetGOTerm* accepts the output value, then we know it is a valid gene ontology ID.

A test case for an output parameter annotation, therefore, looks similar to that for an input annotation. We must provide a collection of values to be used as the inputs to the operation under test, in order to generate a sample output value. We then define the expected response in terms of normal or abnormal termination of a web service. The main difference is that the test case involves the execution of a composition of two operations, rather than just one operation.

All this, however, depends upon the rather rash assumption that the input annotations are correct. Since we have evidence to suggest that quite a high proportion of annotations are inaccurate to some degree or other [2], this approach to verifying output parameters should only be used with web services with trusted input annotations. This suggests a phased approach to testing, in which the annotator first concentrates on verifying the input annotations for a batch of web services, and on correcting any errors found, before beginning to tackle the correctness of output annotations. Another advantage of this phased approach is that we can collect and record the details of the output values generated while testing the input parameters, and thus save some time in the output parameter testing phase.

2.3 Determining Expected Outcome for Parameter Tests Under the Strong Semantics

The annotation of an operation input is said to have a strong semantics, if the operation accepts any instance that belongs to the concept used for annotating the input. This alternative strong semantics for input parameter annotations is implied by the requirements of several applications in which annotations are used, in particular service discovery and workflow composition. Consider, for example, tools that assist designers in the

8. `http://www.inab.org/dproxy/inb.bsc.es/getGOTerm?wsdl`

composition of workflows, e.g., Biomoby Taverna plug-in [25] and Galaxy [7]. Given a service operation in an incomplete workflow, such tools suggest to the designer the service operations that can be composed with it, in order to move towards the completed workflow. Specifically, such tools use the semantic annotations of output parameters to search for services that are able to consume the values output by the service already in the workflow. The implication here is that the suggested service operations are able to consume any instance of the concept used to annotate the output of the preceding service in the workflow. The same implication can be made from tools that diagnose parameters connections to detect mismatches [1].

We can formally define correctness of annotations under this strong semantics, as follows.

Definition 2.2: An input parameter ⟨op,p⟩ is annotated correctly under the strong semantics iff:

- every instance $i \in \mathit{hasInstances}(\mathit{domain}(\langle op, p\rangle))$ is accepted by *op*, for every assignment of valid instances to all other input parameters of *op*, and
- no instance $j \in \mathit{hasInstances}(\neg \mathit{domain}(\langle op, p\rangle))$ exists that is accepted by *op* for some assignment of instances to all other input parameters of *op*.

Note that while the annotations of input parameters specified using languages such as OWL-S and WSMO have a weak semantics, these languages offer a mechanism, namely preconditions, using which the semantics of annotations specified can be upgraded from weak to strong. Specifically, using WSMO, for example, every operation *op* can be associated with a precondition *prec*, that only holds when the values bound to the inputs parameters of the operation *op* are accepted by *op*.

In the rest of this paper, we assume that the input annotations has a strong semantics, since it is the semantics adopted in practice as illustrated in the workflow composition example presented above. That said, we discuss the conclusions that cannot be made when the annotations have a weak semantics.

Regarding output parameters, it does not seem reasonable to insist that an operation with an output annotated with a concept c_{op} must be capable of outputting every possible instance of c_{op}. To illustrate this using an example, consider a service operation that produces a biological sequence. It is neither reasonable nor practical to require such an operation to be able to produce every possible sequence. Instead, service operations that produce outputs of that kind are confined to biological sequences that are either stored in a public database or in-house labs. Also, the set of output instances are usually reduced to the subset of the instances of c_{op} that are output given valid input parameters for the operation. Given the above reasons, we consider only a weak semantics for output parameters.

The above definitions formalise the sets of valid and invalid inputs implied by the annotations, and so provide us with a means of deducing the expected output for a given test case (in terms of normal or abnormal termination of the operation). In the case of input parameters, if the annotations have a strong semantics, then we can be more confident in interpreting the results of the test than we could with the weak semantics. Here, if an instance of a concept c is not accepted by an operation for an input parameter that is annotated with c, then we know that the annotation is definitely incorrect, since all instances of c should be valid for this parameter.

Based on all this, we can now specify the full algorithm for testing semantic annotations for input and output parameters.

3 Annotation Verification Approach

We start this section with an overview of the lifecycle of semantic web service annotations. We then present the details of the phases that constitute the annotation verification process.

3.1 The Lifecycle of Semantic Web Service Annotations

Figure 1 shows that the lifecycle of semantic annotations of web services can be decomposed into three main steps: annotation, verification and use.

In the first step, *Annotation*, web services are annotated either manually, e.g, using Biocatalogue[9] or Radiant [12], or semi-automatically using tools such as Meteor-S [20], APIHUT [11], KINO [21]. These tools provides the user with suggestions for terms that can be used for annotation, and which are inferred using existing schema matching and machine learning techniques. Detailed information about the above annotation tools is given in the related work section.

In the second step, *Verification*, semantic annotations are verified to identify potential defects. A detailed description of this step is given in Section 3.2.

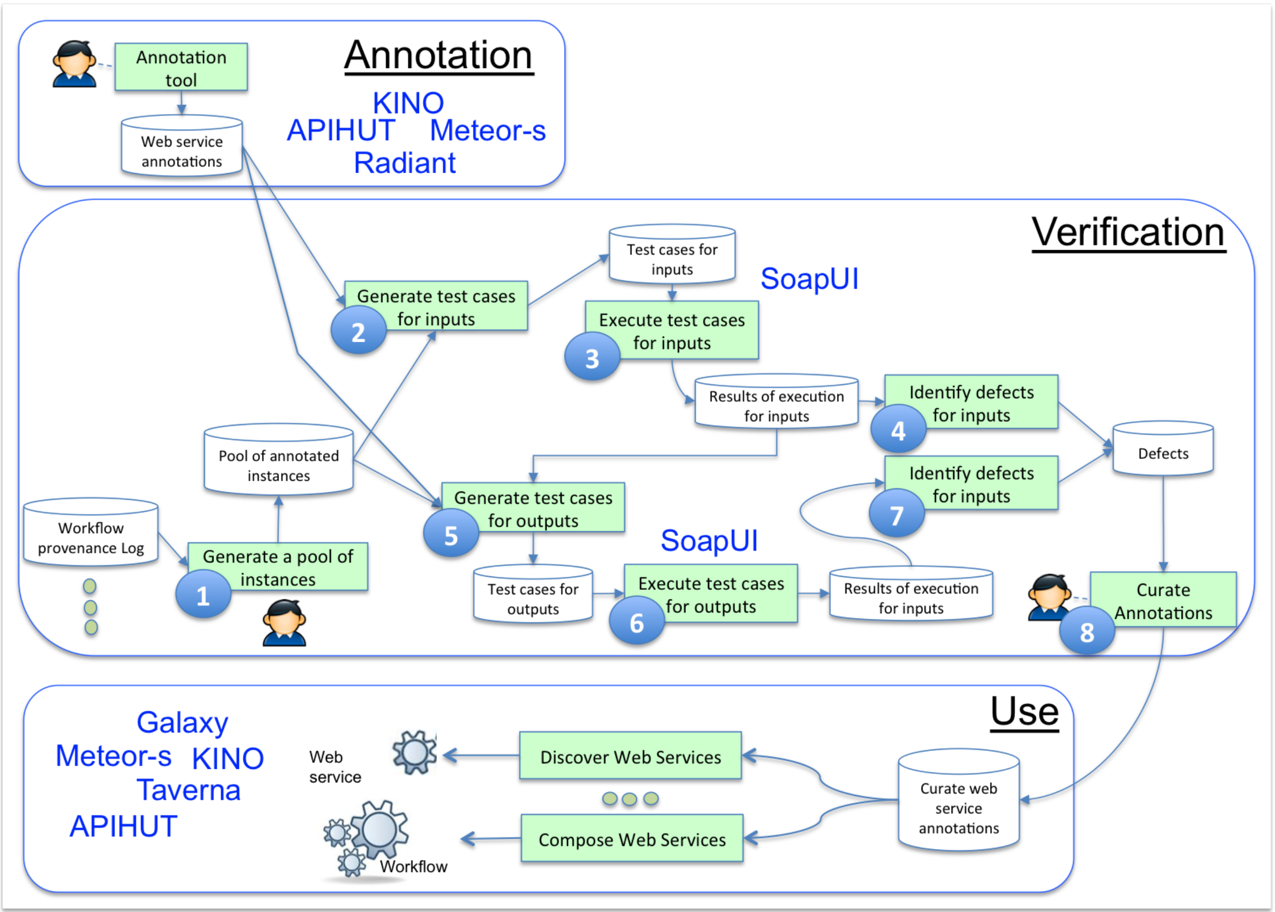
In the the third step *Use*, curated semantic annotations are deployed for public use. In particular, semantic annotations are used to discover and compose web services using tools such as Galaxy[10], Meteor-S [20], APIHUT [11] or KINO [21]. Detailed information about these tools is given in the related work section.

Now that the lifecycle of semantic annotation of web services has been presented, we can proceed to the details of the verification method.

3.2 Verification Process

The phases that constitute the verification process are illustrated and numbered in Figure 1. The verification process starts by generating a pool of annotated instances (phase 1). Once the pool of annotated instances is available, the test cases for input parameters are constructed (phase 2), executed (phase 3), and the results are analyzed to identify defects in inputs annotations (phase 4). Test cases for the annotations of output parameters are then identified (phase 5) by exploiting the result of execution of the test cases for input parameters. The test

9. `www.biocatalogue.org`
10. http://http://galaxyproject.org



Annotation
Annotation tool
Web service annotations
KINO
APIHUT
Meteor-s
Radiant
Verification
Test cases for inputs
SoapUI
Generate test cases for inputs
2
Execute test cases for inputs
3
Results of execution for inputs
Identify defects for inputs
4
Defects
Pool of annotated instances
Identify defects for inputs
7
Workflow provenance Log
Generate test cases for outputs
5
SoapUI
Generate a pool of instances
1
Test cases for outputs
Execute test cases for outputs
6
Results of execution for inputs
Curate Annotations
8
Use
Galaxy
Meteor-s
KINO
Taverna
APIHUT
Web service
Discover Web Services
Compose Web Services
Workflow
Curate web service annotations

Fig. 1. Web Service Annotation Lifecycle and the Verification Approach.

cases for outputs are executed (phase 6), and analyzed to identify defects (phase 7). A curator can then correct annotations in the light of the defects discovered (phase 8). We present, in what follows, each phase in detail.

Phase 1: Generating a pool of annotated instances. In the first phase, a pool of annotated instances is generated using workflow provenance logs. Workflow provenance logs are not used directly to verify parameters annotations, rather they are used as a source of instances of ontological concepts. In doing so, only provenance logs of workflows, that are composed of web services that are known to have correct annotations, are used. Note also that, this phase is not fully automatic; rather, it is done under the control of a human who checks that the instances retrieved from the provenance logs belong to the concepts used for annotating the corresponding parameters.

It is worth mentioning that the use of workflow logs is not mandatory. Other sources of annotated instances can be used. For example, where the ontology used for annotation is already associated with instances that are annotated using that ontology (or what is known as ABox in the literature), then those instances can be used for constructing test cases.

Phase 2: Generating test Cases for Input Parameters. As mentioned earlier, we consider that the annotations of operation inputs have a strong semantics, and that, if needed, operations are associated with preconditions that are used to enforce that semantics. We also discuss the conclusions that cannot be made when the annotation of input parameters have a weak semantics.

Generation of test cases for input parameters is an automatic process. Once the user has specified the set of parameters and the semantic annotation registry to be tested, as well as the location of the ontology used for the semantic annotations, generation is carried out as specified in the algorithm in Figure 2. Interested readers can find example test cases online[11].
Notice that the algorithm does not specify how legal and illegal values for each parameter (variable *pool*) are selected, since this is performed in the first phase. It is worth noting, however, that when the input parameter under test $\langle op, p\rangle$ is associated with a precondition $prec$, then this predicate can be used to reduce the number of legal values used for verifying the accuracy of the annotation of the input parameter to those satisfying the predicate $prec$. Specifically, a legal value v will be used for

11. http://img.cs.man.ac.uk/quasar/example_test_cases.php

```
**Inputs** $ps_i$ := input params to be tested
**Outputs** $ts_i$ := empty test suite
**Begin**
**For each** operation parameter $\langle op, p\rangle \in ps_i$
    *pool* := selected values from pool
    (should be a mixture of legal and illegal)
    **For each** $v$ in *pool*
        **If** $v$ is legal
            **Select** tuple of legal values $lv$ for
            other input params of this operation
            such that the precondition of *op* holds
            *eo* := "*normal*" //expected outcome for $lv \cup v$
        **If** $v$ is illegal
            **Select** tuple of legal values $lv$ for
            other input params of this operation
            *eo* := "*abnormal*"
        *tc* := $\langle\langle op, p\rangle,\ lv \cup v,\ eo\rangle$
        **Add** test case *tc* to $ts_i$
**End**
```

Fig. 2. Algorithm for Generating Test Cases for Testing Input Parameters

testing the annotation of $\langle op, p\rangle$ iff $holds(prec, \langle op, p\rangle, v)$, where $holds(prec, \langle op, p\rangle, v)$ is true iff $prec$ is satisfied when v is used as a value for $\langle op, p\rangle$.

Phase 3: Executing test Cases for Input Parameters. When the test suite has been created, the test cases within it are executed in turn, and the results are logged for subsequent analysis (phase (3) in the overall approach). The steps required are specified by the algorithm in Figure 3. Existing tools such as SoapUI [8] can be used in this phase, since they support large numbers of web service invocations.

```
**Inputs:** $ts_i$ := test suite
**Outputs:** $rs$ := a null tuple
        $inputLog$ := the results of execution of test cases
        $outputLog$ := the results delivered by
                        output parameters
**Begin**
**For each** $tc$ in $ts_i$
    *tc* := $\langle\langle op, p_i\rangle,\ vs,\ eo\rangle$
    **Execute** operation *op* with input params $vs$
    $rs$ := result returned from *op*
    **If** *op* terminated normally **then**
        $ao$ := "normal"
        **For each** output param $\langle op, p_o\rangle$ of *op*
            $r$ := value for $\langle op, p_o\rangle$ in $rs$
            **Add** $\langle\langle op, p_o\rangle, r\rangle$ to $outputLog$
    **Else**
        $ao$ := "abnormal"
    **Add** $\langle op, vs, rs, ao\rangle$ to $inputLog$
**End**
```

Fig. 3. Algorithm for Executing Test Cases for Input Parameters

Notice that when executing an operation *op* using the inputs parameters vs, we may need to transform the values in vs to make them structurally compatible with the representations adopted by *op* parameters. This is because different operation parameters adopt different structures to represent the same value.

Phase 4: Interpreting the results of execution of test cases for input parameters. Once the test cases have been executed, the logged results from the input parameter test cases are analysed. The main task at this point is to compare the expected outcome with the termination status of the task, to determine which test cases have revealed a defect and which have not. As hinted in the algorithm in Figure 3, the result of the execution of a test case for an operation *op* is logged using tuples of the form:

$$\langle op, I_{input}, I_{output}, status\rangle$$

Here, I_{input} is a set of pairs $\langle\langle op, p_i\rangle, ins_i\rangle$ where $\langle op, p_i\rangle$ is an input parameter of *op* and ins_i is the instance used to feed $\langle op, p_i\rangle$ during the test case execution. Similarly, I_{output} is a set of pairs $\langle\langle op, p_o\rangle, ins_o\rangle$ where $\langle op, p_o\rangle$ is an output parameter of *op* and ins_o is the value of $\langle op, p_o\rangle$ that was obtained from the execution of *op* with the input values given in I_{input}. The final component, *status*, has either the value *normal* or the value *abnormal*, describing how the operation terminated.

We retrieve executions from the log using the function:

$$getExecutionsByParamValue : (INS \cup OUTS) \times \mathcal{I} \rightarrow \mathcal{E}$$

where $\mathcal{I}$ is the set of all instances and $\mathcal{E}$ is the set of all operation log tuples just described. The function returns all executions in which the given parameter took the given value.

In interpreting the results of each test case, there are four possible situations to be considered. In the following discussion, *op* is the operation executed during the test case, and $\langle op, p\rangle$ is the parameter being tested. The annotation associated with $\langle op, p\rangle$ is the concept c.

1) *Valid Input/Normal Termination*
In this situation, an instance of c was supplied as the value for $\langle op, p\rangle$ and the operation terminated normally, as expected:
$$\exists\ ins \in domain(\langle op, p\rangle),$$
$$\exists\ e \in getExecutionsByParamValue(\langle op, p\rangle, ins),$$
$$e.status = \text{"}normal\text{"}$$
A test result of this kind presents no evidence for the existence of a defect in the annotation, and it is not reported to the user.
2) *Invalid Input/Normal Termination*
In this situation, an instance of a concept that is disjoint with c was supplied as the value for $\langle op, p\rangle$. We would have expected an abnormal termination from such a test case, but the operation has accepted the supposedly invalid instance.
$$\exists\ ins \in \neg domain(\langle op, p\rangle),$$
$$\exists\ e \in getExecutionsByParamValue(\langle op, p\rangle, ins),$$
$$e.status = \text{"}normal\text{"}$$
This result is evidence of an error in the annotation. The parameter annotation may be over-specialised (i.e. the true annotation is one of its superclasses)

or it may simply be incorrect. This result should be flagged to the user.

3) *Valid Input/Abnormal Termination*
In this situation, an instance of c was supplied as the value for $\langle op, p\rangle$ but the operation unexpectedly terminated abnormally.

$$\begin{gathered}\exists\ ins\ \in\ domain(\langle op, p\rangle),\\ \exists\ e\ \in\ getExecutionsByParamValue(\langle op, p\rangle, ins),\\ e.status\ =\ \text{``}abnormal\text{''}\end{gathered}$$

This result is evidence of the presence of a defect in the annotation of input parameter or the precondition associated with *op*, if such a condition exists. In the case where the operation is not associated with any precondition, and the annotation of the input has a strong semantics, then this test is evidence for the presence of a defect; it could indicate that the annotation for $\langle op, p\rangle$ is over-generalised (i.e. the correct annotation is a subconcept of c) or it may be entirely incorrect.
It is worth mentioning that in the case of the weak semantics, a test run of this kind presents no evidence of any defect in the parameter's annotation.

4) *Invalid Input/Abnormal Termination*
In this situation, an instance of a concept that is disjoint with c was supplied as the value for $\langle op, p\rangle$ and the operation terminated abnormally as expected.

$$\begin{gathered}\exists\ ins\ \in\ \neg domain(\langle op, p\rangle),\\ \exists\ e\ \in\ getExecutionsByParamValue(\langle op, p\rangle, ins),\\ e.status\ =\ \text{``}abnormal\text{''}\end{gathered}$$

A test result of this kind presents no evidence for the existence of a defect in the annotation, and it is not reported to the user.

As we pointed out, in practice, it is more difficult to interpret both normal and abnormal terminations of web services than the above characterisation suggests. In the case of abnormal terminations, there are several other factors that might cause a web service to exit with an exception, besides being supplied with an invalid input. One of these is when an service operation with multiple input parameters terminates with an error. This will occur if any of the parameter values given is invalid; the problem may not be because of an error with the annotation for the parameter under test. To reduce the number of false positives of this kind reported to the user, we consider only cases in which the operation has *always* failed when given this particular value for the parameter under test, regardless of the values of the other parameters. In other words, this kind of test failure is reported only when:

$$\begin{gathered}\exists\ ins\ \in\ domain(\langle op, p\rangle),\\ \forall\ e\ \in getExecutionsByParamValue(\langle op, p\rangle, ins),\\ e.status\ =\ \text{``}abnormal\text{''}\end{gathered}$$

(The only change in the above expression compared with the expression given in (3) is the quantifier in the second conjunct.)

Phase 5: Generating test Cases for Output Parameters. To construct test cases for output parameters, we use the results of execution of the test cases of input parameters. To do so, we examine the results of execution of test cases of input parameters, to filter out parameters for which no output values were produced during the earlier input parameters testing phase. Next, we further filter the parameter set to remove all those for which no annotated web service exists which takes as an input instances of its semantic domain. That is, if we have a service that produces a *ProteinSequence* as an output parameter that we wish to test, then we must also have another service that takes as one of its input parameters (and preferably, the sole input parameter) instances of *ProteinSequence*. Where the web service used for testing has more than one parameter, then we will need to select tuples of legal values for other input parameters for the web service as specified in phase (1) of the overall approach. The algorithm used for generating test cases of output parameters is illustrated in Figure 4. Since we cannot rely on the input annotations being correct, we would further filter out services for which many potential defects were located in the earlier testing phase.

Notice that this method for testing the annotations of outputs assumes that the annotations of the inputs of the web services used for testing have a strong semantics.

```
Inputs: $ps_o$ := output params to be tested
Outputs: $ts_o$ := empty test suite
Begin
For each operation parameter $\langle op, p\rangle \in ps_o$
    $os$ = output values for $\langle op, p\rangle$ from output log
    For each $ov$ in $os$
        Find all services $ss$ with input annotated
        with the same concept as $\langle op, p\rangle$
        Select tuple of legal values $lv$ for
        other input params of this operation
        $eo$ := expected outcome for $lv \cup ov$
        $tc$ := $\langle\langle op, p\rangle,\ lv \cup ov,\ eo\rangle$
        Add test case $tc$ to $ts_o$
End
```

Fig. 4. Algorithm for Generating Test Cases for Output Parameters

Phases 6 and 7: Executing and interpreting the results of the test cases for output parameters. The execution of the constructed test cases is much the same as for the input parameter tests (see the algorithm in Figure 5). Once the algorithm is executed, the results of execution of test cases are then analysed.

Phase 8: Diagnosing errors and correcting annotations. In this phase, the human curator examines the errors discovered to filter out errors that are due to factors other than acceptance or non acceptance of values by service operations. To amend incorrect annotations, the curator examines the ontology and chooses a different concept for annotation. In doing so, the verification results can be exploited to facilitate the curator's task. To illustrate how this can be done, consider that the annotation of an

```
Inputs: $ts_o$ := test suite for outputs
Outputs: $outputLog$ := the results of execution of test cases
Begin
For each $tc$ in $ts_o$
    $tc$ := $\langle\langle op, p_i\rangle,\ vs,\ eo\rangle$
    Execute operation $op$ with input params $vs$
    $rs$ := result returned from $op$
    If $op$ terminated normally then
        $ao$ := 'normal'
    Else
        $ao$ := 'abnormal'
    Add $\langle op, vs, rs, ao\rangle$ to output test log
End
```

Fig. 5. Algorithm for Executing Test Cases for Output Parameters

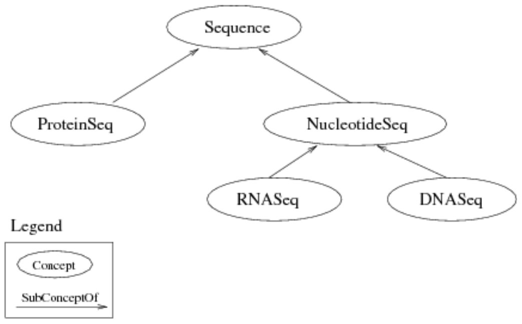


Fig. 6. A Fragment of the $^{\text{my}}$Grid Domain Ontology

input parameter i was found to be erroneous. Instead of examining the whole ontology to identify the correct annotation, the curator can focus on a typically much smaller subset of the ontology composed of concepts, the instances of which were found to be accepted by the input i.

Unfortunately, the above method of reducing the fragment of ontology displayed to the user to annotate the parameter cannot be applied for the annotation of output parameters. This is because, unlike the instances of inputs, the instances of outputs are generally not annotated.

4 Adequacy Criteria

The test generation method described in Section 3 describes a universe of possible test cases, divided into two kinds: those test cases that present valid inputs to operations and those that present invalid inputs.

Just like in software testing [18], to construct a test suite, we partition the domains of legal and illegal values into partitions. In doing so we use the ontology used for annotation. The concepts within an ontology are typically organised into inheritance hierarchies, giving a much finer grained breakdown of the space of instances than the valid/invalid division that we have considered so far in this paper.

This is illustrated in Figure 6, which shows a fragment of the $^{\text{my}}$Grid bioinformatics ontology [26]. Suppose that this ontology was used to annotate the single input parameter of the *SimilaritySearch* web service with the *NucleotideSeq* concept. This annotation could be incorrect in a number of ways. It could be an over-generalisation of the correct annotation if the web service is in fact only designed to search for similar DNA sequences and not RNA sequence. It could be an over-specialisation, if the web service is in fact able to search for similar sequences of a variety of sorts, including protein Sequences and sequences that are neither protein sequences nor DNA Sequences. Or, the web service could have been designed only to operate over protein sequences, in which case the annotation is simply incorrect.

Partitioning the input space for test generation according to the ontology classes holds out the promise of both increasing the diversity of the test suites that we generate and of providing more helpful information to the user regarding the pattern of failures and their possible causes in terms of annotation errors. In effect, the ontology provides us with a means of assessing structural coverage of test suites, thus leading to adequacy criteria, i.e., measures that can be applied to the test generation process for increasing the *representativeness* of the test suite constructed and, thus, increasing their error-detecting power. In the rest of this section, we present two adequacy criteria, that exploit the ontology used for annotation, to partition the domain of legal and illegal values of operation parameters.

4.1 Immediate-Sub-Superconcepts

Our previous work on inference and compatibility checking for semantic web services revealed that over-generalisation and over-specialisation of parameter annotations are both prevalent in practice [2]. This first adequacy criterion is designed to focus on the discovery and diagnosis of exactly this kind of error, by requiring that test suites include, for each parameter, test cases generated from:

- the semantic type c of the parameter (as defined by the current set of annotations),
- all immediate sub-concepts of c, and
- all immediate super-concepts of c.

4.2 All-Disjoint-Concepts

Errors that are not due to over-generalisation or over-specialisation in annotations are unlikely to be discovered using the partitioning strategy just described. For example, if an input parameter accepts instances that do not belong to the concept used for its annotation or to the immediate superconcepts of that concept then this annotation error will not be discovered using test cases generated by the *Immediate-Sub-Superconcepts* strategy.

Also, the extensions of the concepts obtained using the partitioning strategy just described are likely to overlap and, therefore, do not form a partition in the mathematical sense. For example, the domain of nucleotide sequences covers that of DNA sequences. This is not peculiar to our partitioning strategy. It is common, in functional testing, for the domains obtained by partitioning to overlap, e.g., this is often the case with domains obtained by path-coverage partitioning [13].

However, where partitions do overlap there is an increased possibility of generating redundant test cases. To force a more diverse and representative coverage of the space of instances, we can specify an adequacy criterion that requires the test suite to contain inputs from each concept in the ontology that is disjoint with the annotation concept being tested. The set of concepts to be covered in this way can be discovered by subtracting from the domain designated by a given concept, the instances that belong to its subconcepts and siblings[12] in the ontology. For example, using this method, partitioning the domain of *Sequence* results in the set of disjoint sub-domains designated by the following concepts:

- $Sequence \sqcap \neg(NucleotideSeq \sqcup ProteinSeq)$,
- $NucleotideSeq \sqcap \neg(DNASeq \sqcup RNASeq \sqcup ProteinSeq)$,
- $ProteinSeq \sqcap \neg NucletotideSeq$,
- $DNASeq \sqcap \neg RNASeq$,
- $RNASeq \sqcap \neg DNASeq$.

5 Real-World Evaluation

To assess the effectiveness of the verification method described in this paper, we run an experiment with the objective of measuring the defect-detecting power of the test suites generated using the partitioning strategies proposed. In doing so, we used as a base line for comparison, tests conducted using randomly generated test cases. Specifically, given semantic annotations of web services (which will be presented later), we created a test suite for those annotations using the All-Disjoint-Concepts strategy, and created a randomly generated test-suite of the same size. By randomly generated, we mean that the values of the input parameters in the test suite were selected randomly. Similarly, we created a test suite for verifying semantic annotations using the Immediate-Sub-Super-Concepts strategy, and created a randomly generated test-suite of the same size.

To assess the effectiveness of the partitioning strategies, we used the two following measures:

- **Increase in Recall**: we use this measure to compare the defect-detecting power of the test suite generated using the partitioning strategies, with that of a randomly generated test suite of the same size. Increase in recall is defined as the ratio of the number of true positive defects that are discovered using the partitioning strategy to the number of true positive defects that are detected using the randomly generated test suite of the same size. A value above 100% means that the test suite generated using the partitioning strategy performs better than the corresponding randomly generated test suite.
- **Precision**: False positive errors in annotations can be detected due to the difficulties in dealing with real web services, with their varying availability and non-standard termination routes. We used precision to estimate the negative effect that false positive defects may have on the verification method proposed. Precision is defined as the ratio of the number of true positive errors that are detected to the sum of the numbers of true positive and false positive errors. A higher precision means that fewer false positives are reported using the verification method.

12. By "siblings" here, we mean two concepts which share an immediate superclass.

Experiment Setup

A large number of bioinformatics web services are now publicly available, e.g., the ^{my}Grid toolkit provides access to over 3000 third party web services. A domain expert from the ^{my}Grid team annotated a number of these, and the results were made available through the Feta service registry [14]; 53 accessible web services were annotated at the time of writing[13]. These annotations were created in order to facilitate service discovery [14] and to guide workflow composition [15], and we therefore assumed a strong semantics when verifying the semantic annotations of input parameters. Interested readers can find information about the web service operations and semantic annotations that were used in the experiment online[14].

We created a pool of annotated instances using provenance logs supplied by the ^{my}Grid project, and used them to create test cases to verify all parameters of the applicable web services in Feta. In total, we tested 69 web service input parameters. In order to examine the effects of different partitioning strategies on the results of the test execution, we created four different test suites, using the following selection strategies from the initial instance pool:

(i) To assess the effectiveness of the All-Disjoint-Concepts strategy, we selected data from the pool using this partitioning strategy. This produced 1819 test cases.
(ii) As a baseline selection criterion, for comparison, we selected an input data set with the same number of instances as in case *(i)* but where the instances were randomly selected from the input pool. This produced 1819 test cases.
(iii) To assess the effectiveness of the Immediate-Sub-Superconcepts partitioning strategy, we selected data from the pool using this strategy. This produced 165 test cases (a substantial reduction on *(i)*).
(iv) We selected an input data set with the same number of instances as selected in case *(iii)* but where the instances were randomly selected from the input pool. This produced 165 test cases.

Test suites were then generated from these selected value sets, and the test cases executed.

13. The number of web services in the Feta service registry exceeds 53; however, many were unsuitable for use in our experiment since they either had some un-annotated parameters or else were no longer available at the endpoint specified by the WSDL files.
14. img.cs.man.ac.uk/quasar/experiment_info.php

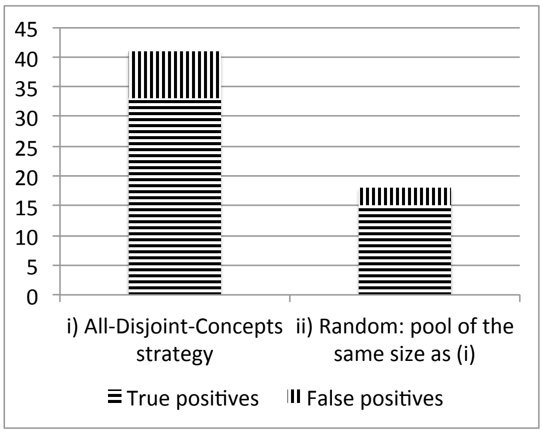


Fig. 7. Comparison of Defects Detected Using the All-Disjoint-Concepts and Random Strategies

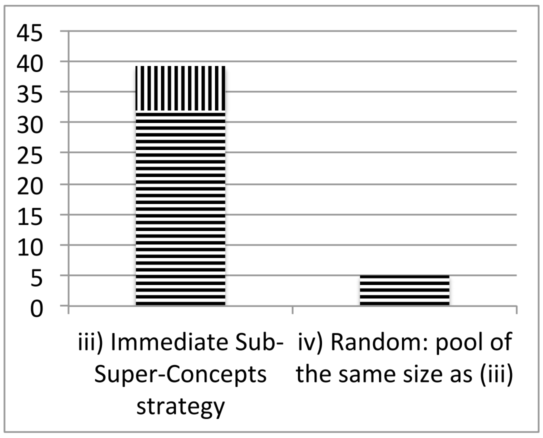


Fig. 8. Comparison of Defects Detected Using the Immediate-Sub-Superconcepts and Random Strategies

Results

For each test suite generated, we examined the results and classified them according to whether they correctly revealed an error (true positives), or incorrectly revealed an error (false positives). Figure 7 shows the results for the test suite generated using the All-Disjoint-Concepts partitioning strategy, and the corresponding randomly generated test suite of the same size, and Figure 8 shows the results for the test suite generated using the Immediate-Sub-Super-Concepts partitioning strategy, and the corresponding randomly generated test suite of the same size. To help analyze such results, Table 1 shows the precision of of the All-Disjoint-Concepts strategy, the precision of the Immediate-Sub-Super-Concepts strategy, as well as the increase in recall that we recorded in the results of the All-Disjoint-Concepts (resp. Immediate-Sub-Super-Concepts) strategy compared with the corresponding random testing strategy.

The results show that, in each case, random testing is significantly less effective than the more systematic approach based on ontology-based partitioning strategies (as suggested by the increase in recall see Table 1), thereby providing evidence in favour of our hypothesis regarding the usefulness of the ontology for partitioning the input domains. Moreover, all errors discovered by the random selection approaches were also discovered by the partitioning strategies. The results also show that the number of false positives is relatively small compared with the number of true positives. Indeed, both partitioning strategies yield a precision that is higher than 80%.

TABLE 1
Analysis of Immediate-Sub-Superconcepts Strategy Results.

Strategy	Precision	Increase in recall
All-Disjoint-Concepts	81%	220%
Immediate-Sub-Superconcepts	82%	640%

The results also show the effectiveness of Immediate-Sub-Superconcepts partitioning. A total of 32 true errors were discovered, a number which compares very favourably with the number of errors found by the All-Disjoint-Concepts partitioning strategy, despite the fact that our more selective partitioning strategy generated less than a tenth of the test cases of the All-Disjoint-Concepts strategy.

These results also show the effectiveness (in this example) of the Immediate-Sub-Superconcepts partitioning strategy in finding over-generalisation errors of this sort. This raised the question of whether the strategy was equally successful at detecting over-specialisation errors, i.e. where the concept used for annotation is a sub-concept of the correct annotation.

Since our collection of annotations did not contain many errors of this kind, we seeded 7 errors, that are representative of over-specialisation, in the annotations of the inputs. Specifically, we randomly picked 7 inputs parameters that were found not to be erroneous using the All-Disjoint-Concepts partitioning method. For each of those input, we modified its annotation by associating it with a concept c that is a direct sub-concept of the concept used for its annotations c. Where the concept c has more than one sub-concept, we randomly picked one of its sub-concepts and used it to annotate the input parameter in question. We then ran the Immediate-Sub-Superconcepts test cases again, and were able to discover 5 of the seeded errors. The remaining 2 errors were not located because we were unable to find instances for one of the partitions proposed by this strategy. Specifically, this was the concept: $DNASequence \sqcap \neg RNASequence$. This is partly due the fact that the definitions of the concepts $DNASequence$ and $RNASequence$ in the ontology used for annotation are not complete: no bioinformatics sequence can be both a DNA sequence and an RNA sequence; however, such a restriction was not specified in the ontology. As a result, even though the repository contains a DNA sequence, we were not able to use that instance to represent the partition: $DNASequence \sqcap \neg RNASequence$.

This kind of problem, i.e., incomplete definitions of concepts, is common in ontology specification [22].

It is worth noting that unlike the All-Disjoint-Concepts strategy, the Immediate-Sub-Superconcepts strategy did not manage to discover errors arising from the acceptance of instances that do not belong to the annotation. This can be explained by the fact that usually to uncover those errors, we need instances of concepts that are not neighbours of the concept used for annotation, but rather are disjoint concepts that are often not descendants or ascendants of the incorrect annotation.

As discussed earlier, to verify the annotations of outputs, we do not generate test data for the outputs using ontology-based partitioning strategies (since we do not have an oracle for detecting errors using generated test data). Instead, we use the data instances delivered by the outputs as a result of the operation invocations performed for testing the annotations of inputs. To test the annotations of outputs, we located, for each of them, a service operation having an input annotated using the same concept as the output under test. In the case where the service registry did not contain any operation with such an input, we located a service operation with an input annotated using a concept that is disjoint with the annotation of the output parameter under test. Using this method, we located 16 operations: 9 having an input with the same annotation as the output and 7 annotated using a concept that is disjoint with the output annotation. We then used the instances delivered by each output to feed the execution of the service operation located for it.

These test case numbers were small, especially when compared with the large number of test cases we had been able to generate for the inputs. Also, we were able to discover only one error in the annotation of the outputs by the execution of this test suite. The output of the operation *get_reaction_by_enzyme* was annotated using the concept *KeggRecord* whereas the instances it delivered were accepted by an input that is annotated using *PathwayReactionID*, which is disjoint with *KeggRecord*.

These poor results can be partly explained by the following observation. The instances used for testing the annotation of an output are not necessarily a good representation of the domain of values of the output: we did not generate the instances using a partitioning strategy, but instead relied on the instances delivered as a result of tests generated with input parameters in mind.

The above observation prompts us to elaborate and assess a new strategy for testing the annotations of output parameters. The proposed strategy exploits the fact that many service operations are used within workflows that produce provenance traces containing instances value of operation outputs. Those readily available values are obtained over time as a result of multiple executions of workflows, and are more likely to yield sets of values that are better representations of the domain of outputs (compared with the case in which we only rely on the output values obtained as a result of testing operation inputs). Of the 53 service operations under test, 19 were used in the composition of workflows in the myExperiment repository[15]. Those 19 operations had in total 27 output parameters. We executed those workflows using as many legal input values as we could harvest, either from the workflow descriptions in myExperiment or from the authors or users of the workflows. On average, we executed each workflow 24 times using different legal input values (we did not use our partitionining method since our objectve is not test the input but the ouput parameters). We then collected provenance traces of those workflows, which contained the instance values produced by the service operation. To test the annotations of the output of those operations, we used the same method as earlier, i.e., by feeding their values to service operations the inputs of which have the same annotation as the operation output subject to test. Using this method, we were able to identify 11 additional errors in the annotation of output parameters. The errors identified resulted from over-specialization in annotations. For example, the curator described the output of the operation $getUNIPROTEntry$ as a protein sequence, whereas that operation was also able to produce protein records as well as protein sequences.

To summarise, the experimental evaluation reported in this section showed that:

- Real world semantic annotations suffer from errors. Almost half of tested inputs were found to have erroneous annotations.
- Partitioning strategies allow a more effective discovery of errors in the annotations of inputs compared with randomly generated test data. The precision of of the tests conducted using the partitioning strategies is higher than 80%, and their effectiveness in terms of recall is much higher than random testing.
- Regarding the annotation of output parameters, the empirical evaluation showed that the use of output values, that are obtained using input values selected based on the proposed partitioning strategies, is not an effective means for testing. Rather, other sources of information, e.g., the provenance traces produced by workflow executions seem to be more effective.

6 Related Work

In this section, we review proposals that are related to ours. In doing so, we structure the section into three subsections: *annotation*, *verification*, and *use*, representing the steps of the semantic annotation lifecycle presented in Section 3.1.

6.1 Web Service Annotation

Several tools have been proposed by the semantic web service community for assisting human curators in the annotation task. For example, Radiant [12] is an Eclipse

15. `http://www.myexperiment.org`

plug-in based tool that provides a graphical user interface for annotating web services, by decorating the web service WSDL document with concepts from domain ontologies. To facilitate the annotation task, the authors of Radiant, developed Meteor-S [20], a tool that semi-automatically annotates web services by suggesting to the user concepts from domain ontologies that can be used for annotation. At the heart of the tool, there are schema matching and machine learning techniques that automatically identify a set of candidate concepts that can be used for the annotation of web services. APIHUT [11] is a tool for the annotation, indexing and discovery of Web APIs, e.g., Rest Web Services. Using APIHUT, the facets of a web service are automatically determined by comparing the description and tags associated with the web services, with the vectors of terms characterizing the facet's values using the cosine similarity metric. Built on APIHUT, KINO [21] is a generic tool for annotating web APIs, including Rest web services. The main difference between APIHUT and KINO, is that the latter allows users to utilize the ontology of their choice for annotation. Furthermore, Kino automatically identifies the ontologies (and concepts) that can be used for annotation.

We have also shown in a previous proposal [2] how information about semantic annotations of web service parameters can be inferred based on their connections to other (annotated) operation parameters within workflows. The idea behind this work is that if an input parameter is connected to an output parameter within a workflow and that the connection is free from mismatch, then the semantic domain of the output is a sub-concept of the semantic domain of the input. This allows inferring information about the semantic annotation of non annotated parameters from the semantic annotation of the parameters, to which they are connected to within workflows.

We regard our work on the verification of semantic annotations of web services, as complementary to the above proposals. While the above proposals focus on assisting the human user in annotating web services, the method proposed in this paper can be used as a systematic means to assess the accuracy of annotations before they are deployed for public use. This is illustrated in the semantic annotation lifecycle in Figure 1.

6.2 Semantic Annotation Verification

To the best of our knowledge, there are no proposals in the semantic web service literature for verifying service annotations. Instead, semantic annotations are used by a handful of proposals as inputs for specifying the test suites used for verifying web services' behaviour [19]. That is, semantic annotations are regarded as being the correct description of the service's behaviour, and the test cases created are intended to reveal defects in the service. For example, Tsai *et al.* propose to extend the WSDL document with semantic annotations that describe, among other things, the dependencies between the web service inputs and outputs, and to use these annotations for specifying the test suites used for verifying web services [23]. Other proposals use semantic annotations to describe service composition, and use these descriptions as input to test the resulting composite service [24]. For example, Wang *et al.* use OWL-S to define workflows specifying composite services [24]. To test a composite service, they generate test cases that exercise all possible paths in its associated workflow.

The above proposals assume that semantic annotations accurately specify web services' behaviour. This is rather a strong assumption that does not always hold in practice, as shown by the experimental results in the previous section. In practice, a web service is often a black box annotated after its deployment by third party annotators that have been involved with neither the design nor the implementation of the service. The method presented in this paper is therefore meant to help these annotators in inspecting their annotations for errors.

Among the existing tools that assist human users in functional testing of web services, there is SoapUI [8]. It is a toolkit that provides the means for specifying and executing test suites to verify the functionality of given web service operations. Although more targeted toward testing the functionality of web services, SoapUI can be used together with QuASAR, the tool that implements the verification method presented in this paper. QuASAR provides functionalities for i) generating test cases, ii) executing web services using the input specified in the test cases, and iii) analysing execution results. The second step (ii) (which corresponds to phases 2 and 3 in the verification process illustrated in Figure 1) can be performed by SoapUI, since it is built to support large numbers of web service calls.

6.3 Uses of Semantic Annotations

Semantic annotations of web services can be used by tools to assist in service discovery and composition [20], [21], [7], [1]. For example, Metor-S [20] provides a selection service that locates web services based on their semantic annotations. Kino [21] also provides the means for locating web services. In Kino, Web APIs (including web services) are indexed based on their semantic annotations. The resulting indexes are then used to answer user queries.

Semantic annotations of web services are also used to guide the composition of scientific workflows [7], [25]. For example, Dhamanaskar *et al.* developed a service suggestion engine [7] that is interfaced with the scientific workflow platform Galaxy. The service suggestion engine guides the composition of workflows by providing the designer with a list of web services that can be used to complete the workflow being designed. The web services are retrieved based on two criteria: i) the functionality implemented by the service: the designer indicates the task that the web service should fullfil, and ii) compatibility in the semantic types between the output of the preceding web service in the workflow and the suggested web service. Semantic annotations of web service parameters are used to check the second condition. Similarly, in previous work [1], we have developed a plugin for the Taverna workflow system, to check that connected input and output parameters are compatible within a workflow. In doing, so we made use of semantic annotations of web services.

It is worth mentioning that semantic annotations of web service parameters are partial descriptions of web services in the sense that they do not describe the transformations carried out by the web service. Yet, they are crucial in service discovery and guiding workflow composition. For example, to guide workflow composition, they are used to suggest web service operations with parameters that are semantically compatible with those in the (incomplete) workflow. The list of service operations retrieved can then be examined by the workflow designer to identify the service operations that are suitable for the task at hand. Where information about the task carried out by the suggested service operations is available, then it can be used to filter out those services operations that do not implement the task required by the workflow designer, e.g., [7].

In summary, the method for verifying semantic annotation of web services that we presented in this paper is complementary to existing proposals in the literature. It closes the gap in the web service annotation lifecycle, by providing a systematic means for assessing the validity of semantic annotations specified using tools such as Meteor-S and Radiant, before the annotations are deployed for public use in web service discovery and composition.

7 Conclusions

In this paper, we have described a first step towards providing low-cost tools to assist annotators in verifying the semantic descriptions they create before they are made publicly accessible. The approach has been implemented within the QuASAR toolkit, and is available as an open source system[16].

Despite the complications of working with real web services, which do not conform to the standard termination models and which are subject to the exigencies of a dynamic and unpredictable communications network, our approach to checking annotation semantics by means of operation acceptance proved to be surprisingly effective as far as annotations for input parameters are concerned. In addition, adequacy criteria based on the use of the ontology for partitioning demonstrated a clear ability to improve the defect-detection power of tests sets when compared with randomly selected suites of the same size.

16. `http://img.cs.manchester.ac.uk/quasar`

References

[1] K. Belhajjame, S. M. Embury, and N. W. Paton. On characterising and identifying mismatches in scientific workflows. In *3rd DILS 06*, pages 240–247. Springer, 2006.

[2] K. Belhajjame, S. M. Embury, N. W. Paton, *et al.*. Automatic Annotation of Web Services Based on Workflow Definitions. *TWeb*, 2(2):1–34, 2008.

[3] M. Brambilla, S. Ceri, F. M. Facca, I. Celino, D. Cerizza, and E. D. Valle. Model-driven design and development of semantic web service applications. *ACM Trans. Internet Techn.*, 8(1):1–31, 2007.

[4] M. H. Burstein, J. R. Hobbs, O. Lassila, *et al.*. DAML-S: Web Service Description for the Semantic Web. In *ISWC*, pages 348–363, London, UK, 2002. Springer-Verlag.

[5] J. Cardoso and A. P. Sheth, editors. *Semantic Web Services, Processes and Applications*. Springer, 2006.

[6] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, ACM, 2008.

[7] A. Dhamanaskar, M. E. Cotterell, J. Zheng, *et al.* Suggestions for Galaxy Workflow Design Using Semantically Annotated Services. In *FOIS*, 2012.

[8] EVIWARE. SoapUI; the Web Services Testing tool. http://www.soapui.org/. accessed 2012-11-07.

[9] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Services. In *ASE*, pages 152–161. IEEE, 2003.

[10] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specification. *Software Engineering Notes*, 24(6):146–162, November 1999.

[11] K. Gomadam, A. Ranabahu, M. Nagarajan, *et al.*. A Faceted Classification Based Approach to Search and Rank Web APIs. In *ICWS*, IEEE, 2008.

[12] K. Gomadam, K. Verma, and D. Brewer, *et al.*. Radiant: A tool for semantic annotation of Web Services. In *ISWC*, (Demo Paper), 2005.

[13] Richard G. Hamlet. Theoretical comparison of testing methods. In *Symposium on Testing, Analysis, and Verification*, pages 28–37. ACM, 1989.

[14] P. W. Lord, P. Alper, C. Wroe, and C. A. Goble. Feta: A light-weight architecture for user oriented semantic service discovery. In *ESWC*, pages 17–31. Springer, 2005.

[15] P. W. Lord, S. Bechhofer, M. D. Wilkinson, *et al.*. Applying semantic web services to bioinformatics: Experiences gained, lessons learnt. In *ISWC*, pages 350–364. Springer, 2004.

[16] E. M. Maximilien and M. P. Singh. A framework and ontology for dynamic web services selection. *IEEE Internet Computing*, 8(5):84–93, 2004.

[17] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *VLDB Journal*, 12(4):333–351, 2003.

[18] B. Meyer. Seven Principles of Software Testing. *IEEE Computer*, 41(8): 99-101 (2008)

[19] G. Oghabi, J. Bentahar, and A. Benharref. On the Verification of Behavioral and Probabilistic Web Services Using Transformation. In *ICWS*, 548–555, IEEE Computer Society, 2011.

[20] A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma. Meteor-s web service annotation framework. In *WWW*, pages 553–562. ACM, 2004.

[21] A. Ranabahu, P. Parikh, M. Panahiazar, *et al.* Kino: A Generic Document Management System for Biologists Using SA-REST and Faceted Search. In *ICSC*, IEEE, 2011.

[22] A. L. Rector, N. Drummond, M. Horridge, *et al.*. Owl pizzas: Practical experience of teaching owl-dl: Common errors & common patterns. In *EKAW*, pages 63–81. Springer, 2004.

[23] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang. Extending wsdl to facilitate web services testing. In *HASE '02*, page 171. IEEE, 2002.

[24] Y. Wang, X. Bai, Juanzi Li, and R. Huang. Ontology-Based Test Case Generation for Testing Web Services. In *ISADS'07*. IEEE, 2007.

[25] D. Withers, E. A. Kawas, E. L. McCarthy, *et al.*. Semantically-Guided Workflow Construction in Taverna: The SADI and BioMoby Plug-Ins. In *ISoLA* (1), pages 301–312, 2010.

[26] C. Wroe, R. Stevens, C. A. Goble, *et al.*. A suite of daml+oil ontologies to describe bioinformatics web services and data. *Int. J. Cooperative Inf. Syst.*, 12(2):197–224, 2003.