Automatic Annotation of Web Services based on Workflow Definitions

Khalid Belhajjame, Suzanne M. Embury, Norman W. Paton, Robert Stevens, and Carole A. Goble

School of Computer Science
University of Manchester
Oxford Road, Manchester, UK
{khalidb,sembury,norm,rds,carole}@cs.man.ac.uk

Abstract. Semantic annotations of web services can facilitate the discovery of services, as well as their composition into workflows. At present, however, the practical utility of such annotations is limited by the small number of service annotations available for general use. Resources for manual annotation are scarce, and therefore some means is required by which services can be automatically (or semi-automatically) annotated. In this paper, we show how information can be inferred about the semantics of operation parameters based on their connections to other (annotated) operation parameters within tried-and-tested workflows. In an open-world context, we can infer only constraints on the semantics of parameters, but these *loose annotations* are still of value in detecting errors within workflows, annotations and ontologies, as well as in simplifying the manual annotation task.

1 Introduction

Semantic annotations of web services have several applications in the construction and management of service-oriented applications. As well as assisting in the discovery of services relevant to a particular task [7], such annotations can be used to support the user in composing workflows, both by suggesting operations that can meaningfully extend an incomplete workflow [3] and by highlighting inappropriate operation selections [1, 9]. As yet, however, few usable semantic annotations exist. Manual annotation is a time-consuming process that demands deep domain knowledge from individual annotators, as well as consistency of interpretation within annotation teams. Because of this, the rate at which existing services are annotated lags well behind the rate of development of new services. Moreover, stable shared ontologies are still comparatively rare, with the result that the annotations produced by one community may be of limited value to those outside it.

Since resources for manual annotation are so scarce and expensive, some means by which annotations can be generated automatically (or semi-automatically) is urgently required. This has been recognised by a handful of researchers, who have proposed mechanisms by which annotations can be learnt or inferred. Heß *et al.* have designed a tool called ASSAM [4], which uses text classification techniques to learn new semantic annotations for individual web services from existing annotations [5]. The tool extracts a set of candidate concepts, from which the user selects the correct annotation. Patil *et al.*, taking inspiration from the classic schema matching problem [10], have constructed a framework for automatically matching WSDL elements to ontology concepts based on their linguistic and structural similarity [12]. The framework was then adapted to make use of machine learning classification techniques in order to select an appropriate domain ontology to be used for annotation [11]. Most recently, Bowers *et al.* have proposed a technique by which the semantics of the output of a service operation can be computed from information describing the semantics of the operation's inputs and a query expression specifying the transformation it performs [2].

All the above proposals attempt to derive new annotations based on the information present in existing annotations. In this paper, we explore the potential uses of an additional source of information about semantic annotations: namely, repositories of trusted data-driven workflows. A workflow is a network of service operations, connected together by data links describing how the outputs of the operations are to be fed into the inputs of others. If a workflow is known to generate sensible results, then it must be the case that the operation parameters that are connected within the workflow are compatible with one another (to some degree). In other words, if one side of a data link is annotated, we can use that information to derive annotation information for the parameter on the other side of the link. Or, if both sides are annotated, we can compare their annotations for compatibility and thus detect errors and inconsistencies in their manually-asserted semantics.

The remainder of the paper is organised as follows. We begin (in Section 2) by formally defining the concept of a data-driven workflow, and the notion of *compatibility* between connected parameters in such workflows. We then discuss how far we can use the information contained within a set of tested workflows in order to automatically derive annotations, and present the derivation algorithm (Section 3). As we shall show, we cannot derive exact annotations using this approach, but it is possible to derive a looser form of annotation which indicates a superset of the concepts that describe the parameters' semantics. We go on to demonstrate that these *loose annotations* have utility, despite their imprecise nature, by showing how they can be used to determine the compatibility of connected service parameters during workflow composition, as well as cutting down the search-space for manual annotators (Section 4). We present a prototype annotation tool that derives loose annotations from workflows (Section 5), and present the results of applying the tool to a collection of real biological workflows and annotations, which show the practicality of our approach (Section 6). Finally, we close the paper by discussing our ongoing work (Section 7).

2 Parameter Compatibility in Data-Driven Workflows

A data-driven workflow is a set of operations connected together using data links. Thus, for our purposes, we regard a data-driven workflow as a triple *swf*

$= \langle$*nameWf, OP, DL*$\rangle$, where *nameWf* is a unique identifier for the workflow, *OP* is the set of operations from which the workflow is composed, and *DL* is the set of data links connecting the operations in *OP*.

Operations: an operation $op \in OP$ is a quadruple $\langle$*nameOp, loc, in, out*$\rangle$, where *nameOP* is the unique identifier for the operation, *loc* is the URL of the web service that implements the operation, and *in* and *out* are sets representing the input and output parameters of the operation, respectively.

Parameters: an operation parameter specifies the data type of an input or output, and is a pair $\langle$*nameP, type*$\rangle$, where *nameP* is the parameter's identifier (unique within the operation) and *type* is the parameter's data type. For web services, parameters are commonly typed using the XML Schema type system, which supports both simple types (such as *xs:string* and *xs:int*) and complex types constructed from other simpler ones.

Data links: a data link describes a data flow between the output of one operation and the input of another. Let *IN* be the set of all input parameters of all operations present in the workflow *swf*, i.e. $IN \equiv \{\ i \mid i \in in\ \wedge\ \langle \_, \_, in, \_\rangle \in OP\}$. Similarly, let *OUT* be the set of output parameters present in *swf*, i.e. $OUT \equiv \{\ o \mid o \in out\ \wedge\ \langle \_, \_, \_, out\rangle \in OP\}$. The set of data links connecting the operations in *swf* must then satisfy:

$$DL \subseteq (OP \times OUT) \times (OP \times IN)$$

Notation: in the remainder of this paper, we will use the following notation:

- *SWF* is the set of tested workflows given as input to the annotation process.
- *OPS* is the set of all operations used in *SWF*, i.e. $OPS = \{\ op \mid op \in OP\ \wedge\ \langle \_, OP, \_\rangle \in SWF)\}$
- *DLS* is the set of all data link connections in *SWF*, i.e. $DLS = \{\ dl \mid dl \in DL\ \wedge\ \langle \_, \_, DL\rangle \in SWF\}$.
- *INS* is the set of all input parameters appearing in *SWF*, i.e. $INS = \{\ i \mid i \in in\ \wedge\ \langle \_, \_, in, \_\rangle \in OPS\}$.
- *OUTS* is the set of all input parameters appearing in *SWF*, i.e. $OUTS = \{\ o \mid o \in out\ \wedge\ \langle \_, \_, \_, out\rangle \in OPS\}$.

2.1 Parameter Compatibility

If a workflow is well-formed then we can expect that the data links within it will link only those parameters that are compatible with one another. In its simplest form, this means that the two parameters must have compatible data types (as described within the WSDL description file for web service operations). However, when services are semantically annotated, it is also necessary to consider the semantic compatibility of connected parameters. Exactly what this means will depend on the form of annotation used to characterise parameter semantics, although the basic principles should be the same in most cases.

For the purposes of this paper, we will consider a particular form of semantic annotation that was developed within the ISPIDER project[1], to facilitate the identification and correction of parameter mismatches in scientific workflows [1]. In ISPIDER, semantic annotations are based upon three distinct ontologies, each of which describes a different aspect of parameter semantics and each of which is defined using the Web Ontology Language (OWL) [8]. These are the *Domain Ontology*, the *Representation Ontology* and the *Extent Ontology*.

The Domain Ontology describes the concepts of interest in the application domain covered by the operation. This is the commonest form of semantic annotation for services, and several domain ontologies have been developed for different application domains. An example is the ontology that was created with the $^{\mathrm{my}}$Grid project, that describes the domain of bioinformatics [13]. Typical concepts in this ontology are *ProteinSequence* and *ProteinRecord*.

Although useful for service discovery, the Domain Ontology is not sufficient by itself to describe parameter compatibility within workflows, hence the need for the two additional ontologies. The first of these, the Representation Ontology, describes the particular representation format expected by the parameter. In an ideal world, the data type of the parameter would give us all the information required about its internal structuring. Unfortunately, however, it is extremely common for the parameters of real web services to be typed as simple strings, on the assumption that the operations themselves will parse and interpret their internal components. This is partly a legacy issue (for older services) but it is also partly caused by the weak type systems offered by many current workflow management systems, which do not encourage web service authors to type operation parameters accurately. Because of this, to determine parameter compatibility, it is necessary to augment the information present in the WSDL data types with more detailed descriptions of the representation formats expected, using concepts from the Representation Ontology. An ontology of this kind for molecular biology formats has already been developed under the aegis of the $^{\mathrm{my}}$Grid project [13], containing such concepts as *UniprotRecord*, which refers to a well known format for representing protein sequences, and *UniprotAC*, which refers to the accession number format dictated by the *Uniprot* database.

The final annotation ontology that we use is the *Extents Ontology*, which contains concepts describing the scope of values that can be taken by some parameter. Although in general it is not possible to accurately describe the extents of all parameters, in some cases this information is known. For example, the TrEMBL database[2] is known to contain information about a superset of the proteins recorded in the SwissProt database[3], and there are several species-specific gene databases that are known not to overlap. Information about the intended extents of parameters can help us to detect incompatibilities of scope in workflows that would otherwise appear to be well-formed. An example concept

[1] http://www.ispider.man.ac.uk/
[2] http://www.ebi.ac.uk/trembl/
[3] http://www.ebi.ac.uk/swissprot

from the Extent Ontology is *UniprotDatastore*, which denotes the set of protein entries stored within the *Uniprot* database.

In order to state the conditions for parameter compatibility in terms of these three ontologies, we assume the existence of the following functions for returning annotation details for a given parameter

$$\begin{aligned} \textit{domain: } & \textit{OPS} \times (\textit{INS} \cup \textit{OUTS}) \rightarrow \theta_{domain} \\ \textit{represent: } & \textit{OPS} \times (\textit{INS} \cup \textit{OUTS}) \rightarrow \theta_{represent} \\ \textit{extent: } & \textit{OPS} \times (\textit{INS} \cup \textit{OUTS}) \rightarrow \theta_{extent} \end{aligned}$$

where θ_{domain} is the set of concepts in the Domain Ontology, $\theta_{represent}$ the set of concepts in the Representation Ontology and θ_{extent} the set of concepts in the Extent Ontology. We also assume the existence of the function *coveredBy()* for comparing extents (since the standard set of OWL operators aren't sufficient for reasoning with Extent Ontology concepts). Given two extents *e1* and *e2*, the expression *coveredBy(e1, e2)* has the value *true* if the space of values designated by *e1* is a subset of the space of values designated by *e2* and *false* otherwise.

Parameter compatibility: Let *(op1,o,op2,i)* be a data link connecting the output parameter *o* of the operation *op1* to the input parameter *i* of the operation *op2*. The parameters *op1.o* and *op2.i* are compatible iff[4]:

(i) *o.type* $\preceq$ *i.type*: the data type of the output *op1.o* is a subtype of the data type of the input *op2.i*; and
(ii) *domain(op1,o)* $\subseteq$ *domain(op2,i)*: the semantic domain of *op1.o* is a subconcept of *op2.i*'s domain; and
(iii) *represent(op1,o)* = *represent(op2,i)*: the output and input parameters adopt the same representation; and
(iv) *coveredBy(extent(op1,o),extent(op2,i))*: the extent of *op1.o* is contained within the extent of *op2.i*.

3 Deriving Parameter Annotations

In addition to using the rules for parameter compatibility to test a workflow for errors, we can also use them in a generative way to infer information about the semantics of linked parameters in workflows that the user believes to be error-free. We will use a simple example to illustrate this idea. Consider the pair of workflows shown in Figure 1. Both these workflows are intended to perform simple similarity searches over biological sequences. The first finds the most similar protein to the one specified in the input parameter. To do this, it retrieves the specified protein entry from the *Uniprot* database, runs the *Blast* algorithm to find similar proteins, and then extracts the protein with the highest similarity score from the resulting *Blast* report. The second workflow finds similar sequences to a given DNA sequence. It retrieves the DNA sequence from the DDBJ database[5], searches for similar sequences using Blast and finally extracts the sequences of all matches from the *Blast* report.

[4] The symbol $\preceq$ stands for a subtype of, and the symbol $\subseteq$ for a subconcept of.
[5] http://www.ddbj.nig.ac.jp

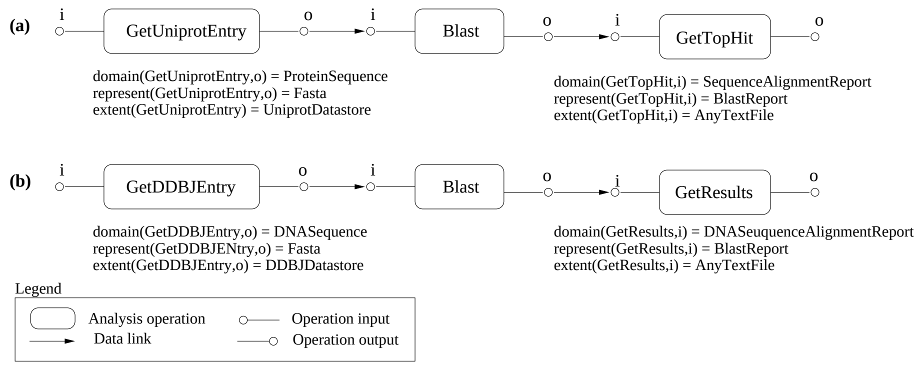

(a)
i
GetUniprotEntry
o
i
Blast
o
i
GetTopHit
o
domain(GetUniprotEntry,o) = ProteinSequence
represent(GetUniprotEntry,o) = Fasta
extent(GetUniprotEntry) = UniprotDatastore
domain(GetTopHit,i) = SequenceAlignmentReport
represent(GetTopHit,i) = BlastReport
extent(GetTopHit,i) = AnyTextFile
(b)
i
GetDDBJEntry
o
i
Blast
o
j
GetResults
o
domain(GetDDBJEntry,o) = DNASequence
represent(GetDDBJENtry,o) = Fasta
extent(GetDDBJEntry,o) = DDBJDatastore
domain(GetResults,i) = DNASeuquenceAlignmentReport
represent(GetResults,i) = BlastReport
extent(GetResults,i) = AnyTextFile
Legend
Analysis operation
Data link
Operation input
Operation output

Fig. 1. Example workflows

Notice that, in this simple example, the parameters of the *Blast* operation have not been annotated, while the parameters of the other operations have. However, since these are thoroughly tested workflows, their data links must all be compatible and we can therefore infer some information about the annotations that the *Blast* operation ought to have. For example, if we focus on just the domain annotations, we can see that the input *Blast* parameter must be compatible with both *ProteinSequence* and *DNASequence*, since parameters conforming to both these concepts are connected to it. In fact, by the rules of compatibility, we can infer that:

$$(\mathit{ProteinSequence} \cup \mathit{DNASequence}) \;\subseteq\; \mathit{domain}(\mathit{Blast}, i)$$

Unfortunately, we cannot infer the exact annotation, as we may not have been given a complete set of workflows (by which we mean a set of workflows that contains every possible connection of compatible parameters). All we can safely do is infer a lower bound on the annotation of the input parameters and an upper bound on the annotation of the output parameters. Thus, in the case of the *Blast* input parameter, we can use the derived lower bound just given to indicate the fragment of the ontology that must contain its true domain annotation (shown in Figure 2)—in this case, all the super-concepts of the union of *ProteinSequence* and *DNASequence*[6].

We call these lower and upper bounds *loose annotations*, to distinguish them from the more usual (*tight*) form of annotation in which the exact concept corresponding to the semantics of the parameter is given. All manually asserted annotations at present are tight annotations (though in the future users may

[6] The ontology fragment shown in Figure 2 does not contain the lower bound concept $\mathit{ProteinSequence} \cup \mathit{DNASequence}$, since it is not a (named) concept within the ontology. However, since OWL language allows the formation of new concepts using, amongst others, the union and intersection operators, the true annotation may in fact be the lower bound itself (i.e. $\mathit{ProteinSequence} \cup \mathit{DNASequence}$). Other, less expressive, ontology languages such as RDFS, do not allow this possibility.

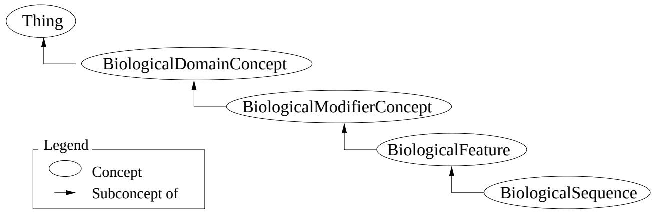


Fig. 2. Fragment of the domain ontology

prefer to assert loose annotations for difficult cases where they are unsure of the correct semantics).

Based on this reasoning, we can derive a method for inferring loose annotations for operation parameters, given a set of tested workflows *SWF* and a set of (loose or tight) annotations for some subset of the operations that appear in *SWF*. Since the compatibility relationship between input and output parameters is not symmetrical, we must use a different method for deriving input parameter semantics from that used for deriving output semantics.

3.1 Derivation of Input Parameter Annotations

Given an input parameter of some operation, we can compute three sets of loose annotations, based on the compatibility rules for each of the three annotation ontologies, as follows.

– *getInputDomains:* $OPS \times INS \rightarrow \mathcal{P}(\theta_{domain})$
This function computes a loose domain annotation, by locating the subset of the ontology that must contain the correct annotation. It first finds all operation outputs that are connected to the given input in *SWF*. It then retrieves the domain annotations for these outputs, unions them and returns all super-concepts of the resulting new concept.

– *getInputRepresentation:* $OPS \times INS \rightarrow \theta_{represent}$
This function computes a representation annotation. Since we assume that each parameter can support only one representation format, we can infer a tight representation annotation for the input parameter, rather than a loose one. To do this, we first find all output parameters that are connected to the given input, and retrieve their representations from the annotation repository. If all the output parameters have the same representation, then this can be returned as the derived annotation for the input parameter. Otherwise, a null result should be returned and the conflict should be flagged to the user. In our example (Figure 1), the representation annotation that is inferred for the *Blast* input parameter is *Fasta*.

– *getInputExtents:* $OPS \times INS \rightarrow \mathcal{P}(\theta_{extent})$
This function computes a loose extent annotation, by locating the fragment of the extent ontology that must contain the correct annotation. It first finds all output parameters that are connected to the input by workflows in *SWF*, and

then retrieves their extent annotations. Finally, it searches the Extent Ontology for all extents known to cover the union of the retrieved extents, and returns the resulting set of concepts. In our example, the extent of the *Blast* input parameter is an extent which covers the union of *UniprotDatastore* and *DDBJDatastore*, if one exists.

3.2 Derivation of Output Parameter Annotations

Derivation of annotations for output parameters follows much the same pattern as for input parameters, except that we infer upper bounds on their semantics rather than lower bounds.

– *getOutputDomains:* $OPS \times OUTS \rightarrow \mathcal{P}(\theta_{domain})$
This function computes a loose domain annotation for the given output parameter. It first finds all input parameters that are connected to it in the workflows in *SWF*, and retrieves their domain annotations. It then returns all domain concepts that are subconcepts of the intersection of the retrieved concepts. In our example, the output parameter of the *Blast* operation must be a subconcept of $(SequenceAlignmentReport \cap ProteinSequenceAlignmentReport)$. Since, according to the domain ontology, the second of these two concepts is a subconcept of the first, this can be simplified to: $domain(Blast, o) \subseteq ProteinSequenceAlignmentReport$.

– *getOutputRepresentation:* $OPS \times INS \rightarrow \theta_{represent}$
As with the inference of input representation annotations, the representation of an output parameter should be the same as that given for all connected inputs, provided there is no conflict. In our example, the annotation inferred for the *Blast* operation output parameter is *BlastReport*.

– *getOutputExtents:* $OPS \times OUTS \rightarrow \mathcal{P}(\theta_{extent})$
This function computes a loose extent annotation by locating the subset of the Extent Ontology that must contain the correct extent. It first finds all input parameters that are connected to the given output and retrieves their extent annotations. It then searches the Extent Ontology for all extents that are covered by the intersection of the retrieved extents, and returns the result. In our example, we can infer that the extent of the *Blast* operation output must be contained within the *AnyTextFile* extent.

3.3 Annotation algorithm

Given the functions for deriving annotations for individual parameters just described, we can construct an algorithm (shown in Figure 3) that will derive all annotations automatically from a set of tested workflows and an incomplete repository of semantic annotations. This algorithm iterates over the parameters present in the workflows, deriving new annotations for each of them using the functions given above. The resulting annotations are then examined by the subroutine presented in Figure 4. If there is no existing annotation for a parameter, then the derived annotation is asserted (i.e. entered into the annotation repository). If a manual annotation is already present, then this is compared with the derived annotation to check for any conflicts. If the two are compatible, then no

```
Algorithm DeriveAnnotations
inputs OPS
outputs OPS
begin
1     for each op ∈ OPS do
2         for each i ∈ op.in do
3             $C_{domain}$ := getInputDomains(op,i)
4             $c_{represent}$ := getInputRepresentation(op,i)
5             $C_{extent}$ := getInputExtents(op,i)
6             ExamineDerivedAnnotation(op,i,$C_{domain}$,$c_{represent}$,$C_{extent}$)
7         for each o ∈ op.out do
8             $C_{domain}$ := getOutputDomains(op,o)
9             $c_{represent}$ := getOutputRepresentation(op,o)
10            $C_{extent}$ := getOutputExtents(op,o)
11            ExamineDerivedAnnotation(op,o,$C_{domain}$,$c_{represent}$,$C_{extent}$)
end
```

Fig. 3. Annotation algorithm

further action need be taken. If not then the discrepancy should be flagged to the user.

Conflicts are detected in the following cases:

- **Domain conflict**: there exists a conflict in the domain semantics when a tight domain annotation does not belong to the subset of the domain ontology indicated by the derived (loose) annotation for the same parameter (*line 5*).
- **Representation conflict**: there exists a conflict in representation if the derived representation concept is different from the asserted representation concept for that parameter (*line 11*).
- **Extent conflict**: there exists a conflict in extent if the tight extent annotation does not belong to the subset of the extent ontology specified by the derived annotation for the same parameter (*line 17*).

There are several situations that can lead to conflicts, each of which requires a different corrective action.
– In the case of domain and extent conflicts, it may be that the manual and derived annotations are in reality compatible, but that an error in the ontology means that this compatibility cannot be detected by our algorithm. In this case, the problem may be corrected by adding new specialisation relationships to the ontology, until the annotations become compatible.
– One or more of the previously asserted annotations for the parameters involved in the conflict may be incorrect. Once the user is confident that the incorrect annotations have been identified, they can be deleted or refined to remove the conflict. However, since the problem parameter may be linked to many services in the workflow repository, determining exactly where the problem lies (i.e. with which parameter annotation) may require some detective work on the part of

```
Algorithm ActOnDerivedAnnotations
inputs (op,p) $\in$ (OPS $\times$ (INS $\cup$ OUTS),
       $C_{domain} \subseteq \theta_{domain}$, $c_{represent} \in \theta_{represent}$, $C_{extent} \subseteq \theta_{extent}$
outputs op $\in$ OPS
begin
1     **if** ($C_{domain} \neq \phi$) **then**
2         **if** (domain(op,p) = null) **then**
3             assertDomain(op,p,$C_{domain}$)
4         **else**
5             **if** (domain(op,p) $\notin C_{domain}$) **then**
6                 domainConflict(op,p,$C_{domain}$)
7     **if** ($c_{represent} \neq$ null) **then**
8         **if** (represent(op,p) = null) **then**
9             assertRepresentation(op,p,$c_{represent}$)
10        **else**
11            **if** (represent(op,p) $\neq c_{represent}$) **then**
12                representationConflict(op,p,$c_{represent}$)
13    **if** ($C_{extent} \neq \phi$) **then**
14        **if** (extent(op,p) = null) **then**
15            assertExtent(op,p,$C_{extent}$)
16        **else**
17            **if** (extent(op,p) $\notin C_{extent}$) **then**
18                extentConflict(op,p,$C_{extent}$)
end
```

Fig. 4. Algorithm for Acting on Derived Annotations

the user. If workflow provenance logs exist, then they can help in this process, since they would allow the user to examine the data values produced by or for the offending parameter during workflow execution. This may reveal the source of the error.
– One of the workflows involved in the conflict may not in fact have been thoroughly tested and may contain some connected parameters that are incompatible. It should be deleted from the workflow repository and the process of annotation derivation begun again from scratch.

4 Uses of Loose Annotations

The loose annotations derived by the method described in the preceding section contain considerably less information than conventional, tight annotations, and they are therefore correspondingly less useful. The question remains, therefore, as to whether the effort in collecting loose annotations is worthwhile. In this section, we demonstrate that loose annotations do have utility, despite their imprecise nature, by considering just two potential applications: the inspection of parameters ' compatibility in workflows and speeding up the process of manual annotation for unannotated service parameters.

4.1 Inspecting Parameter Compatibility in Workflows

One of the original aims of the three annotation ontologies made use of in this paper was to allow mismatched data links, i.e. data links connecting incompatible parameters, in workflows to be detected and flagged to the user for correction. However, this assumes that all annotations are tight. When we have the possibility of loose annotations also being present in the annotation repository, can we still detect parameter compatibility in workflows?

In fact, even with loose annotations, it is still possible to determine compatibility of parameters in the following cases. Let *op1* and *op2* be two linked operations, and *o* and *i* their respective output and input parameters. Suppose that loose annotations for both parameters *op1.o* and *op2.i* have been derived by the algorithm presented in the previous section. In this case, the parameters *op1.o* and *op2.i* are definitely compatible if:

(i) $o.type \preceq i.type$, and
(ii) $\forall\ c_i \in getOutputDomains(op1,o),\ \forall\ c_j \in getInputDomains(op2,i),\ c_i \subseteq c_j$, and
(iii) $represent(op1,o) = represent(op2,i)$, and
(iv) $\forall\ c_i \in getOutputExtents(op1,o),\ \forall\ c_j \in getInputExtents(op2,i),\ coveredBy(c_i,c_j)$.

If we compare these conditions with those for full parameter compatibility (based on tight annotations), we can see that conditions *(i)* and *(iii)* are unchanged. Conditions *(ii)* and *(iv)* have both been altered to take into account the presence of loose annotations. In the case of domain compatibility, for example, we require that all the concepts returned by *getOutputDomains(op1,o)* must be subconcepts of all the concepts returned by *getInputDomains(op2,i)*. This may well be a stronger condition for compatibility than is actually required, but it is conservatively true, given the information we have available in the loose annotations.

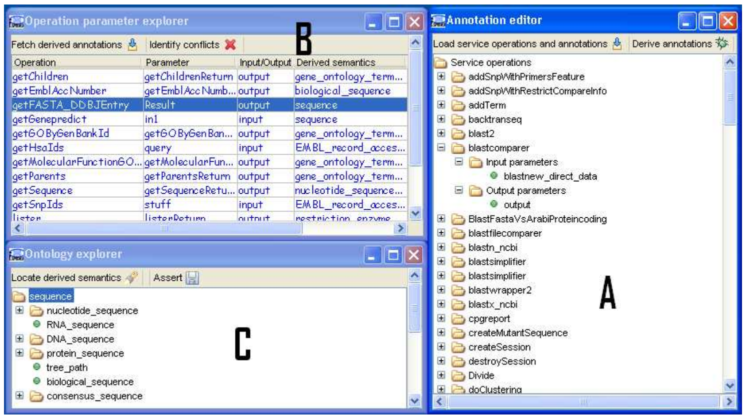
If the conditions given above are not satisfied, however, then we cannot say whether the parameters are compatible or not. We can still flag these connections to the user for their attention, but must allow the user to accept them as correct (i.e. compatible) based on their better knowledge of the real semantics of the parameters involved.

4.2 Supporting the Manual Annotator

Another application for loose annotations is in supporting human annotators in extending the repository of service annotations. If the user starts to annotate an operation parameter that has a loose annotation derived for it, then he or she only has to choose from the (hopefully small) subset of the ontology indicated by the loose annotation, rather than from the full set of ontology concepts. Where the ontology is large and/or complex, this can result in a significant time saving for the human annotator. For example, when specifying the domain semantics of the input parameter belonging to the *Blast* operation given in our earlier example, the user has only to choose from a collection of 5 concepts specified by the loose annotation, rather than the full 1556 concepts in the ^{my}Grid ontology. This also helps to avoid errors and inconsistencies in manual annotation.

5 Implementation

In order to assess the value of this method of deriving annotations, we have developed a prototype annotation tool that infers loose annotations and presents the results to the user through the GUI illustrated in Figure 5. The Annotation Editor, labelled A, shows the contents of the workflow repository being used for annotation derivation, and any existing (tight) annotations presently stored for the operation parameters in the annotation repository. This panel also contains the controls that launch the annotation derivation process.



Operation	Parameter	Input/Output	Derived semantics
getChildren	getChildrenReturn	output	gene_ontology_term...
getEmblAccNumber	getEmblAccNumb...	output	biological_sequence
getFASTA_DDBJEntry	Result	output	sequence
getGenepredict	in1	input	sequence
getGOByGenBankId	getGOByGenBan...	output	gene_ontology_term...
getHsaIds	query	input	EMBL_record_acces...
getMolecularFunctionGO...	getMolecularFun...	output	gene_ontology_term...
getParents	getParentsReturn	output	gene_ontology_term...
getSequence	getSequenceRetu...	output	nucleotide_sequence...
getSnpIds	stuff	input	EMBL_record_acces...
lister	listerReturn	output	restriction_enzyme

Fig. 5. Annotation system (GUI)

The resulting annotations are shown in the Operation Parameter Explorer panel (labelled B). Tight and loose annotations are distinguished here, and any conflicts will be highlighted. The final panel (labelled C) is the Ontology Explorer, which allows the user to view the fragments of the ontology indicated by a loose annotation, and to make a selection of a specific concept, to convert the loose annotation into a tight one.

6 Application to Bioinformatics Web Services

In order to further assess the value of the annotation derivation mechanism described here, we applied the algorithm and tool to a repository of workflows and annotations taken from the domain of bioinformatics. A large number of public web services are available in bioinformatics. For example, the ^{my}Grid toolkit provides access to over 3000 third party bioinformatics web services. The Taverna repository also contains 131 workflow specifications, and the ^{my}Grid web service registry, Feta [6] provides parameter annotations for 33 services[7].

[7] Note here how the number of annotations lags behind the number of available services.

We used these as inputs to our algorithm, and were able to derive 35 domain annotations for operation parameters, a selection of which are shown in Table 1. The concept given in the final column indicates either the upper bound (in the case of an output parameter) or the lower bound (in the case of an input parameter) derived by our algorithm. Upon analysis with the help of a domain expert, 18 of the derived annotations were found to be correct and 11 were found to be incorrect. A further 6 annotations could not be checked as the parameters in question belonged to services that have either moved or no longer exist, and thus could not be examined to determine the semantics of their parameters.

	Service operation	Provider[a]	Parameter	I/O	Derived concept
1	addTerm	EBI	geneOntologyID	I	GeneOntologyTermID
2	blastFileComparer	^{my}Grid	blastResult	I	BlastAlignmentReport
3	getFastaDDBJEntry	DDBJ	result	O	Sequence
4	getGenePredict	VBI	in0	I	Sequence
5	getHsaIds	^{my}Grid	query	I	EMBLAccessionNumber
6	blastx_ncbi	^{my}Grid	query_sequence	I	Sequence
7	lister	^{my}Grid	listerReturn	O	EnzRestReport ∩ DNASeq

[a] EBI stands for European Bioinformatics Institute, DDBJ for DNA Data Bank of Japan, and VBI for Virginia Bioinformatics Institute.

Table 1. Examples of derived parameter annotations

Of the 11 incorrect annotations, 3 were identified thanks to the conflicts automatically detected between the asserted and derived annotations. For example, the annotation given for the input parameter *query_sequence* of the *blastx_ncbi* operation states that it is a *NucleotideSequence*, while the derived annotation specified that it must be a superconcept of *Sequence* (row 6). According to the ^{my}Grid ontology, *NucleotideSequence* is not a super-concept of *Sequence*, hence the conflict. After diagnosis, the derived annotation was found to be incorrect. Two further errors were discovered when the derived loose annotation specifies an empty subset of the ontology, it is the case for the output *listerReturn* (row 7).

The remaining 6 incorrect derived annotations were not detected automatically by our tool, but were diagnosed when we investigated the derived annotations for correctness. They were all found to be due to either incorrect manual annotation or incompatibilities in the input workflows. Of the total 11 errors, 4 were found to be due to errors in the original annotations and 7 due to incompatibilities between connected parameters in the workflows.

This experiment showed that it is possible to derive a significant number of new annotations from even a small annotation repository. We were also able to detect 5 incorrect parameter annotations—quite a high number given the small scale of the inputs. However, the results also show that errors in workflows can lead to errors in derived annotations, and hence the importance of using only tried and tested workflows. This is not a problem where derived annotations can be examined for correctness by a user, but more care must be taken if they are to be created in a wholly automatic manner.

7 Conclusions

In this paper, we have presented an approach for automatically deriving semantic annotations for web service parameters. Our method improves over existing work in this area in that, in addition to facilitating the manual annotation task, it can also be used for examining the compatibility of parameters in workflows.

Our preliminary experiment has provided evidence in support of our annotation mechanism and shown its effectiveness and ability to discover an important number of annotations and help detecting mistakes in existing annotations based on a relatively small set of annotations. The next step is to evaluate the proposed techniques on a larger scale, and to explore their applications in supporting the annotation task more generally. For example, it may be possible to use collections of loose annotations to diagnose problems in ontology design, as well as in semantic annotations and workflows. There are also potential applications in guiding the work of teams of human annotators, to ensure that the most useful services are given priority during annotation.

References

1. K. Belhajjame, S. M. Embury, and N. W. Paton. On characterising and addressing mismatches in scientific workflows. In *International Workshop on Data Integration in the Life Sciences (DILS 06)*. Springer, 2006.
2. S. Bowers and B. Ludäscher. Towards automatic generation of semantic types in scientific workflows. In *WISE Workshops*, 2005.
3. J. Cardoso and A. P. Sheth. Semantic e-workflow composition. *J. Intell. Inf. Syst.*, 21(3), 2003.
4. A. Heß, E. Johnston, and N. Kushmerick. Assam: A tool for semi-automatically annotating semantic web services. In *ISWC*, 2004.
5. A. Heß and N. Kushmerick. Learning to attach semantic metadata to web services. In *ISWC*, pages 258–273, 2003.
6. P. W. Lord, P. Alper, Ch. Wroe, and C. A. Goble. Feta: A light-weight architecture for user oriented semantic service discovery. In *ESWC*, 2005.
7. E. M. Maximilien and M. P. Singh. A framework and ontology for dynamic web services selection. *IEEE Internet Computing*, 8(5), 2004.
8. D. L. McGuinness and F. v. Harmelen. Owl web ontology language overview. In *W3C Recommendation*, 2004.
9. B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *VLDB J.*, 12(4), 2003.
10. P. Mitra, G. Wiederhold, and M. L. Kersten. A graph-oriented model for articulation of ontology interdependencies. In *EDBT*, 2000.
11. N. Oldham, Ch. Thomas, A. P. Sheth, and K. Verma. Meteor-s web service annotation framework with machine learning classification. In *SWSWPC*, 2004.
12. A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma. Meteor-s web service annotation framework. In *WWW*, 2004.
13. Ch. Wroe, R. Stevens, C. A. Goble, A. Roberts, and R. M. Greenwood. A suite of daml+oil ontologies to describe bioinformatics web services and data. *Int. J. Cooperative Inf. Syst.*, 12(2), 2003.