Annotating the Behavior of Scientific Modules Using Data Examples: A Practical Approach

Khalid Belhajjame
PSL, Université Paris Dauphine, LAMSADE
75016 Paris, France
Khalid.Belhajjame@dauphine.fr

ABSTRACT

A major issue that arises when designing scientific experiments (i.e., workflows) is that of identifying the modules (which are often "black boxes"), that are suitable for performing the steps of the experiment. To assist scientists in the task of identifying suitable modules, semantic annotations have been proposed and used to describe scientific modules. Different facets of the module can be described using semantic annotations. Our experience with scientists from modern sciences such as bioinformatics, biodiversity and astronomy, however, suggests that most of semantic annotations that are available are confined to the description of the domain of input and output parameters of modules. Annotations specifying the behavior of the modules, as to the tasks they play, are rarely specified. To address this issue, we argue in this paper that data examples are an intuitive and effective means for understanding the behavior of scientific modules. We present a heuristic for automatically generating data examples that annotate scientific modules without relying on the existence of the module specifications, and show through an empirical evaluation that uses real-world scientific modules the effectiveness of the heuristic proposed.

The data examples generated can be utilized in a range of scientific module management operations. To demonstrate this, we present the results of two real-world exercises that show that: (i) Data examples are an intuitive means for human users to understand the behavior of scientific modules, and that (ii) data examples are an effective ingredient for matching scientific modules.

Categories and Subject Descriptors

H.0 [**Information Systems**]: General

General Terms

Algorithms, Experimentation

Keywords

Data example, scientific module, module annotation, module comparison, scientific workflow, workflow decay.

1. INTRODUCTION

We have recently recorded a dramatic increase in the number of scientists who utilize scientific modules, which are programs that are hosted either remotely, e.g., as web and grid services, or locally, e.g., as Java and Python programs, as building blocks in the composition of their experiments. For example, the European Bioinformatics Institute[1] hosts multiple scientific modules in the form of web services. In 2011, it recorded 21 millions invocations to those scientific modules [29]. Typically, an experiment is designed as a workflow, the steps of which represent invocation to scientific modules, and the edges define data flow dependencies between module invocations [9].

EXAMPLE 1. *Consider the workflow shown in Figure 1, which specifies a simple form of protein identification experiment [2]. The first module (*Identify*) is used to detect the protein that was present in a given sample. To do so, it takes as input peptide masses produced by mass spectrometric analysis of some sample of interest together with an identification error (percentage), and delivers as output the accession of the protein suspected to be present in the sample. The second module (*GetRecord*) takes the accession produced by the first module and returns the corresponding protein record. Finally, the last module (*SearchSimple*) performs an alignment search to identify the proteins that are similar to the one identified by the first module. To do so, it takes as input the record of the protein identified as well as parameters specifying the name of the alignment algorithm to be used (*program*) and the name of the protein database against which the alignment is to be performed (*database*), and produces an alignment report. Such a workflow is used in proteomic studies to identify, e.g., which protein may be responsible for a given infection.*

To assist scientists in the task of identifying the modules that are fit for their needs and experiments, semantic annotations have been proposed and used to describe scientific modules [38]. Such annotations can be used by scientists to discover and compose modules that are relevant for their experiments [14, 24], and to identify interoperability issues between connected modules during the experiment design [23].

A module is semantically annotated by associating it to concepts from ontologies. Different facets of the module can be described using semantic annotations, e.g., input and output parameters, task and quality of service (QoS). In practice, however, we observe that most of semantic annotations that are available are confined to the

[1] http://www.ebi.ac.uk

10.5441/002/edbt.2014.79

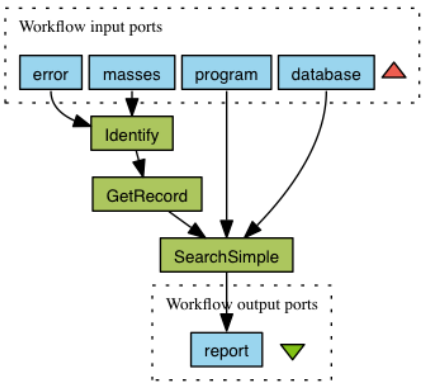

Figure 1: Protein identification workflow.

description of the domain of input and output parameters of modules. Annotations specifying the behavior of the module, as to the task it performs, are rarely specified. Indeed, the number of modules that are semantically described with concepts that describe the behavior of the module lags well behind the number of modules that are semantically annotated in terms of the domains of the input and output parameters, e.g., in Biocatalogue [15]. Even when they are available, annotations that describe the behavior of the module tend to give a general idea of the task that the module implements, and fall short in describing the specifics of its behavior. For example, the modules in Biocatalogue, which is a registry that provides information about scientific modules, are described using terms such as *filtering*, *merging* and *retrieving*. While such terms provide a *rough* idea of what a module does, they do not provide the user with sufficient information to determine if a given module is suitable for the experiment at hand.

The failure in crisply describing the behavior of scientific modules should not be attributed to the designers of task ontologies. Indeed, designing an ontology that captures precisely the behavior of modules, without increasing the difficulty that the human annotators who use such ontologies may face thereby compromising the usability of the ontology, is challenging. Moreover, we note that many scientific modules are polymorphic [30], in the sense that they implement multiple tasks depending on the input values. Describing the behavior of those modules using (named) concepts from existing ontologies can be difficult.

To address the above problem, we investigate in this paper a promising and practical solution that augments the semantic annotations that describe the domain of input and output parameters of a given module with data examples that illustrate the behavior of the module. Given a module m, a data example provides concrete values of inputs that are consumed by m as well as the corresponding output values that are delivered as a result. Data examples provides an intuitive means for users to understand the module behavior: the user does not need to examine the source code of the module, which is often not available, or the semantic annotations, which require the user to be familiar with the domain ontology used for annotation. Moreover, they are amenable to describing the behavior of a module in a precise, yet concise, manner.

EXAMPLE 2. *To illustrate how data examples can be used to understand a module behavior, consider the module* `GetRecord`*,*

GetRecord.i	GetRecord.o
P17110	>sp|P17110|CH36_CERCA Chorion protein S36 OS=Ceratitis capitata MNCFLFTLFFVAAPLATASYGSSSGGGGGGSSYLSSASSNGLDELVQAAAGGAQQAGGTI TPANAEIPVSPAEVARLNQVQAQLQALNSNPVYRNLKNSDAIAESLAESSLASKIRQGNI NIVAPNVIDQGVYRSLLVPSGQNNHQVIATQPLPPIIVNQPALPPTQIGGGPAAVVKAAP VIYKIKPSVIYQQEVINKVPTPLSLNPVYVKVYKPGKKIDAPLVPGVQQNYQAPSYGGSS YSAPAASYEPAPAPSYSAAPAQSYNAAPAPSYSAAPAASYGAAPSASYDAAPAASYGAES SYGSPQSSSSYGSAPPASGY

Figure 2: Data Example.

which has one input and one output. Figure 2 illustrates an input instance that is consumed by `GetRecord` *and the corresponding value obtained as a result of the module invocation. By examining such a data example, a domain expert will be able to understand that the* `GetRecord` *module retrieves the protein record that corresponds to the accession number given as input. It is worth mentioning that we chose an intuitive name for the module that hints to its general behavior. In practice, however, scientific modules often have vague and non-intuitive names. This is partly due to the fact that many modules are generated automatically from existing legacy command lines tools, e.g., SoapLab*[2]*. Because of this, understanding a module behavior from its name becomes a difficult task even for a domain expert.*

The main difficulty when attempting to characterize the module behavior using data examples is the choice of data examples. Enumerating all possible data examples that can be used to describe a given module may be expensive or impossible since the domains of input and output parameters can be large or infinite. Moreover, data examples derived in such a manner may be redundant in the sense that multiple data examples are likely to describe the same behavior of the module. This raises the question as to *which data examples should be used to characterize the functionality of a given module*.

In software engineering, test cases, which can be thought of as data examples, are widely used for verifying that the behavior of a software program conforms with its specification [34]. A software program is tested by using a test suite composed of a collection of test cases that specify data values for feeding the software execution, and the outputs expected as a result according to the specification. We show in this paper how software testing techniques can be adapted to the problem of generating data examples that characterize scientific modules using only the annotations of input and output parameters, without relying on the availability of the module specification, which often is not accessible.

In summary, we make the following contributions:

- **Data example model.** We propose a model of data examples for semantically annotating the behavior of scientific modules (Section 2).
- **A heuristic for generating data examples.** We show how data examples that characterize scientific modules can be automatically constructed without relying on the availability of module specifications (Section 3).
- **Evaluation of the methods proposed.** We report on evaluation exercises that show the effectiveness of the data examples generated using our heuristic to characterize scientific

[2] `http://www.ebi.ac.uk/soaplab/`

modules in terms of completeness and conciseness (Section 4).

- **Usefulness of data examples for human users.** We report on the results of a study that we conducted to gain insight on the extent to which the human user is able to understand the module behavior based on data examples (Section 5).

- **Matching modules based on data examples.** We show how data examples can be used to compare the behavior of two modules (Section 6).

Additionally, we analyze and compare existing works to ours (in Section 7). We conclude the paper (in Section 8) underlining our main contributions and discussing venues for future work.

2. OVERVIEW OF THE SYSTEM

Figure 3 depicts an overview of the system that implements our approach, which distinguishes between the annotation of scientific modules and the use of the resulting annotations. The annotation task is a two-step process. Given a module, in the first step, the curator annotates its input and output parameters by associating them with concepts in the domain ontology used for annotation *(labeled by the number 1 in the figure)*. To do so, the curator can use existing parameter annotation tools such as Radiant [20], Meteor-S [31], APIHUT [19]. For example, Meteor-S [20] allows curators to annotate the parameters of modules using the domain ontology of their choice. It also assists the curators in the annotation of parameters by suggesting an ordered list of concepts. Such a list is constructed by matching the module parameters with the domain ontology used for annotation using schema matching techniques [36].

Once specified, the annotations of module parameters are stored in a module registry. Based on parameter annotations, in the second step, data examples that characterize the module behavior are generated in an automatic manner *(labeled by the number 2 in the figure)*. The resulting data examples are stored together with parameter annotations in the module registry.

The experiment designer can then make use of the module registry to explore and understand the behavior of scientific modules as to the task they perform *(labeled by the number 3 in the figure)*. Once the designer identifies suitable modules, s/he can use them to compose and enact his/her experiment using scientific workflow systems such as Galaxy [18], Taverna [40] and Vistrails [12]. For example, the Taverna system provides a workbench that allows scientists to compose their experiment graphically by linking the modules they choose by means of data links.

As well as assisting designers in composing new experiments, the module registry can be utilized to assist them in the task of repairing existing workflows. Indeed, a problem that frequently arises in scientific workflows is the volatility of the modules that compose the workflow. Such modules are in the majority of cases provided by third parties who are not compelled to continuously supply the functionality of the modules they host. In this respect, an empirical study that was conducted by Zhao *et al.* [42] showed that the majority of scientific workflows stop working few months following their specification because of module volatility. This problem, i.e., module volatility, is widely recognized as one of the main impediments against workflow reuse in the eScience community [17].

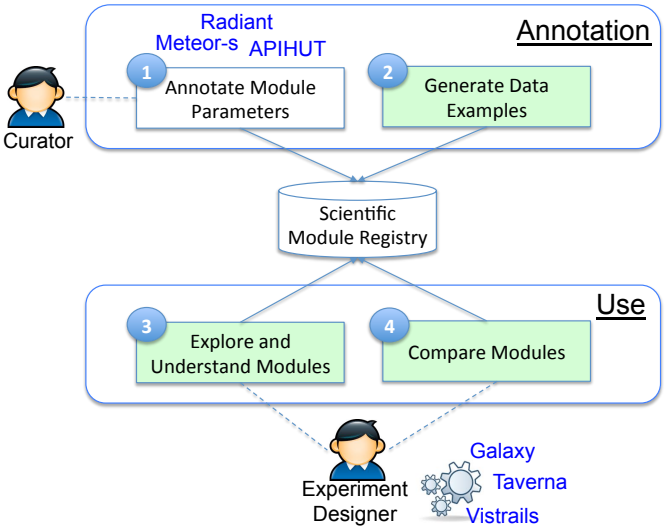

Figure 3: Overall architecture.

To address the above problem, we show that the data examples annotating module behavior can be used to assist workflow designer in repairing broken workflows by identifying available modules that can play the same task as the modules that are no longer available. To do so, we provide workflow designers with an automatic means for comparing the behavior of modules *(labeled by the number 4 in the figure)*.

In the rest of this paper, we present in details the method we propose for generating data examples that characterize scientific modules, we report on a study that we conducted to understand the extent to which data examples help users understand the behavior of scientific modules, and go on to present the method that we propose for automatically comparing the behavior of modules based on data examples. Before doing so, we present in the remaining of this section the data model that we use for specifying data examples.

For the purposes of this paper, we define a scientific module by the pair:

$$\mathtt{m} = \langle \mathtt{id}, \mathtt{name} \rangle$$

where `id` is the module identifier and `name` its name. A module `m` is associated with two ordered sets $\mathtt{inputs}(\mathtt{m})$ and $\mathtt{outputs}(\mathtt{m})$, representing its input and output parameters, respectively. A parameter p of a module m is characterized by a structural type, $str(i)$, and a semantic type, $sem(i)$. The former specifies the structural data type of the parameter, e.g., $String$ or $Integer$, whereas the latter specifies the semantic domain of the parameter using a concept, e.g., $Protein$, that belongs to a domain ontology [21].

A data example δ that is used to describe the behavior a module `m` can be defined by a pair: $\delta = \langle \mathtt{I}, \mathtt{O} \rangle$, where:

$$\mathtt{I} = \{\langle \mathtt{i}, \mathtt{ins}_\mathtt{i} \rangle\} \text{ and } \mathtt{O} = \{\langle \mathtt{o}, \mathtt{ins}_\mathtt{o} \rangle\}$$

`i` (resp. `o`) is an input (resp. output) parameter of m, and $\mathtt{ins}_\mathtt{i}$ and $\mathtt{ins}_\mathtt{o}$ are parameter values. δ specifies that the invocation of the module m using the instances in `I` to feed its input parameters, produces the output values in `O`.

Note that a module m may have optional parameters, in which case, some of the input parameters may be associated with null (or default) values. We use in what follows $\Delta(\mathtt{m})$ to denote the set of data examples that are used to describe the behavior of a module m.

3. ANNOTATING SCIENTIFIC MODULES USING DATA EXAMPLES

Data examples, of the form presented in the previous section, can be used as a means to describe the behavior of scientific modules. However, as mentioned earlier, enumerating all possible data examples that can be used to describe a given module may be expensive, and may contain redundant data examples that describe the same behavior. We present in this section, a method for selecting data examples that characterize the behavior of a given module.

3.1 Identifying the Classes of Behavior of a Scientific Module

To identify the classes of behavior of a given module m, we use and adapt the well established equivalence partitioning technique, which is used in software testing for verifying that a program is conform to its specification [34]. Without loss of generality, consider that m has a single input parameter i. To construct data examples that characterize the behavior of m, the domain of its input i is divided into partitions, $\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n$. The partitioning is performed in a way to cover all classes of behavior of m. For each partition $\mathtt{p}_i$, a data example δ is constructed such that the value of the input parameter in δ belongs to the partition $\mathtt{p}_i$. The issue with the above partitioning method is that it requires the specification of the module m to identify its classes of behavior. However the majority of scientific modules available are not accompanied with specifications [16]. This raises the question as to how the domains of module parameters can be partitioned without using module specifications.

A source of information that we use to overcome the above issue is the semantic annotations used to describe module parameters. Indeed, the input and output parameters of many scientific modules are annotated using concepts from domain ontologies [28]. In its simple form, an ontology can be viewed as a hierarchy of concepts. For example, Figure 4 illustrates a fragment of the myGrid domain ontology used for annotating the inputs and output parameters of bioinformatics modules [15]. The concepts are connected together using the subsumption relationship, e.g., ProteinSequence is a sub-concept of BiologicalSequence, which we write using the following notation: $\mathtt{ProtSequence} \sqsubset \mathtt{BioSequence}$. Such a hierarchy of concepts can be used to partition the domain of parameters. For example, we have shown in previous work that ontology-based partitioning is an effective means for guiding the verification of semantic annotations of web service parameters [3]. In this paper, we exploit the same source of information, i.e., domain ontologies used to annotate module parameters, for a different problem, namely automatic generation of data examples that characterize scientific modules.

EXAMPLE 3. *To illustrate the approach we adopt, using a concrete example, consider the operation* getAccession*, which given an input annotated as biological sequence returns the accession used for its identification. The domain of input of such an operation can be partitioned into the following subdomains using the ontology illustrated in Figure 4:* BiologicalSequence*,* NucleotideSequence*,* RNASequence*,* DNASequence*, and* ProteinSequence*.*

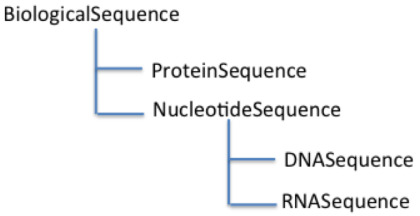

Figure 4: Fragment of the myGrid Ontology.

3.2 Generating Data Examples Covering Input Parameter Partitions

Given the partitions of the input parameter i of a module m identified using the domain ontology, we need to construct data examples that cover those partitions. Such data examples can be specified by soliciting from the human annotator examples input values that belong to the respective partitions, and then invoking the module m to obtain the corresponding output values, necessary for constructing the data examples. The construction of such data examples can, however, be fully automated if a pool of annotated instances is available. Specifically, given pl, a pool of annotated instances, the values of i necessary for constructing data examples that cover the partitions of the input i of the module m can be obtained as follows:

$$\{\langle \mathtt{c}, \mathtt{getInstance}(\mathtt{c}, \mathtt{pl})\rangle \text{ s.t. } \mathtt{c} \sqsubseteq \mathtt{sem}(\mathtt{i})\}$$

where $\mathtt{getInstance}(\mathtt{c}, \mathtt{pl})$ is a function that returns an instance of the concept c from the annotated pool of instances *pl*. Note that this function returns a realization of the concept in question [25], in the sense that the instance of c chosen is not an instance of any strict subconcept of c, i.e. not an instance of any concept $\mathtt{c}' \sqsubset \mathtt{c}$. Note that if it is not possible to have an instance that is a realization of a concept because its domain is covered by the domains of its subconcepts, then we do not create a data example for such a concept, since it is represented by the data examples of its subconcepts. Note also that the data structure (grounding) [26] of the instances selected need to be compatible with the data structure of the input parameter in question, $\mathtt{str}(\mathtt{i})$.

A module may have multiple inputs parameters. This raises the question as to which combinations of input values, that are selected for each input parameter, should be used in the data examples to annotate the module in question. Because different combinations may allow capturing different behaviors of the module, we invoke the module using the different combinations. Note, however, that certain combinations may not be valid. In other words, if they are used to feed the execution of the module, then the module execution throw an error. Therefore, when generating data examples, we only consider the combinations that yield normal termination of the module invocation.

Having specified how the domains of parameters can be partitioned and how input values can be selected for the identified partitions, we can now define the overall procedure whereby the data examples covering the partitions of the input parameters are constructed.

1. Partitioning of the domains of the module inputs based on their semantic annotations.
2. Selection of input values that cover the partitions identified from a pool of annotated instances.
3. Invocation of the module using selected input values.

4. Construction of data examples using the input values and the output values obtained as a result of the module invocations.

Given a module m, the first phase consists in partitioning the domain of each input parameter i of m into the sub-domains that are subsumed by the concept used for annotating i, i.e., sem(i), as illustrated in Section 3.1. In the second phase, for each input i and each partition p_i of i, a value v_i that belongs to the partition p_i is retrieved from an annotated pool of instances pl. The data structure of the value selected v_i needs to be compatible with that of the input i. The module is then invoked using the input values selected. Where the module has multiple input parameters, then the module is invoked using all possible combinations of the values selected for those parameters. In the last phase, data examples are constructed by using the input values and the corresponding values obtained as a result of the module invocations. Where the module has multiple input parameters, data examples are constructed only for the combinations of the values of those parameters that yield a normal termination of the module invocations.

3.3 Generating Data Examples Covering Output Parameter Partitions

Note that so far, we have only considered the domains of the input parameters. The method proposed can be complemented to derive data examples based on the partitioning of the domains of the output parameters. To construct data examples that characterize the behavior of m, the domain of its output o is first divided into partitions, $p_1, p_2, \ldots, p_n$. For each partition p_i, a data example δ is constructed such that the value of the output parameter in δ belongs to the partition p_i.

The method for constructing data examples based on the partitioning of the domains of output parameters is, in principle, similar to that based on the partitioning of the domains of input parameters. However, the former can be difficult to implement. Specifically, given a partition p_o of the output parameter o of a module m, we need to find values that if used to feed the inputs of the module m, the output o generates a value that belongs to the partition p_o. Where a module m' that is known to implement the inverse functionality of m exists, then it can be used to construct data examples that cover the output partitions of the module m. However, our experience suggests that scientific modules often do not have corresponding inverse modules that are available.

Fortunately, there is a source that can be readily used to construct data examples that (at least partially) cover the output partitions, namely the data examples constructed to cover the partitions of the input parameters. Indeed, the empirical evaluation that we report on in the next section shows that, in most cases, the data examples generated to cover the partitions of the input parameters, cover the majority of the partitions of the output parameters. More importantly, the evaluation showed that partitioning of the domains of input parameters yield data examples that completely characterize the classes of the behavior of scientific modules.

4. REAL-WORLD EVALUATION

The method that we have just described is not an *exact* method. Rather, it is a heuristic that provides a working solution for generating data examples based on the partitioning of the domains of module parameters, thereby overcoming the lack of module specifications. Because of this:

• The domain of a given module parameter may be be over-partitioned. Consider for example, a module m that accepts as input biological sequences, and consider that the partitioning method described above divided the domain of biological sequences into the following partitions: Proteinsequences, DNAsequences and RNAsequences. If the module m has the same behavior for DNA and RNA sequences, then the data examples that will be used to cover the DNAsequences and RNAsequences partitions will be redundant as far as the characterization of the module is concerned.

• The domains of a given module parameter may be under-partitioned. This occurs when the module behaves differently for two or more instances of the same partition.

The above discussion calls for an empirical evaluation that assesses the effectiveness of the method proposed for generating data examples in practice. To do so, we ran an experiment that we report on in the remaining of this section.

4.1 Experiment Datasets

We assessed the method we proposed by generating data examples of 252 scientific modules from the life sciences field. Such modules are used for different scientific tasks ranging from pathway analysis, to sequence alignment, to phylogenetic analysis, and are supplied in the forms of: Java and Python programs (56), rest services [37] (60) and soap web services [8] (136) . We selected modules for which documentation describing their specifications is available, to be able to assess the quality of the data examples we generate vis a vis the behavior of the modules. Specifically, given data examples, we were able to identify the classes of behavior of the module that such data examples cover.

For some of the modules, in particular, the SOAP web services, the parameters were annotated using the myGrid domain ontology[3]. Therefore, we directly applied the partitioning strategies and generated the data examples for their characterization. For the remaining scientific modules, we manually annotated their parameters with the assistance of the domain expert using the same ontology, and generated the data examples that characterize them using the method we described in this paper.

Notice that a pool of annotated instances is a key ingredient to the method presented for generating data examples. Such a pool can be obtained by harvesting, e.g., publicly available workflow provenance corpora. For instance, in our experiment, we made use of the Taverna workflow provenance corpus [5]. Such a corpus contains traces of past workflow executions including the data values used as input and obtained as output of the scientific modules that compose the workflows. The input and output parameters of some of those modules are semantically annotated using the myGrid domain ontology. Thanks to those annotations, we were able to semantically annotate the data instances used and produced by such modules in the provenance corpus, thereby constructing the pool of annotated instances necessary for running our experiment.

Using the pool of annotated instances and the semantic annotations of the module parameters, we applied the method presented in this paper to generate data examples that annotate the behavior of the 252 modules.

4.2 Performance Measure

[3] http://www.mygrid.org.uk/ontology/

To assess the performance of the method for generating data example, we use the following metrics.

Coverage. This metric determines the number of partitions of the parameters of m that are covered by the data examples specified for m. Recall that we may not always be able to generate data examples that cover all the partitions identified for output parameters (see Section 3.3). Coverage can be defined by the following ratio:

$$\texttt{coverage}(\texttt{m}) = \frac{\#\texttt{coveredPartitions}(\Delta(\texttt{m}), \texttt{m})}{\#\texttt{partitions}(\texttt{m})}$$

where $\#\texttt{partitions}(\texttt{m})$ is the total number of partitions obtained by partitioning the input and output parameters of m, and $\#\texttt{coveredPartitions}(\Delta(\texttt{m}), \texttt{m})$ is the number of partitions of the parameters of m that are covered by the data examples in $\Delta(\texttt{m})$. A value of 1 means that all partitions identified are covered by the data examples.

Completeness. This metric is used to determine the degree to which the data examples generated for a given module m characterize the classes of its behavior. Here, it is worth stressing that a class of behavior does not refer to a class in the domain ontology used for annotating module parameters. Instead, by classes of behavior, we refer to the different tasks that a given module can perform. The higher the value of completeness, the larger the number of classes of behavior the data examples cover. It can be defined as follows:

$$\texttt{completeness}(\texttt{m}) = \frac{\#\texttt{classesCovered}(\Delta(\texttt{m}), \texttt{m})}{\#\texttt{classes}(\texttt{m})}$$

where $\#\texttt{classes}(\texttt{m})$ is the number of classes of behavior of the module m, and $\#\texttt{classesCovered}(\Delta(\texttt{m}), \texttt{m})$ is the number of classes of the behavior of m that are characterized by the data examples generated for characterizing the module m, i.e., $\Delta(\texttt{m})$. A value of 1 means that the data examples generated characterize all classes of behavior of the module m.

Conciseness. This metric is used to determine the degree to which the data examples specified are free from redundancies. Two data examples are considered redundant if they describe the same class of behavior. The higher the value of conciseness, the lower the number of data examples that are redundant. Conciseness can be defined as follows:

$$\texttt{conciseness}(\texttt{m}) = 1 - \frac{\#\texttt{redundantExamples}(\Delta(\texttt{m}), \texttt{m})}{\#\Delta(\texttt{m})}$$

where $\#\texttt{redundantExamples}(\Delta(\texttt{m}), \texttt{m})$ is the number of redundant examples in $\Delta(\texttt{m})$.

4.3 Experiment Results

Using the partitioning method described in Section 3.1, we partitioned the domains of the module parameters. Following the method described in Section 3.2, we then generated data examples that cover the partitions of the input parameters using the pool of annotated instances. We were able to construct data examples that cover all the partitions of the input parameters.

Moreover, the data examples generated were found to cover most of the partitions of the output parameters. Indeed, with the exception of the partitions of the outputs of 19 modules. e.g., `get_genes_by_enzyme`, `link` and `binfo`, all the partitions of the outputs of the remaining 233 modules were covered by the data examples generated.

# of modules	% of modules	Completeness
236	93.65	1
8	3.18	0.75
4	1.59	0.625
4	1.59	0.6
2	0.8	0.5

Table 1: Data examples completeness.

# of modules	% of modules	Conciseness
192	76.19	1
32	12.7	0.5
7	2.78	0.47
4	1.59	0.4
4	1.59	0.33
8	3.17	0.2
4	1.59	0.17
1	0.4	0.1

Table 2: Data examples conciseness.

To assess the completeness and conciseness of the data examples generated, we examined the data examples generated for the characterization of each module, and checked them against the module's classes of behavior. As mentioned before, the ground truth classes of behavior of the modules were identified using module specifications with assistance from the domain expert. We then computed, for each module, the completeness and conciseness of the data examples.

The results that we obtained in terms of completeness and conciseness are illustrates in Table 1 and Table 2, respectively. The analysis of Table 1 shows that the data examples generated characterize completely the behavior of the 236 out of 252 modules. Only for a small proportion of the modules, namely 16, the data examples did not characterize all classes of behavior. This is an encouraging result, as it means that our methods is effective in identifying data examples that characterize module behavior. This is evidence that data examples derived based on the partitioning of the domains of inputs can be sufficient for characterizing module behavior.

Regarding conciseness, the results were good, but less positive than for the case of completeness. The data examples generated for 192 modules, which represent 76% of the total number of modules, were concise. We identified redundancies in the data examples generated for the remaining 60 modules. The analysis of the data examples generated for those 60 modules revealed that redundancy was due to over-partitioning of the input parameters.

In summary, the above evaluation exercise is evidence that the method presented is effective to a large degree in generating data examples that (completely) characterize the behavior of scientific modules. Although it is possible to obtain redundant data examples as a result of over-partitioning of module parameters, in the majority of cases, the data examples generated are concise. This is a good result, specially considering that the data examples were generated in an automatic manner without access to module specifications or source code, which are generally not available.

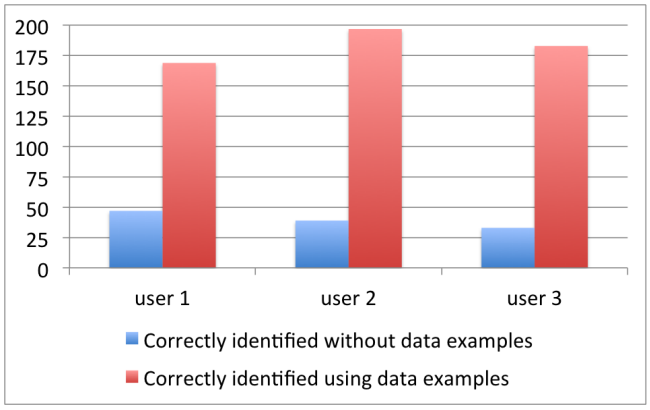

Figure 5: Understanding the behavior of scientific modules with and without data examples.

5. UNDERSTANDING SCIENTIFIC MODULES USING DATA EXAMPLES

We have seen in the previous section that data examples are an effective means for characterizing the behavior of modules. In this section, we report on a study that we conducted to gain insight into the degree to which human users can understand the behavior of modules by examining data examples. To do so, we ran an experiment in which we asked a user to textually describe the behavior of scientific modules by examining data examples. Specifically, given a module `m`, we adopted the following two-step process. In the first step, the user was asked to describe the behavior of a module based on its name, the name of its input and output parameters, and the structural and semantic types of those parameters. In a second stage, the user was given additionally the data examples that characterize the module and was asked to update the module's behavior if s/he deems necessary given the data examples. For the purpose of this experiment, we asked three users with background in the life sciences to textually describe the modules used in the previous experiment. The user had to provide a full account of the behavior of the module for the answer to be counted as correct.

The results of the experiment are shown in Figure 5. The figure shows that the user was able to identify the behavior of a number of modules without access to data examples. For example, $user1$ identified the behavior of 47, which is important as it represents 18% of the total number of modules. This partly due to the fact that those modules are popular modules that are available as web services, and which the user recognized. Note however the thanks to the data examples, the three users identified the behavior of the majority of the modules. For example, $user1$ identified correctly the behavior of 169 modules. That is 67% of the total number of modules. We recorded similar figures for $user2$ and $user3$. It is worth noting none of the modules that were correctly identified without access to data examples was then incorrectly identified using data examples.

Although the number of modules that the user identified thanks to data examples is high, we carried out an analysis of the modules to see why users were unable to identify the behavior of the remaining modules. A careful analysis of our results together with inputs from the user revealed that success or failure in identifying the module behavior is correlated with the nature of the functionality implemented by the module. In particular, we found out that the users were able to identify correctly the majority of the modules implementing data retrieval, format transformation, mapping identifiers. On the other hand, they were not as successful in identifying the behavior of modules implementing data filtering and complex data analysis, such as text mining.

- *F*ormat transformation: these modules are frequently used in scientific experiments (workflows) to resolve mismatches in representation between modules that are developed by independent third parties [35]. An example of a format transformation is that of translating a Uniprot protein record[4] into a Fasta record[5]. The three users were able to identify the behavior of all format transformation modules given the data examples.

- *D*ata retrieval: modules of this kind are used to retrieve records from scientific databases that correspond to an identifier, also known as accession. For example, the module `GetPDBEntry` retrieves the biological DNA record corresponding to a given accession that is provided as input. Data retrieval modules are frequently used in annotation pipeline workflows, which are used to augment input given by the user with annotations from third party data sources. Users were able to identify the behavior of most data retrieval modules. Of the 51 data retrieval modules in our experiment, $user1$ was able to identify 43 by examining the data examples. The user was unable to identify the remaining 8 modules, the reason being the user unfamiliarity with the formats of the outputs of the modules, e.g., $Glycan$[6] ad $Ligand$[7].

- *M*apping identifiers: modules of this kind are used to map identifiers from one data source to another, e.g., from *Uniprot*[8] to *GO*[9]. As such, these modules are used in data integration workflows to combine and link data coming from different sources. The three users were able to identify the behavior of all modules that belong to this category.

- *F*iltering: filtering modules are used to extract from the input values those that meet given criteria or conditions. The three users were able to identify only a small portion of the modules in this category. For example, $user1$ was able to identify the behavior of 5 of the 27 filtering modules.

- *D*ata analysis: modules of this kind apply complex data analysis, such as text mining. As for filtering the three users were able to identify a small portion of the data analysis modules. For example, $user1$ was able to identify 6 of the 59 data analysis modules. For instance, the user was not able to identify the behavior of the `GetConcept` module, which given a text document derives the gene pathway concepts that are subject of the document.

The above experiment shows that data examples are generally a good means for users to grasp the behavior of the data modules. In average the three users were able to correctly identify the behavior of 73% of the modules they were asked to describe. The analysis also showed that for modules that implement data filtering and

[4] `http://web.expasy.org/docs/userman.html`
[5] `http://www.bioinformatics.nl/tools/crab_fasta.html`
[6] `http://www.genome.jp/kegg/glycan`
[7] `http://ligand.info`
[8] `http://web.expasy.org/docs/userman.html`
[9] `www.geneontology.org`

Kind of data manipulation	# of modules
Format transformation	53
Data retrieval	51
Mapping identifiersl	62
Filtering	27
Data analysis	59

Table 3: Kinds of data manipulation carried out by the scientific modules.

complex data analysis, data examples may not have the same value as for other module kinds, as far as the human user is considered. Note, however, that a large proportion of scientific modules implement format transformation, data retrieval and mapping identifiers, which are refereed to in the scientific workflow literature using the term *Shims* [35]. For example, Table 3 classifies the modules that we analyzed in the experiment. It shows that format transformation, data retrieval and mapping identifiers modules represent between them 66% of the total number of modules that we analyzed. That said, it is worth stressing, as we will demonstrate in the next section, that other applications can still benefit from the availability of data examples, even for those modules that implement data filtering and complex data analysis.

6. MATCHING SCIENTIFIC MODULES USING DATA EXAMPLES

As well as understanding scientific modules, users may be interested in comparing the behavior of two or more modules. Module comparison, as a functionality, is particularly requested by workflows curators. Indeed, a problem that frequently occurs within scientific workflows is the volatility of the modules that compose workflows [42]. Generally, there is no agreement that compels the providers to continuously supply their modules. In such situations, users (and curators) of workflows would want to identify available modules that can play the same role as the missing modules.

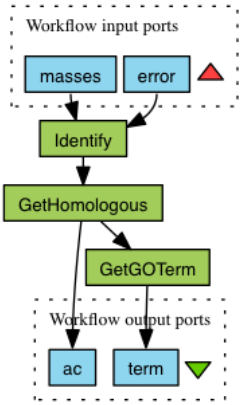

Figure 6: Value-added protein identification

EXAMPLE 4. *To illustrate the problem of scientific module volatility, we will use an example of a real-world experiment, which is variant of the experiment presented earlier in Figure 1. The experiment is used for performing value-added protein identification in which protein identification results are augmented with additional information about the proteins that are homologous to the identified protein. Figure 6 illustrates the workflow that was implemented to automate this experiment. The workflow consists of three modules. The* Identify *module takes as input peptide masses obtained from the digestion of a protein together with an identification error and outputs the Uniprot accession number of the* "best" *match. Given a protein accession, the operation* GetHomologous *performs a homology search and returns the list of similar proteins. The accessions of the homologous proteins are then used to feed the execution of the* GetGOTerm *operation to obtain their corresponding gene ontology term*[10].

This workflow was built in the context of the iSPIDER project [11]*, which ended in 2008. Three years later on, we received a request from a bioinformatician from the myGrid project*[12] *to use the workflow. However, because the module* GetHomologous *that we used for performing the protein homology search did no longer exist, the user was unable to execute the workflow. Therefore, we had to search for an available module that performs homology searches and that we can use instead. This operation turned out to be time consuming. We found several candidate modules for performing homology searches and that are provided by the DNA Databank of Japan*[13]*, the European Bioinformatics Institute*[14] *and the National Center for Biotechnology Information*[15]*. However, we had to try several modules before locating a module that can actually replace the* GetHomologous *operation within the protein identification workflow. The reason is that even though the candidate modules that we found fulfill the task that the unavailable module used to perform (i.e., protein homology search), they use different aligement algorithms and therefore deliver different results from the module used initially in the experiment. In what follow, we show how data examples can be used to address the above problem, by providing a systematic means for comparing the behavior of scientific modules.*

If data examples characterizing the unavailable module are available, then they can be used to identify suitable substitutes, if such substitutes exist. Consider two modules $\mathtt{m}$ and $\mathtt{m}'$, and consider that the inputs and outputs of those modules are semantically and structurally compatible. In other words, there is a 1-to-1 mapping $\mathtt{map}_{\mathtt{param}}$ from $\mathtt{inputs}(\mathtt{m})$ (resp. $\mathtt{outputs}(\mathtt{m})$) to $\mathtt{inputs}(\mathtt{m}')$ (resp. $\mathtt{outputs}(\mathtt{m}')$), such that the parameters connected by such a mapping have the same semantic domain and structure. To be able to compare the behavior of $\mathtt{m}$ and $\mathtt{m}'$, we generate data examples that characterize their behavior using the method presented in Section 3. However, to make the comparison of their behavior straightforward, we generate the data examples of $\mathtt{m}$ and $\mathtt{m}'$ in a way that their data examples have the same input values.

This is better illustrated using an example. Consider that $\mathtt{i}$ is an input of $\mathtt{m}$ and $\mathtt{i}'$ is its corresponding input in $\mathtt{m}'$ according to the mapping $\mathtt{map}_{\mathtt{param}}$. And consider that $\mathtt{i}$ (and therefore $\mathtt{i}'$) are annotated using the semantic domain $\mathtt{c}$. Consider now that the partitioning method that we presented in Section 3, divided the domain of $\mathtt{c}$ into the following partitions $\mathtt{p}_1, \ldots, \mathtt{p}_n$. When selecting the input values that will be used for constructing data examples for $\mathtt{m}$ and $\mathtt{m}'$, we choose the same values for both $\mathtt{i}$ and $\mathtt{i}'$. In other words, for each partitions in $\mathtt{p}_1, \ldots, \mathtt{p}_n$, we choose the same value for both $\mathtt{i}$

[10]http://www.geneontology.org/
[11]http://www.taverna.org.uk/introduction/related-projects/ispider/
[12]http://www.mygrid.org.uk/
[13]http://www.ddbj.nig.ac.jp/
[14]http://www.ebi.ac.uk/
[15]http://www.ncbi.nlm.nih.gov/

and $\mathtt{i}'$. As a result, the data examples generated for characterizing $\mathtt{m}$ and $\mathtt{m}'$ have the same input values. That is, there is a mapping $\mathtt{map}_\Delta$ that maps each data example in $\Delta(m)$ to a data example in $\Delta(m')$, such that the two data examples have the same input values. When comparing the behavior of $\mathtt{i}$ and $\mathtt{i}'$, we distinguish the following three cases:

- Equivalent behavior: If the data examples mapped using $\mathtt{map}_\Delta$ have the same output values, then we conclude that the modules $\mathtt{m}$ and $\mathtt{m}'$ are eventually equivalent. Notice that we use the adverb *eventually*. This is because the method that we propose for generating data examples is a heuristic. As such, there may be corner cases where the data examples for the two modules do not cover all classes of behavior, as illustrated in the experiment reported on in Section 3. Note, however, that if the data examples have output values that are different for the same input values, then we can safely conclude that the two modules do not have equivalent behavior.

- Overlapping behavior: If some, but not all, the data examples mapped using $\mathtt{map}_\Delta$ have the same output values, then we say that the modules $\mathtt{m}$ and $\mathtt{m}'$ have overlapping behaviors. In essence, this means that for a subset of the domains of their inputs, the two modules behave in the same manner. We distinguish this case, as in certain situations, a module that have an overlapping behavior with an unavailable module can play the same role as the unavailable module in a given workflow. To illustrate how this may happen consider the workflow illustrated in Figure 7-(a), which is used to retrieve the gene ontology term of the protein that is most similar to the protein provided as input. The first module returns the accession of the protein that is most similar to the protein given as input to the workflow. The second module retrieves the protein sequence corresponding to the accession delivered by the first module. Finally, the last module *GetGOTerm* returns the gene ontology term of such protein. Consider now that the supplier of the second module *GetProteinSequence* decided to interrupt the supply of the module functionality. To repair the workflow, we can make use of the module *GetBiologicalSequence* (see Figure 7-(b)). Given a Biological Sequence, which is a superconcept of Protein Sequence, it delivers the corresponding Biological Sequence, which is a superconcept of the Protein Sequence concept. *GetBiologicalSequence* have input and output parameters that are semantically different from those of the unavailable module *GetProteinSequence*. However, it behaves in the same way as *GetProteinSequence* for the inputs that are Protein Sequence, which is the kind of inputs that *GetBiologicalSequence* will receive as input in the context of the workflow in Figure 7-(b). This is because the input of *GetBiologicalSequence* is fed using the output of *GetMostSimilarProtein*, which only delivers Protein Sequences.

- Disjoint behavior: if all the data examples mapped using $\mathtt{map}_\Delta$ have different output values, then we say that the modules $\mathtt{m}$ and $\mathtt{m}'$ have disjoint behaviors.

To assess the effectiveness of the above method for comparing modules' behavior, we used it to assist in the curaction of broken workflows. That is workflows for which one or more modules are not available because they are delivered by distributed third party providers that stopped their supply. For our experiment, we used workflows from the popular myExperiment workflow repository[16]. A recent analysis that we conducted revealed that almost half of the workflows (i.e., $\sim$ 1500 workflows) that are stored in that repository could not be enacted because of the unavailability of third party supplied modules [42]. We therefore decided to curate those workflows by locating modules that can play the same role as the unavailable module.

To apply our method for such a purpose, however, we will need data examples that characterize such modules. This is a problem since we cannot construct the data examples, as this operation would require invoking the unavailable modules! Fortunately, there is a source of information that can be utilized to construct the data examples for some, but not all, of those modules, namely workflow provenance traces. Indeed, most of scientific workflow systems are instrumented to capture provenance traces that specify among other aspects the data products used and generated by the module as part of the workflow enactment. We have inspected the publicly available workflow provenance corpus [5], as well as provenance traces captured as part of previous eScience projects, in particular the iSpider project. By trawling those provenance traces, we were able to construct data examples that characterize 72 unavailable scientific modules.

Using the method presented above, we then matched those unavailable modules to the 252 modules that we used the experiment reported on in Section 4. The results of this comparison are depicted in Figure 8. The figure shows that we were able to identify modules with equivalent behavior for 16 unavailable modules, and modules that have overlapping behavior for 23 unavailable modules. 16 modules may sound small. However, such a small number allowed us to curate an important number of workflows 321. This is because some unavailable modules that we identified equivalent modules for are popular modules that are used in multiple workflows. This is, in particular, the case of the KEGG[17] modules provided in the form of SOAP web services, which were interrupted, and for which we identified equivalent modules supplied in the form of Rest web services. Regarding the 23 unavailable modules for which we identified modules with overlapping behavior, we conducted a manual examination of the workflows in which those unavailable modules are used. We were able to detect 13 workflows in which the modules we identified can play the same role as the unavailable modules. Specifically, of the 23 unavailable modules, we identified modules with overlapping behavior for 6 of them such that those modules play the same role as the unavailable ones in 13 workflows.

To verify that the substitute modules discovered have equivalent behavior as the unavailable modules within the workflows in which they were incorporated, we enacted those workflows using samples of randomly selected inputs. We then verified with the help of the domain expert that their invocations do not through any errors and that they deliver results comparable with those that the corresponding missing unavailable modules would deliver. Regarding the workflows in which some but not all the unavailable modules were substituted, we extracted from each of them sub-workflows that contain the substitute modules and verified that the execution of these sub-workflows deliver valid results. This test confirmed that all the discovered substitutes, without exception, have the same

[16] http://www.myexperiment.org
[17] www.genome.jp/kegg

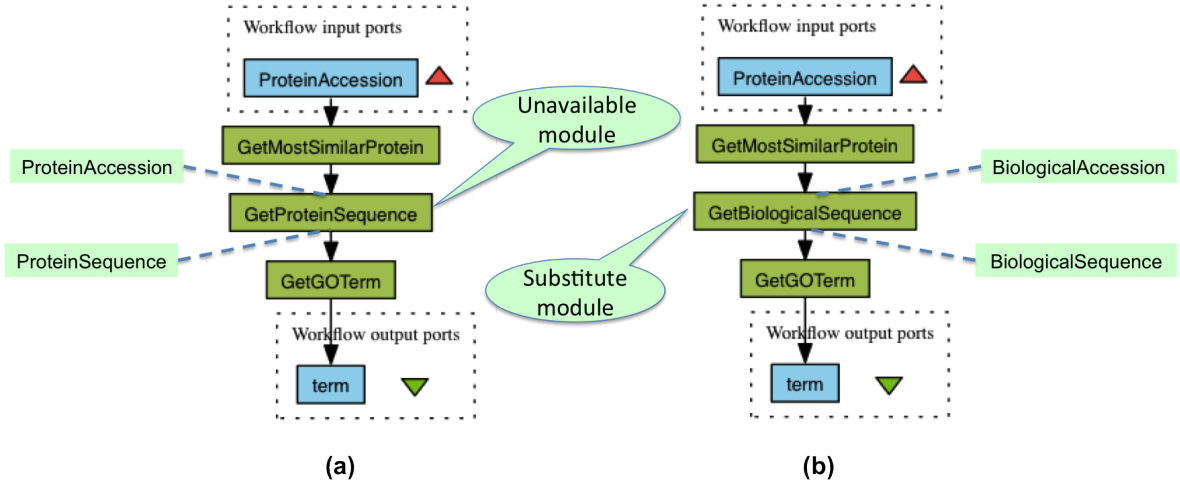

Figure 7: Example of a substitute module that does not have semantically equivalent input and output as the unavailable module.

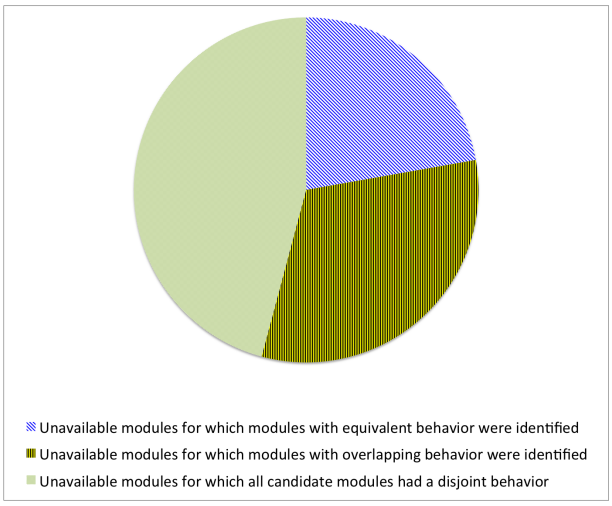

Figure 8: Identifying modules with matching behavior to unavailable modules.

behavior as the modules they replace within the workflows in which they were used.

To summarize, the above experiment showed that, when available, data examples can be used as an effective means for identifying modules with similar behavior. In particular, the experiment showed the practicality of the method in resolving a real problem, i.e., the curation of broken workflows, where the funcionalities of modules are suspended by their third party providers. Although we were able to construct data examples for a subset of the modules that are unavailable (mainly because data examples were not collected for the remaining modules while they were available), we were able to locate suitable substitutes for 22 unavailable modules. The substitutes were used to repair a large number of workflows, 334 in total. Of the 334 workflows, 73 were partly repaired as they contained other unavailable modules for which we did not locate substitute either because we could not construct data examples that characterize them or because the set of the available modules did not contain any suitable substitutes.

The experiment, therefore, demonstrates the utility of data examples when they are available, and therefore can be used to incite module providers and workflow designers to collect data examples characterizing scientific modules they are providing/using with the objective to facilitating their substitution when needed.

7. RELATED WORK

In this section, we analyze and compare existing proposals to ours. We organize the section into four subsections thereby covering the elements of our solution.

7.1 Semantic Annotations of Web Services

Semantic annotations of web services have been proposed as a means for enabling the understanding, discovery and composition of web services [7]. These annotations relate the various service elements (i.e. operations, inputs and outputs) to concepts in ontologies describing their semantics, form and role. However, the literature suggests that, by and large, most of the proposals in this field consider annotations that describe the semantics of the input and output parameters, e.g., [31, 33]. There have been proposals that attempt to describe the behavior of the operations of web services, e.g., the EDAM ontology [18], or more generally computational modules, e.g., [13]. However, such proposals aim to provide a high level description of the behavior. In doing so, they fail to capture the specifics of the transformations carried out by the modules (web services).

7.2 Data Example Generation

Data examples have been used as a means for characterizing queries and schema mappings. In particular, in the area of testing database applications, Binnig *et al.* [6] proposed a method for generating test databases. Given a query and a result, the method they propose produces a database instance that can be used to produce such a result. Similarly, Abdul Khaled *et al.* proposed an algorithm that given a database schema and an SQL query as inputs, generates data to populate the test database as well as the results expected from by issuing the query over the database. Regarding schema mappings, Alexe *et al.* [1] reported on a systematic investigation of universal examples [11], underlying their capabilities and

[18] http://edamontology.org

limitations in understanding schema mappings. The data examples characterizing schema mappings can be used for specifying and refining schema mapping [41].

While related, the scope of our work is different from the above proposals in the sense that we consider existing black box modules (as opposed to query or mapping specifications) for which we do not have any specification, and we aim to elaborate a working solution to derive data examples for their characterization.

The work by Olson *et al.* [32] is perhaps the closest to ours. They investigated the problem of generating example data that illustrate the behavior of data flow programs. In their proposal, Olson *et al.* assume that the specification of the modules is available in the form of *equivalence classes* that characterize the behavior of the steps in the dataflow program. In our work, we do not make such assumption, but rather investigate how such equivalence classes can be automatically identified (or approximated). Also, the work by Olson *et al.* focuses on workflows with modules that resemble relational algebra primitives such as project, filter, join. Instead, in our work, we are targeting black box modules that implement (possibly complex) data analyses and transformations.

Our work is also related to the well established discipline of software testing [34, 27]. Given the specification of a software program, test cases, which are similar to the notion of data examples introduced in this paper, are specified to verify that the software program is conform to its specification. However, software testing techniques assume the availability of the source code [27] of the program and/or its specification [34], neither of which are available for the majority of scientific modules. While inspired by software testing techniques, we propose in this paper a working solution for generating data examples for black-box and un-documented scientific modules.

7.3 Understanding by Means of Data Examples

A number of proposals have investigated characterization of behavior through data examples, as specified in the previous section. Yet, there is no proposal in the literature that investigates if the human user is able to grasp the behavior based on data examples, that we are aware of. For example, there is a reasonable number of proposals that seek to generate and characterize schema mappings using data examples. However, there is no proposal in the literature that investigates the ability of the human user to grasp the behavior of schema mappings using data examples [39]. Our proposal is, therefore, the first to investigate the ability of human users to identify module behavior based on data examples, to our knowledge, and to come up with a classification distinguishing the kinds of behavior that can be identified by the human user from those that are difficult to identify.

7.4 Scientific Module Comparison

Paolucci *et al* [33] is perhaps one of the first proposals to suggest matching web services, which are a kind of module, using semantic annotations. Specifically, the authors of this work used semantic descriptions of web services as defined by the DAML-S language[19]. Two service operations are considered to match if they have compatible input and output parameters. In other words, the task fulfilled by the operations is not taken into account by the matching algorithm proposed in [33].

Hull *et al* proposed an approach for matching modules in which the module task is described using an OWL expression that captures the relationship between the inputs of the module and its outputs [22]. The modules are then matched by comparing their associated expressions. In practice, however, it is difficult to capture the behavior of a module using a mathematical expression, and when it is possible, such expression cannot be formulated because of the absence of the module specification. In those circumstances, data examples remain a cheap resource that can be easily obtained, and can be used to effectively compare the behavior of the module without requiring the availability of the module specifications, which are usually not available, or the use of a task ontology, which often fail to capture the specifics of a module behavior.

In a previous work [4], we have investigated the use of provenance traces as a means for comparing the behavior of modules. However, that method was not guided by any principle. It merely checked if two modules have provenance traces that takes similar inputs and delivers similar outputs. The solutions that we propose in this paper goes beyond (i) by proposing a principled means for identifying the data examples that characterize module behavior taking into account properties such as completeness and conciseness (as opposed to using random data examples), and (ii) by providing a classification that characterizes module comparison into equivalent, overlapping and disjoint behavior.

8. CONCLUSIONS

We showed that it is possible to characterize scientific modules using data examples without relying on module specifications. Central to the method proposed for generating data examples is the partitioning used to divide the domains of module parameters into sub-domains. We showed that, in the majority of cases, partitioning based on the semantic annotations that describe module parameters yields data examples that completely describe the behavior of modules. We also presented two functionalities that can benefit from the generated data examples. Specifically, we showed that human users can understand the behavior of modules based on data examples when such modules do not implement filtering or complex data analyses. Furthermore, we presented a method for comparing the behavior of modules based on data examples, and showed the practical utility of such method in repairing decayed workflows by replacing unavailable modules with modules that can fulfill the same role within the workflow.

This paper constitutes a first step in an important, yet thus far overlooked, research area, namely the characterization of scientific modules using data examples. The evaluation of the method used for generating data examples showed that they are not always concise. We are investigating, as part of our future work, techniques that can be used for detecting redundant data examples. In particular, we envisage examining the use of record linkage techniques, such as those reported on by Elmagarmid *et al.* [10], for this purpose. We also envisage investigating the problem of composition of scientific modules within workflows based on data examples. In other words, how to use data examples to implicitly guide module composition.

Acknowledgments

We warmly thank Dr. Suzanne Embury from the University of Manchester and the members of the myGrid team who were sup-

[19]DAML-S is a service description language, it is the predecessor of the OWL-S language.

portive of the initial idea and helped shape the proposal through the long discussions we have had on the subject of data examples. We would also like to thank the reviewers for their constructive comments that improved the quality of the work.

9. REFERENCES

[1] B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. Characterizing schema mappings via data examples. *ACM Trans. Database Syst.*, 36(4):23, 2011.

[2] K. Belhajjame, S. M. Embury, et al. Proteome data integration: Characteristics and challenges. In *UK All Hands Meeting*, 2005.

[3] K. Belhajjame, S. M. Embury, and N. W. Paton. Verification of semantic web services using ontology-based equivalence partitioning. *IEEE Transactions on Service Computing*, 2013.

[4] K. Belhajjame, C. A. Goble, S. Soiland-Reyes, et al. Fostering scientific workflow preservation through discovery of substitute services. In *eScience*, pages 97–104, 2011.

[5] K. Belhajjame, J. Zhao, D. Garijo, et al. A workflow prov-corpus based on taverna and wings. In *EDBT/ICDT Workshops*, pages 331–332, 2013.

[6] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *ICDE*. IEEE, 2007.

[7] J. Cardoso and A. P. Sheth, editors. *Semantic Web Services, Processes and Applications*, volume 3 of *Semantic Web And Beyond Computing for Human Experience*. Springer, 2006.

[8] F. Curbera, F. Leymann, et al. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR Englewood Cliffs, 2005.

[9] E. Deelman, D. Gannon, M. S. Shields, and I. Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Comp. Syst.*, 25(5):528–540, 2009.

[10] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[11] R. Fagin, P. G. Kolaitis, et al. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[12] J. Freire and C. T. Silva. Making computations and publications reproducible with vistrails. *Computing in Science and Engineering*, 14(4):18–25, 2012.

[13] D. Garijo, P. Alper, et al. Common motifs in scientific workflows: An empirical analysis. In *eScience*, pages 1–8, 2012.

[14] Y. Gil et al. Wings: Intelligent workflow-based design of computational experiments. *IEEE Intelligent Systems*, 26(1):62–72, 2011.

[15] C. Goble et al. Biocatalogue: A curated web service registry for the life science community. In *Microsoft eScience conference*, 2008.

[16] C. Goble and D. D. Roure. Curating scientific web services and workflow. *EDUCAUSE Review*, 43(5), 2008.

[17] C. A. Goble, R. Stevens, D. Hull, K. Wolstencroft, and R. Lopez. Data curation + process curation=data integration + science. *Briefings in Bioinformatics*, 9(6):506–517, 2008.

[18] J. Goecks, A. Nekrutenko, J. Taylor, et al. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol*, 11(8):R86, 2010.

[19] K. Gomadam, A. Ranabahu, M. Nagarajan, A. P. Sheth, and K. Verma. A faceted classification based approach to search and rank web apis. In *ICWS*, pages 177–184, 2008.

[20] K Gomadam, K Verma, D Brewer, AP Sheth, and JA Miller. Radiant: A tool for semantic annotation of web services. In *4th International Semantic Web Conference ISWC*, 2005.

[21] T. Gruber. Ontology. In *Encyclopedia of Database Systems*, pages 1963–1965. Springer US, 2009.

[22] D. Hull, E. Zolin, A. Bovykin, I. Horrocks, U. Sattler, and R. Stevens. Deciding semantic matching of stateless services. In *AAAI*, 2006.

[23] K. Johari and A. Kaur. Interoperability issues in web services. In *CCSEIT*, pages 614–619, 2012.

[24] D. John and M. S. Rajasree. A framework for the description, discovery and composition of restful semantic web services. In *CCSEIT*, pages 88–93, 2012.

[25] S. Koide and H. Takeda. Owl-full reasoning from an object oriented perspective. In *ASWC*. Springer, 2006.

[26] J. Kopecký, D. Roman, M. Moran, and D. Fensel. Semantic web services grounding. In *AICT/ICIW*. IEEE Computer Society, 2006.

[27] B. Korel. Automated software test data generation. *IEEE Trans. Software Eng.*, 16(8):870–879, 1990.

[28] Dominik Kuropka, Peter Tröger, Steffen Staab, and Mathias Weske, editors. *Semantic Service Provisioning*. Springer, Berlin, 2008.

[29] R. Lopez. Personal communication. European Bioinformatics Institute, Cambridge, UK, February 2012.

[30] P. Missier et al. Functional units: Abstractions for web service annotations. In *SERVICES*, pages 306–313, 2010.

[31] N. Oldham, C. Thomas, A. P. Sheth, and K. Verma. Meteor-s web service annotation framework with machine learning classification. In *SWSWPC*, pages 137–146, 2004.

[32] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *SIGMOD Conference*, pages 245–256. ACM, 2009.

[33] M. Paolucci, T. Kawamura, et al. Semantic matching of web services capabilities. In *International Semantic Web Conference*, pages 333–347, 2002.

[34] R. Patton. *Software Testing (2nd Edition)*. Sams, Indianapolis, IN, USA, 2005.

[35] U. Radetzki et al. Adapters, shims, and glue - service interoperability for *in silico* experiments. *Bioinformatics*, 22(9):1137–1143, 2006.

[36] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[37] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly, 2008.

[38] R. Studer, S. Grimm, and A. Abecker, editors. *Semantic Web Services, Concepts, Technologies, and Applications*. Springer-Verlag, Berlin, Heidelberg, 2007.

[39] W. Chiew Tan. Personal communication. University of California, Santa Cruz, USA, March 2013.

[40] K. Wolstencroft et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, 2013.

[41] L. L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD Conference*, pages 485–496, 2001.

[42] J. Zhao et al. Why workflows break - understanding and combating decay in taverna workflows. In *eScience*, pages 1–9, 2012.