Solving Data Mismatches in Bioinformatics Workflows by Generating Data Converters

Mouhamadou Ba, Sébastien Ferré, Mireille Ducassé

To cite this version:

Mouhamadou Ba, Sébastien Ferré, Mireille Ducassé. Solving Data Mismatches in Bioinformatics Workflows by Generating Data Converters. Transactions on Large-Scale Data- and Knowledge-Centered Systems, 2016, LNCS, 9510, pp.88-115. hal-01485059

HAL Id: hal-01485059
https://hal.science/hal-01485059v1

Submitted on 13 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Solving Data Mismatches in Bioinformatics Workflows by Generating Data Converters

Mouhamadou Ba[1], Sébastien Ferré[2], and Mireille Ducassé[1]

[1] IRISA/INSA Rennes
20 Avenue des Buttes de Coesmes, 35708 Rennes cedex, FRANCE
[2] IRISA/Université de Rennes 1
263 Avenue Général Leclerc, 35042 Rennes cedex, FRANCE
mouhamadou.ba@irisa.fr

Abstract. Heterogeneity of data and data formats in bioinformatics entail mismatches between inputs and outputs of different services, making it difficult to compose them into workflows. To reduce those mismatches, bioinformatics platforms propose ad'hoc converters, called shims. When shims are written by hand, they are time-consuming to develop, and cannot anticipate all needs. When shims are automatically generated, they miss transformations, for example data composition from multiple parts, or parallel conversion of list elements.
This article proposes to systematically detect convertibility from output types to input types. Convertibility detection relies on a rule system based on abstract types, close to XML Schema. Types allow to abstract data while precisely accounting for their composite structure. Detection is accompanied by an automatic generation of converters between input and output XML data. We show the applicability of our approach by abstracting concrete bioinformatics types (e.g., complex biosequences) for a number of bioinformatics services (e.g., blast). We illustrate how our automatically generated converters help to resolve data mismatches when composing workflows. We conducted an experiment on bioinformatics services and datatypes, using an implementation of our approach, as well as a survey with domain experts. The detected convertibilities and produced converters were validated as relevant from a biological point of view. Furthermore the automatically produced graph of potentially compatible services exhibited a connectivity higher than with the ad'hoc approaches. Indeed, the experts discovered unknown possible connexions.

1 Introduction

Heterogeneity of data and data formats in bioinformatics entail mismatches between inputs and outputs of different services, making it difficult to compose them into workflows [1]. Formats to represent input and output data can be textual or based on XML technologies. Textual formats, often specific to a few services, have the advantage to be human readable [2]. XML formats despite verbosity are used for their expressiveness. To deal with these different formats in scientific workflows, special services, called *shims* are used for conversion of data between services. Generally, they have to be manually defined (see, for example, Emboss [3], Galaxy [4], Mobyle [5]). Services for conversion of data represent more than 30% of services in life science workflows

according to the analysis of Wassink et al [6] on the Taverna Workflows. When composing services, users can get lost in specific parsers and shims required to transfer data between services. It is difficult to find appropriate shims because they are often mixed with other services. Users can be forced to create new shims, it is time-consuming and error prone. To avoid using shims, some developers implement their services so that they support several formats. It is a burden for service developers because they have to integrate several format conversions in each tool, and each format conversion may be duplicated across many services.

Formats based on XML technologies are proposed as standards to describe data types independently of tools [7, 8]. BioXSD, for example, represents basic bioinformatics data [9]. It also allows meta-information to be added from ontologies, increasing the accuracy of representations. Yet, XML-based formats alone are not sufficient to solve the problem of data matching. On the one hand, most formats are textual, thus it is important to be able to match services using XML *and* textual formats. On the other hand, even if all formats were XML standardized, it remains to solve the n:m matching problem [10], namely matching and conversion between two *composite structures*, i.e. XML trees.

Work related to data mismatches in scientific workflows addresses, among others, classifying mismatches, matching and resolving mismatches. For example, Li et al. [11] classify service composition mismatches. The classification of mismatches enables to understand problems and to find appropriate solutions. Besides approaches that address verifying matching between services (e.g., Lebreton al. [12]), there are approaches that resolve data mismatches by means of shims to insert between services. Among them, some approaches search shims in existing libraries [13, 14] and other approaches automatically generate shims [15, 16]. Many approaches use ontologies to verify matching between service parameters. However, they do not generally guarantee parameter compatibility at syntactic level. Approaches that find shims, for example Elizondo et al. [13] and Hull et al. [14], fix data mismatches but expect the shims to be provided by third-parties. Approaches that generate shims automatically provide data transformers to insert between services. For example, Kaslev et al. [16] generate transformations at workflow execution time. Dibernado et al [17] provide possible transformations during workflow construction. However, these approaches miss transformations such as data composition from multiple parts, or parallel conversion of list elements.

This article proposes a new approach to generate shims to help compose services.[3] It supports complex data representations and transformations. It systematically detects convertibility from output types to input types. Convertibility detection relies on abstract types, close to XML Schema, abstracting data while precisely accounting for their composite structure. The main contribution is the definition of convertibility rules that exploit composition and decomposition as well as specialization and generalization of types. Furthermore, the rules automatically generate a complete constructive specification of the conversion from output to input types. That specification enables to generate executable converters between input and output XML data. We report an experiment on bioinformatics services with an implementation of our approach. We manually specified the inputs and outputs of services using abstract types, where each service is understood

[3] It is a revised and extended version of Ba et al. [18].

as a function from input(s) to output(s) as proposed by Missier et al [19]. The detected convertibilities and produced converters were analyzed with a team of the GenOuest [4] bioinformatics platform. They have been reckoned relevant from a biological point of view. We also led a survey with domain experts, that shows the relevance of connections established between bioinformatics services. When adding a new service, with our approach, it is sufficient to define or reuse abstract types for input and output data, and the new service will be automatically integrated in the global system. At present, in order to achieve the same goal, many converters have to be manually developed for services which are not immediately compatible. That is significantly heavier than specifying a few abstract types. Furthermore, identifying the compatible services by hand is already a challenge while our approach does it automatically. As a consequence, our approach automatically produces a graph of potentially compatible services with a connectivity higher than with the ad'hoc approaches.

In the following, Section 2 introduces the abstract representation of types. Section 3 defines convertibility between abstract types by a rule system, and proves that it forms a reflexive and transitive relationship. Section 4 shows how to instantiate our method and provides a use case in bioinformatics. Section 5 presents an experiment in bioinformatics, and a survey with domain experts. Section 6 compares our approach with related work. Section 7 gives some perspectives.

2 Representation of Types

In this section we present the language used to describe the types of data. It is defined from an open set of primitives and a fixed set of type constructors. From a semantic point of view, a type denotes a set of XML values. An XML value is a sequence of XML nodes. An XML sequence may be empty or contain a single node. An XML node is either an XML element or a textual element (CDATA). An XML element is made of a tag and a content, which is a sequence of XML nodes. Our language of types follows the main XML Schema constructs, but to simplify the presentation we use regular expressions inspired by the work of Hosoya et al. [20]. In the following, types are denoted by uppercase letters (e.g., T, T_1) and function $XML(T)$ defines the semantics of type T by a set of XML values.

- **Primitive types**: $T = p$, where p is a primitive type. Primitive types are the basic ingredients to build other types. Their structure is atomic, they are not decomposable. XML instances of a primitive type are CDATA (text). For example, *int* is a primitive type representing integers.
- **Constructor *tag***: $t[T_1]$ where t is a tag. This expression denotes XML elements whose tag is t and whose content is of type T_1 : $XML(t[T_1]) = \{\text{<t>}\, x_1\, \text{</t>} \mid x_1 \in XML(T_1)\}$. Tags provide semantics for data and can be bound to concepts of ontologies. In XML Schema, constructor *tag* can be expressed by a tag or by the attribute *name* of tag *xs:element*.
- **Constructor *empty***: ε. The empty XML sequence: $XML(\varepsilon) = \{\varepsilon\}$.

[4] http://www.genouest.org/

- **Constructor *tuple***: T_1T_2. This type expression denotes XML sequences that are the concatenation of instances of T_1 and instances of T_2 : $XML(T_1T_2) = \{x_1x_2 \mid x_1 \in XML(T_1), x_2 \in XML(T_2)\}$. That constructor is used to define composite types and sequences. In XML Schema, constructor *tuple* can be expressed by tags *xs:complexType* and *xs:sequence*.
- **Constructor *union***: $T_1|T_2$. This type expression denotes the union of instances of T_1 and instances of T_2 : $XML(T_1|T_2) = XML(T_1) \cup XML(T_2)$. Constructor *union* can be used, for example, to consider several different types and make some treatments without distinction. In XML Schema, that constructor can be expressed by tag *xs:choice*.
- **Constructor *list***: T_1+. This type expression denotes the non-empty sequences of instances of type T_1 (homogeneous lists): $XML(T_1+) = \{x_1 \ldots x_n \mid n \geq 1 \wedge\ x_1, \ldots, x_n \in T_1\}$. In XML Schema, that constructor can be expressed by the *minOccurs* and *maxOccurs* attributes associated with tag *xs:sequence*.
- **Constructor *optional***: $T_1?$. This type expression is equivalent to $T_1|\varepsilon$.

For abbreviation it is possible to bind a name to a type expression. For example, in the expression $T = T_1T_2$, there is a type name T allowing to simplify the re-use of type expressions, where the name T can be used to refer to T_1T_2. We do not allow recursive type expressions yet.

3 Convertibility Rules

By using type theory in the context of workflows, we follow the same line as previous work on web services such as the one of Chen et al. [21]. The novelty of our approach lies in the definition of a rule system to prove convertibility between types, and to apply the proof-as-program paradigm [22] to automatically derive executable converters from convertibility proofs.

In order to motivate the following convertibility rules, we list a few typical cases where there is a mismatch between two types A and B ($A \neq B$), whereas there is a semantic match, i.e. A can be converted to B.

- A and B use tags that are different but have the same meaning, for example `Integer` and `Int`.
- B may be replaced by A, for example `Float` by `Long`.
- B is a concatenation of some components of A, for example `person[name tel email]` from `person[name contact[address tel email]]`.
- A is subsumed by B, for example `protein seq` by `biological seq`.

3.1 Rule System

Figure 1 lists all the rules that specify when a type A is convertible to a type B. They also define associated converters as functions from A to B. Those rules form a natural deduction system whose judgements are in the form $f : A \rightarrow B$, i.e. f is a converter from an XML value of type A to an XML value of type B, and hence A is convertible to B. A judgement $f : A \rightarrow B$ holds true if and only if it is possible to build a

$$\frac{f_p : p_1 \rightarrow_p p_2}{f : p_1 \rightarrow p_2 \quad f(x) = f_p(x)} \quad \text{(PRIMITIVE)}$$

$$\frac{t_a \rightarrow_t t_b \quad f_1 : A \rightarrow B}{f : t_a[A] \rightarrow t_b[B] \quad f(x) = element(t_b, f_1(content(x)))} \quad \text{(TAGCHANGE)}$$

$$\frac{f_1 : A \rightarrow B}{f : t[A] \rightarrow B \quad f(x) = f_1(content(x))} \quad \text{(TAGREMOVAL)}$$

$$\frac{}{f : A \rightarrow \varepsilon \quad f(x) = \varepsilon} \quad \text{(EMPTY)}$$

$$\frac{f_1 : A \rightarrow B_1 \quad f_2 : A \rightarrow B_2}{f : A \rightarrow B_1 B_2 \quad f(x) = concat(f_1(x), f_2(x))} \quad \text{(CONCAT)}$$

$$\frac{f_1 : A_1 \rightarrow B}{f : A_1 A_2 \rightarrow B \quad f(x) = (let\ x_1, x_2 = select(x, A_1, A_2)\ in\ f_1(x_1))} \quad \text{(LEFTSELECTION)}$$

$$\frac{f_2 : A_2 \rightarrow B}{f : A_1 A_2 \rightarrow B \quad f(x) = (let\ x_1, x_2 = select(x, A_1, A_2)\ in\ f_2(x_2))} \quad \text{(RIGHTSELECTION)}$$

$$\frac{f_1 : A_1 \rightarrow B \quad f_2 : A_2 \rightarrow B}{f : A_1|A_2 \rightarrow B \quad f(x) = (case\ (x : A_1)\ then\ f_1(x) \mid (x : A_2)\ then\ f_2(x))} \quad \text{(PRECHOICE)}$$

$$\frac{f_1 : A \rightarrow B_1}{f : A \rightarrow B_1|B_2 \quad f(x) = f_1(x)} \quad \text{(LEFTPOSTCHOICE)}$$

$$\frac{f_2 : A \rightarrow B_2}{f : A \rightarrow B_1|B_2 \quad f(x) = f_2(x)} \quad \text{(RIGHTPOSTCHOICE)}$$

$$\frac{f_1 : A \rightarrow B}{f : A+ \rightarrow B+ \quad f(x) = map(f_1, x)} \quad \text{(MAP)}$$

$$\frac{f_1 : A \rightarrow B}{f : A+ \rightarrow B \quad f(x) = (let\ x_1 = choose(x, A)\ in\ f_1(x_1))} \quad \text{(CHOICE)}$$

$$\frac{f_1 : A \rightarrow B}{f : A \rightarrow B+ \quad f(x) = f_1(x)} \quad \text{(SINGLETON)}$$

Fig. 1. Convertibility rules and definitions of generated converters. For example, rule TagChange reads as follows: if tag t_a is convertible to tag t_b and if f_1 is a converter from A to B, then there is a converter f from $t_a[A]$ to $t_b[B]$ such that applying f to a data x involves extracting its content, applying f_1 to the content, then re-encapsulate the result with tag t_b.

Function	Input	Output	Description
content	XML	XML	returns the content of an XML element
element	tag, XML	XML	builds an XML element given a tag and an XML content
concat	XML, XML	XML	returns the concatenation of two XML sequences
select	XML, type, type	XML, XML	splits an XML sequence in two parts matching given types
map	converter, XML	XML	applies a converter to each node of an XML sequence and returns the concatenation of the results
choose	XML, type	XML	returns any element matching a given type from an XML sequence

Table 1. Utility functions on XML values.

proof tree with that judgement at the root, and where each node instantiates a rule. The deduction system works by structural induction on couples of types (A, B), covering all combinations of type constructors for which convertibility is possible. The rules depend on conversion axioms for tags ($t_a \rightarrow_t t_b$), and on converters between primitive types ($f_p : p_1 \rightarrow_p p_2$). Those base conversions depend on the application domain, and correspond, for instance, to well-known conversion functions (e.g., from floats to integers). By default, we assume that $t \rightarrow_t t$ for every tag t, and $f_p : p \rightarrow_p p$ with $f_p(x) = x$ for every primitive type p. The definitions of converters in rules make use of utility functions on XML values, which are described in Table 1. Figure 2 shows a convertibility proof.

Primitive rule. Rule (PRIMITIVE) allows the use of a primitive converter f_p when the two types are primitive types. That rule handles the conversion of the leaves of XML trees (CDATA nodes).

Tag rules. These rules handle the conversion from and to XML elements. Rule (TAGCHANGE) defines converters from an XML element x to another XML element $element(t_b, f_1(content(x)))$ by applying a domain-dependent tag conversion (here, from t_a to t_b), and by recursively applying a converter f_1 to the content of x. Function $content$ gives access to the content of an XML element, and function $element$ builds the new element from the converted tag and converted content. Rule (TAGREMOVAL) define converters from an XML element to an XML sequence by ignoring the tag, and recursively converting the content.

Empty and tuple rules. These rules handle conversions of XML sequences, i.e. constructors *empty* and *tuple*. Rule (EMPTY) says that any XML value x can be converted to the empty XML sequence ϵ. Rule (CONCAT) defines converters that first apply the converters f_1 and f_2 to the source value x, and then concatenates the two results $f_1(x)$ and $f_2(x)$ with function $concat$, hence producing an XML sequence.

Rules (LEFTSELECTION) and (RIGHTSELECTION) define converters that select respectively the left and right part of the source data, an XML sequence, and convert it to the target data. This is useful when only a part of the source data is necessary to produce the target data. The selection of the parts (function $select$) is guided by the sub-types of the source sequence.

Union rules. These rules handle conversions from and to unions of types. Rule (PRECHOICE) defines converters that produce a target data using a different converter depending on the type of source data (A_1 or A_2). This is useful when the source data can have different structures (union type). Rules (LEFTPOSTCHOICE) and (RIGHTPOSTCHOICE) choose a converter to a target sub-type, when the target type is an union. This is useful when the target data has several acceptable structures.

List rules. The remaining rules handle conversions from and to lists. Rule (MAP) define converters from a source list to a target list where a same converter is applied to each element of the list. Function map is used to perform iteration over list elements, and concatenation of converted elements. Rule (CHOICE) defines converters that first choose an element of a list, and then recursively apply a converter to it. This is useful when a single element is expected while a list is provided. Rule (SINGLETON) defines converters that produce singleton lists from a source element, after recursively applying a converter to it. This is useful when a list is expected while a single element is provided.

For a given couple (A, B) of type expressions, several rules may be applicable. In that case, it is sufficient that one of them leads to a success to prove the convertibility from A to B. Figure 2 details the proof of convertibility between two kinds of biological sequence lists. In the source list, sequences are made of a nucleotide sequence, a species name, and a version number, while in the target list, sequences are made of an organism, and a nucleotide sequence. Another difference is that nucleotide sequences are lowercase in the source list (primitive type *acgt*), and uppercase in the target list (primitive type *ACGT*). In the proof (Figure 2), we assume that a primitive converter is available to convert from lowercase to uppercase (see step 1.2.2.1.2.), and domain knowledge tells us that tag *species* can be replaced by *organism* (see step 1.2.1.1.1.1.). Each item in Figure 2 is the conclusion of a rule, and the sub-items are the hypotheses of the rule. At each item, the converter function is defined with calls to the converter function of sub-items. After inlining the definition of intermediate functions in the main function f, we obtain the full definition of f in Figure 3.

Our rule system exhibits two kinds of non-determinism: (1) in the generation of converters, and (2) in the definition of converters. Firstly, given two types A and B, the system may generate several converters from A to B, i.e. several solutions to the conversion problem. This is a common feature of rule systems. For example, a converter $f : AA \rightarrow A$ can be produced by either Rule (LEFTSELECTION) or Rule (RIGHTSELECTION): in the former, the left part of the source value is selected, while in the latter, the right part is selected. In practice, one converter must be chosen, which could be done through user interaction. Secondly, a generated converter may produce several target values for a same source value. This non-determinism comes from

seq[ns[acgt] species[string] version[int]]+ $\rightarrow$ seq[organism[string] ns[ACGT]]+
(MAP) $f(x) = map(f_1, x)$
1. seq[ns[acgt] species[string] version[int]] $\rightarrow$ seq[organism[string] ns[ACGT]]
(TAGCHANGE) $f_1(x) = element(\text{seq}, f_{1.2}(content(x)))$
1.1. seq $\rightarrow_t$ seq
1.2. ns[acgt] species[string] version[int] $\rightarrow$ organism[string] ns[ACGT]
(CONCAT) $f_{1.2}(x) = concat(f_{1.2.1}(x), f_{1.2.2}(x))$
1.2.1. ns[acgt] species[string] version[int] $\rightarrow$ organism[string]
(RIGHTSELECTION) $f_{1.2.1}(x) = (let\ x_1, x_2 = select(x)\ in\ f_{1.2.1.1}(x_2))$
1.2.1.1. species[string] version[int] $\rightarrow$ organism[string]
(LEFTSELECTION) $f_{1.2.1.1}(x) = (let\ x_1, x_2 = select(x)\ in\ f_{1.2.1.1.1}(x))$
1.2.1.1.1. species[string] $\rightarrow$ organism[string]
(TAGCHANGE) $f_{1.2.1.1.1}(x) = element(\text{organism}, f_{1.2.1.1.1.2}(content(x)))$
1.2.1.1.1.1. species $\rightarrow_t$ organism
1.2.1.1.1.2. string $\rightarrow_p$ string
(PRIMITIVE) $f_{1.2.1.1.1.2}(x) = x$
1.2.2. ns[acgt] species[string] version[int] $\rightarrow$ ns[ACGT]
(LEFTSELECTION) $f_{1.2.2}(x) = (let\ x_1, x_2 = select(x)\ in\ f_{1.2.2.1}(x_1))$
1.2.2.1. ns[acgt] $\rightarrow$ ns[ACGT]
(TAGCHANGE) $f_{1.2.2.1}(x) = element(\text{ns}, f_{1.2.2.1.2}(content(x)))$
1.2.2.1.2. acgt $\rightarrow_p$ ACGT
(PRIMITIVE) $f_{1.2.2.1.2}(x) = uppercase(x)$

Fig. 2. An example proof tree of convertibility between two kinds of sequence lists.

$f(x) = map(f_1, x)$
where $f_1(x) = element(\text{seq}, concat($
$let\ x_1, x_2 = select(content(x), \text{ns[acgt]}, \text{(species[string] version[int])})$
$in\ let\ x_{21}, x_{22} = select(x_2, \text{species[string]}, \text{version[int]})$
$in\ element(\text{organism}, content(x_{21}))$,
$let\ x_1, x_2 = select(content(x), \text{ns[acgt]}, \text{(species[string] version[int])})$
$in\ element(\text{ns}, uppercase(content(x_1)))))$

Fig. 3. The generated converter for example of Figure 2

some utility functions, and the $case$ construct. Function $select$ may find different ways to split the source value in two parts. Function $choose$ has as many results as elements in the list. The $case$ construct has two results when the two conditions are satisfied, when x matches both types A_1 and A_2. This second form of non-determinism could be used to express iteration in a workflow. For example, assuming that service S_1 produces lists of sequences, and service S_2 consumes one sequence at a time, the converter generated by Rule (CHOICE) could be a way to express that S_2 must be iterated over the results of S_1, and the output of S_2 could be considered to be the list of individual results. It will meet needs for job iteration in bioinformatics workflows [23].

3.2 Properties: Reflexivity and Transitivity

Two important properties of the convertibility relationship are *reflexivity* and *transitivity*. First, every type A is convertible to itself and it suffices to take the identity function as a converter. Second, for any types A, B, C, if A is convertible to B, and B is convertible to C, then A is convertible to C and it suffices to compose the two converters from A to B and from B to C to obtain a converter from A to C. We formalize those two properties in the following theorems, and give their proofs. In those theorems, we assume that tag convertibility ($\rightarrow_t$) and primitive convertibility ($\rightarrow_p$) are reflexive and transitive. As a consequence, unlike the approach of Kaslev et al. [16], we do not need rules for transitivity and reflexivity in our rule system. This makes convertibility proofs simpler and more efficient.

Theorem 1. *Let A be a type expression. There is a proof in the rule system of the judgement $f : A \rightarrow A$ where $f(x) = x$.*

Proof. We proceed by induction on type A, considering the six type constructors as different cases. For each of the type constructor, the property is verified because:

1. $A = p$ (primitive): from assumption on primitives ($p \rightarrow_p p$), and by applying Rule (PRIMITIVE).
2. $A = t[A_1]$ (tag): from induction hypothesis on A_1 ($A_1 \rightarrow A_1$), and assumption on tags ($t \rightarrow_t t$), and by applying Rule (TAGCHANGE).
3. $A = \epsilon$ (empty): from Rule (EMPTY).
4. $A = A_1 A_2$ (tuple): from induction hypothesis on A_1 ($A_1 \rightarrow A_1$) and A_2 ($A_2 \rightarrow A_2$), by applying Rule (LEFTSELECTION) to the first, and Rule (RIGHTSELECTION) to the second, and finally by applying Rule (CONCAT) to the consequences of the two previous rules.
5. $A = A_1 | A_2$ (union): from induction hypothesis on A_1 and A_2, by applying Rule (LEFTPOSTCHOICE) to the first (introducing A_2), and Rule (RIGHTPOSTCHOICE) to the second (introducing A_1), and finally by applying Rule (PRECHOICE) to the consequences of the two previous rules.
6. $A = A_1+$ (list): from induction hypothesis on A_1, and by applying Rule (MAP).

In each case, it can be shown that the produced converter function is equivalent to the identity function. For instance, in the *tag* case, assuming $f_1 : A_1 \rightarrow A_1$ is equivalent to the identity function (induction hypothesis), it can be shown that the resulting function $f(x) = element(t, f_1(content(x)))$ is equivalent to $element(t, content(x))$ which is equal to x because x has type $t[A_1]$. ■

Theorem 2. *Let A, B, C be expression types. If there are proofs in the rule system of the judgements $f_1 : A \to B$ and $f_2 : B \to C$, then there is also a proof of the judgement $f : A \to C$ where $f(x) = f_2(f_1(x))$.*

Proof. We proceed by induction on the rules that are used at the root of the proofs of $A \to B$ and $B \to C$. As there are 13 distinct rules, there are potentially 169 distinct cases to consider. Fortunately, many cases have similar proofs and can be grouped together based on the distinction between three kinds of rules:

- Constructors ($\mathcal{C}$): rules where only the target type changes between premises and conclusion (Rules (CONCAT), (LEFTPOSTCHOICE), (RIGHTPOSTCHOICE), (SINGLETON), (EMPTY)),
- Destructors ($\mathcal{D}$): rules where only the source type changes between premises and conclusion (Rules (TAGREMOVAL), (LEFTSELECTION), (RIGHTSELECTION), (PRECHOICE), (CHOICE)),
- Transformers ($\mathcal{T}$): rules where both source and target change but use the same kind of type (Rules (PRIMITIVE), (TAGCHANGE), (MAP)).

Using that grouping, we arrive at 6 meta-cases described using the above group codes $\mathcal{C}$, $\mathcal{D}$, $\mathcal{T}$ and $\mathcal{X}$ to mean any rule. Applying unification constraints on the middle type B, those meta-cases then decompose themselves in 21 elementary cases:

1. $\mathcal{X}$ - $\mathcal{C}$:
   (a) $\mathcal{X}$ - (CONCAT): the proof of $B \to C$ by Rule (CONCAT) implies that $C = C_1C_2$, and that we have proofs for $B \to C_1$, and $B \to C_2$. By induction hypothesis on A, B, C_1 and A, B, C_2, we obtain $A \to C_1$ and $A \to C_2$. Then, by applying Rule (CONCAT) on those judgements, we finally obtain $A \to C$.
   (b) $\mathcal{X}$ - (LEFTPOSTCHOICE): we have $C = C_1|C_2$, and $B \to C_1$. By induction hypothesis on A, B, C_1, we obtain $A \to C_1$. Then, by applying Rule (LEFTPOSTCHOICE) on the later, we obtain $A \to C_1|C_2$.
   (c) $\mathcal{X}$ - (RIGHTPOSTCHOICE): similar to previous case.
   (d) $\mathcal{X}$ - (SINGLETON): we have $C = C_1+$ and $B \to C_1$. By induction hypothesis on A, B, C_1, we obtain $A \to C_1$, from which we obtain $A \to C$ by applying Rule (SINGLETON).
   (e) $\mathcal{X}$ - (EMPTY): we have $C = \epsilon$. We directly obtain $A \to C$ by applying Rule (EMPTY) (everything is convertible to ϵ).
2. $\mathcal{D}$ - $\mathcal{X}$:
   (a) (TAGREMOVAL) - $\mathcal{X}$: we have $A = t[A_1]$ and $A_1 \to B$. By induction hypothesis on A_1, B, C, we obtain $A_1 \to C$. By applying Rule (TAGREMOVAL) on the latter, we obtain $A \to C$.
   (b) (LEFTSELECTION) - $\mathcal{X}$: we have $A = A_1A_2$, and $A_1 \to B$. By induction hypothesis on A_1, B, C, we obtain $A_1 \to C$. By applying Rule (LEFTSELECTION) on the latter, we obtain $A \to C$.
   (c) (RIGHTSELECTION) - $\mathcal{X}$: similar to previous case.
   (d) (PRECHOICE) - $\mathcal{X}$: we have $A = A_1|A_2$, $A_1 \to B$, and $A_2 \to B$. By induction hypothesis on A_1, B, C and A_2, B, C, we obtain $A_1 \to B$ and $A_2 \to B$. By applying Rule (PRECHOICE), we obtain $A \to C$.

(e) (CHOICE) - $\mathcal{X}$: we have $A = A_1+$ and $A_1 \rightarrow B$. By induction hypothesis on A_1, B, C, we obtain $A_1 \rightarrow C$. By applying Rule (CHOICE) to the latter, we obtain $A \rightarrow C$.

3. $\mathcal{C}$ - $\mathcal{D}$:
   (a) (CONCAT) - (LEFTSELECTION): we have $B = B_1B_2$, and the judgements (1) $A \rightarrow B_1$, (2) $A \rightarrow B_2$, (3) $B_1 \rightarrow C$. By induction hypothesis on A, B_1, C and judgements (1) and (3), we obtain $A \rightarrow C$.
   (b) (CONCAT) - (RIGHTSELECTION): similar as previous case.
   (c) (LEFTPOSTCHOICE) - (PRECHOICE): we have $B = B_1|B_2$ and the judgements $A \rightarrow B_1$, $B_1 \rightarrow C$, and $B_2 \rightarrow C$. By induction hypothesis on A, B_1, C, we obtain $A \rightarrow C$.
   (d) (RIGHTPOSTCHOICE) - (PRECHOICE): similar to previous case.
   (e) (SINGLETON) - (CHOICE): we have $B = B_1+$, and the judgements $A \rightarrow B_1$ and $B_1 \rightarrow C$. By induction hypothesis on A, B_1, C, we obtain $A \rightarrow C$.
4. $\mathcal{T}$ - $\mathcal{T}$:
   (a) (PRIMITIVE) - (PRIMITIVE): we have $A = p_1, B = p_2, C = p_3$, and $p_1 \rightarrow_p p2$ and $p_2 \rightarrow_p p_3$. From assumptions on primitives, we obtain $p_1 \rightarrow_p p_3$. By applying Rule (PRIMITIVE) on the latter, we obtain $A \rightarrow C$.
   (b) (TAGCHANGE) - (TAGCHANGE): we have $A = t_A[A_1]$, $B = t_B[B_1]$, $C = t_C[C_1]$, and $t_A \rightarrow_t t_B, t_B \rightarrow_t t_C$, $A_1 \rightarrow B_1$, $B_1 \rightarrow C_1$. From assumptions on tags, we obtain $t_A \rightarrow_t t_C$. By induction hypothesis on A_1, B_1, C_1, we obtain $A_1 \rightarrow C_1$. By applying Rule (TAGCHANGE) to the two latter judgements, we obtain $A \rightarrow C$.
   (c) (MAP) - (MAP): we have $A = A_1+$, $B = B_1+$, $C = C_1+$, and $A_1 \rightarrow B_1$, $B_1 \rightarrow C_1$. By induction hypothesis on A_1, B_1, C_1, we obtain $A_1 \rightarrow C_1$. By applying Rule (MAP) to the latter, we obtain $A \rightarrow C$.
5. $\mathcal{C}$ - $\mathcal{T}$:
   (a) (SINGLETON) - (MAP): we have $B = B_1+$, $C = C_1+$, and $A \rightarrow B_1$, $B_1 \rightarrow C_1$. By induction hypothesis on A, B_1, C_1, we obtain $A \rightarrow C_1$. By applying Rule (SINGLETON) to the latter, we obtain $A \rightarrow C$.
6. $\mathcal{T}$ - $\mathcal{D}$:
   (a) (TAGCHANGE) - (TAGREMOVAL): we have $A = t_A[A_1]$, $B = t_B[B_1]$, and $A_1 \rightarrow B_1$, $B_1 \rightarrow C$. By induction hypothesis on A_1, B_1, C, we obtain $A_1 \rightarrow C$. By applying Rule (TAGREMOVAL) to the latter, we obtain $A \rightarrow C$.
   (b) (MAP) - (CHOICE): we have $A = A_1+$, $B = B_1+$, and $A_1 \rightarrow B_1$, $B_1 \rightarrow C$. By induction hypothesis on A_1, B_1, C, we obtain $A_1 \rightarrow C$. By applying Rule (CHOICE) to the latter, we obtain $A \rightarrow C$.

In each case, it can be shown that the produced converter function is equivalent to the composition of the two converters from A to B, and from B to C. For instance, in the $\mathcal{X}$ - (CONCAT) case, assuming $f_1 : A \rightarrow B$, and $f_2 : B \rightarrow C$, we have $f_2(x) = concat(f_2'(x), f_2''(x))$ where $f_2' : B \rightarrow C_1$ and $f_2'' : B \rightarrow C_2$. By application of three rules as indicated in the proof, we obtain for $f : A \rightarrow C$, the definition $f(x) = concat(f_2'(f_1(x)), f_2''(f_1(x)))$. From the definition of f_2, that definition can be simplified into $f(x) = f_2(f_1(x))$, which is indeed the composition of f_1 and f_2. ■

3.3 Implementation

We implemented our rule system in a program that decides the convertibility between any two type expressions, and generates converters from data matching the first type expression to data matching the second type expression. The algorithm is directly derived from the above rules and combines *pattern matching* on type expressions to identify constructors, and recursive calls on type sub-expressions. The examination of rules shows that recursive calls always involve smaller couples of expressions, which ensures termination of the program in all cases. The computation time required to decide convertibility may be important when the input type is very large, because of the non-deterministic nature of the rule system. However, convertibility is computed once for a set of types. In practice, types are not very large, and we have not encountered any difficulty in our experiments to compute all convertibilities for a set of bioinformatic services (see Section 5.1). Generated converters are efficient. Most rules imply a constant cost per XML node, and hence a linear complexity over the input data. The two cases that may imply additional costs concern node duplication (see Rules LEFTSELECTION and RIGHTSELECTION) and choice handling (see Rule PRECHOICE). Node duplication corresponds to the situation where a node of the input data is converted to several nodes of the output data. In this case, the duplicated node is processed several times, thus exceeding linear complexity. However, the number of duplications is bounded by the number of constructors in the output type. Choice handling corresponds to the situation where a node can have one of two types (A_1 or A_2), and the correct type has to be identified by the converter at execution time. Type identification requires one additional node processing for each choice, thus exceeding linear complexity. However, the number of choices is equal to the number of constructor *union* in the input type, and the additional processing may only concern a fragment of the input data. In practice, input and output types of bioinformatics services make a limited use of duplications and choices, thus in the worst case, the complexity of converters is in the size of the input data multiplied by a small constant. Our generated converters are represented in XQuery, a suitable language to process XML documents, which makes the converters executable. To account for non-determinism, the result of our program is a collection of converters. Each converter will be a function from an XML value to an XML value. In the case of non-deterministic converters, only one value is produced so far. The production of several values is left to future implementation.

4 Instantiation and Use Case

This section presents how we instantiate our type abstraction to bioinformatics data. It also presents a use case that shows how our approach can be used to detect and resolve data mismatches in bioinformatics workflows.

4.1 Instantiation to Bioinformatics

Depending on application requirements and on the nature of the data in bioinformatics platforms, many formats are available. To represent genomics

```
> Accession = accession[string]
> SSeq = simpleSequence[string]
> ProtSeq = ns[sSeq]
> DNASeq = as[sSeq]
> Bioseq =  DNASeq | ProtSeq
> CBioseq = complexBiosequence[
        seq[Bioseq]
        species[string] source[string] name[string]
        version[string] note[string]?]
> CProtSeq = complexProteinSequence[
        seq[ProtSeq]
        species[string] source[string] name[string]
        version[string] note[string]?]
> BioseqList = CBioseq+
```

Fig. 4. Examples of bioinformatics types

data, various textual formats (e.g., FastQ, BED)[5] and XML formats (e.g., BioXSD[6], phyloXML[7]) are provided. Textual formats are the most commonly used. In addition to data formats, ontologies, such as EDAM[8] have been proposed to organize and classify resources including data types and formats. Our work starts from these resources to define input and output types of services. We abstract data types by focusing on the information contents and composite structure of data. Figure 4 shows simple examples of types defined manually from existing bioinformatic formats. $Accession$ and $SSeq$ are simple types representing, respectively, an accession number and a raw sequence, defined with a constructor tag and a primitive. In the same way, $DNASeq$ (representing nucleotide sequences) and $ProtSeq$ (representing amino acid sequences) are defined using $SSeq$, they specialize the sequences. Their union forms $Bioseq$, a biological sequence generalizing the sequences. $CBioseq$ and $CProtSeq$ are composite types holding several types through constructor *tuple*. They are biological sequences containing required (e.g., $sequence[Bioseq]$) and optional (e.g., $note[string]?$) contents, $CProtSeq$ being more specific than $CBioseq$. $CBioseqList$ defines a list of $CBioseq$ using constructor *list*. The other types we use are defined in the same way as the above types. Labels are inspired from the EDAM ontology.

Compared to data types and formats used on platform EMBOSS[9], our types can define accession numbers allowing to represent, for example, sequence and database references. They can represent raw sequences as in plain text format, single sequences as in gcc format, one or several sequences (e.g., alignment of sequences) as in FASTA format, as well as a simple sequence associated to its annotations and features as in

[5] http://genome.ucsc.edu/FAQ/FAQformat.html/
[6] http://bioxsd.org/
[7] http://www.phyloxml.org/
[8] http://edamontology.org
[9] http://emboss.sourceforge.net/docs/Themes

EMBL format. Our types can also represent lists of files and differentiate the nature of information contained in files, for example, nucleotide sequence versus amino acid sequence. We take into account data types and formats commonly used for inputs and ouputs of services on platform EMBOSS. Most platforms we visited use the same categories of data types and formats. Compared to XML formats such as BioXSD, our abtraction represents contents at a higher abtraction level. We only consider information relevant for our matching between input and output data of services. We ignore, for example some type attributes and type restrictions irrelevent for current input and ouput data used in services. If necessary, they can easily be added.

Abstraction of types is straightforward for XML formats thanks to XML schemas. For textual formats, informal specifications must be studied to derive a structural representation. Our experiment with genomics types shows that type expressions recur frequently, they can easily be reused after being defined once. Since the most common data types are defined, there are increasingly less types to define. The BioXSD initiative defines several data types for common bioinformatics web services. Specialized XML formats, such as phyloXML [8] for phylogenetic data and PDBML [24] for systems biology, exist for sub-domains of bioinformatics. Moreover, XML alternatives are provided for some textual formats (e.g., GFF [25]) and some platforms define their own XML format (e.g., Uniprot XML [26]). For our experiment, the abstraction of types is done manually but the spreading of the above mentioned solutions will facilitate the task. We can even expect automatic or semi-automatic abstraction processes.

The defined types represent inputs and outputs of current genomics services. In our approach, adding a new service requires two steps. Firstly, identify the abstract types used as inputs and outputs of the service. Secondly, implement, if they do not already exist, the converters between XML schemas and each format, since our types define an XML. Unlike other approaches, it is not necessary to define converters for all pairs of formats, but only two for each format (from and to XML).

4.2 Use Case : Resolving Data Mismatches in a Bioinformatics Workflow

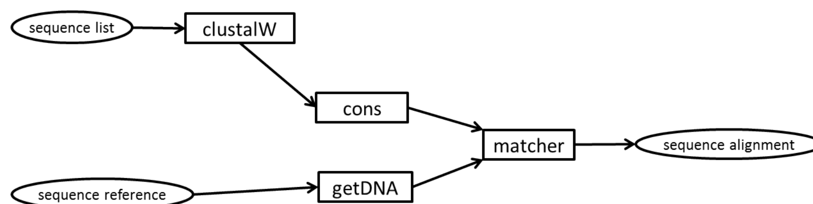

Fig. 5. Workflow at user level (w_u).

We now present a workflow (w), constructed with our approach. It compares a consensus sequence produced from an alignment of a list of sequences with another sequence obtained from an accession number. The workflow consumes sequence lists (tabular form) and references (accessions) to DNA sequences. It produces sequence

alignments. In the following, we describe three representations of workflow w, at three different levels. The workflow at user level is what the user expects to see but we show that it is underspecified, and cannot be executed as such. The workflow at execution level is fully specified and executable, but it is over-detailed for the user because it confuses genuine services and shims. We finally introduce the workflow at a abstract level, from which both user level and execution level representations can be derived automatically.

Figure 5 shows the workflow at a user level (w_u). Boxes represent tasks and ellipses represent input and output data of the workflow. It is an interconnection of inputs and outputs of services from bioinformatics platforms. We assume that a service performs one task, it may have one or several inputs and one or several outputs. To simplify, we do not take into account parameters used to manage service behaviour (e.g., algorithm parameters). The workflow uses the following services:

– **matcher** compares two biological sequences. It takes as inputs two biological sequences in fasta files and returns as output a MSF alignment.
– **getDNA** retrieves a DNA sequence from a database. It takes an accession number and returns an embl file containing a DNA sequence.
– **cons** creates a consensus sequence from a multiple alignment. It takes a MSF alignment and returns a fasta containing a biological sequence.
– **clustalW** makes a multi-alignment of sequences. It takes a multi Fasta containing a list of sequences and returns an alignment.

Workflow w_u shows an ideal view where users have specified only the indispensable information to describe what has to be achieved. However, this view is not directly executable because the workflow uses services whose inputs and outputs do not immediately match. It is necessary to insert shims services to address data mismatches as generally done in the existing platforms.

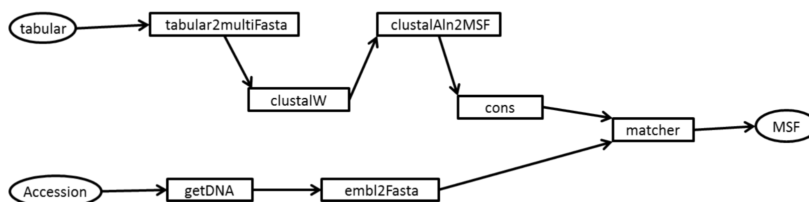

Fig. 6. Executable workflow (w_x).

Figure 6 shows an executable workflow (w_x) where shims have been inserted because of the following mismatches. First mismatch (tabular2multiFasta): the concrete lists of sequences the workflow will consume are in tabular form with additional columns, they must be adapted to feed the input of *clustalW*. Second mismatch (clustalAln2MSF): the output of task *clustalW* must be adapted to match the input of task *cons*. Last mismatch (embl2Fasta): task *getDNA* provides an output that must be fed into the input of task *matcher*, the output of *getDNA* contains more information and is

more specific. In existing approaches, to obtain an executable workflow, users must find and insert format converters between domain tasks for which the formats are different. Data are seen through their formats and they are not decomposable. There is no separation between data, formats and services and no separation between domain services and shim services.

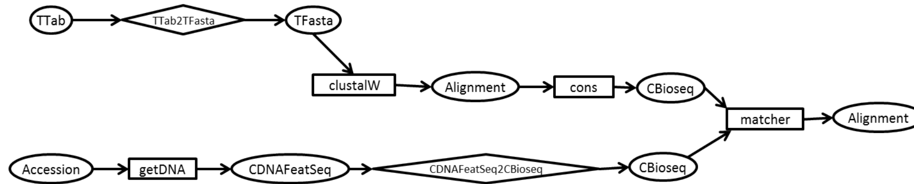

Fig. 7. Abstract workflow (w_a) in our system.

Figure 7 shows the abstract workflow generated by our system (w_a). Ellipses are data types, boxes are domain tasks, and diamonds are generated converters used as shims. Our system sees service inputs and outputs through their composite abstract types. Each service is represented with its input and output types as follows:

- $Alignment$ represents the output type of the service matcher, the input type of the service cons and the output type of the service clustalW. It also represents the output type of the workflow.
- $CBioSeq$ represents the two input types of the service matcher and the output type of the service cons.
- $TFasta = seqs[id[string]\ seq[Bioseq]]+$ represents the input type of the service clustalW.
- $Accession$ represents the input type of the service getDNA, and also an input type of the workflow.
- $CDNAFeatSeq = seq[CDNASeq\ Features]$ represents the output type of getDNA.
- $TTab = seqs[id[string]\ C2[string]\ C3[string]\ C4[string]\ seq[Bioseq]]+$ represents an input type of the workflow.

Our system separates data types, formats and services. It detects and resolves mismatches letting users focus on domain tasks. The generated shims fix the data mismatches presented above. The first generated shim, *TTab2TFasta*, corresponds to the resolution of the tabular2multiFasta mismatch, by transforming each element (row) of the list (table). Transforming each element involves selecting and concatenating subelements (fields). The second shim, *CDNAFeatSeq2CBioseq*, fixes the embl2Fasta mismatch by recognizing a DNA sequence as a biological sequence after ignoring additional information. The clustalAln2MSF mismatch is fixed by reflexivity because clustalAln and MSF correspond to the same abstract type. By ignoring derivable and optional information our system recognises them as equal. Figure 8 shows a generated XQuery source code of shim *CDNAFeatSeq2CBioseq*. It is similar to the theoretical example provided at Figure 2.

```
59 declare function my:convert($v0 as node()*)
60  as node()*
61 {
62 let $v1:=my:content($v0)
63 let $v2:=my:select('CDNASeq',$v1)
64 let $v3:=my:content($v2)
65 return $v3
66 };
```

Fig. 8. Example of source code of the shim converting data matching *CDNAFeatSeq* to data matching *CBioseq*. It is simplified and represented using XQuery. Functions *my:content* and *my:select* are utility functions, their source code is not shown here.

From our abstract workflow (w_a), it is possible to obtain the workflow at user level (w_u) by hiding intermediate types and generated shims. It is also possible to produce the executable workflow (w_x) by inserting converters between concrete formats and XML (e.g., XML from/to each of tabular, multi Fasta, embl and Fasta).

Note that although the users do not have to interfere to generate the converters, they can still check them because all steps of the generation are traceable. In our system, formats are concrete serialization of data, they are not bound to particular data or services. Their use is flexible, for example changing input/output formats do not affect service compatibility or service implementation.

5 Experiments

This section presents a graph of convertibilities where connections between bioinformatics services are detected. It also presents a survey with domain experts to check the relevance of graph connections.

5.1 Convertibility between Bioinformatics Services

Service	Source	Inputs	Outputs	task
Blast	BioXSD	biosequence database URI	biosequence	search
Blastp	EMBL-EBI	Fasta sequence (typed) database URI	BlastResult	search
ClustalWFastaCollection	BioMoby	Fasta files	MSF	alignment
ClustalW	BioXSD	biosequence (≥ 2)	alignment	alignment
maskfeat	EMBOSS	EMBL sequence	Fasta sequence	handling

Table 2. Examples of services

We selected 30 services from platforms EMBOSS [3], EBI (European Bioinformatics Institute) [27], BioMoby [28] and services adopting the BioXSD format. Other

Type	Type in our representation	Represents
Fasta sequence	ComplexProteinSequence | ComplexBiosequence	(typed) sequences
BlastResult, Fasta files	ListOfComplexBiosequences | ListOfBiosequences	a list of sequences
EMBL sequence	AnnotatedSequence	an annotated sequence
BioXSD biosequence	Biosequence | ComplexBiosequence	one sequence
BioXSD Alignment, MSF	SequenceAlignment	an alignement of sequences
Database URI	DatabaseReference	a reference to a database

Table 3. Examples of formats and types

variations and similar categories are provided in platforms but, as mentioned above, their input and output types do not change in general, and they would add little to our experiment. Tables 2 and 3 show exemples of services and types.

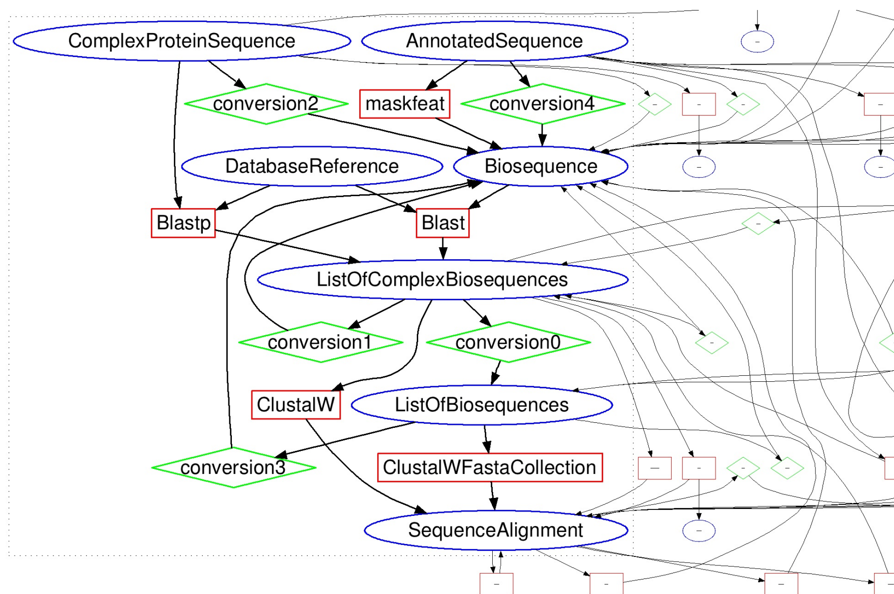

Fig. 9. Excerpt of the graph of links between services

From our selection of services, using our matching algorithm, we automatically generated a graph of the connections between services. Each connection is a convertibility, proved by our rule system, from an output type of a service to an input type of another service. Figure 9 shows an excerpt from the obtained graph. The

complete graph and service list are available online[10]. The graph shows services, input/output types and conversions between types. Services are represented by rectangles, input/output types by ellipses and conversions by diamonds. Services are associated with their inputs and outputs respectively by incoming and outgoing arrows. Similarly, each conversion is associated with a source type and target type, it materializes an automatically detected conversion between two types. Connectivity of services in the graph materializes processing chains where output data are transformed according to the need of the service inputs. Conversions from external formats to our representation are not shown in the graph. Our algorithm finds direct links when the services use the same representation (the same type) to define the same data. This is the case, for example, with the link between services *Blast* and *ClustalW*. Indirect links correspond to a $conversion_i$. In the following, each $conversion_i(A, B)$ specifies a function to transform data of type A to data of type B. With *conversion0(ListOfComplexBiosequences, ListOfBiosequence)*, a list of simple sequences is derived from a list of complex sequences. The function converts each element of the list and produces a new list of the converted elements. The elements of the list are converted using Rule (LEFTSELECTION) (or Rule (RIGHTSELECTION)), and the new list is produced by Rule (MAP). *Conversion1(ListOfComplexBiosequences, Biosequence)* combines rules (CHOICE) and (LEFTSELECTION) (or (RIGHTSELECTION)) to go from a list of complex protein sequences to each simple biological sequence of the list. A simple biological sequence being a component of a complex biological sequence. *Conversion2(ComplexProteinSequence, Biosequence)* shows the generalization and specialization of types. Our algorithm detects that a sequence of proteins is also a biological sequence. The conversion uses (LEFTPOSTCHOICE) (or (RIGHTPOSTCHOICE)) to obtain a complex biological sequence from the complex protein sequence and uses Rule (LEFTSELECTION) (or (RIGHTSELECTION)) to obtain a biological sequence from the complex biological sequence. Our algorithm also individually considers the elements of a list. With *conversion3(ListOfBiosequences, Biosequence)* each element of a list of sequences can be selected, it is done by rule (CHOICE). With *conversion4(AnnotedSequence, Biosequence)*, a simple sequence is derived from an annotated sequence. A composite type is decomposed and some components are selected to feed services. This conversion corresponds to rules (LEFTSELECTION) and (RIGHTSELECTION). In above conversions, rules (TAGCHANGE), (TAGREMOVAL) and (PRIMITIVE) are used to convert between primitives and tags.

The complete graph contains 264 links between services out of the 900 possible links between 30 services (30x30). Our program therefore finds numerous links, but remains specific enough to be useful. Among those links, 88 are direct links, i.e., do not imply any conversion. Our convertibility relation therefore enables a three-fold increase of the number of links between services. The 30 services use 26 different types, among which 10 types are in fact ad-hoc and not decomposable (e.g., pictures, reports). Our program has identified 27 possible conversions between the 16 composite types, out of the 256 possible ones (16x16). This again shows that our approach is both productive and specific.

[10] http://www.irisa.fr/LIS/Members/moba/graph/view

We presented the graph of the experiment to developers and users of the GenOuest bioinformatics platform. They pointed out that "it is remarkable that the central role of sequence alignment is so visible in the graph". They stated that "the produced graph has a pedagogical interest". Indeed, in a typical course handout[11], a graph produced by hand related to a library of bioinformatics services contains similar services and connections as our graph. However, being more complete in the modelling of input and output data, our approach offers more flexibility on input and output types. Thus our algorithm provides more explicit connections and differentiation between categories of services. It also reveals possible conversions between input and output data, that creates new connections. Moreover, our graph is machine processable, it shows a proof of the connections created between services and can quickly take into account new changes on data types and services. With a growing set of currently over 1500 available tools, it is unlikely that people can produce the graph by hand. Our main objective is to guide biologists when composing workflows. One perspective is to take into account other aspects of services. When constructing a workflow, input and output types play a central role in the selection of services by setting constraints on applicable services, but are not the only criteria for selection biologist. It is also necessary to represent the services by their functional and non-functional properties (e.g., bioinformatics task performed, quality of results, provenance, efficiency, popularity). The GenOuest developers nevertheless mentioned that a user with domain knowledge would already find useful support in the produced graph to select services for a workflow among the possibilities given by the graph. Thus, they validated that the graph generated by our approach detects relevant information and produces, in a systematic way, knowledge usually acquired by experience. The costly step of the approach is the production of abstract types, currently done by hand. They highlighted some interesting perspectives. Our data abstractions could be enriched from ontologies, especially EDAM, which will significantly facilitate the type abstraction step and allow integrating others facets. In addition, data in Genomics (e.g., phylogenetic trees) and in others domains (e.g., metabolism) have to be added to the experiment.

5.2 Survey with Domain Experts

We evaluated the graph produced at Section 5.1 by asking domain experts to judge about the relevance of generated connections between services. For two services s_1 and s_2 in the graph, if the type of an output port of s_1 is convertible to the type of an input port of s_2, we suggested to experts to connect the input of s_2 to the output of s_1. We prepared 25 questions, each question being a suggestion to connect two services. Suggestions involved 18 services. The 25 questions concerned all connections on ports of 3 services (output port of blastp, input port of blast and input port of cons). For each question, it was possible to give one among 6 answers: "Yes, I knew", "Yes, I discover", "It may be feasible", "It is not feasible", "No, I disagree", and "I do not know". For questions we provided the description of services and a brief explanation of convertibility that involved the suggestion. To the 25 questions, we added general questions

[11] Presentation of services of Wisconsin Package- Olivier Collin - CNRS Roscoff - Formation Génopole Ouest - november 2002

about user status, background knowledge, the services they know by their names, and their opinion about the relevance of suggestions. The survey was submitted to experts of bioinformatics data, services and workflows via a mailing list.

We collected responses from 6 participants. Among the participants, one did no provide any answer on suggestions, and therefore has been removed from the results on suggestions. One did not finish the survey, thus for him we just consider answered questions. All participants work in bioinformatics. Four of them are computer scientists, four are users of services and workflows, four are developers of workflows, five are developers of services, one is a developer of Workflow Management Systems (WfMS) and one is a researcher. Each participant knew on average 7 services among the 18 services proposed as suggestions. 3 of the services were known to all participants, 7 were known to no participants. The number of known services may look surprisingly low for domain experts. It can be explained by the fact that participants are actually expert in one among many bioinformatics platforms. The same task is often implemented by different services (with different names) across different platforms.

Suggested links	**Cross-platforms**	**SL**	**SR**	**1st pers**	**2nd pers**	**3th pers**	**4th pers**	**5th pers**
blastp > blast	no	5	6	--	-	++	+	o
blastp > blastp	no	5	5	o	-	++	+	o
blastp > pepcoil	yes	5	0	o	-	+	+	+
blastp > diffseq	yes	5	2	o	-	+	+	+
blastp > emma	yes	5	0	o	-	+	+	+
blastp > clustalW	yes	5	5	o	-	++	+	++
blastp > backtranseq	yes	5	0	+	-	+	+	++
blastp > clustalWFastaCollection	yes	5	2	+	-	+	+	++
blastp < **blast**	no	5	6	o	-	++	+	o
cons < **blast**	yes	0	6	+	+	+	o	o
blast < **blast**	no	6	6	++	-	++	++	o
merger < **blast**	yes	0	6	x	+	+	-	?
transeq < **blast**	yes	1	6	x	--	+	++	++
blastn < **blast**	no	6	6	x	-	++	++	o
blastx < **blast**	no	6	6	x	-	++	++	o
extractFeat < **blast**	yes	0	6	x	?	+	+	o
extractAlign < **blast**	yes	1	6	x	+	+	+	o
backtranseq < **blast**	yes	0	6	x	+	+	+	++
gassst < **cons**	yes	5	0	x	o	++	+	--
matcher < **cons**	no	3	0	x	+	o	+	o
maxAlign < **cons**	yes	0	0	x	o	+	+	o
emma < **cons**	no	0	0	x	o	+	+	o
merger < **cons**	no	0	0	x	+	+	+	o
clustalW < **cons**	yes	5	0	x	+	++	+	++
clustalWFastaCollection < **cons**	yes	2	0	x	+	++	+	++

Fig. 10. Reponses of 5 participants to the list of link suggestions between services. 'Cross-platform' means services that are from different platforms. 'SL'/'SR' respectively count the number of participants who already knew the left/right service. Values in last the 5 columns are responses of participants for each suggestion. The responses are represented by (++) for "Yes, I knew", (+) for "Yes, I discover", (o) for "It may be feasible", (-) for "It is not feasible", (--) for "No, I disagree" and (?) for "I do not know". Crosses represent missing responses.

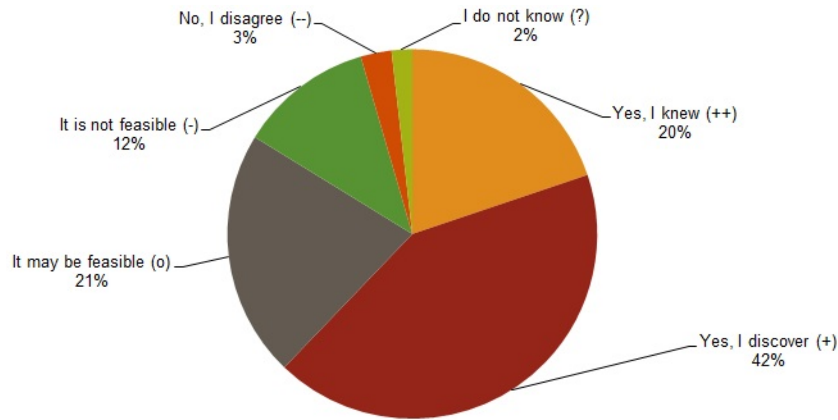

Fig. 11. Pie chat of all participant responses to all suggestions, per response type.

Figure 10 and Figure 11 show the results of the survey. The table of Figure 10 lists the responses for each participant and for each suggestion. The left column of the table contains suggestions established between services, for example '$\mathbf{blastp} > blast$' meant that the system suggested that service blast could consume the output of the service blastp; '$clustalW < \mathbf{cons}$': the system suggested that clustalW could produce the input of the service cons. The second column of the table tells if connected services come from different platforms. Columns SL and SR contain the number of participants who know the tool of either the left part (SL) or the right part (SR). From column four to column eight, are the responses of the participants. The responses are represented by (++) for "Yes, I knew", (+) for "Yes, I discover", (o) for "It may be feasible", (-) for "It is not feasible", (--) for "No, I disagree" and (?) for "I do not know". Missing responses are represented by crosses. Figure 11 aggregates results. It provides the percentage by response type for all responses.

The results show that participants found that 83% of suggested connections are acceptable. They already knew 20% and discovered 42% of suggested connections. They judged that 21% of suggested connections may be feasible. However, they found that 12% of suggested connections as not really feasible. They are in total disagreement with 3% of suggested connections. The participants have no opinion for 2% of suggested connections.

Despite knowing only a few (at most 15%) of the suggested services, participants still recognized most suggestions (83%) as acceptable. That paradoxical observation deserves an explanation. Many services are unknown to participants by their name, because belonging to another platform than the one they are used to, but their functions are known to them. For example, 'emma' has the same function as 'ClustalW', but only the latter is known by the participants. Thanks to the short description that we provided in the survey, and to online information, participants were able to recognize the function of unknown services, and to evaluate them accordingly. For example, the table of Figure 10 shows that 'emma' is evaluated very similarly to 'ClustalW'. This also explains the high proportion of "Yes, I discover" responses (42%), which is much

higher than we expected. It implies two positive results. Firstly, our suggestions cover many relevant service connections that would not be considered by domain experts in the construction of a workflow by lack of knowledge about services. Secondly, most of our suggestions are recognized as relevant by domain experts.

6 Related Work

The development of automatic solutions for service composition is a response to the time-consuming and error prone methods currently used in some platforms to manage service selection and service mediation during composition of services. Mediating incompatible services requires identifying categories of mismatches. The work of Li et al. [11] provides a multi-dimensional classification of mismatches. It identifies syntactic and semantic mismatches of functional and non-functional properties. That systematic classification of service composition mismatches helps understand the problem. It also helps find appropriate solutions for each case (Sirin et al [29], Lin et al [30], Kongdenfha et al [31]). Our work concerns signature mismatches that occur on the structure and on the semantics of service parameters.

There are several approaches that provide solutions for data mismatches in scientific workflows. Besides approaches that address verifying matching between services, there are approaches that resolve data mismatches by means of shims to insert between services in workflows. They can be divided in two categories: A first category relies on semantic annotations to search shims in existing libraries. A second category takes into account syntactic descriptions of data types to automatically generate shims.

Some approaches that address service matching use ontologies such as EDAM [32] and myGrid ontology [33]. They provide methods to access semantic compatibility of workflow components. However, they do not generally guarantee compatibility at syntactic level. For example, Lebreton et al. [12] propose to verify the semantic compatibility of service parameters for web service composition. They address input and output matching but they do not provide solutions to resolve data mismatches that can occur on the data structure. Another approach, by Stroulia et al. [34], uses the structure of data types, messages, operations and textual descriptions to assess similarity between WSDL (Web Service Description Language) specifications. However, it is not designed to adapt data between services.

Compared to previous works, approaches that find shims address resolving data mismatches in workflows. They generally rely on data types, formats and service descriptions. For example, Velasco-Elizondo et al. [13] use data format descriptions to automatically identify relevant shims. In the same manner, the approach of Hull et al [14] relies on the description of shims and data types to retrieve shims transforming data between services. These approaches, besides seeing data as not decomposable, also expect the shims to be provided by third-parties.

In contrast to approaches that search shims, approaches that generate shims automatically provide data transformers to insert between services. They are characterized by the data complexity they support and the transformations between data they offer. Browers et al [15] support structural data transformations based on ontological information. Their approach links a structural type to a corresponding semantic type. The

semantic type is defined with concepts of ontologies. However, the transformations that they offer rely on contextual paths. They do not allow transforming XML documents whose elements are not already attached to domain concepts. They also, do not seem to take into account recursion, which is required for some complex data. In Conveyor, Linke et al. [35] propose generic object-oriented type system to manage nodes, including data types, in workflows. They use interfaces, abstract classes and inheritances to express relations between types but do not mention automated methods ensuring composition and decomposition of complex data types.

Kashlev et al [16] use rules to automatically insert shims as coercions into executable workflows. Their approach and the approach of Dibernado et al [17] are close to our work. The first one because it uses rules, and the second one because it is applied to compose bioinformatics services. We differ from the approach of Kashlev et al. [16] on some points. We envision to mediate data at workflow construction time, not at execution time. We allow more complex data representations, for example type constructors union, tag and list. Thus, we offer additional transformations between input and output data. For example, the approach of Kashlev et al. [16] requires different labels for elements of a tuple while our abstraction authorizes to define a tuple where elements use the same label. This allows us to meet some bioinformatics data representations such as representing lists of biological sequences, and enables parallel transformation on lists. We differ from the approach of Dibernado et al. [17] because we allow decomposing as well as composing data. Dibernado et al. [17] provide the data transformation during workflow composition. They allow generating shims according to connected services. However, they decompose data but do not allow to compose them.

Compared to presented approaches, our approach offers richer type abstractions and more complex transformations between data. It relies on rules that allow to systematically reason on input and output types for managing mismatches during workflow composition. In addition, our approach does not prevent to use existing shims, it provides systematic and automatic mechanisms to re-use some of them (e.g., converters for upper case, lower case or substitution). Unlike many approaches, our approach does not only provide the solution in XML, it also provides mechanisms to link existing textual representations. Furthermore, our approach may easily benefit from mechanisms of matching based on ontologies.

Besides the automatic approaches, platforms such as Galaxy [4] use shims libraries to manage links between services. Matching between inputs and outputs is based on data formats. Services, in their implementation, take into account several formats. When a format is not provided, a format converter defined by hand is used. These techniques make a strong dependency between services and formats, which mixes domain tasks and tasks for the data adaptation. In addition, as discussed above, textual formats do not promote automation. Separating data types and formats is the subject of much work, for example Kalas et al. [9]. To facilitate interoperability of tools, common XML-based formats are proposed to represent bioinformatics data. At present, few implementations use these technologies. A generalization of their use would strengthen our approach, as it would facilitate the specification of abstract data types. Possibilities offered by XML technologies to represent complex data and relationships associated to domain ontolo-

gies may be used to provide a pivot language for conversion between heterogeneous data formats.

7 Perspectives

In the future, we plan to use our convertibility approach to propose an guided approach for composing real-world workflows. Our approach is appropriate for users of platforms such as Taverna [1] and Galaxy [4] that need to benefit more from automation. It can also be used to manage creation, re-use and conversion of parameters in workflow engines such as Bpipe [36], Snakemake [37] that use low-level programming. For our approach to be more beneficial, we will cope with data consistency and deal with non-determinism. We will integrate data consistency checking compared to types. Errors due to the non-compliance between data and constraints are common in automated platforms. For example, little changes in textual formats used to represent input data may cause a program to abort. An advantage of our approach is that it offers means to verify the compliance of data compared to their data type. Enriching abstractions with ontologies will allow to reduce non-determinism. It will be necessary, however, to integrate users in the process to manage multiple choices.

8 Conclusion

Data mismatches between services make it difficult to create scientific workflows. Manually defined shims proposed to fix data mismatches are time consuming and error prone. Existing approaches that automatically insert shim services in workflows are limited in the transformations they provide.

In this paper, we presented an approach that systematically detects convertibility from output types to input types. We have defined convertibility rules that exploit (de)composition as well as specialization and generalization of types. The rules also automatically generate converters between input and output XML data that can be used as shims. An experiment on bioinformatics services, as well as a survey with domain experts, showed that the detected convertibilities and produced converters are relevant from a biological point of view. Furthermore, the automatically produced graph of potentially compatible services exhibited a connectivity higher than with the ad'hoc approaches.

Acknowledgment. We thank Olivier Collin, Yvan Le Bras, Olivier Dameron, Francois Moreews and Olivier Sallou for their expertise in bioinformatics services and workflows, as well as for enriching discussions.

References

1. T. Oinn, M. Greenwood, M. Addis, J. Ferris, K. Glover, C. Goble, D. Hull, D. Marvin, P. Li, and P. Lord, "Taverna: Lessons in creating a workflow environment for the life sciences," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1067–1100, 2006.

2. S. Gundersen, M. Kalas, O. Abul, A. Frigessi, E. Hovig, and G. K. Sandve, "Identifying elemental genomic track types and representing them uniformly," *BMC Bioinformatics*, vol. 12, p. 494, 2011.
3. P. Rice, I. Longden, and A. Bleasby, "Emboss: the european molecular biology open software suite," *Trends in genetics*, vol. 16, no. 6, pp. 276–277, 2000.
4. J. Goecks, A. Nekrutenko, J. Taylor, and T. G. Team, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biology*, vol. 11, no. 8, p. R86, 2010.
5. H. Ménager, V. Gopalan, B. Néron, S. Larroudé, J. Maupetit, A. Saladin, P. Tufféry, Y. Huyen, and B. Caudron, "Bioinformatics applications discovery and composition with the mobyle suite and mobylenet," in *RED*, pp. 11–22, 2010.
6. I. H. C. Wassink, P. E. van der Vet, K. Wolstencroft, P. B. T. Neerincx, M. Roos, H. Rauwerda, and T. M. Breit, "Analysing scientific workflows: Why workflows not only connect web services," in *SERVICES I*, pp. 314–321, 2009.
7. P. N. Seibel, J. Krüger, S. Hartmeier, K. Schwarzer, K. Löwenthal, H. Mersch, T. Dandekar, and R. Giegerich, "Xml schemas for common bioinformatic data types and their application in workflow systems," *BMC Bioinformatics*, vol. 7, p. 490, 2006.
8. M. V. Han and C. M. Zmasek, "phyloxml: Xml for evolutionary biology and comparative genomics," *BMC Bioinformatics*, vol. 10, p. 356, 2009.
9. M. Kalas, P. Puntervoll, A. Joseph, E. Bartaseviciute, A. Töpfer, P. Venkataraman, S. Pettifer, J. C. Bryne, J. C. Ison, C. Blanchet, K. Rapacki, and I. Jonassen, "Bioxsd: the common data-exchange format for everyday bioinformatics web services," *Bioinformatics*, vol. 26, no. 18, 2010.
10. D. W. Embley, L. Xu, and Y. Ding, "Automatic direct and indirect schema mapping: Experiences and lessons learned," *SIGMOD Record*, vol. 33, no. 4, pp. 14–19, 2004.
11. X. Li, Y. Fan, and F. Jiang, "A classification of service composition mismatches to support service mediation," in *GCC*, pp. 315–321, 2007.
12. N. Lebreton, C. Blanchet, D. B. Claro, J. Chabalier, A. Burgun, and O. Dameron, "Verification of parameters semantic compatibility for semi-automatic web service composition: a generic case study.," in *Int. Conf. on Information Integration and Web Based Applications and Services* (D. Taniar, E. Pardede, H.-Q. Nguyen, J. W. Rahayu, and I. Khalil, eds.), pp. 845–848, ACM, 2010.
13. P. V. Elizondo, V. Dwivedi, D. Garlan, B. R. Schmerl, and J. M. Fernandes, "Resolving data mismatches in end-user compositions," in *IS-EUD*, pp. 120–136, 2013.
14. D. Hull, R. Stevens, P. Lord, C. Wroe, and C. Goble, "Treating ?shimantic web? syndrome with ontologies," 2004.
15. S. Bowers and B. Ludäscher, "An ontology-driven framework for data transformation in scientific workflows," in *Data Integration in the Life Sciences, First International Workshop, DILS 2004, Leipzig, Germany, March 25-26, 2004, Proceedings*, pp. 1–16, 2004.
16. A. Kashlev, S. Lu, and A. Chebotko, "Coercion approach to the shimming problem in scientific workflows," in *2013 IEEE International Conference on Services Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*, pp. 416–423, 2013.
17. M. DiBernardo, R. Pottinger, and M. Wilkinson, "Semi-automatic web service composition for the life sciences using the biomoby semantic web framework," *Journal of Biomedical Informatics*, vol. 41, no. 5, pp. 837–847, 2008.
18. M. Ba, S. Ferré, and M. Ducassé, "Generating data converters to help compose services in bioinformatics workflows," in *Database and Expert Systems Applications - 25th International Conference, DEXA 2014, Munich, Germany, September 1-4, 2014. Proceedings, Part I*, pp. 284–298, 2014.

19. P. Missier, K. Wolstencroft, F. Tanoh, P. Li, S. Bechhofer, K. Belhajjame, S. Pettifer, and C. A. Goble, “Functional units: Abstractions for web service annotations.,” in *SERVICES*, pp. 306–313, IEEE Computer Society, 2010.
20. H. Hosoya, J. Vouillon, and B. C. Pierce, “Regular expression types for xml,” in *ICFP*, pp. 11–22, 2000.
21. Z. Chen, J. Wu, S. Deng, Y. Li, and Z. Wu, “Describing and verifying web service using type theory,” in *Proceedings of the 10th International Conference on CSCW in Design, CSCWD 2006, May 3-5, 2006, Southeast University, Nanjing, China*, pp. 746–750, 2006.
22. J. L. Bates and R. L. Constable, “Proofs as programs,” *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 113–136, 1985.
23. F. Moreews and D. Lavenier, “Seamless coarse grained parallelism integration in intensive bioinformatics workflows,” in *20th European MPI Users’s Group Meeting, EuroMPI ’13, Madrid, Spain - September 15 - 18, 2013*, pp. 277–282, 2013.
24. J. D. Westbrook, N. Ito, H. Nakamura, K. Henrick, and H. M. Berman, “Pdbml: the representation of archival macromolecular structure data in xml,” *Bioinformatics*, vol. 21, no. 7, pp. 988–992, 2005.
25. R. D. Dowell, R. M. Jokerst, A. Day, S. R. Eddy, and L. Stein, “The distributed annotation system,” *BMC Bioinformatics*, vol. 2, p. 7, 2001.
26. U. Consortium *et al.*, “The universal protein resource (uniprot) in 2010,” *Nucleic Acids Research*, vol. 38, no. Database-Issue, pp. 142–148, 2010.
27. H. McWilliam, F. Valentin, M. Goujon, W. Li, M. Narayanasamy, J. Martin, T. Miyar, and R. Lopez, “Web services at the european bioinformatics institute-2009,” *Nucleic Acids Research*, vol. 37, no. Web-Server-Issue, pp. 6–10, 2009.
28. M. D. Wilkinson and M. Links, “Biomoby: An open source biological web services proposal.,” *Briefings in Bioinformatics*, vol. 3, no. 4, pp. 331–341, 2002.
29. E. Sirin, J. Hendler, and B. Parsia, “Semi-automatic composition of web services using semantic descriptions,” in *Web services: modeling, architecture and infrastructure workshop in ICEIS*, vol. 2003, Citeseer, 2003.
30. C. Lin, S. Lu, X. Fei, D. Pai, and J. Hua, “A task abstraction and mapping approach to the shimming problem in scientific workflows,” in *2009 IEEE International Conference on Services Computing (SCC2009), 21-25 September 2009, Bangalore, India*, pp. 284–291, 2009.
31. W. Kongdenfha, H. R. M. Nezhad, B. Benatallah, F. Casati, and R. Saint-Paul, “Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters,” *IEEE T. Services Computing*, vol. 2, no. 2, pp. 94–107, 2009.
32. J. C. Ison, M. Kalas, I. Jonassen, D. M. Bolser, M. Uludag, H. McWilliam, J. Malone, R. Lopez, S. Pettifer, and P. M. Rice, “Edam: an ontology of bioinformatics operations, types of data and identifiers, topics and formats,” *Bioinformatics*, vol. 29, no. 10, pp. 1325–1332, 2013.
33. K. Wolstencroft, P. Alper, D. Hull, C. Wroe, P. W. Lord, R. D. Stevens, and C. A. Goble, “The myGrid ontology: bioinformatics service discovery.,” *Int. Journal of Bioinformatics Research and Applications*, vol. 3, no. 3, pp. 303–325, 2007.
34. E. Stroulia and Y. Wang, “Structural and semantic matching for assessing web-service similarity,” *Int. J. Cooperative Inf. Syst.*, vol. 14, no. 4, pp. 407–438, 2005.
35. B. Linke, R. Giegerich, and A. Goesmann, “Conveyor: a workflow engine for bioinformatic analyses,” *Bioinformatics*, vol. 27, no. 7, pp. 903–911, 2011.
36. S. P. Sadedin, B. Pope, and A. Oshlack, “Bpipe: a tool for running and managing bioinformatics pipelines,” *Bioinformatics*, vol. 28, no. 11, pp. 1525–1526, 2012.
37. J. Köster and S. Rahmann, “Snakemake:a scalable bioinformatics workflow engine,” *Bioinformatics*, vol. 28, no. 19, pp. 2520–2522, 2012.