



Semi-automatic web service composition for the life sciences using the BioMoby semantic web framework

Michael DiBernardo^{a,*}, Rachel Pottinger^a, Mark Wilkinson^b

^a University of British Columbia, Department of Computer Science, 201-2366 Main Mall, Vancouver, BC, Canada V6T 1Z4

^b James Hogg iCAPTURE Centre for Cardiovascular and Pulmonary Research, Room 166, St. Paul's Hospital, 1081 Burrard Street, Vancouver, BC, Canada V6Z 1Y6

ARTICLE INFO

Article history:

Received 31 August 2007

Available online 4 March 2008

PACS:

10.000

70.000

90.000

100.00

210.000

240.000

280.000

380.000

Keywords:

Automated web service composition

Web service interoperability

Workflow assembly

Semantic web

BioMoby

ABSTRACT

Researchers in the life-sciences are currently limited to small-scale informatics experiments and analyses because of the lack of interoperability among life-sciences web services. This limitation can be addressed by annotating services and their interfaces with semantic information, so that interoperability problems can be reasoned about programmatically. The Moby semantic web framework is a popular and mature platform that is used for this purpose. However, the number of services that are available to select from when building a workflow is becoming unmanageable for users. As such, attempts have been made to assist with service selection and composition. These tasks fall under the general label of *automated service composition*.

We present a prototype workflow assembly client that reduces the number of choices that users have to make by (1) restricting the overall set of services presented to them and (2) ranking services so that the the most desirable ones are presented first. We demonstrate via an evaluation of this prototype that a unification of relatively simple techniques can rank desirable services highly while maintaining interactive response times.

© 2008 Elsevier Inc. All rights reserved.

1. Introduction

As a result of the shift in emphasis from qualitative to quantitative approaches in the life-sciences, many data sets and analytic services are available only via web services that expose their data or functionality through unstandardized HTML interfaces, using a slew of divergent data formats [1]. Due to the pace and the haphazard nature with which this has occurred, service interoperability has become a major problem. Much research has been done on how to best enable scientists to analyze and retrieve data across disparate web services without having to concern themselves with the details of data conversion and service interoperability.

One way to address the problem is to annotate services and their parameters with semantic information. The interface and function of a web service is thus described with a specificity beyond what standard web service technologies (e.g. WSDL) would allow. For example, a service that declares that it takes a string as input does not provide a client with enough information to

know what format the string should be in. Is it supposed to be a DNA sequence? An output report from another program? A free-form text description? In order to answer these sorts of queries, semantic frameworks provide ontologies that enumerate the various data types and services that are used in life-sciences web services, as well as describing the relationships between them. Programs that are used to build these service compositions, such as the popular Taverna client [2–4], can then be written or extended to leverage this extra information to help users assemble complex service compositions, or *workflows*, that perform functions that are unavailable as atomic services. Specifically, this extra information helps the user compose only those services that are compatible with one another.

In the life-sciences, the Moby semantic web framework is used widely for this purpose [5,6]. As the Moby framework has matured and been increasingly adopted by the research community, a large number of web services have been retrofitted with Moby interfaces. This complicates the task of workflow assembly, as any given data item that the workflow designer is interested in may be produced or consumed by a large number of different services that he or she must then select from (Mark Wilkinson, personal communication, 2007). As of August 22nd, 2007, a workflow designer that is

* Corresponding author. Fax: +1 604 742 0542.

E-mail addresses: mikedebo@gmail.com (M. DiBernardo), rap@cs.ubc.ca (R. Pottinger), markw@illuminae.com (M. Wilkinson).

restricting herself to using only Moby services will still have to select from a set of 1181 services that are declared over a space of 528 data types. In general, as any semantic web framework increases in popularity, one would naturally expect the space of services to become unmanageably large. The goal of our work is to improve the service selection process so that the user is presented only with a small set of the most salient services, thus reducing the time and effort required to build workflows.

This problem has been addressed previously in the more general context of the entire space of services available on the world wide web. These approaches range from incrementally assisting the user at each step of the service composition process (known as *semi-automatic service composition*) to inferring the entirety of the desired workflow with little to no direction from the user (known as *automatic service composition*, or ASC) [7,8]. However, to our knowledge, these approaches have not been applied directly to the life-sciences, and little has been done to demonstrate that the assistance or results provided by these clients result in workflows that are close to what the user actually desires.

This paper describes a prototype solution to the service selection problem in the life-sciences using the Moby environment, but our methods are generally applicable to any semantic web framework. We use a semi-automatic approach to interactively provide the user with a small set of salient services at each step. This contrasts with fully-automatic methods that do a great deal of up-front computation, present the inferred workflow, and then require the user to disassemble and rebuild the segments of the workflow that are found to be unsatisfactory.

Our solution consists of:

- (1) An interaction model similar to that provided by Taverna, so as to enable rapid adoption by those users familiar with this popular client. However, the model has been enhanced to (a) enable the user to ‘conceal’ portions of the workflow that they are not interested in at the moment and (b) present potential useful services incrementally to the user. These enhancements allow our underlying semi-automatic service composition algorithm to reason more quickly and specifically about the problem the user is trying to solve.
- (2) A service selection and ranking algorithm that uses lazy breadth-first search over an implicit graph of the service space. This algorithm has been designed to minimize the time to first results when the user attempts to add a new service to the workflow being constructed, and to present the user only with services that can eventually furnish the unbound input or output parameters in the partially-constructed workflow. A variant of the algorithm is also presented that leverages simple usage statistics to improve the service ranking.

We include an evaluation of how closely the ranking reflects the true saliency of the results when building actual service compositions used by the Moby community.

Our solution is novel in the following respects:

- (1) To our knowledge, we present the most extensive automated workflow assembly approach in life-sciences service integration in a practical environment such as Moby, which has an active user base.
- (2) Previous approaches have not provided explicit support for composite types—that is, types that are described in terms of other component types (e.g. in the Moby ontologies, a DNASequences is ‘made of’ a sequence of type String and a length of type Integer).
- (3) We provide an evaluation of how our method performs on a small number of workflows that were developed by other

authors for purposes of demonstration, and for use in actual life-sciences research.

The paper is organized as follows. Section 2 gives a precise definition of the problem that we are trying to solve, and provides a concrete problem scenario that is referenced throughout the rest of the paper to illustrate key concepts. Section 3 describes the proposed solution. Section 4 presents an evaluation of our work, and Section 5 compares our approach and results to existing solutions. We conclude by identifying future directions for this work in Section 6.

2. Problem description

2.1. Example use case

Throughout the paper we will use the example scenario depicted in Fig. 1 to assist in the explanation of our approach. The example uses a set of simplified data types and services that are similar to ones that can be found in the actual Moby ontologies.

In this use case, the user wishes to produce an aggregate report on different loci in an organism of interest. She refers to these loci using an identifier called an ‘A_ID’. She wishes to retrieve all phenotypes of a locus, as well as all sequences that are similar to the canonical sequence at that locus. This use case is loosely based on the integration tasks described in Wilkinson et al. [6]

The services (possibly unbeknownst to her) that will perform this task are shown in Fig. 1(a). There is no single service that will take an ‘A_ID’ and produce a list of phenotypes; however, there is a service that converts ‘A_ID’s to ‘B_ID’s, and there does exist a service that will produce the phenotypes for a locus represented by a ‘B_ID’. For legacy reasons, this ‘PhenoBank’ is stated to take plain text as input. In the ontology, ‘B_ID’s are a subclass of ‘Text’, and so are compatible with this service. Another service that we name the ‘SeqBank’ takes an ‘A_ID’ as input and produces the canonical sequence for the locus as output. Finally, the BLAST service takes a String as input and produces a report containing similar sequences. Note that the input type of String implies that the DNASeq output of the SeqBank cannot be supplied directly to the BLAST service. Instead, only the String component of the DNASeq should be forwarded to it.

The final composition that will perform the task as specified is shown in Fig. 1(b).

2.2. Problem definition

In this section, we provide a more precise specification of the problem introduced in Section 1 and describe our proposed solution.

Based on our discussion in the introduction, we can specify that, at each step of the service assembly process, we wish to:

- (1) Only present the user with the set of the services that initiate a service chain from the source data to the goal inputs or outputs (i.e. only suggest services that could eventually furnish the goals).
- (2) Order the services within this set such that the most ‘desirable’ services appear first in the list, given the context of the workflow being built.
- (3) Take into account superclass, subclass, and composite/component relationships when performing this derivation—for instance, the algorithm should be smart enough to realize that if one is trying to produce a String output, it should also furnish services that produce DNASequences, since a DNASequences can be decomposed into a String and an Integer.
- (4) Satisfy all of the preceding requirements while returning results to the user within 0.5 s.

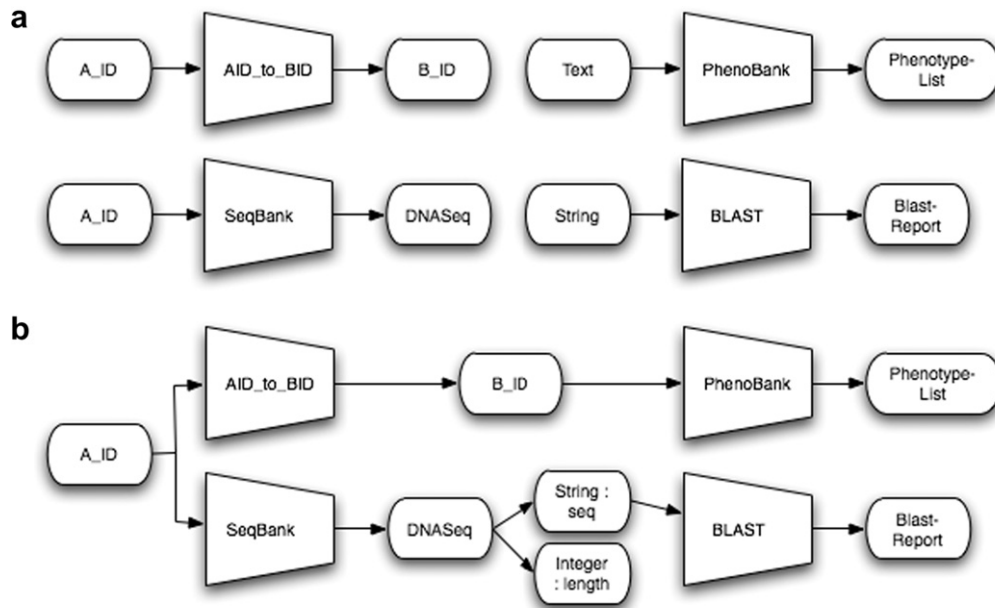


Fig. 1. Individual services desired by user (above) and desired composition of these services (below) in the example use case.

We now describe a solution that is designed to meet these goals.

3. Proposed solution

Our proposed solution consists of two major components: The interaction model, and the underlying algorithms that it invokes. In order to facilitate the description of these components, we first define a series of terms that will be used throughout the paper to discuss our approach in Section 3.1. We then describe these components in detail in Sections 3.2 and 3.3.

3.1. Definitions

A workflow is made up of connections between services. A service exposes zero or more input *parameters* and produces one or more output *data*. When the data of one service is supplied as the value of a parameter of another, we say that the data and parameter have been *bound*.

A *composite* type is one that declares one or more has a relationships to *component* data. These relationships can be declared in and inherited from a superclass.

A parameter or datum is said to be *open* if it has not yet been bound, and is *closed* if it has. A parameter can only be bound to one datum, but a datum may be bound to multiple parameters, and so a closed datum is actually one that has been bound to one or more parameters.

We call the open parameters or data of the workflow the *goals* of the composition algorithm, depending on the context. If one is introducing a service that consumes an output and produces another output that we hope to bind to an open parameter (*forward construction*), then the set of all open parameters are considered to be the goals. The opposite is true for the reverse direction (*reverse construction*).

3.2. The interaction model

The interaction model consists of how the workflow is presented to the user, and what operations the user can perform on its visual representation. Our interaction model differs from existing ones by:

- (1) Providing explicit support for the introduction and manipulation of composites. Specifically, we allow the decomposition of composite items into their components, and recomposition of data items into composites. This permits operations that are difficult or impossible to perform in other clients (e.g. the Taverna BioMoby plugin), such as feeding individual components into different services.
- (2) Allowing the user to direct the behavior of the underlying algorithm by concealing the inputs or outputs of services that she is not currently interested in binding. The intention is to give the user a mechanism to enhance the underlying selection algorithm's ability to restrict the possible service space.
- (3) Adopting a design that presumes that the user is interested in minimizing the time to first results, rather than overall waiting time. The user navigates through the result set by viewing small portions of it at a time, much like one is only able to examine tens of hits at once in a conventional web search engine.

The main challenge in the design of the interaction model is deciding on the set of operations to provide. These operations were selected based on the goals that we are trying to achieve.

The core operation is the *bind* operation, which connects a datum to an open parameter and closes both. It serves as an atomic construction operation that is used to implement the others, but can also be directly invoked by the user. Closed parameters are disqualified from consideration as goals by the suggestion algorithm.

To permit manipulation of composite types, we provide *decompose* and *compose* operations on data nodes. A user that wishes to treat a composite type (e.g. a DNasequence) as a 'bag' of individual components (in this case, a String sequence and Integer length) can choose to decompose the item by exposing its components to the workflow. When this is done, the composite item is closed, and the furnished components are considered open. This permits the user to feed the String sequence and Integer length into different services. If she later decides that she would rather treat the composite type as a single atom (for instance, to feed the DNasequence datum into a service that requires a DNasequence as input), the *compose* operation can be applied to remove the components from the model.

In order to support the local interaction model used by Taverna, we must provide the basic *feed into* and *produce with* operations that are used to permit forward construction from data and backward construction from parameters, respectively. The set of services to be furnished is restricted at this stage by having the user specify what toplevel Moby service type (e.g. Analyses, Retrieval, etc.) they are interested in. This functions as a crude but non-invasive 'keyword search' that does not disrupt the user's navigation.

We support a very basic 'declaration of intent' mechanism by allowing the user to *conceal* and *expose* the open inputs and outputs of services, similar to the way that the components of a composite type can be added to and removed from the workflow via the *decompose* and *compose* operations.

However, even if the user is able to communicate that they are working on a very narrow segment of the workflow, it is still conceivable that the resulting set of services under consideration could be quite large if she is working with a particularly 'popular' data type. We address this concern by considering the algorithm not to be a conventional procedure that takes a set of inputs and produces a set of outputs, but instead to be a *generator* that incrementally returns results until it is exhausted. We support the notion of the algorithm as a generator directly in the client very simply by displaying results to the user a handful at a time, much like a conventional search engine. This makes it possible to perform a great deal of the remaining computation in a separate thread or process while the user is thinking about the salience of the results she has already been presented with.

Fig. 2 gives an example of how the model works in practice. In 2(a) the user has partially-constructed the desired workflow from Fig. 1(b). Deciding to concentrate first on the bottom chain, she 'conceals' the output of the converter service to communicate this focus, and then uses the 'produce with' operation to connect the BLAST service to the BlastReport output (Fig. 2(b)). The BLAST service appears in the ranked list because it consumes an input of type String, and the open DNASEq datum has a String as a component. In Fig. 2(c), the user decomposes the DNASEq and uses the 'bind' operation to connect the String component to the open String parameter of the BLAST service.

3.3. The service selection algorithm

In the previous section, we described an interaction model that (1) supports local assistance but allows the user to provide 'hints' for global reasoning, and that (2) presents batches of results that have (3) been appropriately ranked.

We will now incrementally develop an algorithm that incorporates each of these numbered items in turn. Our algorithm differs from existing ones by:

- (1) Reasoning over services that produce and consume composite types. To our knowledge, existing clients do not take into account the fact that some component of a type may be compatible with a service that leads to a goal. For example, in the discussion of the interaction model we presented an

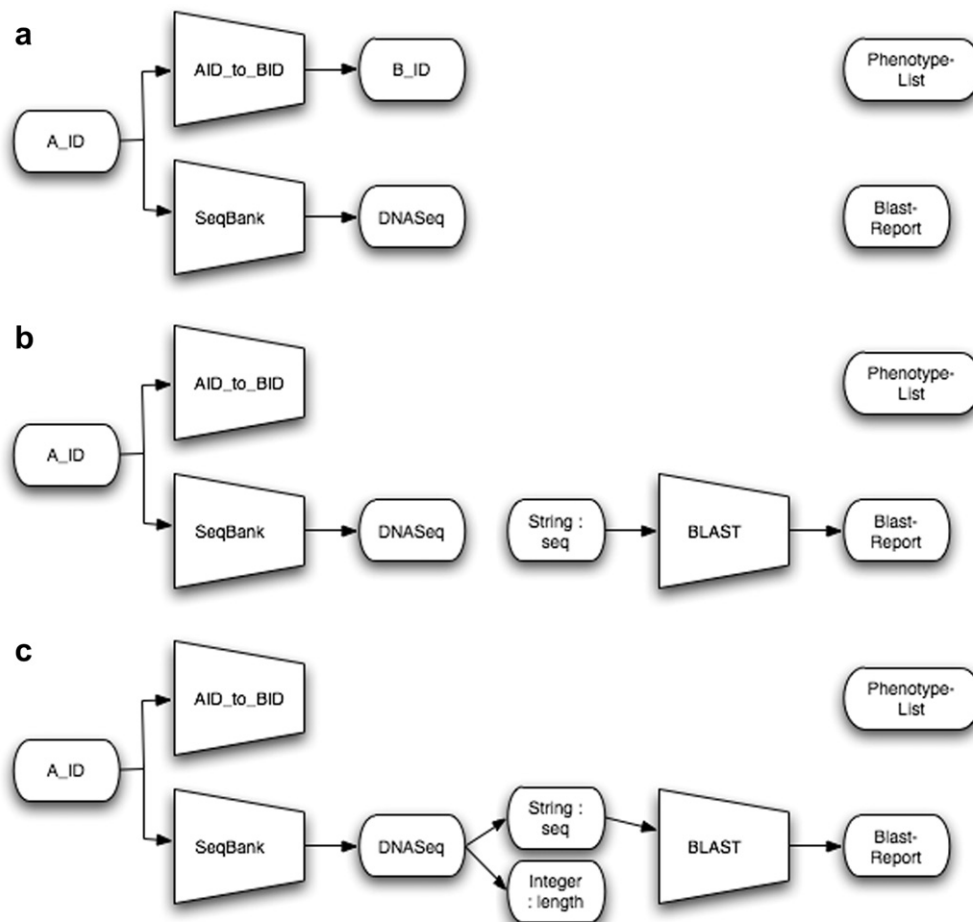


Fig. 2. Example of how interaction model works. (a) Depicts the partial workflow to be built upon. (b) Shows the workflow after the converter output has been concealed and the BLAST service has been selected to produce the report output. In (c), the bottom chain is completed by first decomposing the DNASEq type into its String and Integer components, and then binding the String component of the DNASEq output to the input parameter of the BLAST service.

example where the BLAST service, which has a parameter of type String, was still a desirable choice even though there was no open String datum in the partial workflow (Fig. 2(b)). This is because the open DNaseq datum has a component of type String.

- (2) Minimizing the time to first results instead of minimizing overall computation time. The advantages of this have already been discussed in Section 3.2.

The basic algorithm changes slightly depending on whether the user is searching for a service to feed a datum into (forward construction), or for a service to produce a parameter (reverse construction). In the forward construction case, a ‘forward-chaining’ search algorithm is used, in which the search progresses from inputs to goal outputs, while in reverse construction the opposite procedure, ‘backwards-chaining’, is used. However, the algorithm is almost identical for both the forward-chaining and backwards-chaining scenarios, and so we will simply assume the forward-chaining case for ease of exposition.

3.3.1. Overview of algorithm

We define the *potential service set*, P , as the set of all services that can consume the selected datum (that is, the input is the same as or a superclass of the datum type). We wish to select from P the subset of services R that initiate a service chain that produces an output that is *compatible* with one of the goal outputs. (We defer a precise definition of type compatibility to Section 3.3.2.) Furthermore, we wish to present the services to R in an order that reflects their saliency.

We can compute R by building a graph of all possible service chains that originate at services in P and selecting only those paths that lead to a goal. We build this graph by representing services as nodes, inserting an edge from node s_i to node s_j if the service represented by s_i has an output that is compatible with an input parameter of the service represented by s_j . We then initiate searches from all services in P and retain those that originate a path to a goal output.

For example, consider the depiction of the selection process in Fig. 3. The goal parameters are of type Phenotype-List and BlastReport, and the selected datum is of type ‘A_ID’. Here, P is the set of all services that can consume an ‘A_ID’ or one of its superclasses. R will contain at least the services ‘AID_to_BID’ and ‘SeqBank’, because we know from Fig. 1(b) that they both initiate chains to the goal outputs.

Building this graph in its entirety and then performing searches over it would require space and time at worst cubic in the number

of available services at each invocation, which is unacceptable. We also have yet to address the issue of how the search will be performed so as to furnish results in ranked order. These details are provided in Section 3.3.3: However, in order for this discussion to make sense, we must first provide the definition of type compatibility mentioned earlier.

3.3.2. Type compatibility

Two services in a chain can be connected for the following reasons:

- (1) An output of the first is of the same type as an input of the second.
- (2) An output of the first is the same as one of the subclasses of the inputs of the second.
- (3) Some component (or a component of one the components, etc.) of some output of the first is the same as the type of one of the inputs of the second, or a subclass of one of these inputs.

The first two points are straightforward. One of the novel aspects of our algorithm is that we address the third point by using a recursive definition of compatibility that can be directly implemented as a compatibility detection algorithm.

A datum can bind to an input in some service chain if:

- (1) The type of the input is the same or a superclass of the datum, or
- (2) If any of the components of the datum can similarly bind to the input.

This definition implicitly ‘expands’ all of the components and nested subcomponents of a composite datum to check for compatibility with the input type, thus fulfilling the third requirement.

Note that this definition does not take into account the fact that the inputs under examination may themselves be composites: For simplicity, it is assumed that the connection between two components would be so ‘tenuous’ as to almost certainly be uninformative.

As an example, suppose the datum under consideration is of a (fictional) SequencePair type that is composed of two NucleotideSequences. The algorithm will first check if the SequencePair is of the same type of the input or one of its superclasses; it will then check if the component NucleotideSequences have this property; and finally it will check if the String and Integer components of the NucleotideSequences have this property.

3.3.3. Lazy graph construction and ranking

In order to achieve the desired response time, we modify the algorithm as presented earlier to function as a generator that furnishes search results in small ‘batches’ that are delivered in order of saliency. Therefore, the overall runtime of the entire algorithm has the potential to be great, but the majority of the work is done while the user is browsing the results already presented.

We first discuss how the graph search may be modified to incrementally produce results without constructing the entire graph beforehand. We do this by running breadth-first search one ‘step’ at a time across all of P . That is, instead of sequentially initiating exhaustive searches on each service in P , we repeatedly iterate over P and evolve the state of each search a little bit at a time. This requires us to maintain only the ‘fringe’ of the search, instead of the whole graph. During a ‘step’ in this search, if we find that a path originating from a service $p \in P$ happens to produce an output compatible with a goal, we remove p from P and add it to R . If the size of R is sufficiently large after any particular step (for example, 20), we deliver those services to the user at that time. Other-

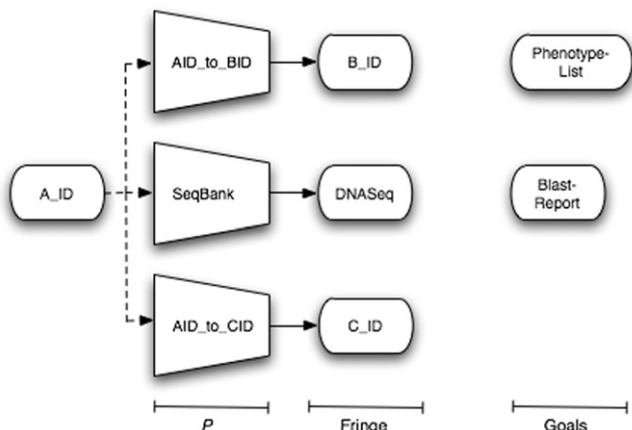


Fig. 3. Initiation of algorithm when the user selects the ‘A_ID’ datum from the use case presented in Fig. 1.

wise, subsequent steps are taken until R has grown sufficiently large or we have determined that the remaining services in P cannot lead to a goal.

We have yet to define what is done in a single ‘step’ in our incremental search. This is because the nature of how these steps are taken affects the order in which the results are furnished, and thus directly determines the ranking of these services. We now specify how the search procedure imposes this ranking.

First, we must decide upon the criteria that are valued in the ordering. One early idea was to mine Moby usage data and statistics in order to discern which services were frequently invoked in sequence; however, at the time this project was started, there was very little usage data being collected, and thus we had to develop an approach that leveraged only the structure of the problem description and of the implicit service graph as built by the generator.

Our intuition was that service chains that (1) lead from the selected datum to a goal output in the fewest number of hops with (2) input/output bindings that were extremely ‘similar’ to one another would be more desirable than those that chains that do not possess these properties. These assumptions are similar to those made in Arpinar et al [11]. We selected breadth-first search as our searching algorithm for precisely this reason, as it will furnish the shortest chains first.

In the first ‘step’ of the search, for each $p \in P$, we examine all services that have an input parameter that exactly matches one of the output datum types of p . If any of these services furnish an output compatible with the goal, we move p to R . In the next step, we examine all services that have an input parameter that subsumes an output of p ; and in the following step, we would consider the *components* of p 's output types using the recursive definition stated earlier. Only after all of these possibilities have been exhausted will the fringe of the search advance by one hop and begin the procedure anew. In this way, we furnish the services in P that can reach the outputs in the shortest number of hops; and among services in P that can reach a goal in the same number of hops, we prefer those that do so with the smallest possible ‘divergence’ in types.

As a concrete example of how this algorithm works in practice, we refer again to Fig. 3. The set P consists of three services. In the first step, we check if any of the outputs of these services is compatible with a goal. This is not the case, so we take another step; for each service p in P , we retrieve the services will consume an output of p of the exact same type. In this example, we assume that there are no operations that specifically consume any of ‘B_ID’, ‘C_ID’, or ‘DNASeq’.

We then take another step, which involves looking at all services that have parameters that *subsume* the outputs of services in P . The ‘PhenoBank’ service takes a Text input, which is a superclass of ‘B_ID’. Further, the output datum of ‘PhenoBank’ is ‘Pheno-type-List’ which is compatible with one of the goals. Thus, we move the ‘AID_to_BID’ service into R . We assume that there are no other services with parameters that subsume outputs of services in P .

In the next step, we look at all services that have input parameters compatible with components of the outputs of services in P . Here we find that the ‘BLAST’ service consumes a String, which is one of the components of the ‘SeqBank’ service, and that BLAST produces an output that is also compatible with a goal. We can thus move the ‘SeqBank’ service from P into R , which ranks it below the ‘PhenoBank’ service that is already there. Again, we assume the ‘AID_to_CID’ service has no such successors.

We have exhausted all possibilities for type compatibility for the types in the fringe, and thus in the next step we will look at all the outputs of all services that can succeed ‘AID_to_CID’ and repeat the process just described for these outputs.

3.3.4. Reranking results based on observations from external data

The algorithm described thus far orders the generated services first by the length of the service chain to the goal that they initiate, and second by the similarity of the types being bound in the service chain. We have mentioned that we chose to use this approach because of the lack of available Moby usage data. However, as the project progressed, some crude usage data was being generated and we wished to make it possible to incorporate it into the algorithm described here.

This was implemented simply by replacing the results queue for R with a priority queue. In the example from the previous section, if we found that the ‘BLAST’ service was more popular than the ‘AID_to_BID’ converter at insertion time, it would be implicitly ranked above the converter even though it was inserted later. We used this approach to leverage basic log data from the Moby central registry that lists the number of times that each service has had an interface description request made of it. This serves as a rough metric of how ‘popular’ each service is. Our comparator positions a service in the queue based on how often the central registry has been asked to provide a description for its interface.

The core algorithm as described relies on incremental generation of results to improve the response time of the client. This makes it impossible to give these comparators a higher precedence on the global ranking than the length of the service chain from the potential service result to the goal. However, this design was chosen exactly *because* such data was lacking at the time that the project was started. As the Moby framework continues to mature and more usage statistics are collected, we will move towards a method that relies more heavily on data mining than on heuristic search to guide the service selection process.

4. Evaluation

This section describes our evaluation methodology, defines the metrics that we used to quantitatively measure the performance of the client and presents the results of our investigation. We conclude this section with a discussion of our results in the context of our stated goals.

4.1. Methodology

We assembled a collection of Moby workflows and attempted to rebuild them using our client prototype. This allowed us to measure (1) the extent to which we would be able to restrict the number of services selected from the potential service set, and (2) how well we ranked the desired service at each step.

The data consists of three example workflows that are used to demonstrate functionality from the Taverna BioMoby plugin [9], and a single large workflow used for promoter analysis [10], which was generously provided by Arnaud Kerhornou for use in this project. The BlastDragonDBWorkflow retrieves sequences from GenBank by ID, BLASTs those sequences against the Antirrhinum DragonDB, and then postprocesses the report to extract the IDs of the similar sequences. The compareKeyword workflow retrieves all sequences related to a locus by similarity and all sequences related to a given keyword, and then takes the intersection, union, and difference of these sets. The repeatMasker workflow retrieves sequences based on a list of identifiers and masks the repeating segments of those sequences. The promoter analysis workflow is relatively complex and is discussed further in the cited publication.

While a user study would have permitted a more thorough evaluation of our approach and its effect on user productivity, we were unfortunately unable to perform one due to resource limitations. We leave such an evaluation as future work.

4.1.1. Challenges encountered with methodology

Many challenges were encountered in the evaluation process. These challenges and the way we addressed them are identified here.

4.1.1.1. Complexity of the interaction model. The interaction model described in Section 3.2 permits many ways of constructing a particular workflow, and our client could conceivably perform quite differently depending on the order in which operations are executed. We accommodated for this by using two rigid processes: We first built each workflow using only forward construction operations, and then we built each workflow using only reverse construction operations. This enables us to discern performance differences between the two aspects of the algorithm, but does not necessarily reflect the way a user would actually interact with the program. In cases where a single workflow was composed of more than one service chain (i.e. contained at least one fork), we completely assembled each chain before beginning on the next.

4.1.1.2. Organization of service selection menu. The service selection menu sub-categorizes services into their top-level service types (e.g. Analysis, Bioinformatics, Parsing, etc). This ambiguates the definition of ‘rank’, since results are only ranked within sub-categories. We made the simplifying assumption that the user would know what category her service of interest belonged to at each step in the assembly process, and we thus report the rank of each service as its position in the sub-category in which it has been placed.

4.1.1.3. Heterogeneity of services in workflows. It is difficult to solve a real-world problem without interfacing with some service that is not described in the Moby framework. For example, machine-local parsing scripts are often written and embedded in workflows to postprocess the results of a service invocation. The consequence of this is that our sample workflows contained a number of non-Moby services that our client is unable to reason over. We thus extracted the sub-workflows of significant size or function that were present in these processes, and built them as individual workflows. The inputs to the initiating services of the sub-workflows were used as the workflow inputs, and the outputs of the terminating services were used as the workflow outputs.

4.1.1.4. Inability to compare to other clients. To our knowledge, Taverna is the only client that operates on Moby services that has an interaction model similar to ours and that also provides some form of workflow assembly assistance (using the BioMoby plugin). However, we could not compare against Taverna to any reasonable extent, because the top-level organization of services in the plugin differs from ours; services are organized by authority instead of by service type. Thus, comparisons between service ranks in each client are not especially informative.

4.1.2. Metrics used

The goal of the evaluation process was to assess how well we are able to restrict the number of services presented to the user (restriction quality), and our ability to position the desired service early in the result list (ranking quality).

We evaluated the restriction quality by taking the ratio of $|R|$ to $|P|$. $|P|$ is the number of results that a typical semi-automatic method that does not consider end-goals would furnish at that step, and so this ratio gives an estimate of how much more effective our approach is in reducing the size of the user’s search space relative to a simpler approach.

We evaluated ranking quality as the position of the desired service in the service list. In order to better understand how the performance of the “structure-based” ranking (i.e. ranking based only on service chain length and type distance) alone compared with

additional “data-based” ranking (i.e. reranking based additionally on usage data), we recorded the ranks of the desired service with the data-based ranking both turned on and off.

4.1.3. Results

The results of the individual trials are presented in Table 1 for forward construction (which invokes the forward-chaining algorithm), and Table 2 for reverse construction (which invokes the backwards-chaining algorithm), respectively. The RepeatMasker workflow required two unrelated inputs to be composed into a single DNASequences type in order for forward-chaining to be used. At the time that the evaluations were performed, this functionality was not yet available in the client, and so it was omitted from the forward-chaining trials.

Each individual *produce with* or *feed into* operation was considered to be a single trial. The ‘Workflow’ column of each table lists the name of the workflow that was being built, while the ‘step number’ identifies the order in which operations were performed. *P* and *R* retain their definitions from Section 3.3: $|P|$ and $|R|$ are the sizes of these sets. $|R|/|P|$ is the restriction ratio defined in Section 4.1.2. A restriction ratio of 1.0 means that all of *P* was considered to be salient, and so no reduction was achieved. Smaller ratios are better. The last two columns in each of Tables 1 and 2 list the rank of the desired service (i.e. the service present in the actual workflow instance) at the given step, both when the structure-based ranking is used alone and when it is compounded with the data-based ranking. Ranks are 0-indexed, and so a position of 0 is the highest possible rank.

Table 1
Results for each step of the forward-chaining experiments

Workflow	Step #	$ P $	$ R $	$ R / P $	Rank (structural)	Rank (data)
BlastDragondbworkflow	1	341	341	1.00	322	22
BlastDragondbworkflow	2	26	19	0.73	14	9
compareKeywords	1	341	341	1.00	222	43
compareKeywords	2	26	26	1.00	13	1
compareKeywords	3	3	3	1.00	2	1
compareKeywords	4	341	341	1.00	171	48
PromoterAnalysisWorkflow	1	342	137	0.40	3	3
PromoterAnalysisWorkflow	2	40	35	0.88	3	0
PromoterAnalysisWorkflow	3	19	14	0.74	2	1
PromoterAnalysisWorkflow	4	19	14	0.74	6	5
PromoterAnalysisWorkflow	5	3	2	0.67	1	1
PromoterAnalysisWorkflow	6	7	6	0.86	0	0

$|P|$ is the size of the potential service set, while $|R|$ is the size of the restricted set. BlastDragondbworkflow, compareKeywords, and repeatMasker are all workflows that are used for demonstration purposes with the Taverna BioMoby plugin, while PromoterAnalysisWorkflow is a ‘real-life’ workflow instance.

Table 2
Results for each step of the backward-chaining experiments

Workflow	Step #	$ P $	$ R $	$ R / P $	Rank (structural)	Rank (data)
BlastDragondbworkflow	1	5	5	1.00	4	0
BlastDragondbworkflow	2	7	7	1.00	1	0
compareKeywords	1	435	396	0.91	162	43
compareKeywords	2	20	16	0.80	4	2
compareKeywords	3	133	94	0.71	9	25
compareKeywords	4	16	16	1.00	0	3
PromoterAnalysisWorkflow	1	1	1	1.00	0	0
PromoterAnalysisWorkflow	2	2	2	1.00	0	0
PromoterAnalysisWorkflow	3	1	1	1.00	0	0
PromoterAnalysisWorkflow	4	3	3	1.00	2	1
PromoterAnalysisWorkflow	5	14	14	1.00	13	4
PromoterAnalysisWorkflow	6	9	9	1.00	8	0
PromoterAnalysisWorkflow	7	14	14	1.00	7	2
repeatmasker	1	5	5	1.00	1	0

See the caption in Table 1 for details on the notation used in this table.

As a concrete example of how this data was generated, we will describe how the trials for the BlastDragondbworkflow instance in Tables 1 and 2 were performed. This workflow consists of an Object input that feeds into a service called MOBYSHoundGetGenBankFasta; this service in turn furnishes a FASTA output that is fed into a getDragonBlastText service instance. This final service instance produces a NCBI_BLAST_Text datum that is bound to the Object output parameter of the workflow.

In both the forward- and backward-chaining cases, we instantiated the workflow in our client with a single Object input and a single Object output. In the forward-chaining case, step 1 was to select the workflow input and choose the *feed into* operation, and then to select the top-level category 'Retrieval', since the MOBYSHoundGetGenBankFasta service is a Retrieval service and we presume that the user would know that this is the category they are interested in. The number of retrieval services that can consume an Object is 341 (that is, $|P| = 341$). Iterating through the entire result set demonstrated that all 341 potential services were retained, and so $|R|$ is also 341, which results in a restriction ratio ($|R|/|P|$) of 1.0. With data-based ranking turned off, the MOBYSHoundGetGenBankFasta service appeared in position 322 of the result set, and so we recorded the rank as 322 and added this service to the workflow. We then expanded its single FASTA output and issued a *feed into* operation on it to begin step 2. We selected the Analysis subcategory of the results menu this time around, since the getDragonBlastText service is an Analysis service. Twenty-six services were in the potential set ($|P| = 26$), 19 of which were displayed in the results menu ($|R| = 19$), which implies a restriction ratio of 0.73 ($|R|/|P| = 0.73$). The getDragonBlastText service appeared in position 14 of the results list, and so we recorded the its structural rank as 14. This procedure was repeated to generate the backward-chaining data, however in this case the first step was to invoke a *produce with* operation on the output Object and to search for the getDragonBlastText service in the results. That is, the order that the services were introduced in the backward-chaining trials is the reverse of the order that they were introduced in the forward-chaining trials.

Summary statistics derived from these tables are found in Tables 3 and 4. The mean restriction ratio, structure-based rank, and data-based rank were computed in order to see how well each method did in general, and standard deviations were included to give an indication of how stable this average behavior is. These results are further discussed in the following section.

The evaluation was conducted on a dual-core 2GHz Intel MacBook with 1GB of RAM. The time to display results for all operations was always found to be under 0.5 s, which meets the requirements specified at the beginning of this paper, and is significantly faster than the multisecond response times that the authors are accustomed to experiencing in clients such as SeaHawk and the

Taverna BioMoby plugin. This demonstrates that, while the theoretical worst-case total runtime of the algorithm may be large, it is in practice quite manageable.

4.2. Discussion of results

Our evaluation was intended to demonstrate our ability to restrict and rank services throughout the workflow building process. We now discuss how well we perform these tasks based on the data presented in Tables 1–4.

4.2.1. Quality of restriction process

The quality of the restriction process varied slightly with the direction of the composition. We first examine forward-chaining. Table 1 shows that 5 of the 12 forward-chaining steps had a restriction ratio of 1.0, that is, no restriction was possible in almost half of the cases. Table 3 indicates that, on average, only 17% of the potential service set is discarded. These poor restriction ratios are a result of imprecise specifications of workflow outputs; most of our workflows had their goal outputs specified only as Objects—that is, an output of *any type* will suffice—which gives the selection algorithm very little information to work with. One notable exception to this trend is the PromoterAnalysisWorkflow. In this use case, the workflow outputs were of infrequently used types (NewickText and List_Text), and so presumably there were few possible service chains from any source datum that would lead to one of these goals. The average restriction ratio here is 0.72, with the most significant restriction taking place in the very first step, where 205 potential services are removed from consideration. This ratio is significantly better than the mean ratio of 0.83 reported in Table 3.

The restriction ratios for backward-chaining operations were worse than those observed for forward-chaining, as is evident in Table 2. Eleven of the 14 backward-chaining trials had restriction ratios of 1.0, and the average restriction ratio is 0.96. While most of our workflow inputs were specified as Objects, this should not in itself be a problem because an Object is a very specific goal when backward-chaining. (That is, if the input to a workflow is identified as an Object, services that require inputs of more specialized types than Object cannot safely bind to it.) The problem lies in the service registry: many services in the registry (over 550 at last count) annotate at least one of their input parameters only as Objects, whereas a more specific annotation should almost certainly have been supplied at the time of registration. Again, since any goal datum could bind to a service parameter of type Object, any such service will be seen as leading to the goal.

Thus, it appears as if having more precise parameter annotations in the service ontology would improve the quality of our restriction process. Improving the specificity of these annotations would also have benefits that extend beyond this work, since the approaches used by other clients could also leverage this higher-quality data to reduce the size of their result sets.

4.2.2. Quality of ranking process

We first discuss the structure-based ranking results. When forward-chaining, Table 3 shows that the average structure-based rank of a desired service is 63 (i.e., the user must look at 63 service names before encountering their desired service from the example workflow), with a large standard deviation. This is likely a consequence of the specificity problem discussed in the previous section. When a workflow output is permitted to be an object of any type, all potential services are one hop away from a goal, and so the only influence left to guide the ranking is the proximity of the output of the service chain to Object in the objects. However, since the output of the workflow is likely not intended to be this generic, the type distance metric is not effective in differentiating service quality, which results in ranking that is largely uninformed. In contrast,

Table 3
Summary statistics for the forward-chaining data presented in Table 1

Metric	Average	Standard deviation
Rank (structural)	63.25	110.63
Rank (data)	11.17	17.21
R/P	0.83	0.19

Table 4
Summary statistics for the backward-chaining data presented in Table 2

Metric	Average	Standard deviation
Rank (structural)	15.07	42.49
Rank (data)	5.71	12.56
R/P	0.96	0.09

backward-chaining can leverage greater specificity, since an Object goal input will directly match less than half of the services in the service space as mentioned in the previous section. This may explain why the structure-based ranking in the backwards-chaining trials is better than that observed for the forward-chaining trials, placing the desired service within the top 20 results on average (Table 4).

We now consider the effect of the data-based reranking relative to the structure-based ranking alone. In both the forward-chaining and backward-chaining data, we observe that the most poorly ranked services from the structure-based ranking have had their ranks significantly improved. The best case is seen in step 1 of the forward-chain for BlastDragondbworkflow (Table 1), with an improvement in rank of roughly 93%, while the smallest improvement of 72% is still impressive (step 4 of compareKeywords for the forward-chain, Table 1). Small improvements are also seen in steps where the structure-based ranking was already performing quite well (e.g. the last six rows of Table 1). Only in one case was the data-based ranking observed to be worse than the structure-based ranking (step 3 of the compareKeywords process in Table 2). The average performance of the data-based ranking for backward-chaining (5.71 as reported in Table 4) is better than that observed for forward-chaining (11.17, Table 3), although both are well within the top 20 results.

Recall that the reranking works by sorting the services within a batch by their ‘popularity’ as evidenced by their frequency of reference in the log data discussed in Section 3.3. This effectively discards the type-distance metric in favor of the data-based ranking within a batch. In the forward-chaining case, most of the services will be furnished within a single batch because they are all one hop from a workflow output that has been specified as an Object, as we have already discussed. Thus, the data-based ranking is essentially the only influence on the service rank. In the backwards-chaining case, where some differentiation is first made based on service chain length, the data-based ranking will only re-rank services within each batch. It could be that this extra influence on the overall ranking is contributing to the better data-based ranking quality that is observed in the backward-chaining case.

There are two interesting implications that we can draw from these results. The first is that the poor restriction ratios reported in the previous section are not especially damaging to our approach as a whole, since our ranking procedure appears to reliably rank the desired service very highly in the service list, especially when the data-based reranking is enabled. Thus, we may be able to simplify the algorithm by not attempting to restrict at all the size of P , since we expect that the user will be supplied with their desired service very early in the navigation process anyhow. Second, the fact that we used a simple method to incorporate very crude data into our approach and achieved significant improvements in ranking seems to bode well for future work in using more sophisticated data mining techniques to improve the ranking. We believe that this is a strong motivation for (1) improving and extending the existing mechanisms for capturing usage statistics within the Moby API implementations and (2) developing methods to infer significant or popular workflow compositions that could be suggested for reuse.

5. Related work

While automatic web service composition methods are uncommon in applications designed specifically for the life-sciences, a significant amount of research has been done in this field for the World Wide Web in general.

Our work is most similar to the approaches presented by Arpinar et al. [11], in which they describe both a fully-automatic and a

semi-automatic process for web service composition. In the automatic method, called interface-matching automatic composition (IMA), a graph-based algorithm is used to forward-chain from the desired workflow inputs to the outputs. Edges in the graph are weighted by a function of service execution time and type similarity, whereas our method does not take execution time into account. The definition of type compatibility is similar to ours, except that it allows for (and ranks highly) ‘unsafe’ matches in which an output parameter is more generic than the input it is being bound to. The use of an edge-weighting function is more flexible than our relatively rigid metrics of path length and type similarity, but presumably requires more effort on the part of the user to configure. Also, as the method is not accompanied by an evaluation, it is not possible to discern whether this extra sensitivity results in an increased probability of correctness. In contrast, the human-assisted (HA) semi-automatic method algorithm produces at each step a ranked list of services, as in our approach. However, this ranking is based only on the type similarity between the datum being bound and the service to bind to it. No extra searching is performed to see if the services being ranked highly will initiate a chain that can lead to an open output. Inputs can be removed from consideration or marked as ‘optional’ in their interaction model, but again these ‘hints’ are only used to determine the compatibility of the datum being examined and the service to succeed it in the chain. There does not appear to be support for bidirectional construction of workflows nor composite types.

Agarwal et al. [12] present an approach that uses a combination of a novel interaction model and a graph algorithm to provide fully-automatic composition. The target service space is that of all services on the WWW. They provide a mechanism for ‘advertising’ these services in an attractive way. A user that has some task in mind must first select a “closed world” of services that she wishes to include in her workflow by browsing these advertisements. Because these services are described by a number of different ontologies, an interface is provided to facilitate user-driven ontology alignment. The assumption is that this closed world will be extremely small, and thus determining the composition that the user already had in mind will be quite simple. This is in contrast to our assumptions, in which our service space is already “closed” by virtue of operating solely within the Moby framework, but this closed space is still large enough that users are in need of assistance at service selection time. Users of the system of Agarwal et al. must presumably exert a great deal of effort to produce this small set of services that they are interested in. Once the world has been thus closed by the user, the composition tool requests that the user specify a set of desired outputs. From here, all of the possible backwards-chained subgraphs of the selected services are displayed for the user to select from. No description is given of how these plans are ranked. It is also not evident how these subgraphs can be united if there are several distinct endpoints to a workflow, whereas our solution allows for arbitrary parameter binding and bidirectional workflow construction. Again, no evaluation is given to demonstrate the efficacy of this method. However, the idea of visualizing possible subflows at each step is one that we would like to incorporate into our method, so that these subflows can be added en bloc if the user determines that she is satisfied with them. This functionality would be particularly useful in a scenario such as the one depicted in Fig. 1(b). In this scenario, when the user is searching for services to consume the initial A_ID input, our algorithm ranks the ‘AID_to_BID’ service highly because it ‘knows’ that it very quickly leads to the goal through the ‘Pheno-Bank’ service. However, the user is not made aware of this connection until she takes further action on the output of the ‘AID_to_BID’ service. It would be much more informative to present her with a depiction of the entire subflow that leads to the goal at this time.

Sirin et al. [13] demonstrate a simple semi-automatic composition method where the user builds the workflow in reverse from the goal outputs towards their desired inputs; however, the basic approach is identical to those already discussed here.

In the Taverna BioMoby plugin [9], one can introduce a Moby object as a ‘processor’ and then request a list of services that directly consume or produce that data. This analysis does not leverage any other information other than the type of the data that is being clicked on, and so services may be suggested that cannot possibly lead the user to their goal inputs or outputs. Also, the service selection process can be quite slow, ranging from seconds to minutes in the experiences of the authors.

All of the aforementioned methods use the same problem specification, in which users describe their desired workflow composition in terms of the types of inputs and outputs it expects. Many other approaches enrich or ambiguate this specification to give the reasoning process more power or to grant more flexibility to the user.

An example of this is presented in Liu et al. [14]. Service interface descriptions are formulated as RDF graphs that function as pre- and post-conditions on the inputs and outputs of a service. The post-conditions can be described in terms of variables that are introduced in the pre-conditions, and these variables can be substituted for at composition time in order to provide more specific workflow-dependent interface descriptions that assist in the reasoning process. For example, a service could specify that it takes as input an ID number from some company *c*, and produces as output an employee name from the same company *c*. If the parameter of this service is then bound to a workflow input that is known to be from IBM, then the service output will be known also to be from IBM. The interesting thing about this formulation is that it implicitly handles composite types via a sort of ‘duck typing’ that does not require the actual composite to be declared in an ontology. Instead, one can specify directly in a pre- or post-condition that an item has two components of atomic types. Their work includes an evaluation, but the dataset is composed entirely of random service descriptions and focuses only on the runtime of the planner, not on the correctness of the resulting workflows. Unfortunately, there seem to be some difficulties with scaling; reasoning over 50 services requires 59 s of “pre-reasoning” time and 2 s of “planning” time, whereas 100 services requires 233 s of pre-reasoning and 29 s of planning time. The Moby service framework has more than 1000 services, and so runtimes using this method could be prohibitively long. However, the idea of providing richer service descriptions via RDF graphs is an appealing one and might perhaps be useful in future versions of BioMoby.

Many other formal problem specifications have also been proposed. The system presented by McIlraith and Son [15] requires the user to represent their desired service composition as a sequence of constraints encoded in a first-order logic language called GOLOG, while Medjahed et al. [16] use a declarative language to describe service composition requests. In both cases, the formulation allows the ASC to be reduced to another problem: The GOLOG encoding permits the use of AI planning techniques, whereas the declarative language encoding enables the application of query planning and query optimization algorithms. Narayanan and McIlraith [17] formalize their service and problem descriptions using Petri nets, and then posit that a reachability analysis could be used to produce an ‘optimal’ workflow that connects user-specified inputs to outputs. The discussion here is largely theoretical, however, and the implementation of the method is only tentatively addressed.

Some approaches use problem description methods that are designed to make the specification process more intuitive for the user, rather than trying to relay more information to the composition process. The SeaHawk client for Moby services [18] provides a

browser in which the user can directly manipulate data and execute services on it, for example by highlighting a biological sequence in a web page and right-clicking on it to invoke a retrieval service on that string. The sequence of user behaviors is recorded and “encoded” as a workflow that can later be executed as a single entity, much like a macro. The Gbrowse-Moby web-based client [19] also uses a similar mechanism. These are both examples of a paradigm that is more generally known as “programming by example” [20]. However, in cases where the user has a general idea of what they want their workflow to do, having to enact the workflow step-by-step within a browser to implicitly record it could potentially be tedious.

6. Conclusions and future work

In this paper, we have presented what is to our knowledge the most extensive automated service composition approach to assist workflow assembly in life-sciences research to date. Our work presents the following novel contributions:

- (1) We have demonstrated a method for assisting workflow assembly at a level beyond what is available in previous clients, within a practical web services integration framework with an active user base.
- (2) The implementation of our method is able to highly rank desirable services at interactive speeds.
- (3) Our interaction model and algorithms permit manipulation of and reasoning over of composite types.
- (4) We provide preliminary evaluations of our methodology that suggest that it may be effective in assisting the user to quickly find their desired service during the assembly process.

There are many ways in which this research could be directly extended. We would like to perform user studies to assess how well the intent-declaration mechanisms work, and to determine if the encouraging results demonstrated in Section 4 translate to increased user productivity, which is the ultimate goal. It is possible that many of the features of the interaction model and the algorithm could be removed or further simplified, and rigorous evaluations would help to identify which of these features should be re-examined. If the results of this evaluation were favorable, the logical next step would be to incorporate the interaction model and algorithms described here into the Taverna workflow client, as this software is used by a relatively large number of scientists to perform actual life-sciences research. Finally, our approach purposely uses a minimal amount of semantic information (data-type matching and menu organization via toplevel service types), in order to prevent frequent interruptions to user navigation. It would be useful to see how our results might be improved if we gave the user the option to perform more ‘advanced’ searches by explicitly specifying keywords that might be present in service or type descriptions.

There are also several tangential directions that this research could take. The current implementation of the client provides only semi-automatic composition. The favorable results that we observed in our evaluations when very simple usage data was incorporated into the ranking suggests that it may be possible to use more and richer data to further automate the assembly process, by identifying service chains that are frequently used and suggesting these ‘prefabricated’ components to the user in the appropriate situations. It is our belief that by combining instance data from large workflow repositories and usage data from central services such as the MobyCentral API implementations, we will be able to infer with high accuracy the workflow compositions that are commonly used in specific cases.

Throughout this paper, we have implicitly assumed that the ability to consistently perform automatic composition based on usage patterns would result in a useful tool. However, such a mechanism may potentially hinder users who are trying to construct completely novel workflows. Characterizing the frequency with which scientists construct similar workflow segments and the effort required to use a client which relied heavily on such patterns to construct very novel workflows would be necessary to produce a tool that adequately serves the entire life-sciences community.

The logical progression of the projects described in this section would be to move the automated service selection and composition logic out of client software and into a universally accessible API or web service so that any client or other service could easily take advantage of it. By providing globally accessible workflow assembly assistance services, we hope to significantly increase the adoption of web services integration technologies such as Moby and Taverna within the life-sciences, and to increase the productivity and effectiveness of researchers in these diverse and important fields.

Acknowledgments

Mark Wilkinson and the BioMoby project are funded through an award from Genome Prairie and Genome Alberta, in part through Genome Canada, a not-for-profit corporation leading Canada's national strategy on genomics. In addition, funding for core BioMoby and client code development in Canada (Mark Wilkinson) was obtained through the CIHR, NSERC, and the Heart and Stroke Foundation for BC and Yukon.

Michael DiBernardo is funded by an NSERC PGS-M grant.

References

- [1] Stein L. Integrating biological databases. *Nature Reviews Genetics* 2003;4:337–45.
- [2] Hull D, Wolstencroft K, Stevens R, Goble C, Pocock MR, Li P, et al. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research* 2006;34(Suppl. 2):W729–32.
- [3] Oinn T, Greenwood M, Addis M, Alpdemir NM, Ferris J, Glover K, et al. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency in Computation: Practical Experiences* 2006;18(10):1067–100.
- [4] Oinn T, Addis M, Ferris J, Marvin D, Senger M, Greenwood M, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* 2004;20(17):3045–54.
- [5] Wilkinson M, Gessler D, Farmer A, Stein L. The BioMOBY project explores open-source, simple, extensible protocols for enabling biological database interoperability. In: *Proceedings of the virtual conference on genomics and bioinformatics*, vol. 3; 2003. p. 17–27.
- [6] Wilkinson M, Schoof H, Ernst R, Haase D. BioMOBY successfully integrates distributed heterogeneous bioinformatics web services: the PlaNet exemplar case. *Plant Physiology* 2005;138:5–17.
- [7] Rao J, Su X. A survey of automated web service composition methods. In: *Proceedings of the 1st international workshop on semantic web services and web process composition*, 2004. p. 43–54.
- [8] Dustdar S, Schreiner W. A survey on web services composition. *International Journal of Web and Grid Services* 2005;1:1–30.
- [9] Kawa EA, Senger M, Wilkinson MD. BioMoby extensions to the Taverna workflow management and enactment software. *BMC Bioinformatics* 2006;7:523+.
- [10] Kerhornou A, Guigo R. BioMoby web services to support clustering of co-regulated genes based on similarity of promoter configurations. *Bioinformatics* 2007;23:1831–3.
- [11] Arpinar IB, Aleman-Meza B, Zhang R, Maduko A. Ontology-driven web services composition platform. In: *CEC'04: proceedings of the IEEE international conference on E-commerce technology (CEC'04)* 2004. p. 146–52.
- [12] Agarwal S, Handschuh S, Staab S. Annotation, composition and invocation of semantic web services. *Journal on Web Semantics* 2005;2(1):1–24.
- [13] Sirin E, Hendler J, Parsia B. Semi-automatic composition of web services using semantic descriptions. In: *Web services: modeling, architecture and infrastructure workshop in conjunction with ICEIS2003*, 2002.
- [14] Liu Z, Ranganath A, Riabov A. modeling web services using semantic graph transformations to aid automatic composition. In: *Proceedings of the 2007 IEEE international conference on web services 2007*, p. 78–85.
- [15] McIlraith SA, Son TC. Adapting GOLOG for composition of semantic web services. In: *KR-02: proceedings of the 8th international conference on principles and knowledge representation and reasoning*, 2002.
- [16] Medjahed Brahim, Bouguettaya Athman, Elmagarmid Ahmed K. Composing web services on the semantic web. *The VLDB Journal* 2003;12(4):333–51.
- [17] Narayanan S, McIlraith SA. Simulation, verification and automated composition of web services. In: *proceedings of WWW'02, the 11th international conference on World Wide Web 2002*, p. 77–88.
- [18] Gordon PMK, Sensen CW. Seahawk: moving beyond HTML in web-based bioinformatics analysis. *BMC Bioinformatics* 2007;8:208+.
- [19] Wilkinson M. Gbrowse Moby: a web-based browser for BioMoby services. *Source Code for Biology and Medicine* 2006;1:4.
- [20] Cypher A, Halbert DC, Kurlander D, Lieberman H, Maulsby D, Myers BA, Turransky Alan, editors. *Watch what I do: programming by demonstration*. MIT Press; 1993.