Fundamenta Informaticae 92 (2009) 279–299
DOI 10.3233/FI-2009-75
IOS Press

Towards a Formal Semantics for the Process Model of the Taverna Workbench. Part I

Jacek Sroka[*][†]
Institute of Informatics, University of Warsaw
Poland
sroka@mimuw.edu.pl

Jan Hidders
Faculty EEMCS, Delft University of Technology
The Netherlands
a.j.h.hidders@tudelft.nl

Abstract. Workflow development and enactment workbenches are becoming a standard tool for conducting *in silico* experiments. Their main advantages are easy to operate user interfaces, specialized and expressive graphical workflow specification languages and integration with a huge number of bioinformatic services. A popular example of such a workbench is Taverna, which has many additional useful features like service discovery, storing intermediate results and tracking data provenance.

We discuss a detailed formal semantics for Scufl - the workflow definition language of the Taverna workbench. It has several interesting features that are not met in other models including dynamic and transparent type coercion and implicit iteration, control edges, failure mechanisms, and incoming-links strategies. We study these features and investigate their usefulness separately as well as in combination, and discuss alternatives.

The formal definition of such a detailed semantics not only allows to exactly understand what is being done in a given experiment, but is also the first step toward automatic correctness verification and allows the creation of auxiliary tools that would detect potential errors and suggest possible solutions to workflow creators, the same way as Integrated Development Environments aid modern programmers. A formal semantics is also essential for work on enactment optimization and in designing the means to effectively query workflow repositories.

[*]Supported by Polish government grant no. N206 007 32/0809
[†]Address for correspondence: Institute of Informatics, University of Warsaw, Banacha 2, 02-097 Warsaw, Poland

This paper is the first of two. It defines, explains and discusses fundamental notions for describing Scufl graphs and their semantics. Then, in the second part, we use these notions to define the semantics and show that our definition can be used to prove properties of Scufl graphs.

Keywords: formal semantics, Scufl, workflows, Taverna workbench

1. Introduction

Taverna [9] is an easy to operate workbench for workflow development and enactment. It allows users to graphically construct workflows from libraries of available components and is intended for use in bioinformatics data analysis experiments. The most important virtues of Taverna are that it is very easy to use, has a specialized and expressive graphical specification language and integrates many data analysis tools. In [10] it is stated that the number of such tools exceeds 1000. It also includes additional useful features like service discovery, storing intermediate results and tracking of data provenance. The workbench is being constantly developed, but it is already considered stable and has been used in real life research, e.g., [13, 8].

The main motivation behind Taverna is to separate users from the implementation details of the communication with the services that they want to use. This is similar to the idea of hiding the details of data storage access in the database management system. Taverna handles communication with the service and its execution. Users are freed of those details and can focus on what is really important for them, which is analyzing the data. The conceptual language that is used to define the data analysis experiments, which we describe in Section 1.4, is intuitive and comprehensible for bioinformaticians, who often have no programming experience. Workflows are specified in a graphical notation specially designed for this purpose, which is called *Simple conceptual unified flow language* (Scufl).

The main concepts of the Scufl syntax and semantics were already presented in [14]. Yet, several Scufl features which are arguably important and unique among scientific workflow specification languages, like control edges used to synchronize services with side effects, dynamic and transparent type coercion and implicit iteration, failure mechanisms, and incoming-links strategies are described at a very high abstraction level or in a highly simplified manner. We give alternative and more detailed definitions of the semantics of Scufl as it is implemented in Taverna 1.7.1. While doing so we pay special attention to the aforementioned features, inspect their usefulness separately as well as in combination and discuss alternatives.

This paper is the first of two. It defines, explains and discusses fundamental notions for describing Scufl graphs and their semantics. Then, in the second part, we use these notions to define the semantics and show that our definition can be used to prove properties of Scufl graphs. To account for side effects the semantics is defined as a transition system. Finally, the second part also includes a more elaborate comparison with the work in [14].

1.1. Why formal semantics

We start with motivating why a precise formulation of a formal semantics for languages such as Scufl is crucial.

Scufl includes high level features and mechanisms, like implicit iteration, that make the construction of real life workflows simpler and allow the programmer to focus on the problem being solved. At the

same time the workflows look less complex and can be used in research papers to convey the main idea of an *in silico* experiment that was conducted. Yet, distributed data-processing experiments are complex in nature and a highly expressive definition language that hides much of the complexity of the workflow behind implicit semantics is not the silver bullet. When problems appear, e.g., while debugging, it is important to exactly understand what computation is being done. And even when the specification of the workflow is successfully finished, it's merit has to be effectively and objectively assessed by reviewers. For this a precise and formal semantics is needed.

It's also obvious that the *in silico* experiments that are being conducted become more and more complex and sooner or later automatic verification procedures, similar to those used for verifying complex business transactions, will have to be developed. For such verification the existence of formal semantics is a necessary first step as well as for the creation of auxiliary tools that would detect potential errors and suggest possible solutions to workflow creators, the same way as Integrated Development Environments aid modern programmers.

Another domain for which the formal semantics is fundamental is enactment optimization. As with database queries the programmer could only specify what has to be done and the determination of the most effective execution strategy would be left to the workflow engine. In addition, with workflows being applied more and more frequently, and being shared in Internet repositories [6], their querying is becoming an interesting scientific problem [4, 2]. A successful workflow query language should take into account the semantics and not just the syntax, i.e., compare what the workflows do and not only how they are defined.

Finally, we argue that the very act of formulating a formal semantics is useful because it forces us to do a complete and thorough analysis of the behavior of Taverna. The formulation of an elegant and natural formal semantics is a good litmus test for checking if the current behavior is consistent and well chosen. Such a test is not unimportant for large, complex and relatively rapidly evolving systems such as Taverna. In addition, as is shown in this paper later on, it may provide inspiration for other interesting alternative semantics. Therefore the formulation of a formal semantics can help in the future design and development of Taverna.

1.2. Scufl type system

As the Taverna authors notice "the problem of data typing in life sciences is simply too hard to attack". There is only one basic type that describes binary data with an attached MIME annotation and we will denote this basic type as $\mathcal{M}$. The MIME annotation is used to determine how a basic type data value is going to be presented to the user, e.g., whether a text, a picture, or its binary representation is going to be displayed. The set of MIME values is denoted as $\mathcal{V}_{\mathcal{M}}$. For our examples we will usually assume it contains at least the natural numbers and strings.

In Taverna we meet in practice only one collection type, namely, ordered lists, even though the documentation suggests that Scufl was designed to support other collection types such as partial orders, trees, bags and sets. Although the user documentation mentions only homogeneous lists, the workbench does not prevent the use of heterogeneous lists, i.e., lists containing elements of different types such as $[1, [2], 3, [[4]]]$. Heterogeneous lists can be obtained from homogeneous ones during the computation. For example, it is possible to specify in a Taverna workflow that an input is computed from different outputs of different processors by combining them into a single list. Therefore we define the set of complex values such that it includes heterogeneous lists.

The *set of complex values*, denoted as $\mathcal{V}_{tav}$, is defined as the smallest set such that (1) $\mathcal{V}_{\mathcal{M}} \subseteq \mathcal{V}_{tav}$ and (2) if $x_1, \ldots, x_n \in \mathcal{V}_{tav}$, then the list $[x_1, \ldots, x_n]$ is in $\mathcal{V}_{tav}$. The values of these list types will be denoted as $[1, 2, 3]$ and $[[1, 2], [3, 4], 5]$, the empty list is denoted as $[]$, and the concatenation of lists is denoted with $+$, so $[1, 2] + [1, 5] + [] = [1, 2, 1, 5]$. Note, that this notion of complex value does not include tuples or records.

Although heterogeneous lists can appear in Taverna, they usually cause processors to fail and otherwise are not always processed coherently, e.g., applying the flatten operation to the list $[[x], [[y]]]$, where x and y are some basic values, results in $[[x], [y]]$ while flattening of $[[[x]], [y]]$ results in $[[x], y]$. It is however quite possible to give an intuitive semantics for Scufl that allows heterogeneous values everywhere and deals with them consistently. Therefore, we will in the formal part of this paper, for the sake of simplicity and consistency, assume that heterogeneous values are allowed everywhere. If heterogeneous values never appear, then the semantics defined in this paper corresponds to the observed behavior of Taverna.

The consistent behavior for the heterogeneous values is owed to the coherent generalization of semantics of product strategies expressions (see Section 2.4) and implicit iteration mechanism (see Section 2.2). Despite this we usually limit the presentation to homogeneous values only and discuss in the second part of this paper the strategies for adapting the semantics such that the heterogeneous values are consistently avoided.

Although Taverna does as little typing as possible it still has a notion of *complex type*, which is defined by the following syntax:

$$\tau \ ::= \ \mathcal{M} \mid [\tau]$$

Examples of such types are $\mathcal{M}$, $[\mathcal{M}]$, $[[\mathcal{M}]]$, *et cetera.* The set of all complex types is denoted as $\mathcal{T}_{tav}$. The semantics of these types are defined with induction on their syntactic structure such that:

- $[\![\mathcal{M}]\!] = \mathcal{V}_{\mathcal{M}}$, and
- $[\![[\tau]]\!] = [\![\tau]\!] \cup \mathcal{L}([\![\tau]\!])$ where $\mathcal{L}(V)$ denotes the set of finite lists over V.

Note, that the given type semantics is more liberal than usual and explicitly allow heterogeneous lists. So not only $[[1], [2]] \in [\![[[\mathcal{M}]]]\!]$ but also $[1, [2]] \in [\![[[\mathcal{M}]]]\!]$ since $1 \in [\![\mathcal{M}]\!] \Rightarrow 1 \in [\![[\mathcal{M}]]\!]$. Effectively the type only restricts the maximum nesting depth of the complex values in its semantics.

Further motivation for the liberal list type semantics is given by the fact that if the nesting depth of a certain value is lower than expected there is always an intuitive interpretation of that value as a more deeply nested one, namely by nesting it in singleton lists. For example, if a certain processor expects on a certain input port a list of protein identifiers and it receives a value that is an unnested single protein identifier, then it can interpret this as a singleton list containing this protein. This principle can be applied to every type, i.e., a value of type τ can always be interpreted as a value of type $[\tau]$ by assuming it is packed in a singleton list. This is reflected in the type semantics by the fact that $[\![\tau]\!] \subseteq [\![[\tau]]\!]$. The idea that types are given a semantics that is related to a coercion mechanism can be found in other work such as [1].

Consistently with the given type semantics and the described type coercion we define a subtyping relation, denoted by $\sqsubseteq$, over complex types such that $\tau \sqsubseteq \sigma$ iff the nesting depth of τ is less than or equal to the nesting depth of σ, i.e., either $\tau = \mathcal{M}$, or $\tau = [\tau']$ and $\sigma = [\sigma']$, where $\tau' \sqsubseteq \sigma'$. For example,

$\mathcal{M} \sqsubseteq [\mathcal{M}]$, and $[\mathcal{M}] \sqsubseteq [[[\mathcal{M}]]]$, but $[[\mathcal{M}]] \not\sqsubseteq [\mathcal{M}]$. Clearly, this notion of subtyping is consistent with the given semantics, i.e., for all complex types τ and σ it holds that $\tau \sqsubseteq \sigma$ iff $[\![\tau]\!] \subseteq [\![\sigma]\!]$.

Since there is only one basic type, viz. $\mathcal{M}$, it is not hard to see that $\sqsubseteq$ defines a linear order over the complex types. So we can define a function $\mathsf{max} : \mathcal{P}(\mathcal{T}_{tav}) \rightarrow \mathcal{T}_{tav}$, such that $\mathsf{max}(T)$ is the least common upper bound of T, i.e., the smallest complex type σ such that for all types $\tau \in T$ it holds that $\tau \sqsubseteq \sigma$. This means, for example, that $\mathsf{max}(\emptyset) = \mathcal{M}$, $\mathsf{max}(\{\mathcal{M}, [\mathcal{M}]\}) = [\mathcal{M}]$, and $\mathsf{max}(\{[\mathcal{M}], [[[\mathcal{M}]]]\}) = [[[\mathcal{M}]]]$.

1.3. Scufl global components

Here we list the Scufl components that are common to all workflows. We postulate a countably infinite set PL of *port labels* that contains all names that can be given to input and output ports of processors as well as to workflow inputs and outputs. The Taverna workbench comes with a huge library of built-in bioinformatics operations, which are mainly external service intermediaries, i.e., programs that call external services. We call this extensible collection of operations the *Taverna services* and model it by a set of service names called TS which can contain an arbitrary number of names.

The interface of a service is defined by tuple types that give the input type and the output type. These tuple types are defined as partial functions $\sigma : PL \rightarrow \mathcal{T}_{tav}$ that map a finite subset $\mathrm{dom}(\sigma) \subseteq PL$, called the *domain* of σ, to complex types. We will denote tuple types $\{(l_1, \tau_1), \ldots, (l_n, \tau_n)\}$ as $\langle\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\rangle$. The set of all tuple types is denoted as $\mathcal{T}_{tup}$ and the set of all tuple values as $\mathcal{V}_{tup}$. The semantics of a tuple type $\sigma = \langle\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\rangle$, denoted as $[\![\sigma]\!]$, is defined as the set all functions $t : \mathrm{dom}(\sigma) \rightarrow \mathcal{V}_{tav}$ such that for each $l_i \in \mathrm{dom}(\sigma)$ it holds that $t(l_i) \in [\![\tau_i]\!]$. Such a function $\{(l_1, x_1), \ldots, (l_n, x_n)\}$ will be denoted as $\langle\, l_1 = x_1, \ldots, l_n = x_n \,\rangle$. For later use we define a notation for the projection of a tuple type σ on a set of labels L as $\sigma|_L$ such that $\sigma|_L = \{(l, \tau) \in \sigma \mid l \in L\}$ and its counterpart for tuple values as $t|_L = \{(l, v) \in t \mid l \in L\}$.

To define the interface of the Taverna services we postulate the functions $type_i : TS \rightarrow \mathcal{T}_{tup}$ and $type_o : TS \rightarrow \mathcal{T}_{tup}$ that give the input type and output type, respectively, of each service as a tuple type. In addition we define the functions $I : TS \rightarrow \mathcal{P}(PL)$ and $O : TS \rightarrow \mathcal{P}(PL)$ such that for every service name $s \in TS$ $I(s)$ gives the set of input port labels and $O(s)$ the set of output port labels, i.e., $I(s) = \mathrm{dom}(type_i(s))$ and $O(s) = \mathrm{dom}(type_o(s))$. For example, the interface for the string concatenation operation "Concatenate_two_strings" $\in TS$ is defined as follows (we abbreviate "Concatenate_two_strings" to "c_t_s"):

$$\begin{aligned} I(\text{“c_t_s”}) &= \{string1, string2\} \\ type_i(\text{“c_t_s”}) &= \langle\, string1 : \mathcal{M}, string2 : \mathcal{M} \,\rangle \\ O(\text{“c_t_s”}) &= \{output\} \\ type_o(\text{“c_t_s”}) &= \langle\, output : \mathcal{M} \,\rangle \end{aligned}$$

The semantics of a service is defined by a non-deterministic function that maps a tuple of the input type of the service to one of possibly many tuples of the output type. There are several reasons why the result might not be functionally dependent on the input. One of them is that the services can have an internal state which influences its result. Also the service can use randomized approximation algorithms, which is often the case in bioinformatics. Finally, the service can be based on a database which is constantly updated. So it seems inappropriate to model services with deterministic functions in the

description of Taverna's semantics. Therefore we associate with each label $s \in TS$ a relation $\mathcal{F}[s] \subseteq [\![type_i(s)]\!] \times [\![type_o(s)]\!]$ such that for each tuple $t \in [\![type_i(s)]\!]$ there is at least one tuple $t' \in [\![type_o(s)]\!]$ such that $(t, t') \in \mathcal{F}[s]$. It should be noted at this point that the current implementation of Taverna does not check if a service call returns a tuple with fields of the correct type, but we chose not to model this in the presented formal semantics.

1.4. Scufl syntax

We start with a brief informal introduction to the Scufl syntax. A small example of a Scufl workflow graph is given in Fig. 1 (a). A set of workflow inputs is indicated by a dotted rectangle with a small triangle pointing upwards, which in this case contains one input labeled pin. The graph also contains a set of workflow outputs indicated by a dotted rectangle with a small triangle pointing downwards, here containing two outputs labeled $ppout$ and $pout$. Furthermore, the graph contains several so-called processors which represent operations from the Taverna services and which are labeled "Get_Nucleotide_FASTA", "Merge_String_list_to_string", "emma", "showalign" and "prettyplot". For each processor, depending on the view settings, the input ports are listed in the top row, as is done here, or in the left column, as in some of the following examples. Similarly, the output ports are listed in the bottom row or in the right column. For example, the processor with label "emma" has one input port labeled $sequence_data_direct$ and one output port labeled $outseq$.

The Scufl workflow graph defines a simple yet often needed experiment. If a pin input port is initiated with a list of nucleotide sequence identifiers, then the "Get_Nucleotide_FASTA" processor implicitly iterates on this list and with the use of an external service that searches the GenBank database [3] returns FASTA formatted nucleotide sequences that correspond to the identifiers. The next processor merges the list of those sequences into one long string, on which the "emma" processor, which is a wrapper for the ClustalW operation of the EMBOSS [11] package, performs a sequence alignment. The final two processors, "showalign" and "prettyplot" are used to present the output respectively in a textual and graphical manner.

As can be noticed, the graphical representation of the Scufl graph communicates well the main intent of the experiment. In the following examples we introduce other important features of the language and then we proceed with formal definitions and discussions of these features.

The second example is abstract and is presented in Fig. 1 (b). The graph has three branches that independently process their own input values. All the computed values, i.e., the results of "foo1", "foo2" and "foo3" processors, are directed to the *out* workflow output. Although it is not visible in the graphical representation of the Scufl graph, for the *out* workflow output an *incoming-links strategy* is specified. It determines how the value for a port is obtained in case of multiple data edges ending in it. This strategy can be either *merge* or *select-first*, where *merge* waits for values to arrive from all incoming data edges and packs them into a list while *select-first* selects the first value that arrives and ignores the others. Example use cases for the different incoming-links strategies in this abstract Scufl workflow graph would be:

- for the merge strategy — obtaining nucleotide sequences from a number of databases and packing them together into a list for further processing, e.g., alignment,
- for the select-first strategy — requesting the same computation with different services and continuing the processing with the result that arrives the quickest.

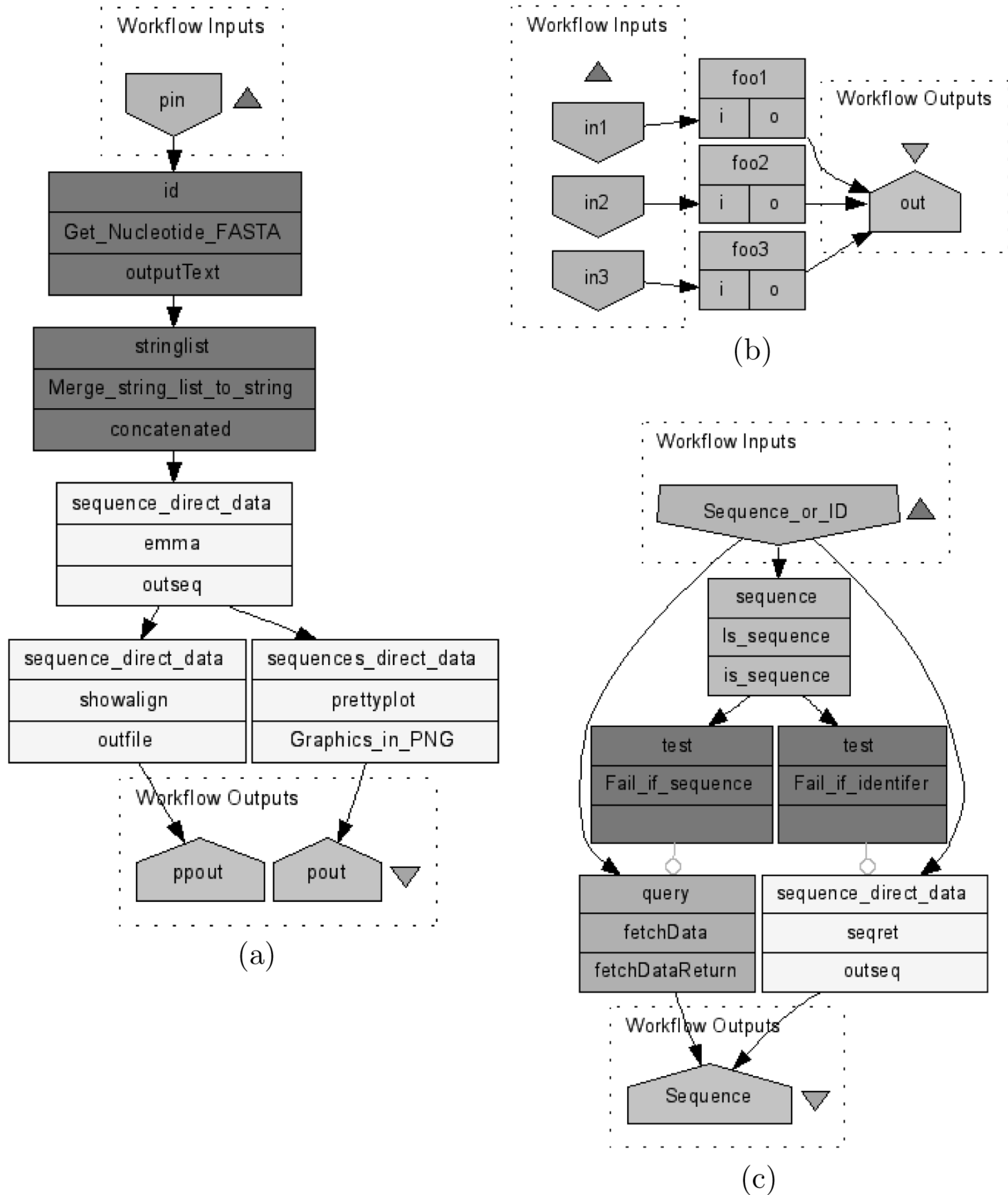

Figure 1. Taverna's visualisation of Scufl workflow graph examples

An extra feature of the select-first strategy is that the workflow is less error prone. In Taverna processors can fail, for example if no connection can be made with them over the Internet. Here the workflow finishes properly if at least one of the used tools, i.e., "foo1", "foo2" or "foo3", finishes with success.

The third example is taken from the myExperiment workflow repository [6]. It is presented in Fig. 1 (c) and is incorporated as a building block into several Scufl workflow graphs defining real-life *in silico* experiments that are also published in the repository. First, it shows that in the Scufl workflow graphs there are two kinds of edges. The *data edges*, indicated by solid edges with an arrow head, represent data flow by connecting workflow inputs or output ports of processors with input ports of processors or workflow outputs. The *control edges* indicated by gray edges ending with a circle represent additional control flow. They connect two processors specifying that one can execute only when the other has successfully finished. Second, the example presents how a combination of failing processors, control edges and ports with many incoming edges and the select-first strategy specified can be used to model conditional behavior. The Scufl workflow graph returns a sequence in a FASTA format that corresponds to a sequence or sequence entry identifier provided as an input. If a sequence identifier, in `database:identifier` format, e.g. `uniprot:wap_rat`, is provided as the input, then the "Fail_if_sequence" processor succeeds but the "Fail_if_identifier" fails and thus the "fetchData" processor uses the EBI's WSDbfetch web service (see `http://www.ebi.ac.uk/Tools/webservices/services/dbfetch`) to retrieve the sequence in FASTA format. Otherwise the "Fail_if_sequence" processor fails but the "Fail_if_identifier" succeeds and the sequence is passed through the Soaplab [7, 12] "seqret" service to force it into a FASTA format. Both conditional branches are joined with the $Sequence$ workflow output for which the select-first strategy is specified.

The last example of this informal introduction to Taverna syntax is presented in Fig. 2. We start with the analysis of the top Scufl graph which may seem incomplete because the $nin2$ has no incoming data edges. For that port a default value is specified, but that again is not visible in the graphical representation.

Another thing that the diagram does not show are the *product strategies* associated with all processors. Such strategies are needed because of the implicit iteration semantics of Scufl that was illustrated by the first processor in Fig. 1 (a). In general the implicit iteration strategy states that if a processor receives a value that is nested deeper than expected, it will iterate over subvalues of the expected nesting depth and combine the results again in a list. For example, if a processor that computes a function $f : [\![\langle\, a : \mathcal{M} \,\rangle]\!] \rightarrow [\![\langle\, b : \mathcal{M} \,\rangle]\!]$ receives on its port labeled a the value ["foo", "bar"], then it will compute the list $[f(\langle\, a = \text{“foo”} \,\rangle), f(\langle\, a = \text{“bar”} \,\rangle)]$. If a processor computes a function that expects many inputs such as $g : [\![\langle\, a : \mathcal{M}, b : \mathcal{M} \,\rangle]\!] \rightarrow [\![\langle\, c : \mathcal{M} \,\rangle]\!]$ and is presented with lists of mime values, then a product strategy such as *cross product* or *dot product* is required to indicate how the input lists are combined into a single list of tuples that represent the combination of complex values to which the function is applied during the iteration. If the list on port a is ["foo", "bar"] and the list on port b is ["x", "y", "z"] then the cross product combines them into $[[\langle\, a = \text{“foo”}, b = \text{“x”} \,\rangle, \langle\, a = \text{“foo”}, b = \text{“y”} \,\rangle, \langle\, a = \text{“foo”}, b = \text{“z”} \,\rangle], [\langle\, a = \text{“bar”}, b = \text{“x”} \,\rangle, \langle\, a = \text{“bar”}, b = \text{“y”} \,\rangle, \langle\, a = \text{“bar”}, b = \text{“z”} \,\rangle]]$ and the dot product combines them into $[\langle\, a = \text{“foo”}, b = \text{“x”} \,\rangle, \langle\, a = \text{“bar”}, b = \text{“y”} \,\rangle]$. For an arbitrary number of input ports a product strategy is defined by an expression in the following syntax:

$$ps \quad ::= \quad \varepsilon \mid PL \mid (ps \otimes ps) \mid (ps \odot ps)$$

in which each label in PL appears at most once. In this expression ε denotes the empty product strategy, a port label product strategy transforms values into tuples, $\otimes$ represents the cross product[1] and $\odot$ represents

[1]For lists the cross product $L_1 \otimes L_2$ is not equivalent with $L_2 \otimes L_1$ because the order of the resulting tuples is not the same,

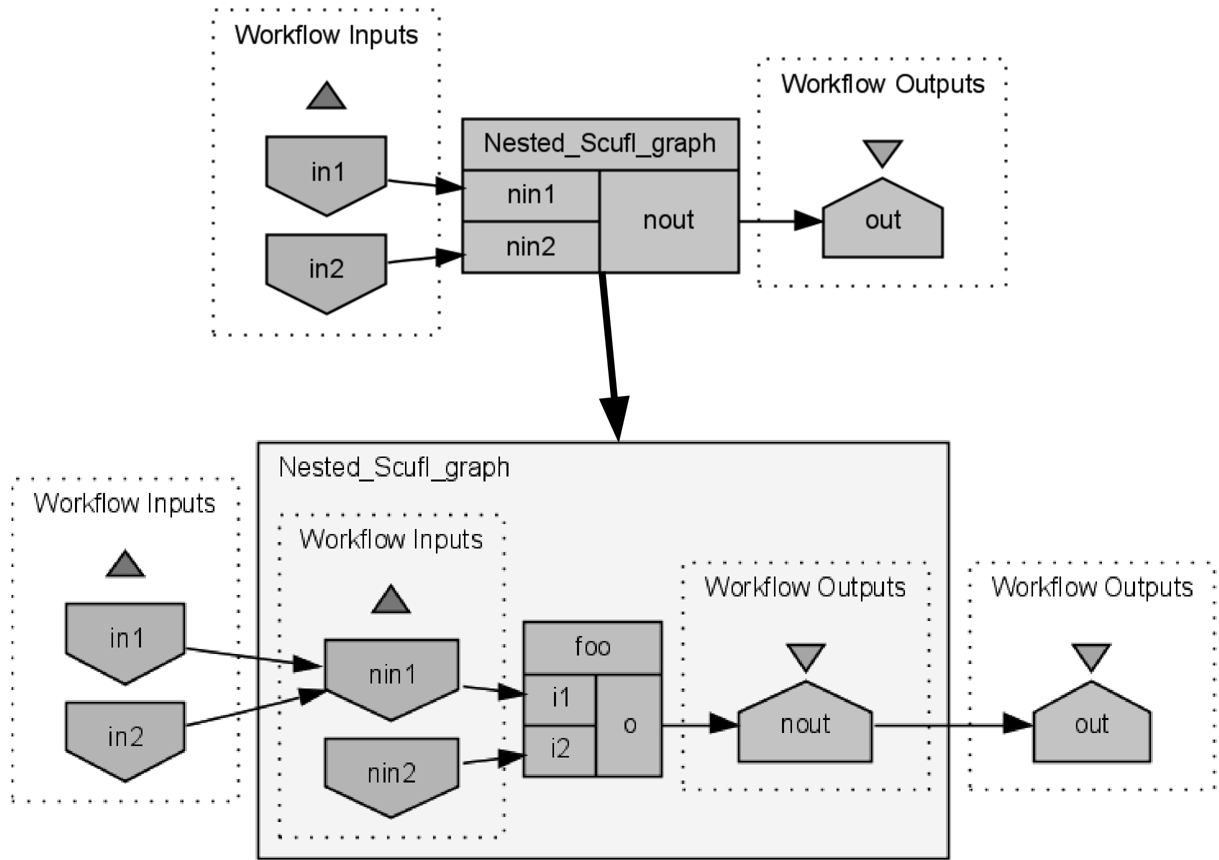

Figure 2. An example of a nested Scufl workflow graph

the dot product. The set of all product strategies is denoted as PS and the set of port labels used in product strategy ps is denoted as $\mathcal{L}(ps)$, i.e., it is defined such that $\mathcal{L}(\varepsilon) = \emptyset$, $\mathcal{L}(a) = \{a\}$ and $\mathcal{L}(ps_1 \otimes ps_2) = \mathcal{L}(ps_1 \odot ps_2) = \mathcal{L}(ps_1) \cup \mathcal{L}(ps_2)$. The result of a product strategy ps is always a possibly nested list of tuples with fields $\mathcal{L}(ps)$, e.g., if $ps = (a \otimes b) \odot c$, then this results in a possibly nested list of tuples of the form $\langle\, a = x, b = y, c = z \,\rangle$.

A product strategy could be relevant for our example if the "Nested_Scufl_graph" processor had a merge strategy specified for its $nin1$ input port, but expected only a single value and not a list. A further explanation of the default value mapping, the incoming-links strategy and the product strategy is provided in Sections 2.3 and 2.4 respectively.

The final feature presented by the example in Fig. 2 is that Scufl workflow graphs are allowed to be recursively nested. The nested Scufl graph is represented by a "Nested_Scufl_graph" processor and its workflow inputs and outputs match the input ports and output ports of the processor. Nesting a part of a Scufl workflow graph into a processor changes its semantics in two ways. The first is that the nested Scufl worklfow is not executed until all input ports are ready, and the second is that it will apply the implicit iteration strategy during its execution.

---

but in Taverna there is also a difference in how the result is nested, as will be explained later on.

The informal discussion until now was illustrated with a notation that is only one of the ways to represent Scufl workflow graphs and more elaborate representations are available in Taverna, although none of them shows all relevant aspects for understanding the complete semantics of the defined Scufl workflow graph. We follow with a comprehensive formal definition of Scufl workflow graphs — Scufl graphs for short. Since these can be recursively nested it will be an inductive definition. For this definition we postulate a countably infinite set $\mathbb{P}$ that contains all possible processor identifiers that we can use in Scufl graphs.

Definition 1.1. (Scufl graph)
The set of Scufl graphs $\mathcal{G}$ is defined as the smallest set such that every Scufl graph composed of Scufl graphs in $\mathcal{G}$ is also in $\mathcal{G}$, where such a Scufl graph is defined as a tuple $(I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ such that

- $I \subseteq PL$ is a finite set of labels representing the workflow inputs,
- $O \subseteq PL$ is a finite set of labels representing the workflow outputs,
- $P \subseteq \mathbb{P}$ is a finite set of processors disjoint with I and O,
- $\pi_i \subseteq P \times PL$ a finite set representing processor input ports,
- $\pi_o \subseteq P \times PL$ a finite set representing processor output ports,
- $E_d \subseteq (I \times \pi_i) \cup (\pi_o \times \pi_i) \cup (\pi_o \times O)$ is a set of data edges,
- $E_c \subseteq P \times P$ is a set of control edges,
- $\lambda : P \to (TS \cup \mathcal{G})$ is the processor labeling function, that maps processors to either a service label in TS or a nested Scufl graph such that for every processor $p \in P$ it holds that $I(\lambda(p)) = \{l \mid (p, l) \in \pi_i\}$ and $O(\lambda(p)) = \{l \mid (p, l) \in \pi_o\}$,
- $ils : (\pi_i \cup O) \to \{\mathsf{first}, \mathsf{merge}\}$ gives the incoming-links strategy for every input port of a processor and the workflow outputs,
- $ps : P \to PS$ gives the product strategy for every processor $p \in P$ such that $\mathcal{L}(ps(p)) = \{l \mid (p, l) \in \pi_i\}$,
- $dv : \pi_i \to \mathcal{V}_{tav} \cup \{\bot\}$ gives a default value[2] for each input port, where $\bot$ represents the lack of default value and is only allowed if the port has at least one incoming data edge, i.e., if $dv((p, l)) = \bot$, then there is a data edge $(x, (p, l)) \in E_d$ for some x,
- there are no cycles in the *dependency graph* which is defined as a directed graph over P such that there is an edge (p_1, p_2) iff there is a control edge $(p_1, p_2) \in E_c$ or there is a data edge of the form $((p_1, l_1), (p_2, l_2)) \in E_d$,

where the I and O functions for labels in TS are generalized for Scufl graphs such that for a Scufl graph g we let $I(g)$ and $O(g)$ denote the I and O component of g, respectively.

There can be no cycles in the dependency graph because it is a fundamental assumption in the semantics of Scufl that each processor starts executing only once. An interesting question is if these semantics

[2] In Taverna 1.7.1, the version that was investigated for this paper, only strings were allowed as default values.

can be adapted such that cycles can have a meaningful and intuitive semantics, but this is not investigated in this paper. An interesting case for allowing such cycles is made in [5] which attempts to show the Turing completeness of Scufl, but does not take into account that they are not allowed.

The restriction that a default value must be specified for input ports that have no arriving data edges is more strict than in the real Taverna 1.7.1, where basic processors are allowed to have input ports with neither an incoming data edge nor a data value. This is an often used feature since basic processors can wrap a service with many optional arguments and flags. However, for the sake of simplicity of presentation we will assume that this is represented in the formal syntax by a basic processor that has exactly the set of input ports that are provided and has the semantics that the real basic processor has for that particular set of input ports.

Next to generalizing I we also extend the function $type_i$ to Scufl graphs, i.e., $type_i : (TS \cup \mathcal{G}) \rightarrow \mathcal{T}_{tup}$. The main purpose of this type is to allow a processor, that is labeled by λ with a Scufl graph, to determine what type it actually expects, and use that to see if for a given complex value it will do an implicit iteration or pass it on to the nested Scufl graph. Recall that if a processor receives a value that is nested deeper than expected, then it will identify the subvalues of the expected nesting depth and iterate over those, i.e., pass them on one by one to the nested Scufl graph.

Informally, the input type of each workflow input is computed by taking the maximum of the types of processor input ports in the nested Scufl graph to which it is connected. So, for example, if the workflow input is connected to two processor input ports that expect $[[\mathcal{M}]]$ and $[\mathcal{M}]$, then the Scufl graph is assumed to expect the type $[[\mathcal{M}]]$ on this input port. The justification for taking the maximum is that this way the processor that contains the nested Scufl graph only starts implicit iteration if it is really necessary, i.e., none of the nested processors to which the value is passed on can deal with it without iteration. For example, assume that the workflow input is connected to a service with input type $\langle\, genes : [\mathcal{M}] \,\rangle$ that expects a list of genes encoded as DNA strands and selects the shortest one. Also assume that another service with input type $\langle\, gen : \mathcal{M} \,\rangle$ is also connected to this workflow input. Then, if the Scufl graph is given a list of genes, the implicit iteration is only needed for the second service and not the whole Scufl graph. This way the first service can find the shortest gene in the whole input list and not in every singleton list resulting from implicit iteration on the workflow input.

Formally, following the induction of $\mathcal{G}$, the input type of a Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ with $I = \{l_1, \ldots, l_n\}$, is defined as $type_i(g) = \langle\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\rangle$, where $\tau_i = \mathsf{max}(\{\sigma(l') \mid (l_i, (p, l')) \in E_d, \sigma = type_i(\lambda(p))\})$. Note, that this is well defined since the domain of $type_i(\lambda(p))$ is $I(\lambda(p))$, which by the definition of Scufl graph is equal to $\{l' \mid (p, l') \in \pi_i\}$.

1.5. Hierarchically nested Scufl graphs

The Scufl graph definition is an inductive definition that builds larger Scufl graphs by using smaller ones as labels of its processors, i.e., as nested Scufl graphs. It allows us to define notions and prove theorems with induction on the structure of a Scufl graph. Over the set of all Scufl graphs $\mathcal{G}$ we can define *the nesting graph* that indicates which Scufl graph is nested in which Scufl graph as follows.

Definition 1.2. (The nesting graph)
The nesting graph is the directed edge-labeled graph $\mathcal{N} = (\mathcal{G}, E)$ where $\mathcal{G}$ is the set of nodes and the set of edges $E \subseteq \mathcal{G} \times \mathbb{P} \times \mathcal{G}$ is defined such that $(g, p, g') \in E$ iff $\lambda(p) = g'$ with λ the labeling function of g and p a processor in g.

It is easy to see that, since $\mathcal{G}$ is required in its definition to be minimal, there are no directed cycles in $\mathcal{N}$. The *set of subgraphs of a Scufl graph* g, denoted as $\mathcal{G}_g$, is defined as the set of nodes reachable in $\mathcal{N}$ from g, including g itself. The *nesting graph for a particular Scufl graph* g is denoted as $\mathcal{N}_g$ and defined as subgraph of $\mathcal{N}$ induced by $\mathcal{G}_g$.

It is allowed that the same Scufl graph is reused as a label of more than one processor in a certain Scufl graph definition, either within the same subgraph or in different subgraphs. However, the definition of a state of a Scufl graph can be simplified if such reuse is not allowed and therefore we introduce the notion of *hierarchically nested Scufl graphs*.

Definition 1.3. (Hierarchically nested Scufl graphs)
A Scufl graph g is said to be *hierarchically nested* iff $\mathcal{N}_g$ is a tree.

Observe that if g is a hierarchically nested Scufl graph then all Scufl graphs in $\mathcal{G}_g$ are also necessarily hierarchically nested.

If a Scufl graph is not hierarchical then it can be made so by replacing each occurrence of a certain Scufl graph with a different but isomorphic Scufl graph. For example, if processors p_1 and p_2 are both labeled with a Scufl graph g, i.e., $\lambda_1(p_1) = \lambda_2(p_2)$, where λ_1 and λ_2 are the processor labeling function of the subgraphs in which p_1 and p_2 appear respectively, then we redefine λ_1 and λ_2 such that $\lambda_1(p_1) = g_1$ and $\lambda_2(p_2) = g_2$, where g_1 and g_2 are different but isomorphic copies of g that do not appear as subgraphs themselves. If we start with a certain Scufl graph and repeat this for every two different processors in subgraphs that are labeled with the same Scufl graph, then we will obtain an equivalent hierarchically nested Scufl graph.

In the remainder of this paper, where we describe the semantics of Scufl graphs, we will do this only for hierarchically nested Scufl graphs, and therefore, when we refer to a Scufl graph, we always mean a hierarchically nested Scufl graph. The semantics of other Scufl graphs is then defined as the semantics of the corresponding hierarchically nested Scufl graphs. The reason for this is that in a hierarchically nested Scufl graph we can describe the total state as a mapping of each Scufl graph that it contains to its particular state. The exponential blow-up that can be caused by making a Scufl graph hierarchical, is in some sense unavoidable, because it is linked to the potentially exponential number of Scufl graph instances for which a state has to be described.

2. Processor execution

2.1. An overview of processor execution

A successful execution of a processor is a complex event best explained by dividing it into several steps. We give here an informal overview of those steps and discuss the first two of them in the rest of this section in further detail by defining the functions that compute them. Then, in the second part of this paper, using those functions and additional prerequisites defined in Section 2.2, we discuss the execution of a Scufl graph as a whole, look into all the steps together, and explore all possible scenarios including the possibility of processor failure.

We now proceed with the informal description of the steps of a successful execution of a processor:

Computing the values in the input ports In the first step an input value for each input port is computed from the values that were sent to it through the incoming data edges. This is done by combining these values into a single complex value according to the incoming-links strategy. The select-first strategy simply takes the first value that arrives and ignores the others, and the merge strategy creates a list containing all the arrived values.

Combining the input port values into the processor input value Next, a *processor input value* is computed, which is a single tuple that can be processed by the service that the processor represents, or a possibly nested list of such tuples. If for every input port of the processor the value computed in the previous step is of the type expected by the processor, i.e., is not overly nested, then the processor input value is a tuple labeled by input port labels and holding the input port values. For example, if the input ports are labeled a and b and their computed input port values v_a and v_b are of the expected type, then the processor input value is $\langle\, a : v_a, b : v_b \,\rangle$. If any of the values computed in the preceding steps is too deeply nested, then the values of the different input ports must be combined into a single nested value, i.e., a list of tuples over which the processor can iterate. For example, assume that v_a is a list of mime values and v_b is a list of lists, while the processor expects types $\mathcal{M}$ and $[\mathcal{M}]$, respectively. The computation of the processor input value can then be thought of as consisting of two steps. First, the values that were computed for the input ports are transformed into values where the subvalues of the type that is expected are identified by packing them in singleton tuples. Continuing the last example, the value for the input port labeled a would be transformed to a list of tuples of type $\langle\, a : \mathcal{M} \,\rangle$ and the value for the input port labeled b would be transformed to a list of tuples of type $\langle\, b : [\mathcal{M}] \,\rangle$. Second, the product strategy of the processor describes which combinations of the identified tuples are taken and how they are nested in the result. For example, a strategy consisting of a single cross product will combine all tuples in the first value with all tuples in the second, resulting in a doubly nested list of tuples of type $\langle\, a : \mathcal{M}, b : [\mathcal{M}] \,\rangle$.

Performing the execution or the iteration If the value computed in the preceding step is a tuple, the processor is executed once, producing one result tuple with values for every output port. If the processor input value is a list, it is iterated over by executing the processor for each tuple in it. The result for each output port contains a list of values from result tuples of subsequent iteration steps that is structured accordingly to the nesting structure of the processor input list. Following the previous example, if the processor has two output ports labeled c and d, and is associated with a Taverna service with output type $\langle\, c : [\mathcal{M}], d : \mathcal{M} \,\rangle$, then the iteration will produce a list of lists with elements of type $[\mathcal{M}]$ for port labeled c, and a list of lists with elements of type $\mathcal{M}$ for port labeled d.

Copying the computed output port values When the normal execution or iteration has finished the values computed in the processor output ports are copied to all processor input ports and workflow outputs to which they are connected.

2.2. Extended complex value construction and deconstruction

As explained in the informal description of the semantics of processor execution in Section 2.1, we can describe the execution of a processor after the processor input value has been computed as a process that

takes a possibly nested list of tuples, iterates over all tuples by executing the processor and while doing so constructs for each output port a value by inserting, at the position of the original tuple, the value that was computed for that output port by the iteration step.

Since $\mathcal{V}_{tup}$ includes tuples, but not lists of tuples, we define an extended complex value set $\mathcal{V}_{ext}$ as the smallest set such that (1) $\mathcal{V}_{tup} \subseteq \mathcal{V}_{ext}$ and (2) if $x_1, \ldots, x_n \in \mathcal{V}_{ext}$ then the list $[x_1, \ldots, x_n]$ is in $\mathcal{V}_{ext}$.

In order to identify the position of tuples and other subvalues in an extended complex value we introduce the notion of subvalue index. By a *subvalue* of an extended complex value v we mean v itself, any element of v, any element of element of v, and so on, up to the tuples. For example, if $v = [[a, b], [c]]$, where a, b and c are tuples, then all subvalues of v are: v, $[a, b]$, $[c]$, a, b and c.

Definition 2.1. (Subvalue index)
A *subvalue index*, or simply *index*, is a list of positive natural numbers. Such indices are denoted by a list of numbers separated by slashes, e.g., $2/3/8$ and $1/1$, and the empty list is denoted as ϵ. The set of all complex value indices is denoted as $\mathcal{I}$.

The numbers in an index are listed from most significant on the left, to the least significant on the right. Following the last example, the subsequent indexes of the mentioned subvalues of v are: ϵ, 1, 2, $1/1$, $1/2$ and $2/1$.

Formally, the subvalue indicated by an index is defined by the function $\mathsf{get} : \mathcal{V}_{ext} \times \mathcal{I} \rightarrow (\mathcal{V}_{ext} \cup \bot)$ such that $\mathsf{get}(v, \epsilon) = v$, and $\mathsf{get}(v, i/\alpha) = \mathsf{get}(v_i, \alpha)$ if $v = [v_1, \ldots, v_n]$ and $1 \leq i \leq n$, and $\mathsf{get}(v, i/\alpha) = \bot$ otherwise. For example, if $v = [[a, b], [c]]$, then $\mathsf{get}(v, 2/1) = c$ and $\mathsf{get}(v, 2/2) = \bot$.

We assume that complex value indices are ordered according to the lexicographical ordering, i.e., the smallest binary relation $\preceq$ over $\mathcal{I}$ such that for every $i, j \in \mathbb{N}$ and $\alpha, \beta \in \mathcal{I}$ it holds that (1) $\epsilon \preceq \alpha$, (2) if $i \leq j$, then $i/\alpha \preceq j/\beta$ and (3) if $\alpha \preceq \beta$, then $i/\alpha \preceq i/\beta$. As usual this defines a linear order over $\mathcal{I}$.

In order to be able to iterate over all tuples in an extended complex value we define a function that retrieves the index of the first tuple and a function to jump to the index of the next tuple. The first function is $\mathsf{first} : \mathcal{V}_{ext} \rightarrow (\mathcal{I} \cup \bot)$ which is defined such that $\mathsf{first}(v) = \alpha$ where α is the smallest index such that $\mathsf{get}(v, \alpha) \in \mathcal{V}_{tup}$, and $\mathsf{first}(v) = \bot$ if there is no such α. The second function is $\mathsf{next} : \mathcal{V}_{ext} \times \mathcal{I} \rightarrow (\mathcal{I} \cup \bot)$ and is defined such that $\mathsf{next}(v, \alpha) = \beta$ if β is the smallest index larger than α such that $\mathsf{get}(v, \beta) \in \mathcal{V}_{tup}$, and $\mathsf{next}(v, \alpha) = \bot$ if such a β does not exist.

Finally, we define a function $\mathsf{put}(v, \alpha, w)$ that inserts into the complex value v at position α the complex value w, which can be used to construct complex values. For example, $\mathsf{put}([x, [y]], 2/1, z) = [x, [z]]$ and $\mathsf{put}([], \epsilon, z) = z$. If the position α does not yet exist in v then it is extended minimally with empty lists to create it. For example, $\mathsf{put}([], 1/1/1, x) = [[[x]]]$ and $\mathsf{put}([], 2/1, x) = [[], [x]]$. Formally, this function $\mathsf{put} : \mathcal{V}_{tav} \times \mathcal{I} \times \mathcal{V}_{tav} \rightarrow \mathcal{V}_{tav}$ is defined such that (1) $\mathsf{put}(v, \epsilon, w) = w$, (2) $\mathsf{put}(v, i/\alpha, w) = \mathsf{put}([], i/\alpha, w)$ if $v \in \mathcal{V}_{\mathcal{M}}$, (3) $\mathsf{put}([], 1/\alpha, w) = [\mathsf{put}([], \alpha, w)]$, (4) $\mathsf{put}([v] + v', 1/\alpha, w) = [\mathsf{put}(v, \alpha, w)] + v'$, (5) $\mathsf{put}([], i/\alpha, w) = [[]] + \mathsf{put}([], (i-1)/\alpha, w)$ if $i > 1$, (6) $\mathsf{put}([v] + v', i/\alpha, w) = [v] + \mathsf{put}(v', (i-1)/\alpha, w)$ if $i > 1$.

2.3. Incoming-links strategy semantics

Here we define the semantics of incoming-links strategy expressions which are used to indicate how to compute the value for a processor input port or workflow output by composing it from values provided from multiple incoming data edges. The computation is done incrementally, that is, a temporary result

is extended each time a new value arrives from one of the data edges that did not already supply a value. The lack of a previous temporary value at the start of the process is represented by $\perp$.

The select-first incoming-links strategy picks the first value to arrive and ignores all the others. This is the default behavior of processor input ports and workflow outputs. The function $[\![\mathsf{first}]\!] : ((\mathcal{V}_{tav} \cup \{\perp\}) \times \mathcal{V}_{tav}) \rightarrow \mathcal{V}_{tav}$ takes as the first argument the current temporary result and as the second the value provided by the next data edge. As a result the new temporary result is returned. Formally:

$$[\![\mathsf{first}]\!](t, v) = \begin{cases} v & \text{if } t = \perp \\ t & \text{otherwise} \end{cases}$$

The merge incoming-links strategy combines all incoming values as elements of a list. It was added to Taverna 1.3.1 to prevent the need for creation of user defined n-argument processors that compose their arguments into a list. As with select-first, the merge function $[\![\mathsf{merge}]\!] : ((\mathcal{V}_{tav} \setminus [\![\mathcal{M}]\!] \cup \{\perp\}) \times \mathcal{V}_{tav}) \rightarrow \mathcal{V}_{tav}$ has two arguments, yet now the temporary value is never of type $\mathcal{M}$ since it is a list of values provided so far. Formally:

$$[\![\mathsf{merge}]\!](t, v) = \begin{cases} [v] & \text{if } t = \perp \\ t + [v] & \text{otherwise} \end{cases}$$

Strictly speaking this is not a merge, but we stick to the Taverna terminology.

2.4. Product strategy semantics

Here we define the semantics of product strategy expressions $ps \in PS$. The product strategy expressions are used to transform values from $\mathcal{V}_{tav}$, that are provided on individual input ports of a given processor p, to extended complex values that contain tuples of type $type_i(\lambda(p))$, i.e., lists of tuples ready to be iterated upon by p.

The values provided on a processors' input ports have to be combined into a processor input value that is either a single tuple which can be processed by the service that the processor represents or a nested list of such tuples. This is done in two steps. The first step transforms each of the values provided on every input port into a single unary tuple or a list of unary tuples. The tuples' field is labeled with the same label as the respective input port and they contain values of the type that is expected on that port. The second step combines such preprocessed values for processors with multiple input ports into a single n-ary tuple or a nested list of those.

We now describe the first step in more detail. Its purpose is to identify the subvalues that are of a nesting depth acceptable by the processor. For example, if the value on the input port with label a is $[[1,2],[],[3]]$ and the processor expects a value of type $[\mathcal{M}]$ on it, then the value is transformed to $[\langle\, a = [1,2]\,\rangle, \langle\, a = []\,\rangle, \langle\, a = [3]\,\rangle]$. If this is the only input port, then the processor will iterate over the three values $[1,2]$, $[]$ and $[3]$. If, on the other hand, a value of type $\mathcal{M}$ is expected, then it is transformed to $[[\langle\, a = 1\,\rangle, \langle\, a = 2\,\rangle], [], [\langle\, a = 3\,\rangle]]$ and the processor will iterate over the three values 1, 2 and 3. This is formalized by the packing function $\mathsf{pack}_{l:\tau} : \mathcal{V}_{tav} \rightarrow \mathcal{V}_{ext}$ that identifies nested values of type τ and packs them into tuples of type $\langle\, l : \tau\,\rangle$. Formally, it is defined as follows:

$$\mathsf{pack}_{l:\tau}(x) = \begin{cases} \langle\, l = x\,\rangle & \text{if } x \in [\![\tau]\!] \\ [\mathsf{pack}_{l:\tau}(x_1), \ldots, \mathsf{pack}_{l:\tau}(x_n)] & \text{if } x = [x_1, \ldots, x_n] \notin [\![\tau]\!] \end{cases}$$

This function is well defined for every $x \in \mathcal{V}_{tav}$, which can be shown with induction on the structure of x and using the fact that $\mathcal{V}_{\mathcal{M}} \subseteq [\![\tau]\!]$ for any $\tau \in \mathcal{T}_{tav}$. It is possible that a value of type τ contains a nested value that is also of type τ. For example, if $\tau = [[\mathcal{M}]]$ and $x = [[[1]]]$, then there are in x three nested values of type τ, namely 1, [1] and [[1]]. In that case the nested value with the largest nesting depth is chosen and so $\mathsf{pack}_{a:\tau}(x) = [\langle\, a = [[1]]\, \rangle]$. For a more elaborate example consider:

$$
\begin{aligned}
&\mathsf{pack}_{a:[\mathcal{M}]}([[1],[[2],3],4]) \\
&\quad = [\mathsf{pack}_{a:[\mathcal{M}]}([1]), \mathsf{pack}_{a:[\mathcal{M}]}([[2],3]), \mathsf{pack}_{a:[\mathcal{M}]}(4)] \\
&\quad = [\langle\, a = [1]\, \rangle, [\mathsf{pack}_{a:[\mathcal{M}]}([2]), \mathsf{pack}_{a:[\mathcal{M}]}(3)], \langle\, a = 4\, \rangle] \\
&\quad = [\langle\, a = [1]\, \rangle, [\langle\, a = [2]\, \rangle, \langle\, a = 3\, \rangle], \langle\, a = 4\, \rangle]
\end{aligned}
$$

Note, that the values 3 and 4 are in $[\![[\mathcal{M}]]\!]$ and therefore also packed in a tuple.

We now proceed to the second step where we deal with the case of processors with multiple input ports. There the extended complex values computed by the packing function have to be combined. For this the cross and dot product strategy expressions are used to represent the $\times$ — cross and $\cdot$ — dot product functions[3]. An intuition of how they work on flat lists has already been given in Section 1.4.

For higher level lists the dot product used in Taverna fully flattens its arguments, operates on the flat lists and structures the result according to the structure of the argument with the highest nesting depth. For example, if a, b, c, d and e are tuples, then $[a, b] \cdot [[c], [d, e]] = [[a \cup c], [b \cup d]]$, where the union of tuple values is a well defined tuple since in product strategy expressions each label from PL appears at most once. In the case where both arguments have the same nesting depth the structuring occurs with respect to the left one. For the formal definition of the dot product we define three auxiliary notions.

The first is the function flat^* that flattens values in $\mathcal{V}_{ext}$, i.e., recursively nested lists of tuples, to lists of tuples, e.g, if x_1, x_2 and x_3 are tuples, then $\mathsf{flat}^*([[[x_1]], [[x_2], [x_3]]]) = [x_1, x_2, x_3]$. Formally, it is defined such that:

$$
\mathsf{flat}^*(x) \quad = \quad \begin{cases} [] & \text{if } x = [] \\ [x] & \text{if } x \in \mathcal{V}_{tup} \\ \mathsf{flat}^*(x_1) + \ldots + \mathsf{flat}^*(x_n) & \text{if } x = [x_1, \ldots, x_n] \end{cases}
$$

The second notion is that of *the tuple nesting depth* of a value x in $\mathcal{V}_{ext}$, denoted as $\mathsf{tnd}(x)$, which can be informally described as the maximum nesting depth of tuples in x. It is formally defined such that (1) $\mathsf{tnd}(x) = 0$ for $x \in \mathcal{V}_{tup}$, (2) $\mathsf{tnd}([]) = 1$, and (3) $\mathsf{tnd}([x_1, \ldots, x_n]) = 1 + max_{1 \leq i \leq n}(\mathsf{tnd}(x_i))$.

Finally, a $\mathsf{replace} : \mathcal{V}_{ext} \times \mathcal{V}_{ext} \rightarrow \mathcal{V}_{ext}$ partial function is defined which replaces all the subsequent tuple subvalues in the complex value provided as the first argument with the subsequent elements of the tuple list provided as the second argument. For example, assuming that every z_i and t_i is a tuple, $\mathsf{replace}([[z_1, z_2], [z_3]], [t_1, t_2, t_3]) = [[t_1, t_2], [t_3]]$. Additionally, if the first argument has more tuples than the second, the extra ones are ignored, for example $\mathsf{replace}([[z_1], [z_2, z_3], [z_4]], [t_1, t_2]) = [[t_1], [t_2]]$. Similarly, we also ignore its subvalues containing no tuples at all but only if it does not change the positions of the other subvalues, for example $\mathsf{replace}([[z_1, z_2], [z_3], []], [t_1, t_2, t_3]) = [[t_1, t_2], [t_3]]$ while $\mathsf{replace}([[[z_1], [z_2]], [[]], [[z_3, z_4]]], [t_1, t_2, t_3]) = [[[t_1], [t_2]], [], [[t_3]]]$. Formally, if z is a complex value

[3]The functions $\times$ and $\cdot$ should not be confused with $\otimes$ and $\odot$, which are the corresponding syntactical constructs in product strategy expressions.

such that $\mathsf{flat}^*(z) = [z_1, \dots, z_m]$ and $t = [t_1, \dots, t_n]$ where $m \geq n$, then $\mathsf{replace}(z, t) = r$ where r is the smallest complex value such that $\mathsf{flat}^*(r) = [r_1, \dots, r_n]$ and $\mathsf{get}(r, \alpha_i) = r_i$ for all $i = 1 \dots n$ and $\alpha_1, \dots, \alpha_n$ being the respective indexes of $z_1, \dots, z_n$ in z. The ordering of the complex values that we refer to in this definition is given such that: (1) if a and b are tuples, then $a \leq b$ iff $a = b$, and (2) $[a_1, \dots, a_n] \leq [b_1, \dots, b_m]$ iff $n \leq m$ and for each $i = 1, \dots, n$ it is true that $a_i \leq b_i$. It is easy to see, that this indeed defines a partial order.

With these notions we can now define the dot product. Let x and y be complex values such that $\mathsf{flat}^*(x) = [x_1, \dots, x_n]$ and $\mathsf{flat}^*(y) = [y_1, \dots, y_m]$. The dot product function $\cdot : \mathcal{V}_{ext} \times \mathcal{V}_{ext} \rightarrow \mathcal{V}_{ext}$ is defined such that $x \cdot y = \mathsf{replace}(z_{x,y}, t_{x,y})$ where $t_{x,y} = [x_1 \cup y_1, \dots, x_{min(n,m)} \cup y_{min(n,m)}]$ and $z_{x,y} = y$ if $\mathsf{tnd}(x) < \mathsf{tnd}(y)$ and $z_{x,y} = x$ otherwise. It is easy to see that $t_{x,y}$ and $z_{x,y}$ are well defined, and because $n \geq min(n, m) \leq m$ so is the dot product.

It should be noted that the pruning of the nested lists with no tuples by the replace function is indeed consistent with how Taverna works, e.g., for tuples a, b, c, d and e, it holds in Taverna that $[[[]], [[a, b]]] \cdot [c, d, e] = [[], [[a \cup c], [b \cup d]]]$. Also note that because of how $z_{x,y}$ is defined it is the tuple nesting depth of the arguments that decides which of the two arguments will determine the nesting structure of the result, as indeed is the case in Taverna. An interesting alternative might be to always let the left argument determine the nesting structure. That way the user can control this by simply changing the order in the product strategy expression.

The generalization of the dot product in Taverna is not the only possible generalization and may sometimes lead to unexpected results. To illustrate this we propose here an alternative where the dot product is generalized recursively. For example, if $x = [x_1, x_2]$ and $y = [y_1, y_2, y_3]$, then $x \cdot_r y = [x_1 \cdot_r y_1, x_2 \cdot_r y_2]$. If $x = [x_1, x_2]$ and y is a tuple, then $x \cdot_r y = [x_1 \cdot_r y]$, and if both x and y are tuples, then $x \cdot_r y = x \cup y$. Formally, we define the recursive dot product function $\cdot_r : \mathcal{V}_{ext} \times \mathcal{V}_{ext} \rightarrow \mathcal{V}_{ext}$ as follows:

$$k \cdot_r l = \begin{cases} [] & \text{if } \mathsf{flat}^*(k) = [] \text{ or } \mathsf{flat}^*(l) = [] \\ [k_1 \cdot_r l] & \text{if } k = [k_1, \dots, k_n] \text{ and } l \in \mathcal{V}_{tup} \\ [k \cdot_r l_1] & \text{if } k \in \mathcal{V}_{tup} \text{ and } l = [l_1, \dots, l_m] \\ [k_1 \cdot_r l_1, \dots, k_{min(n,m)} \cdot_r l_{min(n,m)}] & \text{if } k = [k_1, \dots, k_n] \text{ and } l = [l_1, \dots, l_m] \\ k \cup l & \text{if } k \in \mathcal{V}_{tup} \text{ and } l \in \mathcal{V}_{tup} \end{cases}$$

To motivate the alternative definition let us analyze the example from Fig. 3 where the initial value with an university department identifier, e.g., "informatics", is used by two services, of which one produces a

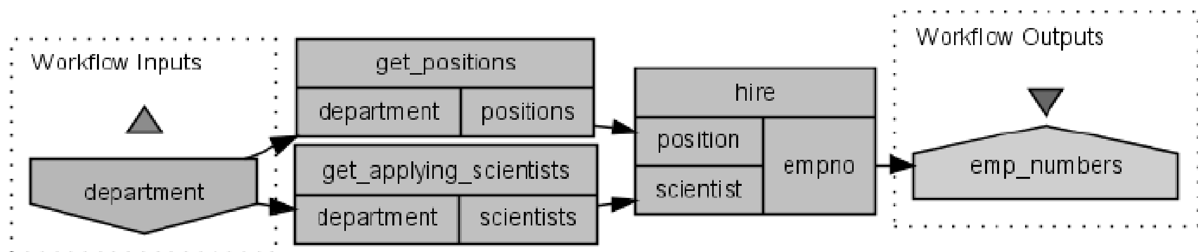

Figure 3. Recursive dot product motivation example

list of positions available in this department and other a list of scientists applying for work there. The list of positions is sorted by their appeal and the scientists are sorted according to their achievements. A third service is used to hire a scientist for a position. To deal with the values of higher types it uses the dot product strategy. This way the best positions are assigned to the best scientists and the hiring occurs while both positions and scientists are still available. Observe now that if this Scufl graph is executed with a list of departments identifiers, e.g., [“physics”, “bioinformatics”, “informatics”] and the implicit iteration over “get_positions” and “get_applying_scientists” returned $p = [[pp_1, pp_2], [pb_1, pb_2, pb_3], [pi_1, pi_2]]$ and $s = [[sp_1, sp_2, sp_3], [sb_1], [si_1, si_2]]$ respectively, then the dot product of Taverna intermixes position and scientists from different departments, i.e., the worst physicist sp_3 will be hired on the best bioinformatics position pb_1 and the best informatician si_1 will be hired on the worst bioinformatics position pb_3. Even if it is the case that informaticians and especially physicists do well as bioinformaticians, the informatics department becomes undermanned and does not get the best people. Clearly the recursive dot product does not intermix the values, so scientists will only be hired by the departments they applied to and the departments will be able to hire all the scientists that applied to them as long as they have enough positions.

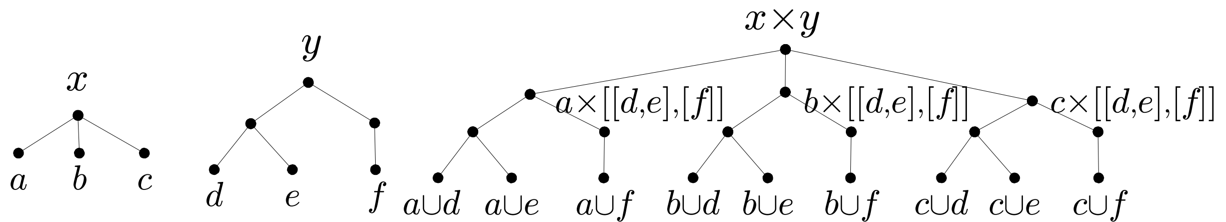

Figure 4. Cross product for higher list types

To understand the cross product of Taverna for higher list types it is convenient to think of the nested lists as ordered trees with the leafs labeled with tuple values. A tree interpretation of values $x = [a, b, c]$ and $y = [[d, e], [f]]$, where a, b, c, d, e and f are tuples, is given in Fig. 4. The cross product of x and y is then obtained by replacing each of the leaf tuples t_x in x by a copy of the y tree that in turn has its every leaf tuple value t_y replaced by $t_x \cup t_y$ (see Fig. 4). This in our case results in $[[[a \cup d, a \cup e], [a \cup f]], [[b \cup d, b \cup e], [b \cup f]], [[c \cup d, c \cup e], [c \cup f]]]$. Formally, we define the cross product function $\mathcal{V}_{ext} \times \mathcal{V}_{ext} \to \mathcal{V}_{ext}$ as follows:

$$k \times l = \begin{cases} [] & \text{if } \mathsf{flat}^*(k) = [] \text{ or } \mathsf{flat}^*(l) = [] \\ [k_1 \times l, \ldots, k_n \times l] & \text{if } k = [k_1, \ldots, k_n] \text{ and } l \neq [] \\ [k \times l_1, \ldots, k \times l_m] & \text{if } k \in \mathcal{V}_{tup} \text{ and } l = [l_1, \ldots, l_m] \\ k \cup l & \text{if } k \in \mathcal{V}_{tup} \text{ and } l \in \mathcal{V}_{tup} \end{cases}$$

Observe that the cross product of Taverna for flat lists is not a natural version of the Cartesian product for lists. Although all the combinations of the argument's tuples are returned, the nesting structure of the result is deeper, i.e., if $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$, then $x \times y = [[x_1 \cup y_1, \ldots, x_1 \cup y_m], \ldots, [x_n \cup y_1, \ldots, x_n \cup y_m]]$, while for the Cartesian product one would expect $[x_1 \cup y_1, \ldots, x_1 \cup y_m, \ldots, x_n \cup y_1, \ldots, x_n \cup y_m]$. A natural generalization of the usual Cartesian product for lists can be

obtained by defining it recursively for higher order lists as follows:

$$k \times_r l = \begin{cases} [] & \text{if } \mathsf{flat}^*(k) = [] \text{ or } \mathsf{flat}^*(l) = [] \\ [k_1 \times_r l, \ldots, k_n \times_r l] & \text{if } k = [k_1, \ldots, k_n] \text{ and } l \in \mathcal{V}_{tup} \\ [k \times_r l_1, \ldots, k \times_r l_m] & \text{if } k \in \mathcal{V}_{tup} \text{ and } l = [l_1, \ldots, l_m] \\ [\ \ k_1 \times_r l_1, \ldots, k_1 \times_r l_m, & \\ \quad \ldots, & \text{if } k = [k_1, \ldots, k_n] \text{ and } l = [l_1, \ldots, l_m] \\ \quad k_n \times_r l_1, \ldots, k_n \times_r l_m] & \\ k \cup l & \text{if } k \in \mathcal{V}_{tup} \text{ and } l \in \mathcal{V}_{tup} \end{cases}$$

Notice that when empty lists don't appear, the nesting depth of the result value for the cross product is the sum of the nesting depths of the arguments and for the generalized Cartesian product it is the maximum. We want to stress that the summing of nesting depths of the arguments in the cross product used in Taverna may be sometimes unexpected for the user. For example, when a Scufl graph with one input port of type $\mathcal{M}$ and one output port type $\mathcal{M}$ is initiated with a list of lists of mime elements, then most users would expect for it to result also with such a list. Yet, if at the start of this Scufl graph a preprocessing of the input value takes place by a binary operation for which a cross product is specified and both input ports are connected to the workflow input, then the result will be a four times nested list of mime elements. Even more interesting is the observation that this will not be the case when such a Scufl graph is nested. Then, a full implicit iteration will occur for the processor representing the nested Scufl graph, i.e., the nested Scufl graph is executed on values of the expected type and the implicit iteration mechanism collects the results into a list of the same structure as the one that was iterated over.

Besides the different nesting of result values, the cross product of Taverna and the generalized Cartesian product order the leaf elements differently, e.g., if a, b, c and d are tuples, $x = [[a, b]]$, and $y = [[c], [d]]$, then $\mathsf{flat}^*(x \times y) = [a \cup c, a \cup d, b \cup c, b \cup d]$, while $\mathsf{flat}^*(x \times_r y) = [a \cup c, b \cup c, a \cup d, b \cup d]$.

Both operations, the cross product and the recursively generalized Cartesian product, may be useful to the user and it is not obvious how to simulate one with the other.

Given the definitions of the cross and dot product we can now define the semantics of a product strategy ps for a processor in a certain Scufl graph. Let τ be the input tuple type of the processor and ps a product strategy such that $\mathcal{L}(ps) = \mathrm{dom}(\tau)$. Then we define for each such product strategy ps and type τ a function $[\![ps]\!]^\tau : (\mathcal{L}(ps) \to \mathcal{V}_{tav}) \to \mathcal{V}_{ext}$ that maps a tuple of complex values containing a field for each port label in ps to an extended complex value which the processor can execute on or iterate over. Formally, we define this function as follows:

$$\begin{aligned} [\![\varepsilon]\!]^\tau(t) &= \langle\rangle \\ [\![l]\!]^\tau(t) &= \mathsf{pack}_{l:\tau(l)}(t(l)) \\ [\![(ps_1 \otimes ps_2)]\!]^\tau(t) &= [\![ps_1]\!]^\tau(t|_{\mathcal{L}(ps_1)}) \times [\![ps_2]\!]^\tau(t|_{\mathcal{L}(ps_2)}) \\ [\![(ps_1 \odot ps_2)]\!]^\tau(t) &= [\![ps_1]\!]^\tau(t|_{\mathcal{L}(ps_1)}) \cdot [\![ps_2]\!]^\tau(t|_{\mathcal{L}(ps_2)}). \end{aligned}$$

All versions of cross and dot products defined here are binary expressions. They can be easily generalized for more arguments thanks to the observation that $x \times (y \times z) = (x \times y) \times z$ and $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ regardless of which, original or recursive, definition is chosen. In fact, it is the generalized versions which are offered in Taverna. Note also, that for higher level lists usually $x \times y \neq y \times x$, so the order of port labels in the product strategy expression is important.

3. Conclusion

In this paper we have presented a formal definition of the syntax of Scufl, the workflow specification language of the Taverna environment, and the fundamental notions that underly the semantics of Scufl. The syntax of Scufl is based on hierarchically nested graphs where nodes describe the processors that perform some computation or call a service, and the edges indicate data flow or control flow between processors. One of the fundamental concepts for the semantics that is discussed are the incoming links strategies that can be specified to deal with the case where multiple incoming values might have to be combined for the same input port. Another presented fundamental concept is the product strategy that is specified to deal with the case where one or more input ports receive values of an incorrect type. These two concepts are at the heart of the semantics of Scufl and set it apart from other workflow languages.

In the follow-up paper we present the full formal semantics of Sculf graphs based on the notions presented here. To take into account that these graphs define processes and not just computations we describe their semantics in terms of a transition system, i.e., we describe the possible states of a Scufl graph and all the transitions between these states. We also show that these formal semantics are effective in the sense that they can be used to prove certain properties of Scufl graphs. Finally, we compare this work to earlier work on the syntax and semantics of Taverna [14].

References

[1] Balsters, H., Fokkinga, M. M.: Subtyping can have a simple semantics, *Theor. Comput. Sci.*, **87**(1), 1991, 81–96, ISSN 0304-3975.

[2] Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying Business Processes., *VLDB* (U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, Y.-K. Kim, Eds.), ACM, 2006, ISBN 1-59593-385-9.

[3] Benson, D. A., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., Wheeler, D. L.: GenBank, *Nucleic Acids Res*, **36**(Database issue), January 2008, ISSN 1362-4962.

[4] Christophides, V., Hull, R., Kumar, A.: Querying and Splicing of XML Workflows, *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, Springer-Verlag, London, UK, 2001, ISBN 3-540-42524-1.

[5] Glatard, T., Montagnat, J.: Implementation of Turing Machines with the Scufl Data-Flow Language, *CC-GRID*, IEEE Computer Society, 2008.

[6] Goble, C. A., De Roure, D. C.: myExperiment: social networking for workflow-using e-scientists, *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*, ACM Press, New York, NY, USA, 2007, ISBN 978-1-59593-715-5.

[7] Kandaswamy, G., Fang, L., Huang, Y., Shirasuna, S., Marru, S., Gannon, D.: Building web services for scientific grid applications, *IBM Journal of Research and Development*, **50**(2/3), 2006, 249–260, ISSN 0018-8646.

[8] Li, P., Hayward, K., Jennings, C., Owen, K., Oinn, T., Stevens, R., Pearce, S., Wipat, A.: *Proceedings of the UK e-Science All Hands Meeting 2004*, Nottingham, UK, September 2004.

[9] Oinn, T., Addis, M., Ferris, J., Marvin, D., Greenwood, M., Carver, T., Wipat, A., Li, P.: Taverna: A tool for the composition and enactment of bioinformatics workflows, *Bioinformatics*, 2004.

[10] Oinn, T., Greenwood, M., Addis, M., Alpdemir, M. N., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M. R., Senger, M., Stevens, R., Wipat, A., Wroe, C.: Taverna: lessons in creating a workflow environment for the life sciences: Research Articles, *Concurr. Comput. : Pract. Exper.*, **18**(10), 2006, 1067–1100, ISSN 1532-0626.

[11] Rice, P., Longden, I., Bleasby, A.: EMBOSS: The European Molecular Biology Open Software Suite (2000), *Trends in Genetics*, **16**(6), 2000, 276–277.

[12] Rice, P. M., Bleasby, A. J., Haider, S. A., Ison, J. C., McGlinchey, S., Uludag, M.: EMBRACE: Bioinformatics Data and Analysis Tool Services for e-Science, *e-science*, **0**, 2006, 146.

[13] Stevens, R., Tipney, H., Wroe, C., Oinn, T., Senger, M., Goble, C., Lord, P., Brass, A., Tassabehji, M.: Exploring Williams-Beuren Syndrome using ^{my}Grid, *Proceedings of 12th International Conference on Intelligent Systems in Molecular Biology*, 2004.

[14] Turi, D., Missier, P., Goble, C., De Roure, D., Oinn, T.: Taverna Workflows: Syntax and Semantics, *e-Science and Grid Computing, IEEE International Conference on*, 2007.