Towards a Calculus for Collection-Oriented Scientific Workflows with Side Effects

Jan Hidders[1] and Jacek Sroka[2,⋆]

[1] University of Antwerp, Belgium
[2] University of Warsaw, Poland

Abstract. In this paper we propose a calculus that can be used to describe the semantics of collection-oriented scientific workflow systems such as the Taverna workbench. Typically such systems focus on the specification and execution of workflows with a relatively simple control flow and a more complex data flow that involves large nested collections of data. An essential operation in such workflows is the instantiation of a certain nested workflow for each element of a collection. We argue that if such workflows call external services, their semantics must be described not only in terms of input-output behavior but also take side effects into account. Based on this assumption a trace semantics is defined that corresponds to the observational equivalence of two workflow specifications. We show that under such a semantics a relatively small calculus with a structural semantics can be defined and used to describe such workflows. This is demonstrated by giving a translation of Taverna workflows in terms of this calculus.

1 Introduction

In many life sciences like chemistry, meteorology, geology, astrology and especially bioinformatics data processing experiments are conducted with the help of the Internet by using services made available by scientific institutions. Such services include databases with information that has been collected and verified by the scientific community, and state of the art domain specific data analysis algorithms. This way hypotheses can be verified before engaging in often expensive and time consuming traditional experiments. For example, in bioinformatics the effectiveness of a new drug or the possibility of mutating of a certain living organism like the budding yeast (saccharomyces cerevisiae) so that it produces desirable chemical substance can be tested *in silico* and then, only if the probability of success is high, proven *in vitro*.

Traditionally services provided on the Internet had interfaces intended for human users and were operated by coping and pasting data between HTML forms. For example, the FASTA Sequence Comparison at the University of Virginia [1] and the Basic Local Alignment Search Tool at NCBI [2] can be used in this way. Yet, nowadays specialized workbenches with simple graphical notations

⋆ Supported by Polish government grant no. N206 007 32/0809.

and integrating many useful tools and services like Taverna [3] and Kepler [4] are popular. Once designed in such a workbench, experiments can be conducted several times with different input data or verified and repeated by independent reviewers.

The notations and languages used in workbenches are similar and sometimes based on the ones known from business process modeling and workflow modeling as well as database research. This is because on one hand control flow and data flow dependencies of the used services have to be specified, and on the other hand nested collections of data are processed, e.g., are iterated upon or filtered. Because of the combination of those two aspects — workflow organization and processing of nested collections of data — such workflows are sometimes distinguished as Collection-Oriented Scientific Workflows or COSWs.

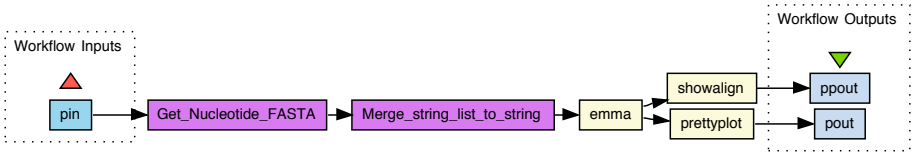


Fig. 1. A collection-oriented scientific workflow

An example COSW defined in Taverna is presented in Figure 1. It defines a simple yet often needed experiment. The *pin* input port on the left hand side can be 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 [5] returns FASTA formated nucleotide sequences that correspond to the identifiers. The next processor merges those sequences into one long string, on which the *emma* processor performs a sequence alignment. This latter processor is a wrapper for the *ClustalW* operation of the *EMBOSS* [6] package. The final two processors, *showalign* and *prettyplot* are used to present the output respectively in a textual and graphical manner.

In this work we are interested in the study of formal models and languages used in COSW tools and Taverna in particular. A formal definition of their semantics is important for several reasons. First, during its definition the consistency of the language design, and to some extent perhaps also its implementation, is verified. Second, it is required for automatic translation of workflows to other models and for research on enactment optimization techniques. Finally, formal semantics is also necessary for the study of the expressive power of such languages. In earlier work by Turi, Missier, Goble, De Roure and Oinn [7] a general syntax and operational semantics were defined formally for *Scufl* — the COSW definition language of Taverna. However, we feel that this work ignores some aspects of Scufl which we argue can be important in certain circumstances. One of these aspects is the fact that the workflow might produce partial results in the

sense that not for all output ports a result value is produced. Another aspect is the failure of a processor, which is used in Scufl as a way of achieving conditional branching. Finally, the aspect of side effects is ignored, since the semantics are described only in terms of input-output behavior, which makes for example the use of control flow links meaningless. The main contribution of this work is that we show that it is possible to define a calculus with a relatively simple syntax and semantics, that does take all these aspects into account.

A fundamental assumption of the calculus is that for some external functions it matters (1) how often they are called, because for example they represent a web service that costs money, and (2) in which order they are called, because for example they are part of the protocol for a stateful web service. Therefore we will model the semantics of workflows in terms of traces that contain function calls of the form $f(t_1)^{t_2}$ where f is the name of the function, t_1 the input value that was supplied and t_2 the value that was returned by the function. In case the function call fails and returns no value, which we would like to take into account explicitly in the formalism, it is of the form $f(t_1)^{\perp}$. In addition we register in the trace separate events for consuming values from input ports and producing values on output ports. These are denoted as $\mathbf{in}_a(v)$, which consumes value v on port a, and $\mathbf{out}_b(w)$, which produces value w on port b. An execution of a workflow might then look for example like the lists $[\mathbf{in}_a(1), \mathbf{in}_b(3), f(\langle a = 1, b = 4\rangle)^{\langle c=5\rangle}, \mathbf{out}_c(5)]$ or $[f(\langle a = 3, b = 6\rangle)^{\langle c=8\rangle}, \mathbf{out}_a(3), \mathbf{in}_b(5)]$. There are no restrictions on the order of the events, so traces do not necessarily begin with input events or end with output events. We also explicitly allow that there are no input and output events for certain input and output ports of a workflow, respectively, but there can be at most one such event for each port. Based in this notion of trace we can then define the semantics of a workflow simply as the set of all traces that describe a possible complete run of the workflow.

These fundamental assumptions set this work apart from other related work on describing workflows. The formal semantics of Ptolemy and Kepler [8, 9] ignore side effects. Work on event algebras [10] and interaction expressions [11] that does take side effects into account, ignores the dataflow aspects such as iteration over lists. Finally, formalisms based on process algebra [12] generally define a kind of bisimulation semantics, which we argue is not appropriate here since only trace semantics corresponds to observational equivalence.

The remainder of this paper is organized as follows. In Section 2 we present the syntax of the calculus. In Section 3 we present the formal semantics of the calculus. In Section 4 we describe informally Scufl and discuss how it can be mapped to the calculus. Finally, in Section 5 we present the conclusion.

2 Formal syntax and informal semantics

2.1 Preliminary definitions

We begin with postulating a countably infinite set of port names $\mathcal{P}$ and a set of function names $\mathcal{F}$, and each name in $\mathcal{F}$ represents an external function or

service that can be called by a workflow. Variants of the variables a, b, c, d and e are used to denote port names, and f and g to denote function names.

For the purpose of this paper we define only a very rudimentary type system that consists of a single basic type s and a list type constructor.

Definition 1 (port type). *The set of port types $\mathcal{T}$ is defined by the abstract syntax rule $T ::= \mathsf{s} \mid [T]$.*

Here s denotes the basic type and $[\tau]$ the type of lists of elements of type τ. We will use variants of the variables τ, σ and ρ to denote port types. We define an ordering $<$ over port types that compares the nesting depth, i.e., $\tau < \sigma$ iff the nesting depth of τ is smaller than the nesting depth of σ. The type of a workflow consists of an input type and an output type which both indicate for each port the expected type. Such a type is called an interface type.

Definition 2 (interface type). *An interface type is defined as a partial function $\iota : \mathcal{P} \to \mathcal{T}$ that is defined for a finite subset of $\mathcal{P}$. Such an interface type is denoted as $\langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle$ and the empty type is allowed and denoted as $\langle\rangle$. The set of all interface types is denoted as Θ.*

We will use variants of the variables ι and κ to denote interface types. The disjoint union of two interface types ι and κ is denoted as ι, κ. For partial functions and other binary relations t such as interface types the *domain* of t is defined as $\mathbf{dom}(t) = \{x \mid (x, y) \in t\}$. We assume that for all function names in $\mathcal{F}$ an input interface type ι and output interface type κ are given, which is denoted as $f : \iota \to \kappa$. Finally we introduce the notion of *interface* in order to describe the input and output ports of workflow without referring to their port type.

Definition 3 (interface). *An* interface *is a finite set $I \subset \mathcal{P}$.*

We will use variants of the variables I and J to denote interfaces.

2.2 The syntax of the calculus

The syntax of the calculus is defined by inference rules for propositions of the form $P : \iota_1 \Rightarrow \iota_2$ which indicates that P is a valid expression with input type ι_1 and output type ι_2. The set of all rules is given in Figure 2. In the following we briefly explain each rule and the operator it defines.

The function name f denotes the workflow that reads on its input ports the required inputs for f, calls the external function that is represented by f and if this call does not fail, returns the results on its output ports. The function is only called if the values in the input ports make up a tuple for which f is defined.

The following four rules define list manipulation operations. The list wrapping workflow $\mathbf{wrap}_{a \to b}$ reads a value v from input port a, and returns the list $[v]$ on output port b. The list concatenation workflow $\mathbf{conc}_{a;b \to c}$ reads a list on input ports a and b, concatenates the list on a with the list on b and returns the result on output port c. The ports a and b are separated by a semicolon

$$\frac{f : \iota \rightarrow \kappa}{f : \iota \Rightarrow \kappa} \qquad \frac{}{\mathbf{wrap}_{a \rightarrow b} : \langle a : \tau \rangle \Rightarrow \langle b : [\tau] \rangle} \qquad \frac{}{\mathbf{conc}_{a;b \rightarrow c} : \langle a : [\tau], b : [\tau] \rangle \Rightarrow \langle c : [\tau] \rangle}$$

$$\frac{}{\mathbf{flat}_{a \rightarrow b} : \langle a : [[\tau]] \rangle \Rightarrow \langle b : [\tau] \rangle}$$

$$\frac{\begin{array}{c} P : \iota_1 \Rightarrow \kappa_1, \kappa_2 \qquad Q : \kappa_2, \iota_2 \Rightarrow \kappa_3 \\ I = \mathbf{dom}(\kappa_2) \qquad \mathbf{dom}(\iota_1) \cap \mathbf{dom}(\iota_2) = \emptyset \qquad \mathbf{dom}(\kappa_1) \cap \mathbf{dom}(\kappa_3) = \emptyset \end{array}}{(P \triangleright_I Q) : \iota_1, \iota_2 \Rightarrow \kappa_1, \kappa_3}$$

$$\frac{\iota = \langle b_1 : \tau, \ldots, b_n : \tau \rangle \qquad I = \mathbf{dom}(\iota)}{\mathbf{ln}_{a \rightarrow I} : \langle a : \tau \rangle \Rightarrow \iota} \qquad \frac{\iota = \langle a_1 : \tau, \ldots, a_n : \tau \rangle \qquad I = \mathbf{dom}(\iota)}{\mathbf{first}_{I \rightarrow b} : \iota \Rightarrow \langle b : \tau \rangle}$$

$$\frac{P : \iota \Rightarrow \kappa \qquad I = \mathbf{dom}(\iota) \qquad J = \mathbf{dom}(\kappa)}{\mathbf{nest}_I^J(P) : \iota \Rightarrow \kappa}$$

$$\frac{\begin{array}{c} P : \langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle, \iota \Rightarrow \langle b_1 : \sigma_1, \ldots, b_m : \sigma_m \rangle \\ I = \{a_1, \ldots, a_n\} \qquad n \geq 1 \qquad J = \{b_1, \ldots, b_m\} \end{array}}{\odot_I^J(P) : \langle a_1 : [\tau_1], \ldots, a_n : [\tau_n] \rangle, \iota \Rightarrow \langle b_1 : [\sigma_1], \ldots, b_m : [\sigma_m] \rangle}$$

Fig. 2. The syntax for the basic calculus

to indicate that their order matters, since $\mathbf{conc}_{a;b \rightarrow c} \not\equiv \mathbf{conc}_{b;a \rightarrow c}$. Finally, the list flattening workflow $\mathbf{flat}_{a \rightarrow b}$ reads a list of lists on port a, concatenates the elements of this list, and returns the result on port b.

The composition operator $\triangleright_I$ denotes the composition of two workflows such that they are connected through interface I. When I is enumerated we will usually omit the brackets and write $P \triangleright_{b,c} Q$ in stead of $P \triangleright_{\{b,c\}} Q$. The meaning of $P \triangleright_I Q$ is that the output ports of P that are in I, are connected with the synonymous input ports of Q. The output ports of P that are not in I become output ports of the result, and the input ports of S that are not in I become input ports of the result. In Figure 3 (a) the result of $P \triangleright_{b,c} Q$ is illustrated, assuming that $P : \iota_1 \Rightarrow \kappa_1$ with $\mathbf{dom}(\iota_1) = \{a, b\}$ and $\mathbf{dom}(\kappa_1) = \{a, b, c\}$, and $Q : \iota_2 \Rightarrow \kappa_2$ with $\mathbf{dom}(\iota_2) = \{b, c, d\}$ and $\mathbf{dom}(\kappa_2) = \{b, c\}$. The requirements for a valid composition of P and Q are that (1) the connected ports in I have the same port type in P and Q, (2) the output ports of P that are not in I must not also be output ports of Q, and (3) input ports of Q that are not in I must not also be input ports input ports of P. The latter two constraints ensure that every port of the total workflow is connected to exactly one port of P or Q. To reduce the number of brackets in expressions we will assume that the operator is left associative.

The linking workflow $\mathbf{ln}_{a \rightarrow I}$ copies a value of its input port a to all its output ports in I, as is illustrated in Figure 3 (b). It's purpose is to copy values to multiple ports and rename ports. The latter is required if we want to connect

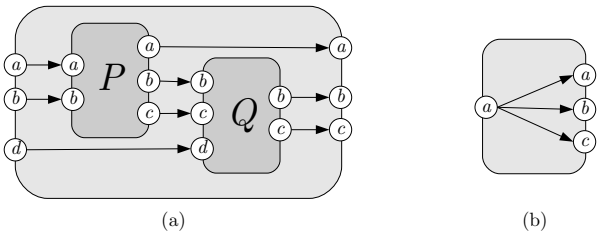
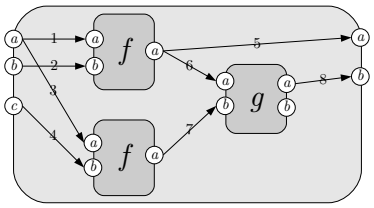


Fig. 3. Illustrations of $P \triangleright_{b,c} Q$ and $\mathbf{ln}_{a \to a,b,c}$

output ports to differently named input ports with the composition operator. Together with the composition operator it allows the representation of acyclic workflow graphs such as shown in Figure 4 where the global input ports and local output ports can have arbitrary many leaving edges, but the global output ports and the local input ports have always exactly one incoming edge. Such workflow graphs can be mapped by introducing a specific port label for each edge, indicated in the drawing by a number i and represented as label e_i. Then, the graph can be represented as:



$$\mathbf{ln}_{a \to e_1,e_3} \triangleright_\emptyset \mathbf{ln}_{b \to e_2} \triangleright_\emptyset \mathbf{ln}_{c \to e_4}$$
$$\triangleright_{e_1} \mathbf{ln}_{e_1 \to a} \triangleright_\emptyset \mathbf{ln}_{e_2 \to b} \triangleright_{a,b} f \triangleright_a \mathbf{ln}_{a \to e_5,e_6}$$
$$\triangleright_{e_3} \mathbf{ln}_{e_3 \to a} \triangleright_\emptyset \mathbf{ln}_{e_4 \to b} \triangleright_{a,b} f \triangleright_a \mathbf{ln}_{a \to e_7}$$
$$\triangleright_{e_6} \mathbf{ln}_{e_6 \to a} \triangleright_\emptyset \mathbf{ln}_{e_7 \to b} \triangleright_{a,b} g \triangleright_a \mathbf{ln}_{a \to e_8} \triangleright_b \mathbf{ln}_{b \to \emptyset}$$
$$\triangleright_{e_5} \mathbf{ln}_{e_5 \to a} \triangleright_{e_8} \mathbf{ln}_{e_8 \to b}.$$

Fig. 4. An acyclic workflow graph and a corresponding calculus expression

The select-first workflow $\mathbf{first}_{I \to b}$ has input ports I and copies the value that arrives first on one of its input ports to the b output port, and ignores any values that may or may not arrive on the other input ports. The purpose of this operation is to allow the definition of the corresponding incoming links strategy as defined in Taverna, as is discussed in Section 4.1.

The nesting constructor $\mathbf{nest}_I^J(P)$ defines a workflow that behaves as a single function, i.e., it first waits until on all ports in I a value has arrived, then executes P, and after P has finished and has produced on all output ports in J a value, then these are copied to the output ports of the total workflow. Since I and J are determined by the type of P, we can omit them in expressions. Note that a workflow P can be unfinished although it has already produced values on all its output ports, because, for example, it still needs to do a function call that has no output ports. This constructor is introduced to describe the behavior of a nested workflow in Scufl.

The iteration constructor $\odot_I^J(P)$ expresses the iteration of workflow P over the lists that are received in the ports in I. The interface J indicates the set of output ports, which must be the same as that of P, and therefore can be omitted in expressions. As an example, consider f such that $f : \langle a : \mathsf{s}\rangle \Rightarrow \langle b : \mathsf{s}\rangle$, then $\odot_a^b(f)$ is of type $\langle a : [\mathsf{s}]\rangle \Rightarrow \langle b : [\mathsf{s}]\rangle$ and applies f to each element of the list received on port a. If multiple ports are mentioned in I then it iterates over the lists received on these ports simultaneously. For example, if $g : \langle a : \mathsf{s}, b : \mathsf{s}\rangle \Rightarrow \langle c : \mathsf{s}\rangle$ and $\odot_{a,b}^c(g)$ receives as input $\langle a = [1,2,3], b = [4,5]\rangle$ then it executes the function calls $g(\langle a = 1, b = 4\rangle)$ and $g(\langle a = 2, b = 5\rangle)$. If P has input ports that are not mentioned in I then the values on these ports are simply copied for each iteration. For example, if $\odot_a^c(g)$ receives as input $\langle a = [1,2], b = 3\rangle$ then it executes $g(\langle a : 1, b : 3\rangle)$ and $g(\langle a : 2, b : 3\rangle)$. This operator can also be used to iterate over the Cartesian product of two lists: $\odot_a^c(\odot_b^c(g))$ executes g for all combinations of elements of the lists received on port a and b. Note, however, that if $P = (\mathbf{ln}_{a\to a} \triangleright_\emptyset \mathbf{ln}_{b\to b})$ then the result of applying $\odot_a^{a,b}(\odot_b^{a,b}(P))$ to $\langle a = [1,2], b = [3,4]\rangle$ is $\langle a = [[1,1],[2,2]], b = [[3,4],[3,4]]\rangle$ and not $\langle a = [1,1,2,2], b = [3,4,3,4]\rangle$.

3 Formal semantics

3.1 Preliminary definitions

Lists play a central role in the semantics of the calculus since traces are lists and also the values that are manipulated by the workflows are lists. They are denoted as $[v_1, \ldots, v_n]$, with the empty list denoted as $[\,]$. The concatenation of two lists v and w is denoted as $v \cdot w$. If $w = [w_1, \ldots, w_n]$ then we write $v \in w$ to denote that v appears in w, i.e., there exists an $1 \leq i \leq n$ such that $w_i = v$. We will use the list comprehension notation, such as $[v \mid v \leftarrow w, v > 5]$, which constructs the sublist of list w that consists of all elements larger than 5. The length of a list v is denoted as $|v|$.

In order to define the semantics of port types we postulate the set of string values $\mathcal{S}$ which will be semantics of the basic type s. In our examples we will assume that $\mathcal{S}$ also includes numbers.

Definition 4 (type semantics). *The semantics of a port type τ, denoted as $[\![\tau]\!]$, is defined with induction on the structure of τ such that (1) $[\![\mathsf{s}]\!] = \mathcal{S}$ and (2) $[\![[\tau]]\!]$ is the set of all lists $[v_1, \ldots, v_n]$ with $n \geq 0$ and $v_i \in [\![\tau]\!]$ for all $1 \leq i \leq n$. The set of all possible* port values *is the union of all semantics of all port types and is denoted as $\mathcal{V}$, i.e., $\mathcal{V} = \bigcup_{\tau\in\Theta} [\![\tau]\!]$.*

We will use variants of the variables v, w, x and y to range over port values.

The semantics of interface types is based on the notion of tuples over port values.

Definition 5 (interface value). *An* inteface value *is a partial function $t : \mathcal{P} \to \mathcal{V}$ that is defined for a finite subset of $\mathcal{P}$. Such a value is denoted as $\langle a_1 = v_1, \ldots, a_n = v_n\rangle$.*

The semantics of interface types is then roughly defined as the interface values that match the interface type. However, since we want to allow that workflows can run without values on all input ports or can can finish without producing values on all output ports, we allow that some fields of the tuples are undefined. This leads to the following definition.

Definition 6 (interface type semantics). *The semantics of an interface type* $\iota = \langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle$*, denoted as* $[\![\iota]\!]$*, is defined as the set of all interface values* $t = \langle b_1 = v_1, \ldots, b_m = v_m \rangle$ *such that* $\{b_1, \ldots, b_m\} \subseteq \{a_1, \ldots, a_n\}$ *and* $v_i \in [\![\iota(b_i)]\!]$ *for all* $1 \leq i \leq m$ *.*

For the function names f in $\mathcal{F}$ which represent external services that can be called by workflows, we postulate that if $f : \iota_1 \Rightarrow \iota_2$ then their semantics is defined by a binary relation $[\![f]\!] \subseteq [\![\iota_1]\!] \times [\![\iota_2]\!]$. Note that these binary relations are not necessarily total or functional, as might be expected for functions. We allow them to be partial in order to model that the associated service is only called if the input values satisfy certain preconditions, apart from belonging to the input types. We allow them to be non-functional to model the fact that the result of a call may be non-deterministic from the point of view of the workflow. Also note that, because of the chosen semantics of interface types, it depends on the semantics of the function name whether it is allowed to call a service without all input ports having a value and if it can return a result on only some output ports.

Finally, we define the traces that represents a particular run of a workflow.

Definition 7 (workflow trace). *A* workflow trace *is a list of events where an event is either*

- *a successful function call:* $f(t_1)^{t_2}$ *where* $(t_1, t_2) \in [\![f]\!]$*,*
- *a failed function call:* $f(t_1)^{\perp}$ *where* $(t_1, t_2) \in [\![f]\!]$ *for some* t_2*,*
- *an input event:* $\mathbf{in}_a(v)$ *with* $a \in \mathcal{P}$ *and* $v \in \mathcal{V}$*, or*
- *an output event:* $\mathbf{out}_a(v)$ *with* $a \in \mathcal{P}$ *and* $v \in \mathcal{V}$*,*

and for each port label a *there is at most one* $\mathbf{in}_a(v)$ *and* $\mathbf{out}_a(v)$ *in the list.*

We will use variants of the variables α, β and γ to denote traces. In function calls we will often omit the angular brackets and write $f(\langle a = 1, b = 3 \rangle)^{\langle a=4, b=5 \rangle}$ as $f(a = 1, b = 3)^{a=4,b=5}$.

Given a trace α we define the *input value of* α as the interface value $\mathbf{val_{in}}(\alpha) = \{(a, v) \mid \mathbf{in}_a(v) \in \alpha\}$. In a similar fashion the *output value of* α is defined as the interface value $\mathbf{val_{out}}(\alpha) = \{(a, v) \mid \mathbf{out}_a(v) \in \alpha\}$. For example, if $\alpha = [\mathbf{in}_a(1), \mathbf{in}_b(3), f(c = 3)^{a=4}, \mathbf{in}_c(4), \mathbf{out}_d(5), g(a = 1)^{\langle\rangle}, \mathbf{out}_e(6)]$ then $\mathbf{val_{in}}(\alpha) = \langle a = 1, b = 3, c = 4 \rangle$ and $\mathbf{val_{out}}(\alpha) = \langle d = 5, e = 6 \rangle$. The *function call content* or *body of* α is defined as $\mathbf{body}(\alpha) = [E \mid E \leftarrow \alpha, (\exists f, v, w : E = f(v)^w \vee E = f(v)^{\perp})]$, i.e., the sublist of α that contains all successful and failed function calls. For example, if α is as before then $\mathbf{body}(\alpha) = [f(c = 3)^{a=4}, g(a = 1)^{\langle\rangle}]$.

We define for two traces α and β the interleaving set, denoted as $(\alpha \mid\mid\mid \beta)$, such that $(\alpha \mid\mid\mid \beta) = \{\alpha_1 \cdot \beta_1 \cdot \ldots \cdot \alpha_n \cdot \beta_n \mid n \geq 1, \alpha = \alpha_1 \cdot \ldots \cdot \alpha_n, \beta =$

$\beta_1 \cdot \ldots \cdot \beta_n\}$. Note that this indeed defines all possible interleavings of the events in α and β since α_i and β_j can be the empty trace. For example, the interleaving set of $\alpha = [\mathbf{in}_a(3), \mathbf{out}_b(9)]$ and $\beta = [\mathbf{in}_c(2), \mathbf{out}_d(3)])$ contains $[\mathbf{in}_c(2), \mathbf{in}_a(3), \mathbf{out}_b(9), \mathbf{out}_d(3)]$ since $\alpha = [\,] \cdot [\mathbf{in}_a(3), \mathbf{out}_b(9)]$ and $\beta = [\mathbf{in}_c(2)] \cdot [\mathbf{out}_d(3)]$. We generalize the interleaving set for more than two traces, denoted as $(\alpha_1 \,|||\, \ldots \,|||\, \alpha_n)$, such that $\beta \in (\alpha_1 \,|||\, \ldots \,|||\, \alpha_n)$ iff there is a $\gamma \in (\alpha_2 \,|||\, \ldots \,|||\, \alpha_n)$ and $\beta \in (\alpha_1 \,|||\, \gamma)$. For $n = 1$ we define $(\alpha_1 \,|||\, \ldots \,|||\, \alpha_n)$ as $\{\alpha_1\}$, and for $n = 0$ as $\{[\,]\}$, i.e., the singleton set containing the empty trace.

3.2 Semantical inference rules

The semantics are defined by inference rules for deriving propositions of the form $P \Downarrow \alpha$ where P is a workflow expression and α a valid trace of P. A valid trace gives a possible sequence of events that might occur between the moment the workflow becomes scheduled and the moment it stops. This not only includes the successful runs, but also the unsuccessful ones, and the ones where it received a few input values but not enough to be enabled.

The presented inference rules follow the recursion of the definition of valid expression and in that sense define the semantics in a structural way, i.e., the traces of an expression are defined in terms of traces of subexpressions.

Function calls The semantics of a function call is defined by the following three rules:

$$\frac{(t_1, t_2) \in [\![f]\!] \qquad t_1 = \langle a_1 = v_1, \ldots, a_n = v_n \rangle \qquad t_2 = \langle b_1 = w_1, \ldots, b_m = w_m \rangle}{f \Downarrow [\mathbf{in}_{a_1}(v_1), \ldots, \mathbf{in}_{a_n}(v_n), f(t_1)^{t_2}, \mathbf{out}_{b_1}(w_1), \ldots, \mathbf{out}_{b_m}(w_m)]}$$

$$\frac{t_1 = \langle a_1 = v_1, \ldots, a_n = v_n \rangle \qquad t_1 \in \mathbf{dom}([\![f]\!])}{f \Downarrow [\mathbf{in}_{a_1}(v_1), \ldots, \mathbf{in}_{a_n}(v_n), f(t_1)^{\perp}]}$$

$$\frac{t_1 = \langle a_1 = v_1, \ldots, a_n = v_n \rangle \qquad t_1 \notin \mathbf{dom}([\![f]\!])}{f \Downarrow [\mathbf{in}_{a_1}(v_1), \ldots, \mathbf{in}_{a_n}(v_n)]}$$

The first rule defines the run of a successful function call. First the arguments are read in an arbitrary order, then, if the input events define a tuple in the domain of the function, the function is called, and finally the resulting interface value is returned by output events. The second rule defines the run of a failed function call. It starts like a successful one, but after the function call fails it stops and does not return any output. The final rule defines the trace for the case where not enough ports receive an input value to make the function call. Whether this is the case is determined by the domain of $[\![f]\!]$.

Note that the actual function call and the production of its result on the output ports are distinct events and there may therefore be other events that happen between them. So if two function calls for f and g happen in parallel it can happen that the function call event of f is first but that g produces results on its ports earlier.

List wrapping The semantics of the list wrapping workflow is defined by the following rules:

$$\frac{}{\mathbf{wrap}_{a\to b} \Downarrow [\mathbf{in}_a(v), \mathbf{out}_b([v])]} \qquad \frac{}{\mathbf{wrap}_{a\to b} \Downarrow [\,]}$$

The first rule defines a successful run where the input value from port a is consumed, wrapped in a list and produced on output port b. Note that contrary to function calls there is no function call event in the trace and only input and output behavior is shown. The second rule defines the empty trace as a possible trace, to cover the case where no input value is present on the input port a.

List concatenation The semantics of the list concatenation workflow is defined by the following rules:

$$\frac{w = v_1 \cdot v_2 \qquad \gamma \in ([\mathbf{in}_a(v_1)] \,|||\, [\mathbf{in}_b(v_2)])}{\mathbf{conc}_{a;b\to c} \Downarrow \gamma \cdot [\mathbf{out}_c(w)]} \qquad \frac{d \in \{a, b\}}{\mathbf{conc}_{a;b\to c} \Downarrow [\mathbf{in}_d(v)]}$$

$$\frac{}{\mathbf{conc}_{a;b\to c} \Downarrow [\,]}$$

The first rule derives successful runs where first both input ports a and b are read in arbitrary order and then the result of the concatenation is produced on c. The second rule defines the traces for the case where only one input port contains a value, and here no output is produced. Finally, the last rule defines the trace for the case where no input port contains a value.

List flattening The semantics of the list flattening workflow is defined by the following rules:

$$\frac{w = [v_1, \ldots, v_n] \qquad w' = v_1 \cdot \ldots \cdot v_n}{\mathbf{flat}_{a\to b} \Downarrow [\mathbf{in}_a(w), \mathbf{out}_b(w')]} \qquad \frac{}{\mathbf{flat}_{a\to b} \Downarrow [\,]}$$

The first rule defines the trace where a list of lists is read from the input port a and the flattened result is produced on output port b. The second rule again defines the trace for when no input is supplied.

Composition For the composition operation the semantics is defined by

$$\frac{\begin{array}{c} P \Downarrow \alpha \qquad Q \Downarrow \beta \\ \{b_1, \ldots, b_m\} = \mathbf{dom}(\mathbf{val_{out}}(\alpha)) \cap I = \mathbf{dom}(\mathbf{val_{in}}(\beta)) \cap I \\ \alpha = \alpha_1 \cdot [\mathbf{out}_{b_1}(v_1)] \cdot \ldots \cdot \alpha_m \cdot [\mathbf{out}_{b_m}(v_m)] \cdot \alpha_{m+1} \\ \beta = \beta_1 \cdot [\mathbf{in}_{b_1}(v_1)] \cdot \ldots \cdot \beta_m \cdot [\mathbf{in}_{b_m}(v_m)] \cdot \beta_{m+1} \qquad \{\gamma_i \in (\alpha_i \,|||\, \beta_i)\}_{i=1..m+1} \end{array}}{(P \triangleright_I Q) \Downarrow \gamma_1 \cdot \ldots \cdot \gamma_{m+1}}$$

Recall that the intended meaning of $P \triangleright_I Q$ with $I = \{a_1, \ldots, a_n\}$ is that for each port name $a_i \in I$ the output port of P with that name is connected with the input port of Q with that name. We explain the rule with an example where $P : \langle a : \mathsf{s} \rangle \Rightarrow \langle a : \mathsf{s}, b : \mathsf{s}, c : \mathsf{s}, d : \mathsf{s} \rangle$ and $Q : \langle b : \mathsf{s}, c : \mathsf{s}, d : \mathsf{s}, e : \mathsf{s} \rangle \Rightarrow \langle d : \mathsf{s} \rangle$. We consider now the expression $P \triangleright_{b,c,d} Q$.

The first line of the rule says that α and β must be traces of P and Q, respectively. Let us assume that
$\alpha = [\mathbf{in}_a(3), f(t_1)^{t_2}, \underline{\mathbf{out}_b(2)}, \mathbf{in}_b(8), g(t_3)^{t_4}, \underline{\mathbf{out}_c(4)}, h(t_5)^{t_6}, \mathbf{out}_a(3), f(t_7)^{t_8}]$
$\beta = [\mathbf{in}_e(4), g(t_4)^{t_3}, \underline{\mathbf{in}_b(2)}, \underline{\mathbf{in}_c(4)}, g(t_6)^{t_1}, \mathbf{out}_d(7)]$.

The second line of the rule says that the subset of I for which there are output events in α is the same as the subset of I for which there are input events in β. In this case this set is in both cases $\{b, c\}$ and the concerning events, which we will call here the synchronization events, are underlined in the traces. The third line and the beginning of the fourth line require that these events appear in both traces in the same order, in this case first b then c, and with the same port value, in this case 2 for port b and 4 for port c. Moreover, the traces are split into parts according to these events:
$\alpha_1 = [\mathbf{in}_a(3), f(t_1)^{t_2}], \alpha_2 = [\mathbf{in}_b(8), g(t_3)^{t_4}], \alpha_3 = [h(t_5)^{t_6}, \mathbf{out}_a(3), f(t_7)^{t_8}]$
$\beta_1 = [\mathbf{in}_e(4), g(t_4)^{t_3}], \beta_2 = [\,], \beta_3 = [g(t_6)^{t_1}, \mathbf{out}_d(7)]$.

Finally the corresponding parts are interleaved and concatenated, i.e., we construct the traces in the language $(\alpha_1 \mid\mid\mid \beta_1) \cdot \ldots \cdot (\alpha_3 \mid\mid\mid \beta_3)$. This can also be described as interleaving α and β while synchronizing on the synchronization events and removing them. For example, in the given example a possible trace for $P \triangleright_{b,c,d} Q$ is $\gamma_1 \cdot \gamma_2 \cdot \gamma_3$ where $\gamma_1 = [\mathbf{in}_a(3), \mathbf{in}_e(4), f(t_1)^{t_2}, g(t_4)^{t_3}]$, $\gamma_2 = [\mathbf{in}_b(8), g(t_3)^{t_4}]$ and $\gamma_3 = [g(t_6)^{t_1}, h(t_5)^{t_6}, \mathbf{out}_a(3), \mathbf{out}_d(7), f(t_7)^{t_8}]$.

Note that in the previous result trace the events of α and β occur in each interleaved part. It is not the case that in $P \triangleright_I Q$ first P executes, and when it is finished, then Q starts executing. Rather, P and Q execute in parallel but the parts of Q that need input on the ports in I will wait until this input is produced by P. In the extreme case where $I = \emptyset$ the workflows run completely in parallel. Only if I contains all the output ports of P and input ports of Q and Q cannot start any activity until it has received an input on all its ports, then the composition is truly sequential. So the composition operation is a generalization of both the sequential composition and the parallel composition as is usually found in process algebra[12].

Linking The semantics of the linking workflow is defined by:

$$\frac{\{c_1, \ldots, c_n\} = I}{\mathbf{ln}_{a \to I} \Downarrow [\mathbf{in}_a(v), \mathbf{out}_{c_1}(v), \ldots, \mathbf{out}_{c_n}(v)]} \qquad \frac{}{\mathbf{ln}_{a \to I} \Downarrow [\,]}$$

The first rule states that there is a valid trace in which the input port a is read and the result is written to the output ports in I in arbitrary order. The second rule deals with the case where no input value is supplied.

Select-first The semantics of the select-first workflow is defined by:

$$\frac{\{a_1,\ldots,a_m\} \subseteq I \qquad m > 0}{\mathbf{first}_{I\to b} \Downarrow [\mathbf{in}_{a_1}(v_1), \mathbf{out}_b(v_1), \mathbf{in}_{a_2}(v_2), \ldots, \mathbf{in}_{a_m}(v_m)]} \qquad \frac{}{\mathbf{first}_{I\to b} \Downarrow [\,]}$$

The first rule describes a run where at least one of the input ports has an input value. As soon as the first input value is read, it is copied to the output port, and then the other input values are read. Note that not all input ports need to have an input value for this. The second rule again describes the case where there are no input values at all.

Nesting The semantics of the nesting constructor is defined by:

$$\frac{\begin{array}{c} P \Downarrow \alpha \qquad I = \{a_1,\ldots,a_n\} \qquad J = \{b_1,\ldots,b_m\} \\ \gamma_1 = [\mathbf{in}_{a_1}(v_1),\ldots,\mathbf{in}_{a_n}(v_n)] \qquad \mathbf{val}_{\mathbf{in}}(\gamma_1) = \mathbf{val}_{\mathbf{in}}(\alpha) \\ \gamma_2 = [\mathbf{out}_{b_1}(w_1),\ldots,\mathbf{out}_{b_m}(w_m)] \qquad \mathbf{val}_{\mathbf{out}}(\gamma_2) = \mathbf{val}_{\mathbf{out}}(\alpha) \end{array}}{\mathbf{nest}^J_I(P) \Downarrow \gamma_1 \cdot \mathbf{body}(\alpha) \cdot \gamma_2}$$

$$\frac{\begin{array}{c} P \Downarrow \alpha \qquad I = \{a_1,\ldots,a_n\} \qquad \gamma_1 = [\mathbf{in}_{a_1}(v_1),\ldots,\mathbf{in}_{a_n}(v_n)] \\ \mathbf{val}_{\mathbf{in}}(\gamma_1) = \mathbf{val}_{\mathbf{in}}(\alpha) \qquad \mathbf{dom}(\mathbf{val}_{\mathbf{out}}(\alpha)) \subsetneq J \end{array}}{\mathbf{nest}^J_I(P) \Downarrow \gamma_1 \cdot \mathbf{body}(\alpha)}$$

$$\frac{\begin{array}{c} P \Downarrow \alpha \\ \gamma_1 = [\mathbf{in}_{c_1}(v_1),\ldots,\mathbf{in}_{c_n}(v_n)] \qquad \mathbf{val}_{\mathbf{in}}(\gamma_1) = \mathbf{val}_{\mathbf{in}}(\alpha) \qquad \{c_1,\ldots,c_n\} \subsetneq I \end{array}}{\mathbf{nest}^J_I(P) \Downarrow \gamma_1}$$

The first rule describes the case where on all input ports input values are available and on all output ports a value is produced. It does this by taking a trace α of P for which this holds. It then takes all input and output events out of the trace, and places them in arbitrary order respectively at the beginning and at the end of the trace. The resulting trace is then a valid trace of the total workflow. The second rule describes the case where there are sufficient input values, but P does not produce a value on all its output ports. In that case the resulting trace produces no output events at all. The final rules describes the case where not all input ports have a value available. In that case the input values are consumed, but P is not executed.

The nesting operation can also express synchronization. For $I = \{a_1,\ldots,a_n\}$ with $n > 0$ we define the workflow $\mathbf{sync}_I = \mathbf{nest}^I_I(\mathbf{ln}_{a_1\to a_1} \triangleright_\emptyset \ldots \triangleright_\emptyset \mathbf{ln}_{a_n\to a_n})$ which waits until all input ports have an input value, and then copies these values to the synonymous output ports. The semantics of nesting is not exactly the same as synchronizing the input and output, i.e, for P of type $\iota \Rightarrow \kappa$ with $\mathbf{dom}(\iota) = I$ and $\mathbf{dom}(\kappa) = J$ it is not always true that $\mathbf{nest}^J_I(P) \equiv \mathbf{sync}_I \triangleright_I P \triangleright_J \mathbf{sync}_J$. For example if $P : \langle a : \mathsf{s}\rangle \Rightarrow \langle b : \mathsf{s}\rangle$ has a valid trace $\alpha = [\mathbf{in}_a(1), \mathbf{out}_b(2), f(a = 2)^{c=3}]$ then this is not a valid trace of $\mathbf{nest}^b_a(P)$ but it is a valid trace of $\mathbf{sync}_a \triangleright_a P \triangleright_b \mathbf{sync}_b$.

Iteration The semantics of the iteration constructor is defined by:

$$\frac{\begin{array}{c} I = \{a_1, \ldots, a_n\} \qquad \gamma_1 \in ([\mathbf{in}_{a_1}(v_1), \ldots, \mathbf{in}_{a_n}(v_n)] \,|||\, [\mathbf{in}_{c_1}(w_1), \ldots, \mathbf{in}_{c_p}(w_p)]) \\ \{v_j = [v_{j,1}, \ldots, v_{j,r_j}]\}_{j=1..n} \qquad q = \min(r_1, \ldots, r_n) > 0 \qquad \{P \Downarrow \alpha_i\}_{i=1..q} \\ \{\langle a_1 = v_{1,i}, \ldots, a_n = v_{n,i}, c_1 = w_1, \ldots, c_p = w_p\rangle = \mathbf{val_{in}}(\alpha_i)\}_{i=1..q} \\ \gamma_2 \in (\mathbf{body}(\alpha_1) \,|||\, \ldots \,|||\, \mathbf{body}(\alpha_q)) \qquad \{d_1, \ldots, d_s\} = \cap_{i=1..q}\mathbf{dom}(\mathbf{val_{out}}(\alpha_i)) \\ \{\langle d_1 = x_{1,i}, \ldots, d_s = x_{s,i}\rangle \subseteq \mathbf{val_{out}}(\alpha_i)\}_{i=1..q} \\ \{x_k = [x_{k,1}, \ldots, x_{k,m}]\}_{k=1..s} \qquad \gamma_3 = [\mathbf{out}_{d_1}(x_1), \ldots, \mathbf{out}_{d_s}(x_s)] \end{array}}{\odot_I^J(P) \Downarrow \gamma_1 \cdot \gamma_2 \cdot \gamma_3}$$

$$\frac{\begin{array}{c} I = \{a_1, \ldots, a_n\} \qquad J = \{b_1, \ldots, b_m\} \\ \gamma_1 \in ([\mathbf{in}_{a_1}(v_1), \ldots, \mathbf{in}_{a_n}(v_n)] \,|||\, [\mathbf{in}_{c_1}(w_1), \ldots, \mathbf{in}_{c_p}(w_p)]) \\ \min(|v_1|, \ldots, |v_n|) = 0 \qquad \gamma_3 = [\mathbf{out}_{b_1}([\,]), \ldots, \mathbf{out}_{b_m}([\,])] \end{array}}{\odot_I^J(P) \Downarrow \gamma_1 \cdot \gamma_3}$$

$$\frac{I \not\subseteq \{c_1, \ldots, c_p\}}{\odot_I^J(P) \Downarrow [\mathbf{in}_{c_1}(w_1), \ldots, \mathbf{in}_{c_p}(w_p)]}$$

The three rules deals with the cases of an iteration with at least one iteration step, an iteration with zero iteration steps, and a failed iteration where not all lists over which we want to iterate are available. We explain each of the rules in more detail below.

In the first rule the condition for γ_1 established that the trace begins with reading a value from each iteration port in I plus perhaps some extra ports. The condition $\{v_j = [v_{j,1}, \ldots, v_{j,r_j}]\}_{j=1..n}$ then ensures that all values on the iteration ports are lists and that the length of the list on port a_j is r_j. Note that $v_{j,i}$ denotes the ith element of the list on port a_j. Then in $q = \min(r_1, \ldots, r_n) > 0$ the minimum length of these lists is determined and required to be non-zero. This will be the number of iteration steps in this run. Since we simultaneously iterate over these lists, we can indeed iterate only as often as the length of the shortest list. The condition $\{P \Downarrow \alpha_i\}_{i=1..q}$ defines the separate traces for each iteration step. The condition $\{\langle a_1 = v_{1,i}, \ldots, a_n = v_{n,i}, c_1 = w_1, \ldots, c_p = w_p\rangle = \mathbf{val_{in}}(\alpha_i)\}_{i=1..q}$ ensures that each of these traces has the right interface input value, i.e., the value on each port a_j is the ith element of the list read on this port in the global trace, and each port c_j contains the same value as read on this port in the global trace. The condition $\gamma_2 \in (\mathbf{body}(\alpha_1) \,|||\, \ldots \,|||\, \mathbf{body}(\alpha_q))$ states that the body of the global trace is equal to an interleaving of the bodies of the traces of each iteration step, i.e., all iteration steps are executed in parallel. Then $\{d_1, \ldots, d_s\} = \cap_{i=1..q}\mathbf{dom}(\mathbf{val_{out}}(\alpha_i))$ determines the set of output ports for which each iteration step has produced an output, and $\{\langle d_1 = x_{1,i}, \ldots, d_s = x_{s,i}\rangle \subseteq \mathbf{val_{out}}(\alpha_i)\}_{i=1..q}$ computes for each iteration step the output interface value restricted to these ports. This is done because we will only generate an output value for the ports for which each iteration step returned a value. Note that $x_{j,i}$ denotes the output of port d_j after the ith iteration step. Then in $\{x_k = [x_{k,1}, \ldots, x_{k,m}]\}_{k=1..s}$ the output value x_k for port d_k is determined by

enumerating the results of each iteration step on port d_k. Finally, the condition for γ_3 translates the output value to a trace with output events.

The second rule for $\odot_I^J(P)$ begins with a similar condition for γ_1 as the first rule, but the condition $\min(|v_1|, \ldots, |v_n|) = 0$ establishes that there are no elements to iterate over, and so the global trace immediately finishes with output events that produce an empty list on all output ports.

The third rule defines the traces that describe the case where not all iteration ports in I have a value, so there can be no iteration and the trace stops immediately after reading the input ports.

The semantics of the iteration constructor is similar to the nesting constructor in several ways: it also synchronizes the input events and the output events, and it does not produce an output until the nested workflows are completely finished. It is however different in that it does not necessarily require for a successful run that all input ports are supplied with a value, except for the ports in I, and it may produce an output on only some of the output ports.

4 Representing Taverna workflows in the calculus

In the section we discuss informally how to describe the semantics of Taverna by mapping Scufl graphs to calculus expressions.

4.1 Scufl graphs

We briefly sketch here the core features of Scufl. Workflows are expressed in this language by an annotated *Scufl graph* such as is shown in Figure 5.

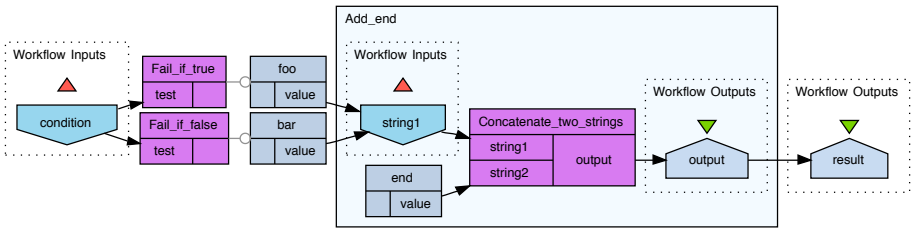


Fig. 5. A Scufl workflow graph with conditional branching and nesting

A Scufl graph contains zero or more *workflow inputs* and *workflow outputs* that indicate the input ports and output ports of the workflow. In the Scufl graph in Figure 5 there is a workflow input condition and a workflow output result. The components of a Scufl graph are *basic processors* such as Fail_if_true and foo, and *nested processors* such as Add_end whose behavior is specified by a nested Scufl graph. A processor can have zero or more input and output ports.

For example, Fail_if_true has an input port test and no output ports, and Concatentate_two_string has two input ports, string1 and string2, and one output port output. The ports of a nested processor are defined by the workflow inputs and outputs of the associated Scufl graph, so Add_en has input port string1 and output port output. Each processor waits until all its input ports have received a value, then executes and either returns a value on all its output ports or fails. This also holds for nested processors, and so, if the nested workflow does not produce a value on all its workflow outputs, the nested processor fails and produces no output values at all.

Basic processors can represent calls to external web services or local functions and their input and output interface type is specified. In this case the basic processors Fail_if_true and Fail_if_false either fail or succeed depending on the value read on port test. The basic processors foo, bar and end always succeed and produce their name as a string on their output ports. The basic processor Concatenate_two_strings concatenates the strings it receives on its input ports.

The links in Scufl graphs are either *data links*, which connect workflow inputs and processor output ports with workflow outputs and processor input ports, or *control links*, which connect a processor with another processor. The data links are indicated by arrows with a solid head, and their meaning is that the value produced on its source port is copied to its destination port. The control links are indicated by a gray edge ending with a circle, and their meaning is that the processor at the end of the control link cannot start execution unless the processor at the beginning of the link has finished execution without failure. These edges, when all seen as edges from processors to processors, must define an acyclic graph, and all input ports and workflow outputs must have at least one incoming data link. In Taverna it is possible to specify a default value for an input port, in which case no incoming data link is required, but since default values can be straightforwardly simulated in the presented fragment we will not consider them here.

In the example in Figure 5 it is clear that although port string1 of Add_end as two incoming data links, only one value will be copied to it. In order to deal with the case where there are multiple incoming data links and therefore multiple values are copied to the same input port or workflow output an *incoming links strategy* must be specified for each input port and workflow output, which is either the *select-first strategy*, which says that the first value that arrives is kept and the rest ignored, or the *merge strategy*, which waits until through each incoming data link a value has arrived, and collects these as elements into a single list. Although this is not possible in Taverna, we will assume that the merge strategy also indicates the order over the incoming data links that determines the order of the values in the resulting list. In Taverna the order is determined by the order in which the values arrive. Although it is not hard to define a special merge operator with this semantics, our version seems equally valid and can be described with the given operators. Note that if the merge strategy is specified for the string1 port of Add_end then this processor would never execute.

If a received value is more deeply nested then the expected type, then it will solve this by applying the so-called *implicit iteration strategy*. This means that if a processor expects values of a certain type τ on an input port a but is offered a value of type $[\tau]$, then it deals with these values by iterating over each element. More precisely, if its normal behavior is described by the calculus expression P then it will behave in that case as workflow $\odot_a(P)$. If the processor has two or more ports then it can be indicated with an *iteration strategy* how these lists will be combined for the iteration. The strategy is specified by an expression such as $(a \otimes b) \odot c$, which is a combination of the input ports of the processor, each appearing exactly once, and two binary operators, the dot product $\odot$ which indicates simultaneous iteration, and the cross product $\otimes$ which indicates iteration over all possible combinations.

4.2 Mapping Scufl graphs

As was indicated in the discussion of iteration strategies, the semantics of a Scufl graph and therefore the calculus expression to which it is mapped depends on the type of the offered input values. In Scufl, ports of different port types can be connected by data links, and this is solved by adapting the semantics of the receiving processor. Very loosely speaking this adaption can be described by saying that if the received value is nested too deeply then the processor will iterate according to the iteration strategy and if the value is nested not enough then it is wrapped in a singleton list.

A crucial notion for the mapping is that of the *expected input type* of the ports of a processor. For a basic processor these are the port types in the specified input interface type. For a nested processor this is determined by determining for each workflow input the maximum of the expected types of the input ports to which it is connected in the nested workflow. So, if a workflow input a is connected in the nested workflow to port b with expected type $[\mathsf{s}]$ and c with expected type s, then the expected type of a is $[\mathsf{s}]$.

For the mapping of the merge strategy we define a macro $\mathbf{collect}_{a_1;\ldots;a_n \to b}$ with induction on n such that $\mathbf{collect}_{a \to b} = \mathbf{wrap}_{a \to b}$ and $\mathbf{collect}_{a_1;a_2;\ldots;a_n \to b} = \mathbf{wrap}_{a_1 \to a_1} \triangleright_\emptyset \mathbf{collect}_{a_2;\ldots;a_n \to a_2} \triangleright_{a_1,a_2} \mathbf{conc}_{a_1;a_2 \to b}$. Now consider the Scufl graph in Figure 6 and assume that the incoming links strategy for workflow output b is the merge strategy that places the output of R before that of S. If no iteration or coercion is necessary then this can be mapped to $\mathbf{ln}_{a \to c,e} \triangleright_c P \triangleright_e Q \triangleright_d \mathbf{ln}_{d \to h,k} \triangleright_g \mathbf{ln}_{g \to l} \triangleright_f \mathbf{ln}_{f \to i} \triangleright_{h,i} R \triangleright_{k,l} S \triangleright_{j,m} \mathbf{collect}_{j;m \to b}$ where S is mapped to $\mathbf{nest}^m_{k,l}(T)$.

There can be two reasons to insert operations to coerce values to a greater, i.e., more deeply nested, type. The first is that a processor receives a value that is nested less deeply then the expected type. For example, $\mathsf{P.d}$ has type s but $\mathsf{R}.h$ expects $[\mathsf{s}]$, then R is mapped not to R but to $\mathbf{wrap}_{h \to h} \triangleright_h R$. The second reason is that the operator for the incoming links strategy does not receive values of the same type. In that case wrapping operations are inserted to coerce all the incoming values to the same type. For example, if $\mathsf{R.j}$ has type s and $\mathsf{T.m}$ has type $[\mathsf{s}]$ then the incoming link strategy is not mapped to $\mathbf{collect}_{j;m \to b}$ but to $\mathbf{wrap}_{j \to j} \triangleright_j \mathbf{collect}_{j;m \to b}$. In the current implementation of Taverna, this coercion

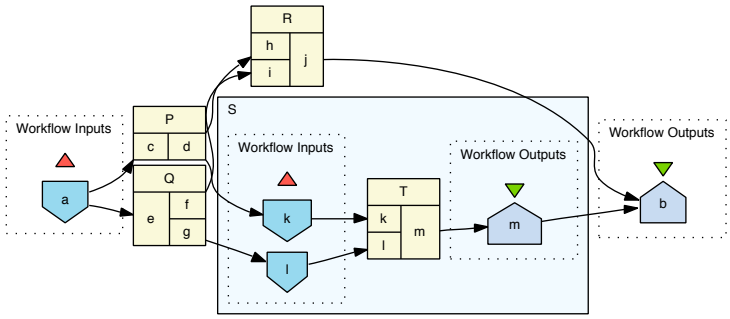


Fig. 6. A Scufl workflow graph

is not done, which in the case of the merge strategy can lead to heterogeneous lists such as $[1,[2]]$, which strictly speaking are not legal port values. For the select-first strategy there is no such risk, but in a strictly typed calculus as ours it is necessary.

For the translation of an iteration strategy s we define the macro $\mathbf{is}_s(P)$ such that $\mathbf{is}_a(P) = \odot_a(P)$, $\mathbf{is}_{s_1 \odot s_2} = \mathbf{is}_{s_1}(\mathbf{id}_{\Sigma(s_1)}) \triangleright_\emptyset \mathbf{is}_{s_2}(\mathbf{id}_{\Sigma(s_2)}) \triangleright_{\Sigma(s_1 \odot s_2)} \odot_{\Sigma(s_1 \odot s_2)}(P)$ and $\mathbf{is}_{s_1 \otimes s_2} = \mathbf{is}_{s_1}(\mathbf{is}_{s_2}(P))$ where $\Sigma(is)$ is the set of port names used in is and $\mathbf{id}_{a_1,\ldots,a_n} = \mathbf{ln}_{a_1 \to a_1} \triangleright_\emptyset \ldots \triangleright_\emptyset \mathbf{ln}_{a_n \to a_n}$. Then, if processor R with iteration strategy $\mathsf{h} \otimes \mathsf{i}$ expects on both h and i values of type s but receives values of type [s], it is mapped to $\mathbf{nest}_{h,i}(\mathbf{is}_{h \otimes i}(P)) = \mathbf{nest}_{h,i}(\odot_h(\odot_i(P)))$. Unfortunately the mapping is more difficult and probably not possible for some other combinations of expected and received types. For example, if the product strategy of R is $\mathsf{h} \odot \mathsf{i}$ and the received values on h and i are $[[1,2],[3]]$ and $[4,5,6,7]$ then in Taverna the processor iterates over the combinations $(1,4)$, $(2,5)$ and $(3,6)$, and if these result in the values 8, 9 and 10, then the final result is nested like the deepest input list, i.e., $[[8,9],10]$. This does not seem expressible in the current basic calculus and requires the addition of extra operators.

Finally we briefly discuss the mapping of control links. Their semantics depends on the notions of failed and successful execution. In Taverna a failed execution of a processor is in general defined as an execution which, after finishing, has not produced values on all output pots. The only exception is made for basic processors without output ports, such as Fail_if_true and Fail_if_false, which could otherwise not fail. For simplicity we will assume that such basic processors are simulated by a processor with a dummy output port, and so all basic processor will have at least one output port. Then, we can describe the semantics of the control links as a transformation of the Scufl graph as follows. Assume there is a control link from P to Q. Then P is nested in a nested processor P' with an extra freshly named output port a on which an additional basic processor with no input ports and one output port produces a dummy value. Moreover, Q is nested in a nested processor Q' with an extra freshly named input port b which is not connected to anything inside Q'. Finally a data link is added from port a

of P' to port b of Q'. Note that the calculus currently has no operator to describe the semantics of a processor that produces a dummy value, but it is not hard to see how it could be added.

5 Conclusion

In this paper we have presented a calculus that can be used to describe the semantics of collection-oriented scientific workflows such as can be defined in systems like Taverna. The syntax and the semantics have been formally defined, with the semantics being defined in a structural way and in terms of traces that contain input and output events and function calls. The usability of the calculus is shown by discussing informally how the semantics of the Scufl workflow language can in principle be described in terms of the calculus.

References

1. University of Virginia: FASTA Sequence Comparison. (`http://wrpmg5c.bioch.virginia.edu/fasta_www2/fasta_list2.shtml`)
2. National Center for Biotechnology Information: NCBI Blast. (`http://www.ncbi.nlm.nih.gov/blast/Blast.cgi`)
3. Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M.R., Li, P., Oinn, T.: Taverna: a tool for building and running workflows of services. Nucl. Acids Res. **34** (2006) W729–732
4. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the kepler system: Research articles. Concurr. Comput. : Pract. Exper. **18** (2006) 1039–1065
5. Benson, D.A., Karsch-Mizrachi, I., Lipman, D.J., Ostell, J., Wheeler, D.L.: Genbank. Nucleic Acids Res **36** (2008)
6. Rice, P., Longden, I., Bleasby, A.: EMBOSS: the European Molecular Biology Open Software Suite. Trends in Genetics **16** (2000) 276–277
7. 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) 441–448
8. Liu, X., Lee, E.A.: CPO semantics of timed interactive actor networks. Technical Report UCB/EECS-2007-131, EECS Department, University of California, Berkeley (2007)
9. Lee, E.A., Sangiovanni-Vincentelli, A.: Comparing models of computation. In: ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design, Washington, DC, USA, IEEE Computer Society (1996) 234–241
10. Singh, M.P., Meredith, G., Tomlinson, C., Attie, P.C.: An event algebra for specifying and scheduling workflows. In: Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA), World Scientific Press (1995) 53–60
11. Heinlein, C.: Workflow and process synchronization with interaction expressions and graphs. In: Proceedings of the 17th International Conference on Data Engineering, Washington, DC, USA, IEEE Computer Society (2001) 243–252
12. Baeten, J.C.M.: A brief history of process algebra. Theor. Comput. Sci. **335** (2005) 131–146