Taverna Workflows: Syntax and Semantics

Daniele Turi, Paolo Missier, Carole Goble
School of Computer Science, University of Manchester, Manchester, UK
{pmissier,dturi,cgoble}@cs.manchester.ac.uk

David De Roure
School of Electronics and Computer Science, University of Southampton, UK
dder@ecs.soton.ac.uk

Tom Oinn
European Bioinformatics Institute, Cambridge, UK
tmo@ebi.ac.uk

Abstract

This paper presents the formal syntax and the operational semantics of Taverna, a workflow management system with a large user base among the e-Science community. Such formal foundation, which has so far been lacking, opens the way to the translation between Taverna workflows and other process models. In particular, the ability to automatically compile a simple domain-specific process description into Taverna facilitates its adoption by e-scientists who are not expert workflow developers. We demonstrate this potential through a practical use case.

1 Introduction

Past accounts of Taverna [OGAea06, SWAea07], a workbench for the definition and execution of scientific workflows, have been focusing mainly on its practical applications to e-Science. However, a formal foundation of the Taverna model has so far been lacking. In this paper we fill this gap by presenting both the formal syntax of Taverna, and its operational semantics (Sections 2 and 3).

With this work, Taverna joins the ranks of other scientific workflow management systems for which formal models have been developed, notably Kepler [LABe05, MBL06] based on Process Networks [KM77], and BPEL [OVvdA$^+$07], using Petri Nets. Some of the benefits of providing a formal syntax and semantics for a workflow language are well-known, i.e., to apply process analysis techniques [OVvdA$^+$07], and to enable unambiguous mappings between models. Mappings, in turn, make inter-model *dataflow repositories* a practical possibility. A recent proposal for such a repository [HKS$^+$07] uses Nested Relational Calculus (NRC) [BNTW95] to describe dataflows.

As a further research contribution, in the second part of this paper (Section 4) we illustrate a less-explored practical use of the workflow model, namely the formal description of workflows that result from the automated translation from a high-level, domain-specific process model. This is motivated by the need to provide users who are not expert workflow developers, with a simple way to specify a standard coordination among processors that they are familiar with.

Informally, a Taverna workflow consists of a collection of processors with data and control links among them. Processors may have multiple inputs and outputs; a data link establishes a dependency between the output of a processor and the input of another. A control link indicates that a processor can only begin its execution after some other processor has successfully completed its execution. Processors are implemented either as local Java classes, or as Web Services, with input and output *ports* that correspond to the operations defined in the service's WSDL interface. The workflow execution engine schedules the invocation of the service operations, making sure that the dependencies are not violated, and manages the flow of data among the processors. A simple workflow, used as a running example throughout the paper, is shown in Figure 1.

In this paper, the Taverna language is defined using the *computational lambda calculus* [Mog91]. The use of lambda calculus is motivated by the fact that Taverna workflow language can be defined in functional terms, although it uses web services as its building blocks. The use of functional languages to give formal meaning to workflows is not new; an example is the use of the Haskell functional pro-

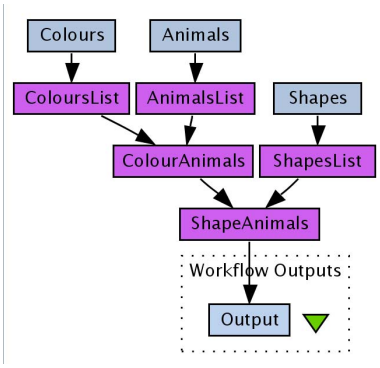

Figure 1. Workflow Diagram

gramming language to give a formal definition of a particular workflow for the Ptolomey II system [LA03] (a precursor to Kepler). The computational lambda calculus is obtained by augmenting the lambda calculus with suitable *monads* [Wad90] to model real-life behaviour of functional programs. The list operator, mapping a set A to the set $L(A)$ of all lists formed with elements of the set A, is one such monad and Taverna *is* the corresponding computational lambda calculus. This is a striking result, especially since Taverna was not designed with the computational lambda calculus in mind. Moreover, even relatively low level implementation details, such as the way Taverna deals with data cardinality mismatches, are accounted for by the theory.

2 Formal Syntax

2.1 Types

Taverna has base types like $\mathbf{s}$, the set of strings; without loss of generality, we will consider $\mathbf{s}$ to be the only base type. One can construct arbitrarily nested lists starting from the base types, ie $L(\mathbf{s})$, $L^2(\mathbf{s})$, etc.[1] Taverna also allows for multiple inputs and outputs, hence products have also to be included. For instance: $\mathbf{s}\times\mathbf{s}$, $\mathbf{s}\times L^3(\mathbf{s})$, $L^2(\mathbf{s})\times\mathbf{s}\times\mathbf{s}\times L(\mathbf{s})$. Finally, we also need the 0-ary product type 1 for the special case of workflows with no output. Formally:

$$\tau ::= \mathbf{s} \mid L(\tau) \mid \tau \times \tau \mid 1$$

We use σ and τ to range over types.

In the example, ShapesList has two string inputs, `string` and `regex`, and one output `split`, a list of strings. Since data in Taverna is XML-formatted, the types $\mathbf{s}$ and $L(\mathbf{s})$ are represented in the implementation using mime types, i.e., `'text/plain'` and `l('text/plain')`, respectively. Given this simple type system, type mismatches reduce to list nesting cardinality mismatches. As we will see, some of these mismatches are dealt implicitly through iterations and wrapping.

[1] We write L^n for n applications of L.

2.2 Language

Contexts. A context Γ is a list of (typed) inputs:

$$\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$$

where $x_1, \ldots, x_n$ are input variables of type $\sigma_1, \ldots, \sigma_n$. Given context Γ above, we write $\Gamma, x : \sigma$ to denote the context $x_1 : \sigma_1, \ldots, x_n : \sigma_n, x : \sigma$. Note that contexts can be empty, i.e., n can be 0. We write $\mathit{Type}(x_i) = \sigma_i$ to denote the type of variable x_i.

For example, here is a context consisting of two input variables $genes$ and url, with $\mathit{Type}(genes) = L(\mathbf{s})$ and $\mathit{Type}(url) = \mathbf{s}$:

$$genes : L(\mathbf{s}), url : \mathbf{s}$$

Workflows and Processors. We represent workflows with inputs Γ and output of type τ as *sequents* of the form

$$\Gamma \vdash P : \tau \tag{1}$$

We extend the function Type from variables (i.e., workflow inputs) to workflow outputs. Thus, in (1), $\mathit{Type}(P) = \tau$. A workflow consists of a collection of processors with linked inputs and outputs. The remainder of this section formalises the linking process using a *sequent calculus*, capturing the order in which the linking is done. The language accounts for the linking of processors with mismatching cardinalities, as in the example shown in §1.

Processors are *axioms* of the form

$$\Gamma \vdash \mathsf{p} : \tau$$

A processor is a special case of a workflow. (Note the difference between a workflow variable P and a processor constant p.) We use product types for multiple outputs. For instance, the KEGG service *get enzymes by pathways* expects a pathway id as input and returns a list of enzyme ids as output. The sequent calculus notation for the corresponding Taverna processor is:

$$\mathit{pathwayId} : \mathbf{s} \vdash \mathsf{get\_enzymes\_by\_pathways} : L(\mathbf{s}) \tag{2}$$

In practice, each processor has a unique identifier, eg its WSDL address, instead of just get_enzymes_by_pathways. Also note that:

$$\mathit{Type}(\mathsf{get\_enzymes\_by\_pathways}) = L(\mathbf{s})$$

Taverna has a *String constant* predefined service that is used to provide predefined inputs to other processors. This gives a processor for each possible string. We denote these

processors using the quoted string itself. Thus, for instance, the **constant processor** for the string "foo" is:

$$\vdash \text{“foo”} : \mathbf{s} \tag{3}$$

Our example workflow in Figure 1 has three constant processors, namely:

$\vdash$ “red, green” : $\mathbf{s}$
$\vdash$ “cat, rabbit” : $\mathbf{s}$
$\vdash$ “square, circular, triangular” : $\mathbf{s}$

The remaining five processors in the workflow are:

$x_1 : \mathbf{s} \vdash \mathsf{ColoursList} : L(\mathbf{s})$
$x_2 : \mathbf{s} \vdash \mathsf{AnimalsList} : L(\mathbf{s})$
$x_3 : \mathbf{s} \vdash \mathsf{ShapesList} : L(\mathbf{s})$
$x_4 : \mathbf{s}, x_5 : \mathbf{s} \vdash \mathsf{ColourAnimals} : \mathbf{s}$
$x_6 : \mathbf{s}, x_7 : \mathbf{s} \vdash \mathsf{ShapeAnimals} : \mathbf{s}$

Here is an example of a processor with no input and two outputs (a list of strings and a string):

$$\vdash \mathsf{p}_2 : L(\mathbf{s}) \times \mathbf{s} \tag{4}$$

Note the product type. Conversely, here is a processor with two inputs and no output (the absence of output is denoted by the type 1):

$$\mathit{genes} : L(\mathbf{s}), \mathit{url} : \mathbf{s} \vdash \mathsf{p}_3 : 1$$

2.2.1 Syntax rules

Workflows are built using processors in conjunction with the following rules.

Pairing. We have seen in (4) that a processor with more than one output has product type. One can also obtain a product type by pairing two workflows, which amounts to having no link between them.

$$\frac{\Gamma \vdash P : \sigma \qquad \Gamma \vdash Q : \tau}{\Gamma \vdash \langle P, Q \rangle : \sigma \times \tau} \tag{5}$$

For instance, the pairing of (3) with (2) yields:

$$\mathit{pathwayId} : \mathbf{s} \vdash \langle \text{“foo”}, \mathsf{get\_enzymes\_by\_pathways} \rangle : \mathbf{s} \times L(\mathbf{s}) \tag{6}$$

Projections. The dual to pairing is to project. This is needed in order to select one of multiple outputs.

$$\frac{\Gamma \vdash P : \sigma \times \tau}{\Gamma \vdash \mathtt{fst}(P) : \sigma} \qquad \frac{\Gamma \vdash P : \sigma \times \tau}{\Gamma \vdash \mathtt{snd}(P) : \tau} \tag{7}$$

For example if we apply the second projection to (6) we get a workflow with the same type as (2) and, following the operational rules in §3, we can show that they have the same behaviour. Note that n-ary products can be defined in terms of binary ones.

Simple Composition The basic rule of our calculus shows how to compose two workflows by linking one workflow's output to another workflow's input:

$$\frac{\Gamma \vdash P : \sigma \qquad \Gamma, x : \sigma \vdash Q : \tau}{\Gamma \vdash \mathtt{let}\ x \leftarrow P\ \mathtt{in}\ Q : \tau} \tag{8}$$

Note that the P and Q above are variables rather than concrete workflows. Hence the rule applies to any pair of workflows with matching types. The syntax: $\mathtt{let}\ x \leftarrow P\ \mathtt{in}\ Q$ stands for: "link the output of P to the input x of Q". One can use the projection rule to select the intended output when there P has more than one.

A simple example of this rule is the composition of (3) with (2):

$$\vdash \mathtt{let}\ \mathit{pathwayId} \leftarrow \text{“foo”}\ \mathtt{in}\ \mathsf{get\_enzymes\_by\_pathways} : L(\mathbf{s})$$

This links the output of (3) to the only input of (2).

The Taverna workbench also supports *nested workflows*, namely workflows that are reused inside another workflow. From the point of view of our formal language there is no difference with the workflow fragments we have used so far. It is just a naming convention.

Control link Control links denote that a processor (the controlled) cannot start execution before another processor (the controller) has terminated. This is type of sequential composition and can be easily simulated by adding to the controlled processor an extra new input of the same type as the output of the controller. Formally, sequential composition is syntactic sugar for a let on a fresh variable:

$$P;Q \ \equiv\ \mathtt{let}\ x \leftarrow P\ \mathtt{in}\ Q \tag{9}$$

where x does not occur in P nor in Q.

Taverna can deal with two dual types of cardinality mismatches on the data, namely, when a list $L(\tau)$ is provided to a processor that expects input of type τ, and viceversa, input of type τ is supplied instead of $L(\tau)$. The following two composition rules account for these mismatches.

Iterative Composition. This rule uses a combination of composition and list replication. The list replication operation takes an element a and a list $[b_1, \ldots, b_n]$ and maps them to the list of pairs $[\langle a, b_1 \rangle, \ldots, \langle a, b_n \rangle]$. Its type is thus $A \times L(B) \to L(A \times B)$, for every pair of sets A and B.

$$\frac{\Gamma \vdash P : L(\sigma) \qquad \Gamma, x : \sigma \vdash Q : \tau}{\Gamma \vdash \mathtt{let}\ x \leftarrow P\ \mathtt{in}\ Q : L(\tau)} \tag{10}$$

For instance, if

$$\vdash \mathsf{pathways} : L(\mathbf{s}) \tag{11}$$

is a processor giving a list of pathway ids, we can compose it with (2) using the above rule, obtaining:

$$\vdash \texttt{let } \mathit{pathwayId} \leftarrow \mathsf{pathways} \\ \texttt{in } \mathsf{get\_enzymes\_by\_pathways} : L^2(\mathbf{s}) \tag{12}$$

Note that the input of (2) is a string therefore there is a cardinality mismatch with the output of (11) which is a list: the rule ensures that the two can still be composed. The semantic rules presented in §3 define how that this is dealt with by iterations. Note that the output type of (2) was already a list of strings, so the type of the composition is $L^2(\mathbf{s})$, a list of lists of strings.

Wrapped Composition. This rule combines composition with the list wrapping operation `up`, which maps an element a to the one element list $[a]$. Its type is $A \to LA$, for every set A.

$$\frac{\Gamma \vdash P : \sigma \qquad \Gamma, x : L(\sigma) \vdash Q : \tau}{\Gamma \vdash \texttt{let } x \leftarrow P \texttt{ in } Q : \tau} \tag{13}$$

This rule promotes the output of the first argument to a higher cardinality by wrapping it into a one-element list and then composes it. As an example, consider the *String list union* local Java widget, taking two lists of strings in input and unioning them:

$$x_1 : L(\mathbf{s}), x_2 : L(\mathbf{s}) \vdash \mathsf{String\_list\_union} : L(\mathbf{s}) \tag{14}$$

By composing the first input with, for instance, the string constant processor "red, green", we get

$$x_2 : L(\mathbf{s}) \vdash \\ \texttt{let } x_1 \leftarrow \text{“red, green”} \texttt{ in } \mathsf{String\_list\_union} : L(\mathbf{s})$$

Flattening. The third canonical operation for lists flattening, defined by the rule:

$$\frac{\Gamma \vdash P : L^2(\tau)}{\Gamma \vdash \texttt{flatten}(P) : L(\tau)} \tag{15}$$

This flattens a list of lists into a single list. If we apply the latter to (12) we thus obtain the following, where the output is only one big list of pathways:

$$\vdash \texttt{flatten}(\\ \texttt{let } \mathit{pathwayId} \leftarrow \mathsf{pathways} \\ \texttt{in } \mathsf{get\_enzymes\_by\_pathways}) : L(\mathbf{s})$$

Note that, although not a primitive, Taverna has a local Java widget called *Flatten list* that does exactly this.

Cross Product The cross product iteration is syntactic sugar for double application of the iterative composition rule. Thus if we define

$$\texttt{let } x_1 \ltimes x_2 \leftarrow P_1 \ltimes P_2 \texttt{ in } \equiv \\ \texttt{let } x_1 \leftarrow P_1 \texttt{ in } (\texttt{let } x_2 \leftarrow P_2 \texttt{ in } Q) \tag{16}$$

we obtain the following derived rule:

$$\frac{\Gamma \vdash P_1 : L(\sigma_1) \qquad \Gamma \vdash P_2 : L(\sigma_2) \qquad \Gamma, x_1 : \sigma_1, x_2 : \sigma_2 \vdash Q : \tau}{\Gamma \vdash \texttt{let } x_1 \ltimes x_2 \leftarrow P_1 \ltimes P_2 \texttt{ in } Q : L^2(\tau)}$$

Dot Product In contrast with the cross product, the dot product is a primitive. This is the only rule that does not follow from the general computational lambda calculus framework.

$$\frac{\Gamma \vdash P_1 : L(\sigma_1) \qquad \Gamma \vdash P_2 : L(\sigma_2) \qquad \Gamma, x_1 : \sigma_1, x_2 : \sigma_2 \vdash Q : \tau}{\Gamma \vdash \texttt{let } x_1 \odot x_2 \leftarrow P_1 \odot P_2 \texttt{ in } Q : L(\tau)} \tag{17}$$

In our example, if we set

- $P_1 =$ `let` $x_1 \leftarrow$ "red, green" `in` ColoursList
- $P_2 =$ `let` $x_2 \leftarrow$ "cat, rabbit" `in` AnimalsList : $L(\mathbf{s})$

then

$$\frac{\vdash P_1 : L(\mathbf{s}) \qquad \vdash P_2 : L(\mathbf{s}) \qquad x_4 : \mathbf{s}, x_5 : \mathbf{s} \vdash \mathsf{ColourAnimals} : \mathbf{s}}{\vdash \texttt{let } x_4 \odot x_5 \leftarrow P_1 \odot P_2 \texttt{ in } \mathsf{ColourAnimals} : L(\mathbf{s})} \tag{18}$$

We conclude the section by presenting the formal syntax for the entire workflow in Figure §1:

$$\vdash \texttt{let } x_6 \ltimes x_7 \leftarrow \\ (\texttt{let } x_3 \leftarrow \text{“square, circular, triangular”} \texttt{ in } \mathsf{ShapesList}) \\ \ltimes (\texttt{let } x_4 \odot x_5 \leftarrow \\ (\texttt{let } x_1 \leftarrow \text{“red, green”} \texttt{ in } \mathsf{ColoursList}) \\ \odot (\texttt{let } x_2 \leftarrow \text{“cat, rabbit”} \texttt{ in } \mathsf{AnimalsList}) \\ \texttt{in } \mathsf{ColourAnimals}) \\ \texttt{in } \mathsf{ShapeAnimals} : L^2(\mathbf{s})$$

3 Operational Semantics

We can now give the operational semantics for Taverna corresponding to the rules in §2. The rules again follow from the general computational lambda calculus theory. They are *structural* [Plo81] in the sense that they describe how complex workflows behave in terms of their components, using the behaviour of processors as the basis of the structural induction. Thus there is an operational semantics

rule for each syntactic rule in §2. These rules allow us to compute, on paper, what the outcome of running a workflow is (see example in §3).

We use the notation $P \Downarrow u$ to denote that the workflow P successfully terminates with output u. The latter can of course be a tuple of outputs and possibly contain lists.

In order to execute a workflow, this must be *closed*, i.e., its context must be empty. Thus all the rules below apply for closed workflows.

Processors. Most processors are web services, hence their operational semantics is defined entirely by their input/output behaviour, since no structural inspection of their content is possible. We would for instance observe that if we provide the KEGG web service (see §2) with the pathway id "path:bsu00010" as input, the output consists of a list:

$$[\text{ec:1.1.1.1}, \text{ec:1.1.1.2}, \text{ec:1.1.1.27}, \ldots, \text{ec:6.2.1.1}]$$

Formally, we write this as:

$$\begin{aligned}&\texttt{let } \mathit{pathwayId}\\ &\quad \leftarrow \text{“path:bsu00010”} \texttt{ in } \mathsf{get\_enzymes\_by\_pathways}\\ &\quad \Downarrow [\text{ec:1.1.1.1}, \text{ec:1.1.1.2}, \text{ec:1.1.1.27}, \ldots, \text{ec:6.2.1.1}]\end{aligned}$$

This generalises to every closed processor P.

A special case is given by the string constant processors, whose semantics is the string itself, for example:

$$\text{“red, green”} \Downarrow \text{“red, green”}$$

Finally, the ShapesList, AnimalsList and ColourLists processors all have the effect of splitting a string into a list of tokens: (using "," as the regular expression):

$$\texttt{let } x_1 \leftarrow \text{“red, green”} \texttt{ in } \mathsf{ColoursList} \Downarrow [\text{“red”}, \text{“green”}] \tag{19}$$

$$\texttt{let } x_2 \leftarrow \text{“cat, rabbit”} \texttt{ in } \mathsf{AnimalsList} \Downarrow [\text{“cat”}, \text{“rabbit”}] \tag{20}$$

Pairing. The semantics for the syntax pairing rule establishes that if two workflows P and Q terminate with respective outputs u and v, then the workflow pair $\langle P, Q\rangle$ terminates with pair $\langle u, v\rangle$ as output:

$$\frac{P \Downarrow u \quad Q \Downarrow v}{\langle P, Q\rangle \Downarrow \langle u, v\rangle} \tag{21}$$

Projections. Similarly for the two projections rules:

$$\frac{P \Downarrow \langle u, v\rangle}{\texttt{fst}(P) \Downarrow u} \qquad \frac{P \Downarrow \langle u, v\rangle}{\texttt{snd}(P) \Downarrow v} \tag{22}$$

Flattening.

$$\frac{P \Downarrow [[w_{11}, \ldots, w_{1m}], \ldots, [w_{n1}, \ldots, w_{nm}]]}{\texttt{flatten}(P) \Downarrow [w_{11}, \ldots, w_{ij}, \ldots, w_{nm}]} \tag{23}$$

Thus the output of $\texttt{flatten}(P)$ is obtained by removing the inner brackets from the output of P.

The following rules deal with our three different forms of composition. Their application depends on the types involved.

Simple Composition. Let Q have input x, and $\mathit{Type}(P) = \mathit{Type}(x)$. If P terminates with output u and if we send u through the input x of Q and obtain v, then their composition $\texttt{let } x \leftarrow P \texttt{ in } Q$ also terminates with output v:

$$\frac{P \Downarrow u \quad Q[u/x] \Downarrow v}{\texttt{let } x \leftarrow P \texttt{ in } Q \Downarrow v} \tag{24}$$

Note the notation $Q[u/x]$ which stands for "substitute the value u for the variable x in Q".

Iterative Composition. If $\mathit{Type}(P)$ is $L(\mathit{Type}(x))$, if P terminates with a list value $\vec{u} = [u_1, \ldots, u_n]$ and if each Q with of u_i substituted for x terminates as v_i, then $\texttt{let } x \leftarrow P \texttt{ in } Q$ terminates with output the list $\vec{v} = [v_1, \ldots, v_n]$. (Note that the v_i's might themselves be lists if that is the type of Q.) Formally:

$$\frac{P \Downarrow \vec{u} \quad \{Q[u_i/x] \Downarrow v_i\}_{i=1..|\vec{u}|}}{\texttt{let } x \leftarrow P \texttt{ in } Q \Downarrow \vec{v}} \tag{25}$$

Wrapped Composition. Conversely, let $\mathit{Type}(x)$ be $L(\mathit{Type}(P))$. If P terminates with output u and if Q terminates with value v when the singleton list $[u]$ is substituted for x, then $\texttt{let } x \leftarrow P \texttt{ in } Q$ terminates with v:

$$\frac{P \Downarrow u \quad Q[[u]/x] \Downarrow v}{\texttt{let } x \leftarrow P \texttt{ in } Q \Downarrow v}$$

Cross Product Definition (16) and rule (25) imply the following:

$$\frac{\begin{array}{c}P_1 \Downarrow \vec{u} \quad P_2 \Downarrow \vec{v}\\ \{Q[u_i/x_1][v_j/x_2] \Downarrow w_{ij}\}_{j=1..m}^{i=1..n}\\ |\ \vec{u}\ | = n \quad |\ \vec{v}\ | = m\end{array}}{\begin{array}{l}\texttt{let } x_1 \ltimes x_2 \leftarrow P_1 \ltimes P_2 \texttt{ in } Q\\ \Downarrow [[w_{11}, \ldots, w_{1m}], \ldots, [w_{n1}, \ldots, w_{nm}]]\end{array}} \tag{26}$$

Dot Product In the dot product of P_1 and P_2 in Q the assumption is that the length of the list produced by P_1 as output is the same as the length of that produced by P_2. We

write this as: if $P_1 \Downarrow \vec{u}$ and $P_2 \Downarrow \vec{v}$ then $|\ \vec{u}\ | = |\ \vec{v}\ |$. The rule is:

$$\frac{P_1 \Downarrow \vec{u} \quad P_2 \Downarrow \vec{v} \quad |\ \vec{u}\ | = |\ \vec{v}\ | \quad \{Q[u_i/x_1, v_i/x_2] \Downarrow w_i\}_{i=1..|\vec{u}|}}{\texttt{let}\ x_1 \odot x_2 \leftarrow P_1 \odot P_2\ \texttt{in}\ Q \Downarrow \vec{w}} \tag{27}$$

For example, using (19) and (20) and since the processor ColourAnimals concatenates its two input strings, we have:

$$\begin{array}{l}\texttt{let}\ x_4 \odot x_5 \leftarrow \\ \quad (\texttt{let}\ x_1 \leftarrow \text{“red, green”}\ \texttt{in}\ \text{ColoursList}) \\ \quad\ \odot(\texttt{let}\ x_2 \leftarrow \text{“cat, rabbit”}\ \texttt{in}\ \text{AnimalsList}) \\ \quad\ \texttt{in}\ \text{ColourAnimals} \Downarrow [\text{“red cat”}, \text{“green rabbit”}]\end{array}$$

Example (continued) Here is the operational semantics of our entire example workflow:

$$\begin{array}{l}\texttt{let}\ x_6 \ltimes x_7 \leftarrow \\ \quad (\texttt{let}\ x_3 \leftarrow \text{“square, circular, triangular”}\ \texttt{in}\ \text{ShapesList}) \\ \quad \ltimes (\texttt{let}\ x_4 \odot x_5 \leftarrow \\ \quad\quad (\texttt{let}\ x_1 \leftarrow \text{“red, green”}\ \texttt{in}\ \text{ColoursList}) \\ \quad\quad \odot(\texttt{let}\ x_2 \leftarrow \text{“cat, rabbit”}\ \texttt{in}\ \text{AnimalsList}) \\ \quad\quad \texttt{in}\ \text{ColourAnimals}) \\ \quad \texttt{in}\ \text{ShapeAnimals} \Downarrow \\ [[\text{“square red cat”}, \text{“square green rabbit”}], \\ \ [\text{“circular red cat”}, \text{“circular green rabbit”}], \\ \ [\text{“triangular red cat”}, \text{“triangular green rabbit”}]]\end{array}$$

Note that the output of the workflow is, as expected from its type, a list of lists.

4 Application of the Taverna model: workflows as computed artifacts

One use of the formal Taverna language specification is to enable formal proofs that involve a workflow definition. An interesting example consists of an application scenario where users provide a high-level, declarative specification of a process P, which is then automatically translated into a Taverna workflow T_P by a compiler comp. Given an apriori functional definition $f_P()$ of P, a formal specification for T_P enables one to prove the correctness of the compiler comp. In this section, we present the specification of T_P for a particular case with practical applications in e-science. Note however that the complete proof is beyond the scope of the paper.

Producing a workflow specification as the result of a compilation step, rather than by direct user input, is desirable whenever a family of workflows, all performing a similar function, can be described using few domain-specific, user-friendly primitives. In such cases, it is possible to define a workflow template and then specify rules for translating the user specification into an instance of the template. By automating the translation process, we produce error-free workflows, at the same time reducing the burden to the user.

4.1 Quality workflows

A practical example of this scenario is provided by the *Qurator* framework for Information Quality management in e-Science, described in detail in [MEG+06]. Consider an e-scientist who has defined a workflow to perform some *in silico* experiment, and is aware that the quality of the result can be adversely affected by the potentially poor quality of the data at some critical step in the process. The Qurator framework provides a simple way for the scientist to add a variety of tailormade quality filters to the original workflow, by specifying (i) which quality functions are to be applied to the data, and (ii) the quality control points in the original workflow where these functions are to be applied. This specification is known as a *Quality View*.

Qurator provides an ontology of Information Quality functions as well as a simple, XML-based language to describe Quality Views using the ontology classes. It also defines a workflow template that corresponds to a Taverna translation of a Quality View, and a compiler that takes a Quality View specification and computes an executable workflow, i.e., an instance of the workflow template.

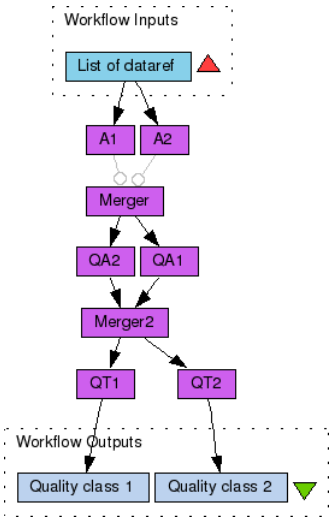

Figure 2. Generic Quality Workflow

One simple instance of quality workflow is shown in Figure 2. Its input consists of a list of unique data identifiers called *dataref*s, which represent the data to which the quality functions are applied. In a first step, the data is annotated

using *annotation processors* $\mathsf{A_i}$, $i : 1 \ldots n$, which associate metadata triples of the form $\langle name, class, value\rangle$ to each input dataref. For example, $\langle$C, q:coverage, $32.5\rangle$ denotes an annotation with name C, type "q:coverage" (a reference to an ontology class) and value 32.5.

Next, a collection of quality functions processors $\mathsf{QA_h}$, $h : 1 \ldots l$[2] take the annotations as input, and assign a score to each dataref. This is represented as a new annotation that is derived from the values for coverage C, along with those for other annotations.

Finally, the template accounts for yet a third layer of processors, denoted QT for "Quality Testing", which partition datarefs into quality classes according to logical expressions that predicate on the annotations accumulated through the previous steps. The classification {"accept", "reject"}, for instance, can be used to indicate which data elements should be discarded from the workflow. In addition to filtering, more expressive classifications can be used, in combination with other types of actions. For example, colour labels can be assigned to data elements to indicate how they should be presented to the user. To accommodate such expressiveness, Quality Views allow for the specification of multiple Quality Test processors.

A quality workflow is designed to be a sub-workflow embedded within a host scientific workflow, making the latter "quality-aware". In particular, the responsibility of performing the actual actions on the data is with the enclosing host workflow. Further details of the model, as well as of the example, are omitted for brevity.

4.2 Specification of Quality workflows

The following formal specification of a generic quality workflow is to be taken as an illustration of use of the Taverna model that may enable the formal proof of the Qurator Taverna translator.

Data annotations are represented as triples of the form $\langle name, class, value\rangle$, of type $\mathbf{s} \times \mathbf{s} \times \mathbf{s}$, denoted $\mathbf{s}^3$.

An *annotation processor* $\mathsf{A_i}$, $i : 1 \ldots n$, maps a dataref d to a list of annotation triples:

$$d : s \vdash \mathsf{A_i} : L(\mathbf{s}^3)$$

A *Quality Assertion* processor $\mathsf{QA_h}$, $h : 1 \ldots l$, computes a new annotation triple for a dataref, that is derived from a list of existing annotations previously computed by some $\mathsf{A_i}$:

$$y : L(\mathbf{s} \times L(\mathbf{s}^3)) \vdash \mathsf{QA_h} : L(\mathbf{s} \times L(\mathbf{s}^3))$$

A *Quality Test* processor $\mathsf{QT_k}$, $k : 1 \ldots r$, assigns a quality class label to a dataref according to the values of the available metadata, both primitive and derived:

$$w : \mathbf{s} \times L(\mathbf{s}^3) \vdash \mathsf{QT_k} : \mathbf{s} \times \mathbf{s}$$

[2]These processors are called *Quality Assertions* in Qurator, hence the term QA.

Note that the result of applying r independent Quality Test processors is a list of lists of class-labelled datarefs, for instance $[\langle d_1, \text{"accept"}\rangle, \langle d_2, \text{"reject"}\rangle]$ and $[\langle d_1, \text{"green"}\rangle, \langle d_2, \text{"red"}\rangle]$ (so that d_1 is both "accept" and "green", etc.).

The following rules define the quality workflow template.

Merging of annotations A Merger processor consolidates multiple list of annotation triples, each computed by a different annotation processor for the same input dataset, into a single list of annotations:

$$x_1 : \mathbf{s} \times L(\mathbf{s}^3) \ldots x_n : \mathbf{s} \times L(\mathbf{s}^3) \vdash \mathsf{Merger} : \mathbf{s} \times L(\mathbf{s}^3)$$

Composition of each A_i with Merger uses a dot product to provide multiple inputs to Merger in the correct order:

$$\frac{\{D : L(s) \vdash A_i : L(\mathbf{s} \times L(\mathbf{s}^3))\}_{i:1\ldots n} \quad x_1 : L(\mathbf{s}^3) \ldots x_n : \mathbf{s} \times L(\mathbf{s}^3) \vdash \mathit{Merger} : \mathbf{s} \times L(\mathbf{s}^3)}{D : L(s) \vdash \texttt{let}\ x_1 \odot \ldots \odot x_n \leftarrow A_1 \odot \ldots \odot A_n\ \texttt{in}\ \mathsf{Merger} : L(\mathbf{s} \times L(\mathbf{s}^3))}$$

The corresponding semantic rule is:

$$\frac{\{\mathsf{A_i}[\vec{e}/D] \Downarrow \vec{a_i}\}_{i:1\ldots n} \quad \mathsf{Merger}[\{\langle d, \vec{a}_j\rangle / x_i\}_{j:1\ldots n}] \Downarrow \langle d, \vec{a}\rangle}{(\texttt{let}\ x_1 \odot \ldots \odot x_n \leftarrow A_1 \odot \ldots \odot A_n\ \texttt{in}\ \mathsf{Merger})[\vec{e}/D] \Downarrow \vec{w}}$$

where $\vec{w} = [\langle d_1, \vec{a}'_1\rangle \ldots \langle d_m, \vec{a}'_m\rangle]$ includes the input dataset along with the consolidatd annotations. We write $\mathsf{WF_1}$ to denote the resulting workflow up to this point.

Computing Quality Assertions $\mathsf{WF_1}$ is composed with a QA processor $\mathsf{QA_h}$, $h : i \ldots l$, using simple composition:

$$\frac{D : L(\mathbf{s}) \vdash \mathsf{WF_1} : L(\mathbf{s} \times L(\mathbf{s}^3)) \quad y_h : L(\mathbf{s} \times L(\mathbf{s}^3)) \vdash \mathsf{QA_h} : L(\mathbf{s} \times L(\mathbf{s}^3))}{D : L(\mathbf{s}) \vdash \texttt{let}\ y_h \leftarrow \mathsf{WF_1}\ \texttt{in}\ \mathsf{QA_h} : L(\mathbf{s} \times L(\mathbf{s}^3))}$$

Here is the corresponding semantics rule:

$$\frac{\mathsf{WF_1}[\vec{e}/D] \Downarrow \vec{w} \quad \mathsf{QA_h}[\vec{w}/y_h] \Downarrow \vec{v}_h}{(\texttt{let}\ y_h \leftarrow \mathsf{WF_1}\ \texttt{in}\ \mathsf{QA_h})[\vec{e}/D] \Downarrow \vec{v}_h}$$

where $\vec{v}_h = [\langle d_1, \vec{a}''_1\rangle \ldots \langle d_m, \vec{a}''_m\rangle]$ includes the new annotations computed by $\mathsf{QA_h}$.

Merging of Quality Assertion values Let $\mathsf{P_{QA_h}}$ denote the workflow fragment resulting from this latest derivation rule. A Merger processor is again used to consolidate the annotations computed by all $\mathsf{P_{QA_h}}$ into a single list of annotations:

$$\frac{\{D : L(\mathbf{s}) \vdash P_{QA_h} : L(\mathbf{s} \times L(\mathbf{s}^3))\}_{h:i\ldots l} \quad x_1 : L(\mathbf{s}^3) \ldots x_n : \mathbf{s} \times L(\mathbf{s}^3) \vdash \mathit{Merger} : \mathbf{s} \times L(\mathbf{s}^3)}{D : L(s) \vdash \texttt{let}\ x_1 \odot \ldots \odot x_n \leftarrow P_{QA_1} \odot \ldots \odot P_{QA_l}\ \texttt{in}\ \mathsf{Merger} : L(\mathbf{s} \times L(\mathbf{s}^3))}$$

with semantics:

$$\frac{\{P_{QA_h}[\vec{e}/D] \Downarrow \vec{v}_h\}_{h:i\ldots l} \quad \mathsf{Merger}[\langle d, \vec{a}_j\rangle/x_j\}_{j:1\ldots l}] \Downarrow \langle d, \vec{a}\rangle}{(\texttt{let}\ x_1 \odot \ldots \odot x_n \leftarrow P_{QA_1} \odot \ldots \odot P_{QA_l}\ \texttt{in}\ \mathsf{Merger})[\vec{e}/D] \Downarrow \vec{v}}$$

where $\vec{v} = [\langle d_1, \vec{v}'_1\rangle \ldots \langle d_m, \vec{v}'_m\rangle]$ is the dataset with consolidated Quality Assertion annotation values.

Performing Quality Tests As a final step the resulting workflow, denoted WF_2, is composed with each of the r Quality Test processors QT_k using iterative composition:

$$\frac{D : L(\mathbf{s}) \vdash \mathsf{WF}_2 : L(\mathbf{s} \times L(\mathbf{s}^3)) \quad w : \mathbf{s} \times L(\mathbf{s}^3) \vdash \mathsf{QT}_\mathsf{k} : \mathbf{s} \times \mathbf{s}}{D : L(\mathbf{s}) \vdash \texttt{let}\ w \leftarrow \mathsf{WF}_2\ \texttt{in}\ \mathsf{QT}_\mathsf{k} : L(\mathbf{s} \times \mathbf{s})}$$

The corresponding semantics is:

$$\frac{\mathsf{WF}_2[\vec{e}/D] \Downarrow \vec{v} \quad \mathsf{QT}_\mathsf{k}[\langle d_i, \vec{a}'_i\rangle/w] \Downarrow \langle d_i, c_i\rangle}{(\texttt{let}\ w \leftarrow \mathsf{WF}_2\ \texttt{in}\ \mathsf{QT}_\mathsf{k})[\vec{e}/D] \Downarrow [\langle d_1, c_1\rangle, \ldots \langle d_m, c_m\rangle]}$$

This step yields a set of $k : 1 \ldots r$ independent workflows, denoted $\mathsf{WF}_{3,\mathsf{k}}$:

$$\mathsf{WF}_{3,\mathsf{k}} \equiv D : L(\mathbf{s}) \vdash \texttt{let}\ w \leftarrow \mathsf{WF}_2\ \texttt{in}\ \mathsf{QT}_\mathsf{k} : L(\mathbf{s} \times \mathbf{s})$$

The final outputs from the entire quality workflow is obtained by pairing. Assuming w.l.o.g. $r = 2$, this is written:

$$\frac{D : L(\mathbf{s}) \vdash \mathsf{WF}_{3,1} : L(\mathbf{s} \times \mathbf{s}) \quad D : L(\mathbf{s}) \vdash \mathsf{WF}_{3,2} : L(\mathbf{s} \times \mathbf{s})}{D : L(\mathbf{s}) \vdash \langle \mathsf{WF}_{3,1}, \mathsf{WF}_{3,2}\rangle : L(\mathbf{s} \times \mathbf{s}) \times L(\mathbf{s} \times \mathbf{s})}$$

with corresponding semantics:

$$\frac{\mathsf{WF}_{3,1}[\vec{e}/D] \Downarrow u \quad \mathsf{WF}_{3,2}[\vec{e}/D] \Downarrow v}{\langle \mathsf{WF}_{3,1}, \mathsf{WF}_{3,2}\rangle[\vec{e}/D] \Downarrow \langle u, v\rangle}$$

where each of the final lists represents an independent class labelling of the input dataset.

5 Conclusions and further work

We have presented two contributions in this paper, (i) a new formal syntax and semantics for the Taverna workflow management system, and (ii) an application of the formalism to precise characterize *quality workflows* that are automatically generated from a simpler, domain-specific process model.

The main focus of current work is on extending the model to describe data streams, currently unavailable in Taverna but essential to deal with large volumes of data, and one the main new features to be offered in the forthcoming Taverna 2 management system.

References

[BNTW95] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Computer Science*, 149(3), 1995.

[HKS+07] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. Van den Bussche. A formal model of dataflow repositories. In *DILS*, pages 105–121, 2007.

[KM77] G. Kahn and D.B. MacQueen. Coroutines and networks of parallel processes. In *IFIP congress*, 1977.

[LA03] B. Ludscher and I. Altintas. On providing declarative design and programming constructs for scientific workflows based on process networks. Technical Report SciDAC-SPA-TN-2003-01, San Diego Supercomputer Center, 2003.

[LABe05] B. Ludscher, I. Altintas, C. Berkley, and el. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, Special Issue on Scientific Workflows, 2005.

[MBL06] T. McPhillips, S. Bowers, and B. Ludscher. Collection-oriented scientific workflows for integrating and analyzing biological data. In *Proceedings DILS*, LNCS/LNBI. Springer, 2006.

[MEG+06] P. Missier, S. M. Embury, M. Greenwood, A. D. Preece, and B. Jin. Quality views: Capturing and exploiting the user perspective on data quality. In *VLDB*, pages 977–988, Seoul, Korea, September 2006.

[Mog91] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[OGAea06] T. Oinn, M. Greenwood, M. Addis, and M. Nedim Alpdemir et al. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, August 2006.

[OVvdA+07] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[SWAea07] S.Pettifer, K. Wolstencroft, P. Alper, and T. K. Attwood et al. myGrid and UTOPIA: An integrated approach to enacting and visualising in silico experiments in the life sciences. In *DILS*, pages 59–70, 2007.

[Wad90] P. Wadler. Comprehending monads. In *LISP and Functional Programming*, pages 61–78, 1990.