Journal of Computer and System Sciences 76 (2010) 490–508

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

# A formal semantics for the Taverna 2 workflow model

Jacek Sroka[a], Jan Hidders[b], Paolo Missier[c,*], Carole Goble[c]

[a] *University of Warsaw, Poland*
[b] *Delft University of Technology, The Netherlands*
[c] *University of Manchester, United Kingdom*

ARTICLE INFO

*Article history:*
Received 10 March 2009
Received in revised form 20 August 2009
Available online 22 November 2009

*Keywords:*
Taverna 2
Scientific workflows
Trace semantics
Formal semantics

ABSTRACT

This paper presents a formal semantics for the *Taverna* 2 scientific workflow system. Taverna 2 is a successor to Taverna, an open-source workflow system broadly adopted within the e-science community worldwide. The new version improves upon the existing model in two main ways: (i) by adding support for data pipelining, which in turns enables input streams of indefinite length to be processed efficiently; and (ii) by providing new extensibility points that make it possible to add new operators to the workflow model. Consistent with previous work by some of the authors, we use trace semantics to describe the effect of workflow computations, and we show how they can be used to describe the new features in the Taverna 2 model.

## 1. Introduction

Taverna [9] is a workflow language and computation model designed to support the automation of complex, service-based processes, specifically in the domain of science. Version 1 was first described in 2004 [17] and has since proven to be a useful tool for scientists, mainly in bioinformatics [4,12]. The formal semantics of Taverna was described first by Turi et al. [23], and then by Hidders et al. [7]. The latter in particular uses trace semantics to take into account possible side-effects that are associated with the services.

As a language, Taverna is strongly user-focused, as it attempts to strike a balance between expressivity of its operators and simplicity of use, with the ultimate goal of empowering users, who have only a rudimentary understanding of programming, to assemble complex workflows. The re-design of the workflow engine, which is now called "Taverna 2", reflects the need to reassess this balance as a result of years of practical usage, and of changing requirements.

This paper presents a new trace semantics for Taverna 2, following a similar approach to that of Hidders et al. [7]. We specifically formalise some of the model's new features, namely (i) support for data streaming, through pipelined execution of workflows; and (ii) support for extensibility of the set of workflow operators. Despite the fact that techniques for exploiting implicit data parallelism in workflows have been known for a long time [25], pipelining was not available in Taverna 1, resulting in a limitation in the scalability of applications where data consists of large collections.

Although we describe these features formally, and characterize the types of parallelism that are available in Taverna, our presentation is focused entirely on the development of the theoretical model, while a quantitative performance analysis of the workflow engine is beyond the scope of this paper.

We also pay special attention to Taverna's support of repeated processor execution by iteration over lists. This feature is important for scientific workflows and is shared among some other workflow systems. Notably, one of the computation

* Corresponding author.
*E-mail addresses:* j.sroka@mimuw.edu.pl (J. Sroka), a.j.h.hidders@tudelft.nl (J. Hidders), pmissier@cs.man.ac.uk (P. Missier), carole.goble@manchester.ac.uk (C. Goble).

0022-0000/$ – see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2009.11.009

models in the Kepler system [13], namely the COMAD model, supports streaming of data collections by letting workflow processors manipulate XML documents [14]. This has been recently advocated as a way to better meet the requirements of non-expert users [15].

The importance of defining formal models of various types of workflow computation has been recognised for a long time [1], including a formal model of Kepler and its precursor, the Ptolemy system [5]. Such models can be used to support the design of workflow languages and of their interpreters, compilers and optimizers as well as of debuggers, and to support the definition of verification procedures, similar to those used for verifying the correctness of complex business transactions. In the case of Taverna, a system whose design was largely driven by the practical need to support process automation for a variety of users types, such formalisation is even more important, in that it provides workflow designers, developers as well as users with an *a posteriori* description of the system's behaviour that is both unambiguous and complete.

The first question that needs to be answered before a formal semantics can be defined is what it is that we would like the semantics to describe about the specified workflows and at what abstraction level. It can be argued, for example, that from the point of view of the user a Taverna workflow just describes a computation that maps a set of input values to a set of output values. In that case the semantics of a Taverna workflow would be a non-deterministic function that maps the input values to the output values, or in other words, if two workflows express the same mapping then they are equivalent as far as the user is concerned. This is for example the perspective that was taken in earlier work by Turi et al. [23]. However, in this paper we assume that in some cases the user does care about which steps are executed by the computation and in what order. This is for example the case if these are calls to functions with side effects, e.g., an update to a database, or calls to web services that require a certain protocol. In this case the formalism should not just indicate to which output the input is mapped, but also which steps are taken to do so, and hence a process formalism is more appropriate. We will assume that the relevant events that are to be described are the consumption of the input values, the invocation of functions that have side effects or call external web services and the production of output values.

There are many formalisms and languages already available to describe processes and in particular processes that compose web services. Many of these have a syntax and semantics based on either the $\pi$-calculus [16] or on Petri nets [21]. Based on $\pi$ calculus are for example WS-CDL [10], WSCI [3] and XLANG, and based on Petri nets are BPMN [6], YAWL [24] and WSFL [11]. A somewhat special case is BPEL [2], which is based on both XLANG and WSFL. Its semantics is at least partially describable in terms of both Petri nets, e.g., [18], and $\pi$-calculus, e.g., [27], and it is explicitly aimed at defining an executable process specification.

Despite the existence of the previously described range of languages and formalisms, we have chosen to define a specific tailored set of operators for describing the semantics of Taverna 2 in terms of traces. We have done so for several reasons:

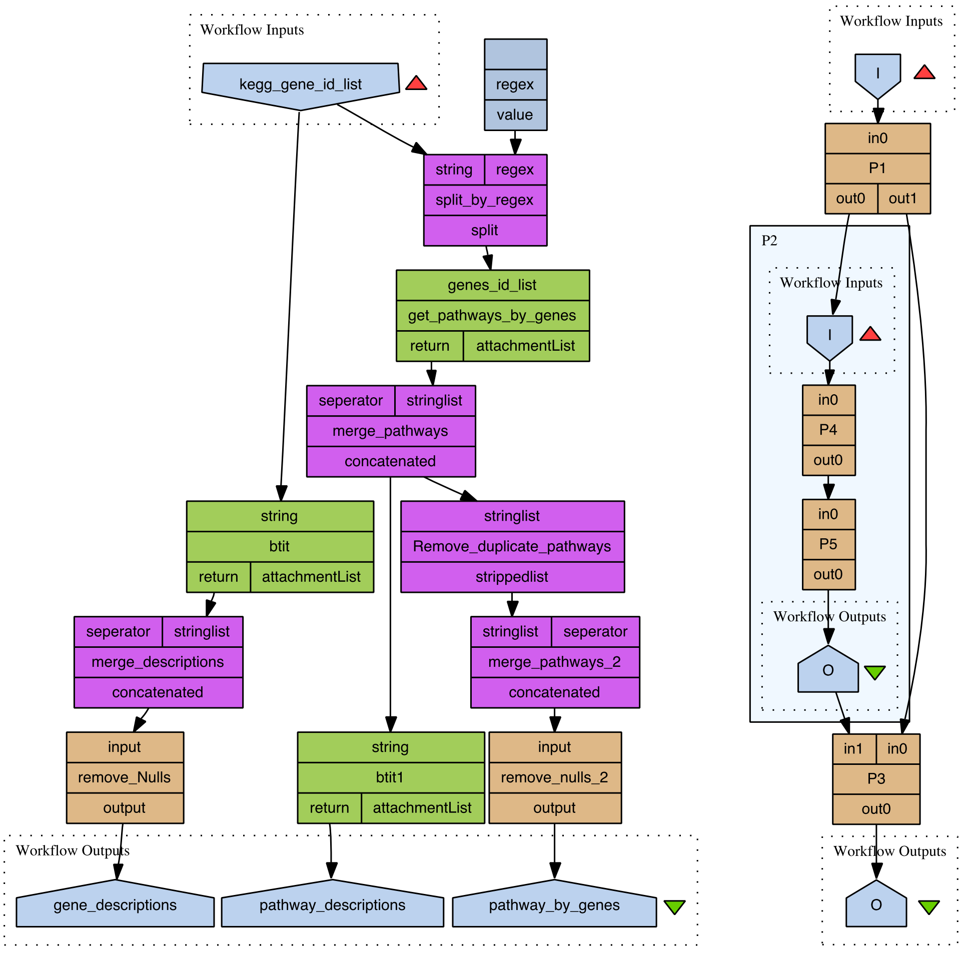
*Observational equivalence.* The main goal of the presented formalism is to define when two workflow specifications are equivalent, and to allow reasoning over what can and cannot be expressed in the Taverna 2 workflow language. This requires the chosen language to have a complete formal semantics that indeed describes when two workflows are observationally equivalent. Note that for formalisms that are based on transition systems, such as Petri nets and $\pi$-calculus, this means that next to the transition system an equivalence relation also needs to be defined. See for example [26] for a range of such equivalence relations and an explanation of when they are appropriate depending on the type of processes that are being described. In the case of workflow enactment such as defined in Taverna, it is not hard to see that trace semantics is the correct equivalence relation since the user can observe the events, i.e., the calls to services, but cannot influence them. As a consequence, formalisms based on bisimulation semantics, such as $\pi$-calculus, are less suitable or at least require a proof that this does not lead to incorrect equivalencies. It will also be clear that a formalism that directly defines the set of possible traces, rather than indirectly through Petri nets which in turn define sets of traces, will be easier to understand and reason over.

*Taverna* 2 *specific features.* Typically, Taverna workflows involve computations on recursively nested lists, i.e., list composition and iteration. Although extensions exist of $\pi$-calculus, e.g., Pict [20], and of Petri nets, e.g., DFL [8], that can describe this, these require major extensions of the basic formalism. Also here we claim that a tailored formalism that directly defines the traces for such behavior is more compact and easier to comprehend. A similar argument can be made for the two new features in Taverna 2 that are the focus of this paper: the pipelined execution of processors and the dispatch stack which defines an extensible mechanism to alter the behavior of individual processors.

The rest of the paper is organised as follows. We present an informal overview of the Taverna model and the calculus in Section 2, followed by a definition of trace, in Section 3. Traces are then used in Section 4 to describe the semantics of workflow graphs, in Section 5 to formalise Taverna's iteration strategies, and again in Section 6 to describe the extensibility features of the model. Finally, in Section 7 we describe a translation of a Taverna 2 specification into our calculus, and we conclude in Section 8.

## 2. Overview of Taverna and introduction to the calculus

Informally, a Taverna workflow graph is a directed acyclic graph where nodes, called *processors*, represent software components, for instance Web services or local scripts, with an interface that consists of input and output *ports*. Arcs in the graph connect pairs of ports, and specify a data dependency from the output port of one processor to the input port of



(a) A simple bioinformatics Taverna workflow (b) A nested workflow

**Fig. 1.** Examples of Taverna workflows.

another. Additionally, a control link between a source and a sink processor can be used to specify that the sink processor cannot execute until the source processor has produced all its output.

A simple example of a Taverna workflow for bioinformatics is illustrated in full detail notation with processor and port names in Fig. 1(a).[1] The workflow takes a user-supplied list of genes, in the format expected by the KEGG database,[2] and it retrieves information about the metabolic pathways which the input genes are involved in, according to the database. It uses the `KEGG` Web Service to fetch the data, as well as a number of ancillary scripts, known as "shims", as adapters between the output of a service and the input of the next.

Workflow computation is mostly data-driven: when there are no control links, a processor is ready to execute as soon as all of its input ports are bound to a value, and its execution causes its output ports to be bound to the result values. These values are then propagated along any arcs that originate in any of the output ports to create new bindings downstream. In particular, workflow execution begins when the workflow inputs are bound to initial input values, e.g., Kegg_ gene_id_list in the example, and terminates when output values cannot be propagated any further down the graph. In the example, the final results are bound to the output values gene_descriptions, pathway_descriptions, and pathway_by_genes.

Note that processors need not have any input ports, for instance rexeg in the figure, which is used to produce a constant value. Similarly a processor may have no output ports, and would only be executed for its side effects. Note also that the separator input ports of merge_descriptions, merge_pathways and merge_pathways_2 processors have no incoming arcs. For such input ports a default value is provided, but this is not visible in the graphical notation.

With processors we associate *activities*, which define the core functionality of the processor. These can be either *basic activities*, i.e., representing a local program or script or the invocation of an external service, or workflows themselves, i.e.,

[1] The workflow is published in the myExperiment website, please see http://www.myexperiment.org/workflows/611.
[2] KEGG is available at http://www.genome.jp/kegg/pathway.html.

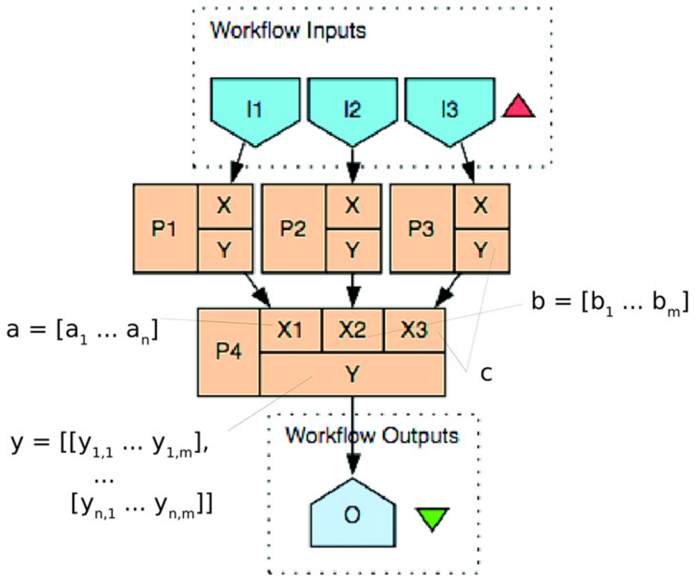


**Fig. 2.** List values in Taverna.

*nested workflows*, as shown in Fig. 1(b) where processor P2 contains a workflow with processors P4 and P5. All activities are characterized by a pair of interfaces that are both a set of port names. Basic activities in particular perform some kind of computation using the values on their input ports and produce new values on the output ports.

Let $\mathcal{B}$ be a countably infinite set of basic activity names and $\mathcal{P}$ a countably infinite set of port names. For example, with reference to Fig. 1(a), $\mathsf{Kegg\_query} \in \mathcal{B}$, and $\mathsf{string}, \mathsf{return}, \mathsf{attachmentList} \in \mathcal{P}$.[3] An *interface* is a finite set $I \subset \mathcal{P}$ We use variants of the variables $I$ and $J$ to denote interfaces. Interfaces describe the input and output ports of activities (including sub-workflows).

The *type* of an interface is based upon the type of its ports, which consists of a single basic type s containing basic values such as strings, booleans and numbers, and a list type constructor:

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

Here s denotes the basic type and $[\tau]$ the type of lists of elements of type $\tau$. For instance, port return in processor Kegg_query is of type [s]. Note that lists can be nested, as for example Y for processor P4 in Fig. 2. We will use variants of the variables $\tau$, $\sigma$ and $\rho$ to denote port types.

**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 $\langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle$, for example $\langle \mathsf{return} : [\mathsf{s}], \mathsf{attachmentList} : [\mathsf{s}] \rangle$, and the empty type is allowed in order to accommodate processors with no input (see for instance processor regex, which produces a constant value), and is denoted as $\langle\ \rangle$. The set of all interface types is denoted as $\Theta$.

We will use variants of the variables $\iota$ and $\kappa$ to denote interface types. The disjoint union of two interface types $\iota$ and $\kappa$ is denoted as $\iota, \kappa$. 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 basic activity names $f \in \mathcal{B}$, an input interface type $\iota$ and output interface type $\kappa$ are given, which is denoted as $f : \iota \to \kappa$, for instance $\mathsf{Kegg\_\ query} : \langle \mathsf{string} : [\mathsf{s}] \rangle \to \langle \mathsf{return} : [\mathsf{s}], \mathsf{attachment\_list} : [\mathsf{s}] \rangle$.

Lists, i.e., values of type $[\tau]$, play a central role in the semantics of the calculus since the values that are manipulated by the workflows are mostly lists. Furthermore, we also use lists to describe traces. Lists 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 $v \cdot w$. We generalize this notation such that $\bullet_{i=1..n} v_i$ denotes $v_1 \cdot \cdots \cdot v_n$ if $n > 0$ and $[\,]$ if $n = 0$. If $w = [w_1, \ldots, w_n]$ then we write $a \in w$ to denote that $a$ appears in $w$, i.e., there exists an $1 \leqslant i \leqslant n$ such that $w_i = a$. The length of a list $v$ is denoted as $|v|$.

In order to define the semantics of port types we postulate the set of basic values $\mathcal{S}$ which will be semantics of the basic type s. We will assume that the basic values include strings, booleans and numbers, but not a special error value **err**.

[3] As activities are often wrappers for third party Web Services, the sometimes uninspiring names of the ports are not chosen by the workflow designer, but rather they are mandated by the service's WSDL interface definition.

The semantics of a port type $\tau$, denoted as $[\![\tau]\!]$, defines the set of values that can be associated to the port, and is defined with induction on the structure of $\tau$ such that (1) $[\![\mathrm{s}]\!] = \mathcal{S} \cup \{\mathbf{err}\}$ and (2) $[\![[\tau]]\!]$ is the union of $\{\mathbf{err}\}$ and the set of all lists $[v_1, \ldots, v_n]$ with $n \geqslant 0$ and $v_i \in [\![\tau]\!]$ for all $1 \leqslant i \leqslant n$. 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 3** *(Interface value).* An *interface 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 $t = \langle a_1 = v_1, \ldots, a_n = v_n \rangle$.

An example is the value given by the tuple

$$\langle \mathsf{return} = [\texttt{"Thyroid cancer..."}, \texttt{"PPAR signaling pathway..."}], \mathsf{attachmentList} = []\rangle$$

produced by processor Kegg_query. The set of all interface values is denoted as $\mathcal{I}$. We will use variants of $t$ to represent interface values. Element $a_i$ of tuple $t$ is denoted $t(a_i)$.

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 $t = \langle a_1 = v_1, \ldots, a_n = v_n \rangle$ such that $v_j \in [\![\iota(a_j)]\!]$ for all $1 \leqslant j \leqslant n$.

For the basic activities $f : \iota_1 \to \iota_2$ we assume their semantics to be defined by a binary relation $[\![f]\!] \subseteq [\![\iota_1]\!] \times [\![\iota_2]\!]$. For example, $[\![\mathsf{Kegg\_query}]\!]$ includes:

$$\langle \mathsf{string} = [\texttt{"path:hsa05216"}, \texttt{"path:hsa03320"}]\rangle$$

$$\langle \mathsf{return} = [\texttt{"Thyroid cancer..."}, \texttt{"PPAR signaling pathway..."}], \mathsf{attachmentList} = []\rangle$$

Note that this binary relation is not required to be either total or functional. We allow it to be partial to model that the associated service always fails for certain input values even if they belong to the required input type. This might for example be the case if they do not satisfy certain additional preconditions that the service requires. We allow the semantics to be non-functional to model that the result of a service call may be non-deterministic from the point of view of the workflow, i.e., not be completely determined by the arguments.

## *2.1. List values and repeated processor invocation*

Although the workflow is an acyclic graph, and thus does not permit users to explicitly describe loops over groups of processors, the Taverna model specifies how a processor can execute repeatedly, by iterating over its input ports list values. Here we give an informal account of this mechanism, while a complete formal model will be given in Section 5.

Let $pvd(v)$ denote the depth of a port value $v$, for example, $pvd(\texttt{"cat"}) = 0$, $pvd([\texttt{"cat"}, \texttt{"dog"}]) = 1$, $pvd([[\texttt{"cat"}], [\texttt{"dog"}]]) = 2$, et cetera. Similarly, let $ptd(\tau)$ denote the depth of a port type $\tau$. Let $A : \iota \to \kappa$ be an activity with a simple input interface type $\iota = \langle a : \tau \rangle$, and let $t = \langle a = v \rangle$ be an interface value that describes the input to $A$. Consider the case where $pvd(v) = ptd(\tau) + 1$, that is, the depth of the value assigned to $a$ is one greater than the depth of the type $\tau$ of $a$. This *depth mismatch* is dealt with by executing $A$ repeatedly, once for each element $v_i \in v$. This can be described as the application of the well-known map function to $A$ and $v$, i.e., $(\mathsf{map}\ A\ v) = [(A\ v_1) \ldots (A\ v_{|v|})]$. More generally, map is applied recursively $pvd(v) - ptd(\tau)$ times to $A$. For example, if $\iota = \langle a : \mathrm{s} \rangle$ and $t = \langle a = v \rangle$ where $v = [[\texttt{"cat"}, \texttt{"dog"}], [\texttt{"black"}, \texttt{"white"}]]$, then the workflow computes

$$(\mathsf{map}\ (\mathsf{map}\ A)\ [[\texttt{"cat"}, \texttt{"dog"}], [\texttt{"black"}, \texttt{"white"}]]) =$$

$$[(\mathsf{map}\ A\ [\texttt{"cat"}, \texttt{"dog"}]), (\mathsf{map}\ A\ [\texttt{"black"}, \texttt{"white"}])] =$$

$$[[(A\ \texttt{"cat"}), (A\ \texttt{"dog"})], [(A\ \texttt{"black"}), (A\ \texttt{"white"})]]$$

This *implicit iteration* rule is designed to facilitate the composition of individual services into workflows. For example, it makes it possible to feed the results of a database lookup service P1, which returns a collection of records, to a service P2 that is designed to analyse a single record. In this case, implicit iteration is a natural interpretation of the data dependency between P1 and P2.

The iteration rule extends to the case where depth mismatches appear on multiple inputs. Fig. 2 shows a workflow fragment, where all input ports (presented on the top right side of the processor) of P4 expect simple values. When only the value on port $c$ is basic as expected and the list-valued inputs $a = [a_1, \ldots, a_n]$ and $b = [b_1, \ldots, b_m]$ are provided instead, the multiple port mismatches $pvd(a) = ptd(\mathsf{P4{:}X1}) + 1$ and $pvd(b) = ptd(\mathsf{P4.X3}) + 1$ are solved with the help of the *cross product* $a \boxtimes b = [[\langle a_1, b_1 \rangle, \ldots, \langle a_1, b_m \rangle], \ldots, [\langle a_n, b_1 \rangle, \ldots, \langle a_n, b_m \rangle]]$. We can describe the result using standard λ-calculus notation, as follows:

$$y = \big(\mathsf{map}\ \big(\mathsf{map}\ \big(\lambda\, x_a\, x_b.(\mathsf{P4}\ x_a\ x_b\ c)\big)\big)\ (a \boxtimes b)\big)$$

where $|y| = n$, and $\lambda\, x_a\, x_b.(\mathsf{P4}\ x_a\ x_b\ c)$ denotes a function with arguments $x_a$ and $x_b$ that applies P4 to $x_a$, $x_b$, and $c$. The map operator repeatedly applies this function, by iteratively binding $x_a$ and $x_b$ to each element of the product $a \boxtimes b$. In the example, the result is a list of the form $[[y_{1,1},\dots,y_{1,m}],\dots,[y_{n,1},\dots,y_{n,m}]]$ where $y_{i,j} = (\mathsf{P4}\ a_i\ b_j\ c)$. Note that here $c$ is treated as a constant parameter, because there is no port mismatch on X3, therefore the same value $c$ is used in each iteration.

Additionally, Taverna workflow designers have the option to choose a different iteration strategy, based on the dot product $a \boxdot b = [\langle a_1, b_1\rangle, \langle a_2, b_2\rangle, \dots, \langle a_k, b_k\rangle]$ where $k = \min(n, m)$. The description in λ-calculus notation is

$$y = \bigl(\mathsf{map}\ \bigl(\lambda\, x_a\, x_b.\bigl(\mathsf{P4}\ x_a\ x_b\ c\bigr)\bigr)\ \ (a \boxdot b)\bigr)$$

where $|y| = k$. The overall iteration strategy of a processor can be specified by an *iteration strategy expression*, such as for example $(a \boxtimes b) \boxdot c$, where the symbol $\boxtimes$ indicates the cross product strategy and the $\boxdot$ indicates the dot product strategy. In Section 5 we further generalize and formalize the cross and dot product operators, showing in particular that the cross product operator can be simulated using the dot product iteration and one additional operator of the calculus.

## 2.2. Pipelining

The Taverna 2 execution engine provides support for the pipelined execution of processors along a path in the workflow, an important feature not previously available in Taverna 1, as mentioned in the introduction. This feature is strictly connected with the implicit iteration mechanism just discussed, and is aimed at improving efficient execution over large data collections. In Taverna, iterations over items in a list may be executed concurrently if sufficient resources are available to support parallel execution threads. When this happens, the elements of the result list may be produced in arbitrary order, due to different speeds of each of the threads. These possible different arrival orders are captured in workflow traces, described in Section 3, by recording events that represent individual *locations* (see Definition 4) of an output list being assigned a value, and values at input locations being used by an activity.

In Taverna 1, once the output list is complete, the engine activates each of the downstream processors that consume the list. While this behaviour is correct, it does not take advantage of additional implicit parallelism that is available due to the mutual independence between data elements within a list. In contrast, Taverna 2 allows for the consumer processors to begin to process individual elements of the result without the need to wait for the entire list to be complete. The most common and useful case occurs when consumers are also subject to implicit iteration. Suppose for example that, in the fragment of Fig. 2, P4:Y has an outgoing arc to some port P5:X, with port type depth equal to 0. As we know, P5 will iterate on each of the elements $y_{i,j}$ in $y$, and furthermore, these iterations will all be independent of one another. Since the sub-lists as well as the simple values in $y$ are themselves independently generated, each of them can be forwarded to P5 as soon it becomes available on port P4:Y, independently from the others. By greedily pushing partial results to downstream processors, the speed at which individual values are computed is no longer bounded by the slowest thread that accounts for one iteration over P4.

We can frame this behaviour in terms of a collection of workflow patterns for parallel computing, proposed in [19]. Within this framework, Taverna implements a *superscalar* pipeline execution semantics, whereby a new thread is allocated to each list element in the input to an iterating processor. At the same time, however, an upper bound on the number of available threads can be set for each processor (through an advanced configuration option). With this limitation, some pipelines can be *blocking*, if the upstream processors are systematically faster than downstream processors.

In addition, Taverna also supports *streaming pipelines*. In this context, a stream is defined as an unpredictably long collection of discrete input data elements, an increasingly common occurrence in applications such as process monitoring, or sensor data processing. Indeed, Biomart (www.biomart.org) [22] is an example of a service that supplies its output in a streamed fashion. Taverna is able to consume the stream as it appears on the output port, either by feeding it incrementally to downstream services that also accept streams, or by buffering the output prior to forwarding it to services that expect complete data as input. Conceptually, buffers have unlimited capacity, thanks to a two-tier data architecture that involves an in-memory buffer with a database overflow. A complete description of the data architecture is beyond the scope of this paper, however.

## 3. Trace semantics

As anticipated in the introduction, we model the semantics of workflows in terms of traces. Informally, traces record sequences of three types of elementary events: (i) input events, representing values arriving on input ports; (ii) the atomic execution of a basic activity, and (iii) output events, i.e., the availability of a value on an output port. In this section we introduce our notation for describing traces, while in the next section we introduce the set of operators of our calculus that describe Taverna 2 workflows and define the semantics of such operators as *the set of all legal traces that the use of the operator can produce*. As a workflow is described by an expression that combines operators of the calculus, by extension the semantics of a workflow is defined as the set of all legal traces that the expression can produce. Note that by describing the semantics in terms of traces that contain basic activity executions, rather than just input and output events, we account for the fact that these activities involve possibly stateful service invocations, a common occurrence in scientific workflows.

Traces carry information about the relative ordering of events which are assumed to be atomic, but they are not concerned with the actual relative *duration* of parts of the computation. Although a basic activity execution can in principle

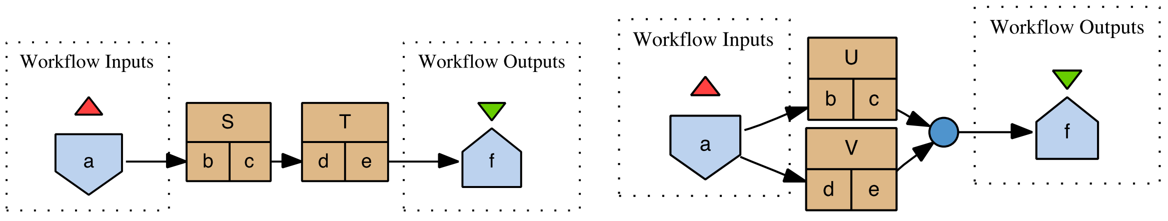


**Fig. 3.** Basic workflow graphs to illustrate sequential and parallel composition.

be thought of as the combination of two steps: sending a request and receiving a response, we are going to assume for simplicity that basic activity executions are atomic and they are therefore described by a single event.

As mentioned in the previous section, collections consist of elementary values that can, in principle, be consumed and produced, and thus appear in input and output trace events, in any order. We use *locations* to indicate the position of an elementary basic value within a containing collection value, as follows.

**Definition 4** *(Location).* A *location* is a possibly empty list of non-zero natural numbers. These lists are denoted by separating the numbers with dots, such as in 1.4.2.1 and 2.3, and the empty location is denoted as $\epsilon$. The set of all locations is denoted as $\mathcal{L}$. The concatenation of two locations $\lambda_1$ and $\lambda_2$ is denoted as $(\lambda_1.\lambda_2)$.

The numbers further from the location head give the indexes in deeper nested lists. As an example, each element $y_{i,j}$ of the list $[[y_{1,1},\ldots,y_{1,m}],\ldots,[y_{n,1},\ldots,y_{n,m}]]$ is at location $i.j$, for all $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$.

**Definition 5** *(Workflow trace).* A workflow trace is a list of events. An event is either

- a successful basic activity execution: $f(t_1) \mapsto t_2$ where $f \in \mathcal{B}$ and $t_1, t_2 \in \mathcal{I}$,
- a failed basic activity execution: $f(t_1) \mapsto \bot$ where $f \in \mathcal{B}$ and $t_1 \in \mathcal{I}$,
- an input event: $\mathbf{in}_a(\lambda, v)$ with $a \in \mathcal{P}$, $\lambda \in \mathcal{L}$ and $v \in \mathcal{S} \cup \{\mathbf{err}, [\,]\}$,
- an output event: $\mathbf{out}_a(\lambda, v)$ with $a \in \mathcal{P}$, $\lambda \in \mathcal{L}$ and $v \in \mathcal{S} \cup \{\mathbf{err}, [\,]\}$,
- a fail event: **fail**.

Since a workflow execution involves both parallelism across processors, and pipelining across iterations of a processor, different executions may generate different traces for the same inputs, even when the services involved are stateless.

Fig. 3 includes two simple workflows. We use them to demonstrate: (1) complete workflow trace examples and (2) that traces can have different relative ordering of events. Here are two possible traces for two executions of the workflow of Fig. 3(a):

$$\begin{aligned}\alpha = \big[&\mathbf{in}_{\mathsf{a}}(1,x_1), \mathbf{in}_{\mathsf{a}}(2,x_2), F_{\mathsf{S}}\big(\langle \mathsf{b}=x_1\rangle\big) \mapsto \langle \mathsf{c}=y_1\rangle, F_{\mathsf{S}}\big(\langle \mathsf{b}=x_2\rangle\big) \mapsto \langle \mathsf{c}=y_2\rangle,\\ &F_{\mathsf{T}}\big(\langle \mathsf{b}=y_1\rangle\big) \mapsto \langle \mathsf{c}=z_1\rangle, F_{\mathsf{T}}\big(\langle \mathsf{b}=y_2\rangle\big) \mapsto \langle \mathsf{c}=z_2\rangle, \mathbf{out}_{\mathsf{f}}(1,z_1), \mathbf{out}_{\mathsf{f}}(2,z_2)\big]\\ \beta = \big[&\mathbf{in}_{\mathsf{a}}(1,x_1), \mathbf{in}_{\mathsf{a}}(2,x_2), F_{\mathsf{S}}\big(\langle \mathsf{b}=x_1\rangle\big) \mapsto \langle \mathsf{c}=y_1\rangle, F_{\mathsf{T}}\big(\langle \mathsf{e}=y_1\rangle\big) \mapsto \langle \mathsf{e}=z_1\rangle,\\ &\mathbf{out}_{\mathsf{f}}(1,z_1), F_{\mathsf{S}}\big(\langle \mathsf{b}=x_2\rangle\big) \mapsto \langle \mathsf{c}=y_2\rangle, F_{\mathsf{T}}\big(\langle \mathsf{d}=y_2\rangle\big) \mapsto \langle \mathsf{d}=z_2\rangle, \mathbf{out}_{\mathsf{f}}(2,z_2)\big]\end{aligned}$$

Here $F_{\mathsf{S}}$ and $F_{\mathsf{T}}$ are the basic activities associated with processor S and T. Note that in both traces these processors iterate over their input, a list with two elements, but in $\beta$ processor T already starts before S is finished, as might be expected in a pipelined execution.

We also give two traces for two executions of the workflow of Fig. 3(b):

$$\begin{aligned}\alpha &= \big[\mathbf{in}_{\mathsf{a}}(\epsilon,x), F_{\mathsf{U}}\big(\langle \mathsf{b}=x\rangle\big) \mapsto \langle \mathsf{c}=y_1\rangle, F_{\mathsf{V}}\big(\langle \mathsf{d}=x\rangle\big) \mapsto \langle \mathsf{e}=y_2\rangle, \mathbf{out}_{\mathsf{f}}(1,y_1), \mathbf{out}_{\mathsf{f}}(2,y_2)\big]\\ \beta &= \big[\mathbf{in}_{\mathsf{a}}(\epsilon,x), F_{\mathsf{V}}\big(\langle \mathsf{d}=x\rangle\big) \mapsto \langle \mathsf{e}=y_2\rangle, F_{\mathsf{U}}\big(\langle \mathsf{b}=x\rangle\big) \mapsto \langle \mathsf{c}=y_1\rangle, \mathbf{out}_{\mathsf{f}}(2,y_2), \mathbf{out}_{\mathsf{f}}(1,y_1)\big]\end{aligned}$$

Note that the complete traces include only initial input events and final output events. As will become clear with the presentation of the semantics rules, the corresponding intermediate input and output events cancel themselves out when the composition operator is used.

To describe all possible orderings of events in a trace in a compact way, we write $(\alpha \parallel\!\mid \beta)$ to denote the set of all possible *interleavings* of two traces $\alpha$ and $\beta$. Formally: $(\alpha \parallel\!\mid \beta) = \{\alpha_1 \cdot \beta_1 \cdot \cdots \cdot \alpha_n \cdot \beta_n \mid n \geqslant 1,\ \alpha = \alpha_1 \cdot \cdots \cdot \alpha_n,\ \beta = \beta_1 \cdot \cdots \cdot \beta_n\}$. Note that this indeed defines all possible interleavings of the events in $\alpha$ and $\beta$ since $\alpha_i$ and $\beta_j$ can be the empty trace. We generalize the interleaving set for more than two traces, denoted as $(\alpha_1 \parallel\!\mid \ldots \parallel\!\mid \alpha_n)$ or $\parallel\!\mid_{i=1..n} \alpha_i$, such that $\beta \in (\alpha_1 \parallel\!\mid \ldots \parallel\!\mid \alpha_n)$

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.

As we have seen from the previous example, multiple events may be used to represent the arrival or generation of list values. For convenience, we summarize the entire set of such events by denoting the encoding of an entire input (resp., output) value $v$ in a trace $\alpha$ as $\mathbf{in}_a(v) \Downarrow \alpha$ (resp. $\mathbf{out}_a(v) \Downarrow \alpha$). Note that the elements of the list may be produced in any order, therefore different executions may show different encodings for the same values.

It will also be necessary to describe list values of depth $n$, which are obtained by wrapping an existing list of depth $n-1$, e.g., a list $[\![a_1, a_2]\!]$ obtained by wrapping list $[a_1, a_2]$. If $\mathbf{in}_a(\lambda, v)$ is the event corresponding to the arrival of an input $v$ at location $\lambda$, then the corresponding location in the wrapped list is $i.\lambda$ for some positive $i$. Therefore we let $i.\alpha$ denote the trace that we obtain if in $\alpha$ we replace each event $\mathbf{in}_a(\lambda, v)$ with $\mathbf{in}_a(i.\lambda, v)$ and each event $\mathbf{out}_a(\lambda, v)$ with $\mathbf{out}_a(i.\lambda, v)$. Formally, $\mathbf{in}_a(v) \Downarrow \alpha$ is defined with induction on the structure of $v$ by the following inference rules:

$$\frac{v \in \mathcal{S} \cup \{\mathbf{err}, [\,]\}}{\mathbf{in}_a(v) \Downarrow [\mathbf{in}_a(\epsilon, v)]} \qquad \frac{\forall_{i=1..n}(\mathbf{in}_a(v_i) \Downarrow \alpha_i) \qquad \gamma \in ⫴_{i=1..n}(i.\alpha_i)}{\mathbf{in}_a([v_1, \ldots, v_n]) \Downarrow \gamma}$$

These are examples of inference rules that will be used extensively throughout the paper. The first rule states that if $v$ is a basic value (or **err** or $[\,]$), then the single event $\mathbf{in}_a(v)$ generates the single-event trace $[\mathbf{in}_a(\epsilon, v)]$. The second rule states that if $n$ basic values $v_1, \ldots, v_n$ arrive on port $a$, and each such arrival is encoded as $\alpha_i$, then the arrival of a list $[v_1, \ldots, v_n]$ is encoded by a trace $\gamma$, which can be any interleaving of traces $i.\alpha_i$. The rules for output traces, i.e., $\mathbf{out}_a(v) \Downarrow \alpha$, are similar and omitted for brevity.

Where basic values and lists are associated with streams on individual input and output ports, tuple values are used to model the complete input or output of a processor and in that sense arrive at or are produced by a processor as a whole. Correspondingly, as a further shorthand we introduce the notation $\mathbf{in}^*(t) \Downarrow \beta$ and $\mathbf{out}^*(t) \Downarrow \beta$ to denote that trace $\alpha$ encodes the tuple interface value $t$ in input events or output events, respectively. Formally, such an encoding $\beta$ is an interleaving of traces $\alpha_i$, where each $\alpha_i$ encodes the arrival (resp., generation) of one field $a_i$ of the tuple $t$ on port $a_i$:

$$\frac{\begin{array}{c}\mathbf{dom}(t) = \{a_1, \ldots, a_n\} \\ \forall_{i=1..n}(\mathbf{in}_{a_i}(t(a_i)) \Downarrow \alpha_i) \qquad \beta \in ⫴_{i=1..n}\alpha_i\end{array}}{\mathbf{in}^*(t) \Downarrow \beta}$$

The rules for output traces, i.e., $\mathbf{out}^*(t) \Downarrow \beta$, are similar and omitted for brevity.

Finally, we introduce the notation $\alpha|_I$ to denote the filtering of the stream $\alpha$ on the input events on ports from $I$ disregarding all other event types, i.e., $\alpha|_I = \{\mathbf{in}_a(\lambda, v) \mid \mathbf{in}_a(\lambda, v) \in \alpha,\ a \in I\}$ and a counterpart notation for output events $\alpha|^I = \{\mathbf{out}_a(\lambda, v) \mid \mathbf{out}_a(\lambda, v) \in \alpha,\ a \in I\}$. Note that a filtering is a set, rather than a trace itself. We use $\alpha|$ to denote the set of *all* events in $\alpha$, i.e., $\alpha| = \{e \mid e \in \alpha\}$.

## 4. Semantics of workflow graphs

In the calculus we distinguish two types of expressions, *fragments* and *activities*. Fragments represent parts of workflow graphs, i.e., single processors and subgraphs. Activities represent wrappers for real-world services that are executed by workflows, e.g., web service clients. The difference between the two is that fragments never fail, while activities may terminate with failure. We need to make this distinction explicit because T2 assumes that nodes in the graph represent processors that never fail. However, since in reality the services that are invoked when the processors are executed can fail, this potential mismatch is resolved by the processor execution and the semantics accounts for this.

In this section we give the formal semantics for both types of expressions. The fragments correspond to subgraphs of workflow graphs, which can be composed into bigger graphs. The operators that describe such fragments have types of the form $\iota \twoheadrightarrow \kappa$, where $\iota$ and $\kappa$ are interface types that describe the input ports and the output ports, respectively. The semantics of these operators is defined by judgments of the form $e \Downarrow \alpha$, where $e$ is a fragment expression and $\alpha$ a possible trace that can include only input events, output events and basic activity executions but no failure events. The intended semantics of a type $\iota \twoheadrightarrow \kappa$ is that if a fragment has this type then it holds for the associated set of traces that (1) no trace in the set contains **fail**, (2) for each interface value $t$ of interface type $\iota$ there is at least one trace $\alpha$ in this set such that $\alpha|_{\mathcal{P}}$, i.e., the set of input events, encodes $t$ and (3) for each trace $\alpha$ in this set if it holds that $\alpha|_{\mathcal{P}}$ encodes a values of type $\iota$, then $\alpha|^{\mathcal{P}}$, i.e., the set of output events in $\alpha$, encodes a value of type $\kappa$.

Activities are executed during the run of a workflow and can be either basic activities that represent real world services or nested workflow graphs. Operators that describe activities have types of the form $\iota \Rightarrow \kappa$ and their semantics is defined by judgments of the form $e \Downarrow \alpha$, where $e$ is an activity expression and $\alpha$ a possible trace that can include **fail**. The intended semantics of a type $\iota \Rightarrow \kappa$ is that if a fragment has this type then it holds for the associated set of traces that (1) for each interface value $t$ of interface type $\iota$ there is at least one trace $\alpha$ in this set such that $\alpha|_{\mathcal{P}}$ encodes $t$, and (2) for each trace $\alpha$ in this set it holds that if $\alpha|_{\mathcal{P}}$ encodes a values of type $\iota$ then either $\alpha$ contains **fail** or $\alpha|^{\mathcal{P}}$ encodes a value of type $\kappa$ and (3) for each trace in this set it holds that if it contains a **fail** then this is the last event in the trace. Note that it holds that every fragment of type $\iota \twoheadrightarrow \kappa$ is also an activity of type $\iota \Rightarrow \kappa$. Although we do not formally prove this in this paper,

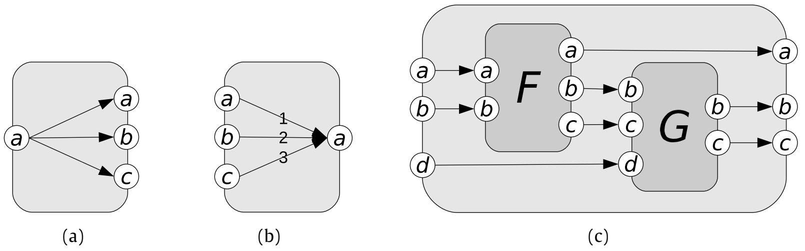


**Fig. 4.** Illustrations of $\mathbf{ln}_{a\to a,b,c}$, $\mathbf{merge}_{a;b;c\to a}$ and $F \rhd_{b,c} G$.

it can be verified that the sets of traces that are defined by the presented operators indeed satisfy the requirements by the types of the operators.

In Section 4.1 we present the basic operators for defining fragments. Then, in Section 4.2, we explain how to translate basic workflow graphs defined in Taverna 2 into these basic operators. Finally, in Section 4.3 we introduce *activities* and define the conversions between fragments and activities.

### 4.1. Basic fragment operators

The syntax of the fragment operators is given by:

$$\frac{\iota = \langle b_1 : \tau, \ldots, b_n : \tau\rangle \qquad I = \mathbf{dom}(\iota)}{\mathbf{ln}_{a\to I} : \langle a : \tau\rangle \twoheadrightarrow \iota} \qquad \frac{\iota = \langle a_1 : \tau, \ldots, a_n : \tau\rangle}{\mathbf{merge}_{a_1;\ldots;a_n\to b} : \iota \twoheadrightarrow \langle b : [\tau]\rangle}$$

$$\frac{\begin{array}{c} F : \iota_1 \twoheadrightarrow \kappa_1, \kappa_2 \qquad G : \kappa_2, \iota_2 \twoheadrightarrow \kappa_3 \qquad I = \mathbf{dom}(\kappa_2) \\ \mathbf{dom}(\iota_1) \cap \mathbf{dom}(\iota_2) = \emptyset \qquad \mathbf{dom}(\kappa_1) \cap \mathbf{dom}(\kappa_3) = \emptyset \end{array}}{(F \rhd_I G) : \iota_1, \iota_2 \twoheadrightarrow \kappa_1, \kappa_3}$$

The linking operator $\mathbf{ln}_{a\to I}$ is used to express arcs that transport data values, and defines a fragment in which values from the port $a$ are copied to all the ports in $I$. This is illustrated in Fig. 4(a). The merge operator $\mathbf{merge}_{a_1;\ldots;a_n\to b}$ represents the (almost) symmetrical situation where values from many arcs are combined into a list produced on one port. Here the ordering of $a_1; \ldots; a_n$ in the operator matters, as empahsized by the use of semicolons, and determines the ordering of elements in the result list. This is illustrated in Fig. 4(b) where the numbers on edges depict this ordering. Finally, the composition operator $F \rhd_I G$, illustrated in Fig. 4(c), constructs a fragment by combining the fragments $F$ and $G$ on the interface $I$ so that the values produced by $F$ on this interface are consumed by $G$.

The semantics of the linking workflow graph is defined by:

$$\frac{\begin{array}{c} \{c_1, \ldots, c_n\} = I \qquad \gamma = [\mathbf{in}_a(\lambda_1, v_1)] \cdot \beta_1 \cdot \cdots \cdot [\mathbf{in}_a(\lambda_m, v_m)] \cdot \beta_m \\ \forall_{i=1..m}(\beta_i = [\mathbf{out}_{c_1}(\lambda_i, v_i), \ldots, \mathbf{out}_{c_n}(\lambda_i, v_i)]) \end{array}}{\mathbf{ln}_{a\to I} \Downarrow \gamma}$$

The rule states that there is a valid trace in which the stream from the input port $a$, i.e., its subsequent events $\mathbf{in}_a(\lambda_1, v_1), \ldots, \mathbf{in}_a(\lambda_m, v_m)$, is read and written to each of the output ports in $I$, and this is decoded by corresponding subtraces $\beta_1, \ldots, \beta_m$. The output ports in $I$ are chosen in an arbitrary order but the streams are not reordered.

The semantics of the merge operator is defined by:

$$\frac{\begin{array}{c} n > 0 \qquad \forall_{j=1..m}(1 \leqslant i_j \leqslant n) \\ \alpha \in \Vert\!|_{j=1..m}[\mathbf{in}_{a_{i_j}}(\lambda_j, v_j), \mathbf{out}_b(i_j.\lambda_j, v_j)] \end{array}}{\mathbf{merge}_{a_1;\ldots;a_n\to b} \Downarrow \alpha} \qquad \frac{}{\mathbf{merge}_{\to b} \Downarrow \mathbf{out}_b(\epsilon, [\,])}$$

Here we have $n$ input ports on which $m$ values arrive. $\forall_{j=1..m}$ $(1 \leqslant i_j \leqslant n)$ specifies a port on which every single of those values arrives, i.e., value $v_j$ is assumed to arrive on port $a_{i_j}$. Then in the trace $\alpha$ the value from port $a_{i_j}$ is outputed as the $i_j$th element of the result list. Thus the ordering of the result list is determined by the ordering of the input ports. Nevertheless the order of events in the stream can change. This is because $\alpha$ is constructed by interleaving, so this definition allows the merge to buffer certain events. The second rule covers the case of $n = 0$ where the merge becomes the empty list constructor.

The semantics of the composition operation is based on the notion of *composing* traces for a certain set of port names $I$. This means that two traces are interleaved such that the output events of one trace for the ports in $I$ coincide with the input events for these ports in the other trace. For example, consider the traces $\alpha = [\mathbf{in}_a(\epsilon, v_1), f(t_1) \mapsto t_2, \mathbf{out}_b(1, v_2), g(t_2) \mapsto t_4, \mathbf{out}_a(\epsilon, v_3), \mathbf{out}_b(2, v_4), \mathbf{in}_b(\epsilon, v_5), \mathbf{out}_c(\epsilon, v_6)]$ and $\beta = [\mathbf{in}_d(\epsilon, v_7), h(t_3) \mapsto t_4, \mathbf{in}_b(1, v_2), f(t_6) \mapsto t_7, \mathbf{out}_b(\epsilon, v_8), \mathbf{in}_b(2, v_4), \mathbf{in}_c(\epsilon, v_6), g(t_8) \mapsto t_9, \mathbf{out}_c(\epsilon, v_9)]$. The output events of $\alpha$ for the

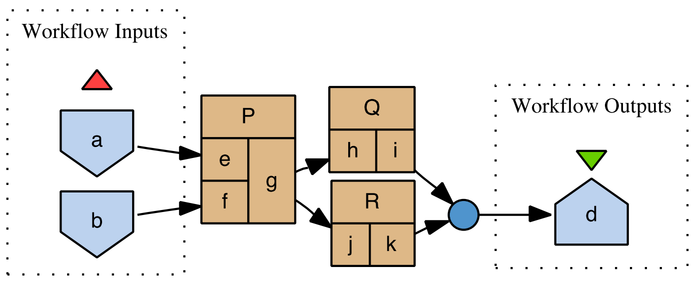


**Fig. 5.** A basic workflow graph to illustrate the representation in calculus.

port names in $\{b,c\}$ are $[\mathbf{out}_b(1,v_2),\mathbf{out}_b(2,v_3),\mathbf{out}_c(\epsilon,v_4)]$. The input events in $\beta$ for this set of port names are $[\mathbf{in}_b(1,v_2),\mathbf{in}_b(2,v_3),\mathbf{in}_c(\epsilon,v_4)]$. Observe that the considered output events of $\alpha$ exactly match the considered input events of $\beta$ in the same order. This means that if $\alpha$ and $\beta$ are traces of fragments $F$ and $G$ (Fig. 4(c)), where indeed the ports $b$ and $c$ are connected, then $\alpha$ could describe a run of $F$, while $\beta$ describes a possible simultaneous run of $G$. A run of such a composition of $F$ and $G$ could be constructed by interleaving $\alpha$ and $\beta$ such that the corresponding output and input events coincide and then removing these output and input events. To illustrate this, we first identify the subtraces $\alpha$ and $\beta$ that lay between the relevant output and input events:

| | $\alpha$ | $\beta$ |
|---|---|---|
| 1 | $[\mathbf{in}_a(\epsilon,v_1), f(t_1)\mapsto t_2]$ | $[\mathbf{in}_d(\epsilon,v_7), h(t_3)\mapsto t_4]$ |
| – | $[\mathbf{out}_b(1,v_2)]$ | $[\mathbf{in}_b(1,v_2)]$ |
| 2 | $[g(t_2)\mapsto t_4, \mathbf{out}_a(\epsilon,v_3)]$ | $[f(t_6)\mapsto t_7, \mathbf{out}_b(\epsilon,v_8)]$ |
| – | $[\mathbf{out}_b(2,v_4)]$ | $[\mathbf{in}_b(2,v_4)]$ |
| 3 | $[\mathbf{in}_b(\epsilon,v_5)]$ | $[\,]$ |
| – | $[\mathbf{out}_c(\epsilon,v_6)]$ | $[\mathbf{in}_c(\epsilon,v_6)]$ |
| 4 | $[\,]$ | $[g(t_8)\mapsto t_9, \mathbf{out}_c(\epsilon,v_9)]$ |

Here the numbered rows indicate the subtraces that will be interleaved and the unnumbered rows indicate the events on which the traces are synchronized. A possible result could for example be $[\mathbf{in}_a(\epsilon,v_1),\mathbf{in}_d(\epsilon,v_7),h(t_3)\mapsto t_4, f(t_1)\mapsto t_2]\cdot[f(t_6)\mapsto t_7, g(t_2)\mapsto t_4,\mathbf{out}_b(\epsilon,v_8),\mathbf{out}_a(\epsilon,v_3)]\cdot[\mathbf{in}_b(\epsilon,v_5)]\cdot[g(t_8)\mapsto t_9,\mathbf{out}_c(\epsilon,v_9)]$.

The *trace composition operator*, denoted $\alpha \circ_I \beta$, describes all possible compositions of traces $\alpha$ and $\beta$ that synchronize on the port names in $I$. We define the set $\alpha \circ_I \beta$ using the following inference rule:

$$\frac{\begin{array}{c}\{\mathbf{out}_{b_1}(\lambda_1,v_1),\ldots,\mathbf{out}_{b_m}(\lambda_m,v_m)\}=\alpha|^I \qquad \{\mathbf{in}_{b_1}(\lambda_1,v_1),\ldots,\mathbf{in}_{b_m}(\lambda_m,v_m)\}=\beta|_I \\ \alpha=\alpha_1\cdot[\mathbf{out}_{b_1}(\lambda_1,v_1)]\cdot\cdots\cdot\alpha_m\cdot[\mathbf{out}_{b_m}(\lambda_m,v_m)]\cdot\alpha_{m+1} \\ \beta=\beta_1\cdot[\mathbf{in}_{b_1}(\lambda_1,v_1)]\cdot\cdots\cdot\beta_m\cdot[\mathbf{in}_{b_m}(\lambda_m,v_m)]\cdot\beta_{m+1} \qquad \forall_{i=1..m+1}(\gamma_i\in(\alpha_i\parallel\!\mid\beta_i))\end{array}}{\gamma_1\cdot\cdots\cdot\gamma_{m+1}\in(\alpha\circ_I\beta)}$$

The first two premises of the rule identify the output and input events that should be synchronized, i.e., $\alpha|^I$ and $\beta|_I$. The next two premises identify the subtraces between the events that are synchronized, namely $\alpha_1,\ldots,\alpha_{m+1}$ and $\beta_1,\ldots,\beta_{m+1}$. The final premise states that $\gamma_1$ is an interleaving of $\alpha_1$ and $\beta_1$, $\gamma_2$ is an interleaving of $\alpha_2$ and $\beta_2$, etc. Then, in the conclusion, it is stated that the string $\gamma_1\cdot\cdots\cdot\gamma_{m+1}$ is in the set $\alpha\circ_I\beta$.

With the help of the trace composition operator, we can now define the semantics of the fragment composition operator, as follows.

$$\frac{F\Downarrow\alpha \qquad G\Downarrow\beta \qquad \gamma\in\alpha\circ_I\beta}{(F\rhd_I G)\Downarrow\gamma}$$

## 4.2. Representing basic Taverna 2 workflow graphs

The preceding operators are sufficient to model the semantics of simple Taverna 2 workflow graphs, such as the one shown in Fig. 5. In this workflow graph the small circle indicates merging of values arriving from different edges into a list. In Fig. 6 we illustrate how this workflow graph can be represented in the calculus.

We assume that the behavior of the processors P, Q and R as fragments is described by the calculus expressions $F_\mathsf{P}$, $F_\mathsf{Q}$ and $F_\mathsf{R}$. Then the total workflow graph can be represented as $F_7=(((((\mathbf{ln}_{a\to e}\rhd_\emptyset\mathbf{ln}_{b\to f})\rhd_{e,f}F_\mathsf{P})\rhd_g\mathbf{ln}_{g\to h,j})\rhd_h F_\mathsf{Q})\rhd_j F_\mathsf{R})\rhd_{i,k}$ $\mathbf{merge}_{i,k\to d}$. This expression builds up the workflow graph from left to right. It starts with the edge connecting input $a$ to input port $e$, which translates to $F_1=\mathbf{ln}_{a\to e}$. Note that this represents a workflow graph with inputs $\{a\}$ and outputs $\{e\}$. This is combined with the other edge from $b$ to $f$, giving $F_2=F_1\rhd_\emptyset\mathbf{ln}_{b\to f}$. Note that the operator $\rhd_\emptyset$ indicates that $F_1$ and

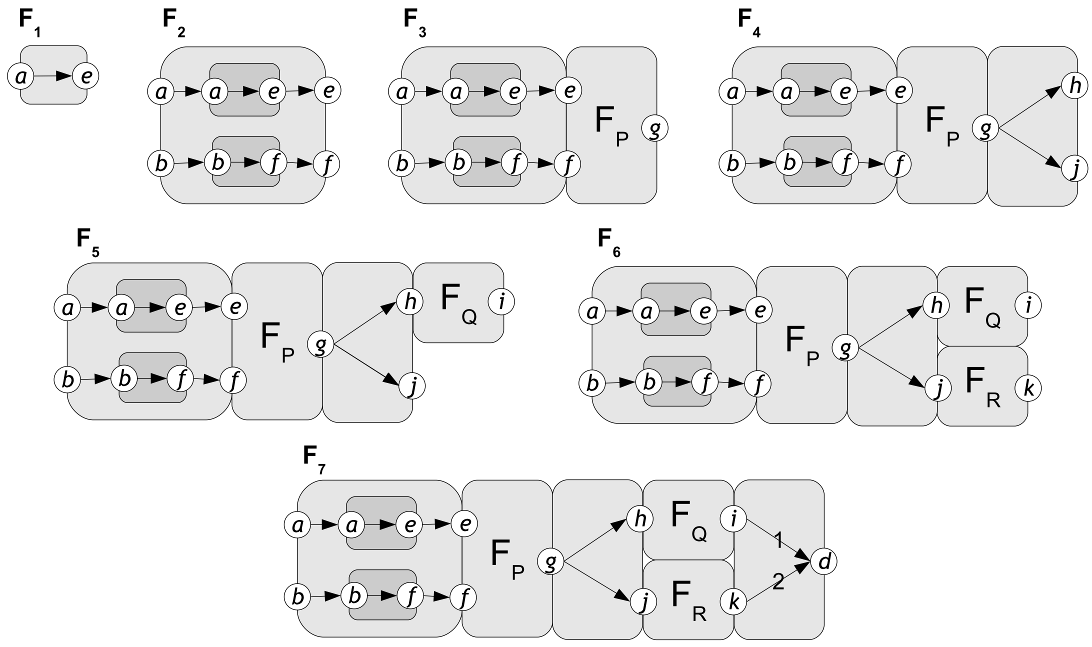


**Fig. 6.** Illustration of a calculus expression for the workflow graph from Fig. 5.

$\mathbf{ln}_{b\to f}$ do not synchronize on any ports, and so operate in parallel. Also note that $F_2$ has inputs $\{a,b\}$ and outputs $\{e,f\}$. In the next step we construct $F_3 = F_2 \rhd_{e,f} F_{\mathsf{P}}$ which connects the output ports $\{e,f\}$ of $F_2$ with the input ports $\{e,f\}$ of $F_{\mathsf{P}}$ and therefore has outputs $\{g\}$. We then construct $F_4 = F_3 \rhd_g \mathbf{ln}_{g\to h,j}$ with outputs $\{h,j\}$ being copies of the output $g$ of $F_3$. Then $F_5 = F_4 \rhd_h F_{\mathsf{Q}}$ has outputs $\{i,j\}$, $F_6 = F_5 \rhd_j F_{\mathsf{R}}$ has outputs $\{i,k\}$, and finally $F_7 = F_6 \rhd_{i,k} \mathbf{merge}_{i,k\to d}$ has outputs $\{d\}$.

Let us illustrate how to use the semantic rules to derive a possible trace of the expression. We assume that $\mathsf{P}$ executes a basic activity $\mathsf{f_P}$ which performs an addition over its inputs, and so $F_{\mathsf{P}} \Downarrow [\mathbf{in}_e(\epsilon,3), \mathbf{in}_f(\epsilon,4), \mathsf{f_P}(\langle e=3, f=4\rangle) \mapsto \langle g=7\rangle, \mathbf{out}_g(\epsilon,7)]$. We assume that $\mathsf{Q}$ doubles its input and therefore $F_{\mathsf{Q}} \Downarrow [\mathbf{in}_h(\epsilon,7), \mathsf{f_Q}(\langle h=7\rangle) \mapsto \langle i=14\rangle, \mathbf{out}_i(\epsilon,14)]$. Finally we assume that $\mathsf{R}$ squares its input, which means that for example $F_{\mathsf{R}} \Downarrow [\mathbf{in}_j(\epsilon,7), \mathsf{f_R}(\langle j=7\rangle) \mapsto \langle k=49\rangle, \mathbf{out}_k(\epsilon,49)]$. We can now derive a trace for $F_7$ as follows. For $F_1 = \mathbf{ln}_{a\to e}$ it holds that $F_1 \Downarrow [\mathbf{in}_a(\epsilon,3), \mathbf{out}_e(\epsilon,3)]$, and similarly $\mathbf{ln}_{b\to f} \Downarrow [\mathbf{in}_b(\epsilon,4), \mathbf{out}_f(\epsilon,4)]$. It then follows by applying the rule for $\rhd_I$ with $I=\emptyset$ that $F_2 = (F_1 \rhd_\emptyset \mathbf{ln}_{b\to f}) \Downarrow [\mathbf{in}_b(\epsilon,4), \mathbf{in}_a(\epsilon,3), \mathbf{out}_e(\epsilon,3), \mathbf{out}_f(\epsilon,4)]$. Clearly the output events for ports $e$ and $f$ in this final trace match with the input events for ports $e$ and $f$ in the mentioned trace of $F_{\mathsf{P}}$, i.e., they happen in the same order for the same ports, locations and values, and so the $\rhd_{e,f}$ operator can combine them, which allows us to derive that $F_3 = (F_2 \rhd_{e,f} F_{\mathsf{P}}) \Downarrow [\mathbf{in}_b(\epsilon,4), \mathbf{in}_a(\epsilon,3), \mathsf{f_P}(\langle e=3, f=4\rangle) \mapsto \langle g=7\rangle, \mathbf{out}_g(\epsilon,7)]$. Since $\mathbf{ln}_{g\to h,j} \Downarrow [\mathbf{in}_g(\epsilon,7), \mathbf{out}_h(\epsilon,7), \mathbf{out}_j(\epsilon,7)]$ it follows that $F_4 = (F_3 \rhd_g \mathbf{ln}_{g\to h,j}) \Downarrow [\mathbf{in}_b(\epsilon,4), \mathbf{in}_a(\epsilon,3), \mathsf{f_P}(\langle e=3, f=4\rangle) \mapsto \langle g=7\rangle, \mathbf{out}_h(\epsilon,7), \mathbf{out}_j(\epsilon,7)]$. Continuing this for the whole expression we can for example derive that $F_7 \Downarrow [\mathbf{in}_b(\epsilon,4), \mathbf{in}_a(\epsilon,3), \mathsf{f_P}(\langle e=3, f=4\rangle) \mapsto \langle g=7\rangle, \mathsf{f_Q}(\langle h=7\rangle) \mapsto \langle i=14\rangle, \mathsf{f_R}(\langle j=7\rangle) \mapsto \langle k=49\rangle, \mathbf{out}_d(1,14), \mathbf{out}_d(2,49)]$. Note that since the final result is a list with two elements this is encoded as a stream with two events for port $d$.

## 4.3. Activities

The syntax of basic activities and translations between activities and fragments is given by:

$$\frac{f \in \mathcal{B} \qquad f : \iota \to \kappa}{f : \iota \Rightarrow \kappa} \qquad \frac{A : \iota \Rightarrow \kappa \qquad J = \mathbf{dom}(\kappa)}{\mathbf{fr}_J(A) : \iota \twoheadrightarrow \kappa}$$

The first rule from the left gives the syntax for the basic activities for which, recall from Section 2, we assume $f$ to be the basic activity name with an input interface $\iota$ and output interface $\kappa$. The second rule defines the fragment operator which transforms an activity into a fragment, which means that it converts fail events into error values for all the output ports. Here $J$ denotes the output interface of the activity and the resulting fragment.[4] We do, however, need to state this, using the following sub-typing rule:

[4] Note that we do not need an additional operator that turns a fragment into an activity, because fragments *are* activities by definition.

$$\frac{e : \iota \twoheadrightarrow \kappa}{e : \iota \Rightarrow \kappa}$$

Since basic activities can fail, we need two rules to define their semantics. to deal with a successful execution and with failure, respectively:

$$\frac{(t_1, t_2) \in [\![f]\!] \qquad \mathbf{in}^*(t_1) \Downarrow \alpha \qquad \mathbf{out}^*(t_2) \Downarrow \beta}{f \Downarrow \alpha \cdot [f(t_1) \mapsto t_2] \cdot \beta} \qquad \frac{\mathbf{in}^*(t_1) \Downarrow \alpha}{f \Downarrow \alpha \cdot [f(t_1) \mapsto \bot] \cdot [\mathbf{fail}]}$$

The semantics of the fragment operator is defined by the following two rules. The first rule states that traces of an activity $A$ that do not contain a fail event are also traces of the corresponding fragment. The second rule describes what happens if the trace does contain a fail event. In this case, the result trace begins with the events in $\alpha$, followed by the output events in $\gamma$ representing an output tuple which contains error values for all output ports in $J$.

$$\frac{A \Downarrow \alpha \ \mathbf{fail} \notin \alpha}{\mathbf{fr}_J(A) \Downarrow \alpha} \qquad \frac{\begin{array}{c} A \Downarrow \alpha \cdot [\mathbf{fail}] \qquad J = \{a_1, \ldots, a_m\} \\ \beta = [\mathbf{out}_{a_1}(\epsilon, \mathbf{err}), \ldots, \mathbf{out}_{a_m}(\epsilon, \mathbf{err})] \end{array}}{\mathbf{fr}_J(A) \Downarrow \alpha \cdot \beta}$$

Note that we use the assumption that failing activities have no output events (so no such events need to be removed), and that the fail event is the last event.

## 5. Iteration strategies and their trace semantics

In Section 2.1 we introduced and motivated iteration strategies. Here we formalize the iteration strategies in terms of calculus operators, and show that those operators are sufficient to represent all the iteration strategies defined in Taverna 2.

### 5.1. Operators for expressing iteration strategies

Only two additional operators are needed to express Taverna 2 iteration strategies, namely $\odot_{I \to J}(F)$ and $\mathbf{rpt}_{a;b \to c}$. Their syntax is given by:

$$\frac{\begin{array}{c} F : \langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle \twoheadrightarrow \langle b_1 : \sigma_1, \ldots, b_m : \sigma_m \rangle \\ n > 0 \qquad I = \{a_1, \ldots, a_n\} \qquad J = \{b_1, \ldots, b_m\} \end{array}}{\odot_{I \to J}(F) : \langle a_1 : [\tau_1], \ldots, a_n : [\tau_n] \rangle \twoheadrightarrow \langle b_1 : [\sigma_1], \ldots, b_m : [\sigma_m] \rangle}$$

$$\frac{}{\mathbf{rpt}_{a;b \to c} : \langle a : \tau, b : [\sigma] \rangle \twoheadrightarrow \langle c : [\tau] \rangle}$$

The **dot product iteration** operator $\odot_{I \to J}(F)$ expects a list on each input port in $I$ and iterates on all of them at the same time, i.e., the fragment $F$ is executed once for every combination of elements that are on the same indexes in the input lists. This means that if the input lists are of unequal length then the unmatched elements from the longer lists are ignored. The repeat operator $\mathbf{rpt}_{a;b \to c}$ repeats the value on port $a$ in a list as often as the list on port $b$ has elements, e.g., it maps the input $\langle a = [1], b = [[2], [3, 4], []] \rangle$ to the output $\langle c = [[1], [1], [1]] \rangle$.

With these two operators we can simulate complex iteration strategies. For example, assume we would like to compute the combinations for the strategy $(a \boxtimes b)$ where for ports $a$ and $b$ the expected port types are s and s and the offered port types are [s] and [s], respectively. If the overall input is, for example $\langle a = [1, 2], b = [3, 4] \rangle$, then we need to produce the pseudo value $[[\langle a = 1, b = 3 \rangle, \langle a = 1, b = 4 \rangle], [\langle a = 2, b = 3 \rangle, \langle a = 2, b = 4 \rangle]]$. Note that this is not really a value because we do not allow tuples in lists, only lists and basic values. However this pseudo value indicates for which combinations the processor will execute, and how the result values are constructed, i.e., each tuple is replaced with the result for that combination. We can encode this pseudo value as two real values, one containing the $a$ values, and the other the $b$ values. The first value would be $[[1, 1], [2, 2]]$ and the second $[[3, 4], [3, 4]]$. The first value can be produced by $\mathbf{ln}_{a \to a, a'} \rhd_{a', b} \mathbf{rpt}_{b; a' \to c} \rhd_{a, c} \odot_{a, c \to a}(\mathbf{rpt}_{a; c \to a})$ and the second value by $\mathbf{rpt}_{b; a \to b}$. As will be illustrated later, this is in fact possible for any iteration strategy and combination of expected and offered port types.

The semantics of the dot product iteration operator is given by three rules. We start with the case where all the input lists are non-empty:

$$\frac{\begin{array}{c} p > 0 \qquad \forall_{j=1..p}(F \Downarrow \alpha_j) \qquad \alpha \in |||_{j=1..p}(j.\alpha_j) \qquad \{c_1\} \cup \cdots \cup \{c_k\} \subsetneq \{a_1, \ldots, a_n\} \\ \beta = [\mathbf{in}_{c_1}(\lambda_1, v_1), \ldots, \mathbf{in}_{c_k}(\lambda_k, v_k)] \qquad \delta \in \alpha \ ||| \ \beta \end{array}}{\odot_{a_1, \ldots, a_n \to b_1, \ldots, b_m}(F) \Downarrow \delta}$$

Here $p$ is the length of the shortest of the input lists, the trace $\alpha_j$ for each $j \in 1, \ldots, p$ is the trace of the execution of fragment $F$ for position $j$, the event $\mathbf{in}_{c_i}(\lambda_i, v_i)$ for each $i \in 1, \ldots, k$ is an input events that describes parts of the input lists beyond the first $p$ elements and will therefore be ignored. The trace of the dot product iteration is then composed from (i) $\alpha$, which is a combination of traces $\alpha_j$ extended in such a way that the input and output events refer to sub-values of the value at position $j$, and (ii) $\beta$ which includes the possible additional input events trimmed from longer input lists.

The second rule deals with the case where there is at least one empty input list, which means that an empty list is also produced on all the output ports. Observe that the result, encoded by $\beta$, can be produced as soon as an empty list appears on any of the input ports.

$$\frac{\begin{array}{c}\mathbf{in}^*(\langle a_1 = v_1, \ldots, a_n = v_n\rangle) \Downarrow \alpha \qquad v_i = [\,] \\ \alpha = \alpha_1 \cdot [\mathbf{in}_{a_i}(\epsilon, [\,])] \cdot \alpha_2 \qquad \mathbf{out}^*(\langle b_1 = [\,], \ldots, b_p = [\,]\rangle) \Downarrow \beta \qquad \gamma \in \alpha_2 \parallel\!| \beta\end{array}}{\odot_{a_1,\ldots,a_n \to b_1,\ldots,b_m}(F) \Downarrow \alpha_1 \cdot [\mathbf{in}_{a_i}(\epsilon, [\,])] \cdot \gamma}$$

The final rule deals with the case where there are no empty input lists, but at least one error value, which means that error values are produced on all the output ports. Here $\beta$ starts after at least one input event for every input port has arrived in $\alpha_1$, because at that moment it is clear that on port $a_i$ there arrived an error value and that none of the input values is an empty list.

$$\frac{\begin{array}{c}\mathbf{in}^*(\langle a_1 = v_1, \ldots, a_n = v_n\rangle) \Downarrow \alpha_1 \cdot \alpha_2 \\ \forall_{j=1..n}(v_j \neq [\,]) \qquad \forall_{j=1..n}\exists_{\lambda, v'}(\mathbf{in}_{a_j}(\lambda, v') \in \alpha_1) \\ v_i = \mathbf{err} \qquad \mathbf{out}^*(\langle b_1 = \mathbf{err}, \ldots, b_p = \mathbf{err}\rangle) \Downarrow \beta \qquad \gamma \in \alpha_2 \parallel\!| \beta\end{array}}{\odot_{a_1,\ldots,a_n \to b_1,\ldots,b_m}(F) \Downarrow \alpha_1 \cdot \gamma}$$

Finally, the semantics for the repeat operator is given by:

$$\frac{\begin{array}{c}\mathbf{in}^*(\langle a = v_a, b = [v_1, \ldots, v_n]\rangle) \Downarrow \gamma_1 \cdot \cdots \cdot \gamma_{m+1} \\ \mathbf{out}^*(\langle c = \bullet_{k=1..n}[v_a]\rangle) \Downarrow [\mathbf{out}_c(i_1.\lambda_1, w_1), \ldots, \mathbf{out}_c(i_m.\lambda_m, w_m)] \\ \forall_{j=1..m}(\exists_{w',\lambda'}(\mathbf{in}_b(i_j.\lambda', w') \in \gamma_1 \cdot \cdots \cdot \gamma_j)) \qquad \forall_{j=1..m}(\mathbf{in}_a(\lambda_j, w_j) \in \gamma_1 \cdot \cdots \cdot \gamma_j)\end{array}}{\mathbf{rpt}_{a;b\to c} \Downarrow \gamma_1 \cdot [\mathbf{out}_c(i_1.\lambda_1, w_1)] \cdot \gamma_2 \cdot \cdots \cdot [\mathbf{out}_c(i_m.\lambda_m, w_m)] \cdot \gamma_{m+1}}$$

The first precondition specifies that input events in the concatenation of $\gamma_1, \ldots, \gamma_{m+1}$ encode exactly the input tuple. The second gives $m$ output events encoding the list with $v_a$ repeated $n$ times, from which it follows that the sequence of pairs $(\lambda_i, w_i)$ includes $n$ repetitions of the values needed to encode $v_a$. The output trace is constructed by interleaving the subsequent parts of the input trace with the subsequent output events. The last two preconditions ensure that an output event $\mathbf{out}_c(i_j.\lambda_j, w_j)$ is produced only after (i) on the input $b$ we have seen the index $i_j$ and (ii) on the input $a$ we have seen the relevant piece of $v_a$, i.e., $(\lambda_j, w_j)$. Note that the final part of the output trace $\gamma_{m+1}$ is necessary because each of the values $v_1, \ldots, v_n$ might be encoded with more than one input event, but to produce the output event for its index we only need to have consumed one of those input events. Note also that this definition is not as greedy as it could be, because in principle the operator could output events encoding $v_a$ on all the positions $1, \ldots, i$, if there was an input event for $v_{i+1}$, but the given definition matches the current implementation observed in Taverna 2.

### *5.2. Translating iteration strategy expressions*

In this section we show that the presented operators and primitives are sufficient to represent all the iteration strategies in Taverna. As an example, consider a processor with input interface type $\langle a : \mathrm{s}, b : \mathrm{s}, c : \mathrm{s}\rangle$ which is offered values of type [s], [s] and [[s]] on ports $a$, $b$ and $c$, respectively, with iteration strategy $(a \boxtimes b) \boxdot c$. The semantics of the iteration strategy is defined in terms of a calculus expression $IS_{(a[1]\boxtimes b[1])\boxdot c[2]} : \langle a : [\mathrm{s}], b : [\mathrm{s}], c : [[\mathrm{s}]]\rangle \twoheadrightarrow \langle a : [[\mathrm{s}]], b : [[\mathrm{s}]], c : [[\mathrm{s}]]\rangle$ that consumes the offered values and produces three streams that contain all the combinations over which the processor is to iterate. The integer numbers associated with each port indicate the difference in nesting depth between the *offered type* and the *expected type*. This is a positive number if the offered value is nested too deep and negative if it is not nested deep enough. For example, if the input value is $\langle a = [1, 2], b = [3, 4], c = [[5, 6], [7]]\rangle$ then the output streams of $IS_{(a[1]\boxtimes b[1])\boxdot c[2]}$ encode $\langle a = [[1, 1], [2]], b = [[3, 4], [3]], c = [[5, 6], [7]]\rangle$. The processor then iterates over these three lists, simultaneously processing each time the combinations it finds at a certain position. In this case it processes for position 1.1 the value $\langle a = 1, b = 3, c = 5\rangle$, for position 1.2 the value $\langle a = 1, b = 4, c = 6\rangle$ and for position 2.1 the value $\langle a = 2, b = 3, c = 7\rangle$.

Expressing $IS_{(a[1]\boxtimes b[1])\boxdot c[2]}$ requires first expressing $IS_{a[1]\boxtimes b[1]}$. The output for the $b$ port can be computed as $F_b = \mathbf{rpt}_{b;a\to b}$. Note that this maps $\langle a = [1, 2], b = [3, 4]\rangle$ to $\langle b = [[3, 4], [3, 4]]\rangle$. The output for the $a$ port can be computed by $F_a = (\mathbf{ln}_{a\to a,a'} \triangleright_{a'} \mathbf{rpt}_{b;a'\to b'}) \triangleright_{a,b'} \odot_{a,b'\to a}(\mathbf{rpt}_{a;b'\to a})$. Given $\langle a = [1, 2], b = [3, 4]\rangle$ the expression $(\mathbf{ln}_{a\to a,a'} \triangleright_{a'} \mathbf{rpt}_{b;a'\to b'})$ produces $\langle a = [1, 2], b' = [[3, 4], [3, 4]]\rangle$ and from this $\odot_{a,b'\to a}(\mathbf{rpt}_{a;b'\to a})$ produces $\langle a = [[1, 1], [2, 2]]\rangle$. It is then not hard to see that $IS_{a[1]\boxtimes b[2]} = \mathbf{ln}_{a\to a,a'} \triangleright_{\emptyset} \mathbf{ln}_{b\to b,b'} \triangleright_{a,b} F_b \triangleright_{a',b'} (\mathbf{ln}_{a'\to a} \triangleright_{\emptyset} \mathbf{ln}_{b'\to b} \triangleright_{a,b} F_a)$. Indeed, it can be shown that it is possible to express an operator $\otimes^{i,j}_{I;J}$ that computes the cross product iteration strategy assuming that the left-hand side is encoded in the ports in the set $I$ at nesting depth $i$ and the right-hand side is in ports $J$ at depth $j$.

The next step in expressing $IS_{(a[1]\boxtimes b[1])\boxdot c[2]}$ consists in combining the output of $IS_{a[1]\boxtimes b[1]}$ with that of port $c$. Note that this output encodes the iteration values at nesting depth 2. So we need to take the dot product of ports $a$, $b$ and $c$ at nesting depth 2, which can be done by nesting the iteration operator twice and applying it to the identity fragment: $\odot_{a,b,c \to a,b,c}(\odot_{a,b,c\to a,b,c}(\mathbf{ln}_{a\to a} \rhd_\emptyset \mathbf{ln}_{b\to b} \rhd_\emptyset \mathbf{ln}_{c\to c}))$. Note that this indeed maps $\langle a = [[1,1],[2,2]], b = [[3,4],[3,4]], c = [[5,6],[7]]\rangle$ to $\langle a = [[1,1],[2]], b = [[3,4],[3]], c = [[5,6],[7]]\rangle$. Also here it is possible to express a general operator $\odot^i_{I;J}$ that computes the dot product iteration strategy assuming that the left-hand side is encoded in the ports in $I$, the right-hand side in ports $J$ and at depth $i$ for all ports. At the time of writing Taverna 2 indeed requires that at the point that the dot product is computed the nesting depth of the expected values is on both sides equal; a workflow graph in which this is not the case is considered incorrect. However, in future versions this might be taken care of by inserting extra list-wrapping operators that increase the nesting depth.

A complete iteration strategy annotated with positive nesting depth differences can be relatively straightforwardly translated using the operators $\otimes^{i,j}_{I;J}$ and $\odot^i_{I;J}$, e.g., $IS_{(a[1]\boxtimes b[1])\boxdot c[2]}$ is translated to $\otimes^{1,1}_{a;b} \rhd_{a,b,c} \odot^2_{a,b;c}$. If the nesting depth differences are negative then this is remedied by inserting extra wrapping operations that increase the nesting depth. For example, $IS_{a[-1]\boxtimes b[1]} = \mathbf{merge}_{a\to a} \rhd_a IS_{a[0]\boxtimes b[1]}$.

In the complete processor semantics, the iteration expression is followed by the actual iteration of the processor over the computed combinations. Observe that in this case this means that it iterates at nesting depth 2. For this we introduce the notation $\odot^i_{I\to J}(F)$ which indicates the iteration at nesting depth $i$ and is defined such that $\odot^0_{I\to J}(F) = F$ and $\odot^{i+1}_{I\to J}(F) = \odot_{I\to J}(\odot^i_{I\to J}(F))$. So in the example presented earlier, assuming that the body of the processor is represented by the expression $F' : \langle a : \mathrm{s}, b : \mathrm{s}, c : \mathrm{s}\rangle \twoheadrightarrow \langle d : [\mathrm{s}]\rangle$, its total semantics becomes $IS_{(a[1]\boxtimes b[1])\boxdot c[2]} \rhd_{a,b,c} \odot^2_{a,b,c\to d}(F')$.

## 6. The dispatch stack

As mentioned in the introduction, one of the interesting features of the new Taverna 2 model is the extensibility of the process activation sequence. In practice, each processor $P$ in the graph is configured by means of a stack of execution layers and the processing that takes place when the workflow execution reaches $P$ depends on this configuration. Each layer in the stack receives a request from the layer above, it performs a certain function that transforms the request and then it either forwards the request to the layer below or it "bounces" it back up the stack as a response. Taverna 2 comes with a choice of prebuilt *standard* layers but the stack can also include *custom* layers of a user provided type. This gives a way to implement additional control operators in Taverna 2.

Each processor's stack is configured independently from the others, using any of the available layers. This makes it possible to associate a different behavior to different nodes in the graph. The semantics of each processors's execution is completely defined by the semantics of each layer, along with the interaction amongst the layers in the stack.

The layers in the stack can be variously configured, but at the bottom is always the **Invoke** layer, which is responsible for launching the execution of a basic activity and collecting its result. When the activity is a Web Service, for example, this layer implements a client that initiates the service invocation, collects its result and detects error conditions. Other layers perform additional tasks, intended mostly to provide quality of service to the execution. The $\mathbf{Retry}_k$ layer, for example, accounts for the possibility that a service is temporarily unreachable, the **Bounce** layer returns a failure state without forwarding the request to the layer below it, if the incoming request from layer contains an error, while the **Failover** layer tries several alternative activities in turn, hoping that one of them will succeed. We are going to describe the function of each layer more precisely in the rest of the section. Specifically, we show how the behavior of each standard layer can be formalized in terms of our trace semantics. We then provide a formal definition for two additional custom layers, namely the **Branch** layer and the **Loop** layer, which can be used to add new control flow operators to Taverna, an if-then-else and a while-loop operator respectively. This shows how one can exploit the stack-based architecture to extend the workflow definition language of Taverna 2 and furthermore how such extensions can be described within the trace semantics framework.

### 6.1. Dispatch stack activities

In the following we describe the semantics of the dispatch stack in terms of activities, i.e., given a list of activities $\mathcal{A} = [A_1, \dots, A_n]$ and a dispatch stack $S = [L_1, \dots, L_m]$ we describe the behaviour of activity $S(\mathcal{A})$. Later on in Section 7 we show how the total behavior of a processor is described based on this, but in terms of fragments and taking the iteration strategy into account.

The syntax for the dispatch stack layers is as follows:

$$\frac{A_1 : \iota \Rightarrow \kappa}{[\,]([A_1]) : \iota \Rightarrow \kappa} \qquad \frac{S(\mathcal{A}) : \iota \Rightarrow \kappa}{[\mathbf{Retry}_k | S\,](\mathcal{A}) : \iota \Rightarrow \kappa}$$

$$\frac{\mathcal{A} = [A_1, \dots, A_n] \qquad \forall_{i=1..n}(S([A_i]) : \iota \Rightarrow \kappa)}{[\mathbf{Failover} | S\,](\mathcal{A}) : \iota \Rightarrow \kappa} \qquad \frac{S(\mathcal{A}) : \iota \Rightarrow \kappa}{[\mathbf{Bounce} | S\,](\mathcal{A}) : \iota \Rightarrow \kappa}$$

$$\frac{\mathcal{A}=[A_1,\ldots,A_n] \quad \forall_{i=1..n}(S([A_i]) : \iota \Rightarrow \kappa) \quad c \in \mathbf{dom}(\iota)}{[\mathbf{Branch}_c|S](\mathcal{A}) : \iota \Rightarrow \kappa} \qquad \frac{S(\mathcal{A}) : \iota \Rightarrow \iota \quad c \in \mathbf{dom}(\iota)}{[\mathbf{Loop}_c|S](\mathcal{A}) : \iota \Rightarrow \iota}$$

Here we adopt the notation $[H|T]$ to denote a list with head $H$ and tail $T$. In the following we describe the behavior of each layer in the same order.

Note that, if a stack does not include either e **Failover** or **Branch**$_c$ layers, then its activity list must be a singleton, because the bottom **Invoke** layer requires exactly on activity. Note also that, when **Failover** or **Branch**$_c$ are present, the activity list can be of an arbitrary length, even empty, yet all activities in it will be of the same type.

## 6.2. Standard layers

The **Invoke** layer expects a single activity in the list, and executes it. Since it is always required as the last layer it is not represented in the formal definition: its behavior is described by the behavior of the empty stack.

$$\frac{S=[\,] \quad \mathcal{A}=[A_1] \quad A_1 \Downarrow \alpha}{S(\mathcal{A}) \Downarrow \alpha}$$

The role of the **Retry**$_k$ layer is to submit the activities in its input list at most $k$ times to the layers below it. The layer returns with failure, if the layers below it in the stack fail all $k$ times, and it returns with success as soon as one of the attempts succeeds.

$$\frac{S=[\mathbf{Retry}_0|R] \; \mathbf{in}^*(t) \Downarrow \alpha}{S(\mathcal{A}) \Downarrow \alpha \cdot [\mathbf{fail}]} \qquad \frac{S=[\mathbf{Retry}_k|R] \quad k>0 \quad R(\mathcal{A}) \Downarrow \alpha \quad \mathbf{fail} \notin \alpha}{S(\mathcal{A}) \Downarrow \alpha}$$

$$\frac{S=[\mathbf{Retry}_k|R] \quad k>0 \quad R(\mathcal{A}) \Downarrow \alpha \cdot [\mathbf{fail}] \quad S'=[\mathbf{Retry}_{k-1}|R] \quad S'(\mathcal{A}) \Downarrow \beta \quad \beta \in \beta_1 \,|||\, \beta_2 \quad \alpha|_{\mathcal{P}} = \beta|_{\mathcal{P}} = \beta_1|}{S(\mathcal{A}) \Downarrow \alpha \cdot \beta_2}$$

The first rule explicitly allows that 0 retries are specified, which means that the processor always fails after consuming the whole input. The second rule describes a successful execution where the trace of the layers remaining in $R$ does not contain a failure and becomes the result trace. The third rule deals with the case where the trace of the layers remaining in $R$ contains a failure, i.e., equals $\alpha \cdot [\mathbf{fail}]$. The resulting trace is obtained by removing the failure event and replacing it with a new trace $\beta_2$. This new trace is obtained by removing the input events from a possible trace $\beta$ of $S'(\mathcal{A})$ where $S'$ is equal to $S$ except the number of retries is decreased by one. The equation $\alpha|_{\mathcal{P}} = \beta|_{\mathcal{P}}$ guarantees that $\beta$ includes the same input events as $\alpha$, albeit possibly in some different order. The part of $\beta$ that is included in the resulting trace, namely $\beta_2$, contains all events in $\beta$ except the input events. This is guaranteed by $\beta|_{\mathcal{P}} = \beta_1|$ which implies that all input events of $\beta$ are in $\beta_1$ and therefore not in $\beta_2$.

The role of the **Failover** layer is to try and submit each activity in the input list to the layer below. It returns with success when an activity succeeds, and it fails when all activities fail. This is similar to the **Retry**$_k$ layer, except that the number of attempts is determined by the length of the list $\mathcal{A}$, rather than by a user-defined parameter $k$.

In principle, this layer may implement one of a number of strategies to prioritize the activities. We illustrate the simple model where the activities are tried in the order in which they appear in the list.

$$\frac{S=[\mathbf{Failover}|R] \quad \mathcal{A}=[\,] \quad \mathbf{in}^*(t) \Downarrow \alpha}{S(\mathcal{A}) \Downarrow \alpha \cdot [\mathbf{fail}]} \qquad \frac{S=[\mathbf{Failover}|R] \quad \mathcal{A}=[A|\mathcal{B}] \quad R([A]) \Downarrow \alpha, \quad \mathbf{fail} \notin \alpha}{S(\mathcal{A}) \Downarrow \alpha}$$

$$\frac{S=[\mathbf{Failover}|R] \quad \mathcal{A}=[A|\mathcal{B}] \quad R([A]) \Downarrow \alpha \cdot [\mathbf{fail}] \quad S(\mathcal{B}) \Downarrow \beta \; \beta \in \beta_1 \,|||\, \beta_2 \quad \alpha|_{\mathcal{P}} = \beta|_{\mathcal{P}} = \beta_1|}{S(\mathcal{A}) \Downarrow \alpha_1 \cdot \beta_2}$$

The rules for the **Failover** layer are analogous those for the **Retry**$_k$ layer.

The **Bounce** layer returns a failure state without forwarding the request to the layer below it, if the incoming request from the previous layer contains an error. Otherwise it returns the value provided by the layer below it.

$$\frac{S=[\mathbf{Bounce}|R] \quad \mathbf{in}^*(t) \Downarrow \alpha \quad \exists_{a,\lambda}(\mathbf{in}_a(\lambda,\mathbf{err}) \in \alpha)}{S(\mathcal{A}) \Downarrow \alpha \cdot [\mathbf{fail}]} \qquad \frac{S=[\mathbf{Bounce}|R] \quad R(\mathcal{A}) \Downarrow \alpha \quad \neg\exists_{a,\lambda}(\mathbf{in}_a(\lambda,\mathbf{err}) \in \alpha)}{S(\mathcal{A}) \Downarrow \alpha}$$

The left rule deals with the case where the value on some input port $a$ contains an error on any index $\lambda$. In this case, the request is bounced off and a fail event is appended to the complete input. The right rule deals with the complementary case where non of the input values was an error nor contained an error as any of its subvalues. The resulting trace is then the complete trace of the rest of the stack $R(\mathcal{A})$.

## 6.3. Extension layers

So far we have described the layers that comprise the standard configuration of the dispatch stack. A powerful feature of Taverna 2 is its ability to accept new layers as part of a stack's configuration. As each processor's layer is configured independently from the others, new layers can be selectively added to provide specific processors with new functionality. Here we describe two such layers, which add control operator functionality to Taverna 2.

The first of the two, the $\mathbf{Loop}_c$ layer, is used to implement a conditional loop processor $P$. We assume that the input and output ports of $P$ are identical and include one distinguished port named $c$. If the layer receives a request with an interface value $t$ where $t(c)$ is true, then it forwards the request to the next layer. If this returns a result then it feeds this back as a request to itself, but if there is no result it returns failure. If however in the request $t(c)$ is not true, then the input is immediately returned unchanged.

$$\frac{\begin{array}{c} S = [\mathbf{Loop}_c|R] \qquad \alpha = \alpha_1 \cdot [\mathbf{in}_c(\epsilon, v)] \cdot \alpha_2 \\ v \neq \mathbf{true} \qquad \alpha \in |||_{i=1..n}([\mathbf{in}_{a_i}(\lambda_i, v_i), \mathbf{out}_{a_i}(\lambda_i, v_i)]) \qquad \alpha_1| = \alpha_1|_{\mathcal{P}} \end{array}}{S(\mathcal{A}) \Downarrow \alpha}$$

$$\frac{\begin{array}{c} S = [\mathbf{Loop}_c|R] \qquad R(\mathcal{A}) \Downarrow \alpha \\ \alpha = \alpha_1 \cdot [\mathbf{in}_c(\epsilon, \mathbf{true})] \cdot \alpha_2 \qquad \alpha_1| = \alpha_1|_{\mathcal{P}} \qquad S(\mathcal{A}) \Downarrow \beta \qquad \gamma \in \alpha \circ_{\mathcal{P}} \beta \end{array}}{S(\mathcal{A}) \Downarrow \gamma}$$

The first rule covers the case where the value provided on the port $c$ is not true. The operation described by the result trace $\alpha \in |||_{i=1..n}([\mathbf{in}_{a_i}(\lambda_i, v_i), \mathbf{out}_{a_i}(\lambda_i, v_i)])$ is simply coping the input to output, possibly with some buffering. Note that we do not allow output events in $\alpha_1$, i.e., before we know that the value on port $c$ is not the boolean true.

In the second rule $\alpha$ describes the initial iteration of the loop, which occurs because on port $c$ a boolean true is provided. Here we also not allow output events in $\alpha_1$. The result trace of the whole iteration is obtained by, recall the trace composition operator $\circ_{\mathcal{I}}$ from Section 4.1, composing $\alpha$ with the trace of the remaining iterations of the loop described by $\beta$ on the whole domain of input ports $\mathcal{P}$.

The second of the two custom layers enables workflow designers to create *If-then-else* processors. The layer's behavior is similar to that of the **Loop** layer, in that it assumes a conditional value is bound to a particular variable $c$, i.e., $(c, i) \in t$. This time the value, a positive integer in the range $1 \ldots |\mathcal{A}|$, is used to select the $i$th element $\mathcal{A}(i)$ of list $\mathcal{A}$. This activity is forwarded to the layer below for execution. A value that is out of range results in a failure.

$$\frac{S = [\mathbf{Branch}_c|R] \qquad \alpha| = \alpha|_{\mathcal{P}} \qquad \mathbf{in}_c(\epsilon, v) \in \alpha \qquad v \notin \{1, \ldots, |\mathcal{A}|\}}{S(\mathcal{A}) \Downarrow \alpha \cdot [\mathbf{fail}]}$$

$$\frac{\begin{array}{c} S = [\mathbf{Branch}_c|R] \\ v \in \{1, \ldots, |\mathcal{A}|\} \qquad R([\mathcal{A}(v)]) \Downarrow \alpha \qquad \alpha = \alpha_1 \cdot [\mathbf{in}_c(\epsilon, v)] \cdot \alpha_2 \qquad \alpha_1| = \alpha_1|_{\mathcal{P}} \end{array}}{S(\mathcal{A}) \Downarrow \alpha}$$

The first rule deals with the case where the value $v$ provided on port $c$ is out of range. The resulting trace contains all the input events followed by failure. Note that we allow for the input event for port $c$ to be followed by some other input events. The second rule deals with the case that the value $v$ provided on port $c$ is in the correct range. The result trace is then a valid trace for $R([\mathcal{A}(v)])$, i.e., the remaining layers and the $v$th activity, but we require that in the part of the trace before the input event for port $c$, that is in $\alpha_1$, there are no other events except input events.

## 7. Translating Taverna 2 processor specifications

In this section it is explained how to represent in a calculus expression the behavior of a single processor. Recall that in Section 4.2 it was shown how to represent a workflow graph where the behavior of the individual processors was represented by abstract expressions such as $F_{\mathsf{P}}$, $F_{\mathsf{Q}}$ and $F_{\mathsf{R}}$. Here we explain how these expressions are constructed from the processors' specification.

The complete specification of a processor's behavior in Taverna 2 consists of the following components: (1) the expected types on each of its input ports, (2) the iteration strategy expression, (3) the dispatch stack $S$ and (4) a list of activity specifications. In the default configuration, the stack $S$ is of the form $S = [\mathbf{Bounce}, \mathbf{Failover}, \mathbf{Retry}_k]$. This has the effect that a request submitted to the processor succeeds if at least one of the activities in $\mathcal{A}$ succeeds, possibly after failing at most

$k-1$ times, and it fails otherwise. The additional extension layers, like the $\mathbf{Loop}_c$ layer, are typically placed at the top of the stack. The list of activity specifications can contain basic activity names and workflow graphs.

The construction of the corresponding calculus expression is done in three steps. In the first step we construct an expression for the *core semantics* which describes the behavior of the dispatch stack and the activities list. In the second step we extend this expression such that it describes the *step semantics* which describes the behavior of a single iteration step of the processor. In the last step the expression is again extended such that it describes the *iteration semantics* by incorporating the iteration strategy and describing how the processor iterates over the provided input values.

We begin with the translation of the list of activity specification which is translated to a list of activity expressions $\mathcal{A}$. The basic activity names are simply translated to themselves. Each workflow graph is represented by a fragment expression that gives the semantics of this workflow graph, as explained in Section 4.2. The core semantics of the processor is then described by the activity expression $S(\mathcal{A})$.

The core semantics falls short of describing the step semantics of a processor, i.e., the semantics for a single iteration step, in two ways. The first is that it might still fail, and this can be remedied by applying the fragment operator $\mathbf{fr}_J()$ which replaces failure with the production of error values. The second is that a processor always waits with executing an iteration step until all the input values for that step have completely arrived. To represent this we introduce the input synchronization operator:

$$\frac{F : \iota \twoheadrightarrow \kappa}{\mathbf{sync}(F) : \iota \twoheadrightarrow \kappa} \qquad \frac{F \Downarrow \alpha \qquad \alpha = \alpha_1 \interleave \alpha_2 \qquad \alpha_1| = \alpha|_P = \beta_1|}{\mathbf{sync}(F) \Downarrow \beta_1 \cdot \alpha_2}$$

We then can represent the step semantics of a processor as $\mathbf{sync}(\mathbf{fr}_J(S(\mathcal{A})))$ where $J$ is the set of output ports of the processor.

We now proceed with representing the iteration semantics of a processor. An iteration strategy *is* that is annotated with nesting depth differences can be expressed by an expression $IS_{is}$, as discussed in Section 5.2. The nesting depth difference can be computed from the expected type, which is given, and the offered type, which can be relatively straightforwardly computed. For example, if the input port is connected to an output port of a preceding processor, then it is the produced output type of that output port, and if it is connected to a workflow input then it is the expected type of that workflow input. Assuming that in the result of $IS_{is}$ the expected values are found at nesting depth $i$, the full semantics of the processor can be described as $IS_{is} \rhd_I \odot^i_{I \to J}(\mathbf{sync}(\mathbf{fr}_J(S(\mathcal{A}))))$ where $I$ and $J$ are the set of input ports and output ports, respectively, of the processor.

Together with the translation given in Section 4.2 for the workflow graph that contains the processors, this demonstrates how the semantics of a complete Taverna 2 workflow graph can be represented as a fragment expression in the calculus. As an example consider again the workflow graph in Fig. 5 in Section 4.2. Recall that the whole workflow graph could be represented as $F_7 = (((((\mathbf{ln}_{a \to e} \rhd_\emptyset \mathbf{ln}_{b \to f}) \rhd_{e,f} F_\mathsf{P}) \rhd_g \mathbf{ln}_{g \to h,j}) \rhd_h F_\mathsf{Q}) \rhd_j F_\mathsf{R}) \rhd_{i,k} \mathbf{merge}_{i,k \to d}$. Now assume that we know for processor P that its dispatch stack is the default one, i.e., $S = [\mathbf{Bounce}, \mathbf{Failover}, \mathbf{Retry}_4]$, and its list of activities contains one basic service viz. $s_1$. In that case the core activity of the processor is given by $S(\mathcal{A}) = [\mathbf{Bounce}, \mathbf{Failover}, \mathbf{Retry}_4]([s_1])$. Assume that for P the specified iteration strategy is $(e \boxdot f)$ and that the types of the workflow inputs $a$ and $b$ are [s] and [s], and the expected input types for the ports $e$ and $f$ of service $s_1$ are s and s, respectively. It then follows that for P for ports $e$ and $f$ the offered types are [s] and [s], respectively, and the expected types are s and s, respectively. So the annotated iteration strategy is $(e[1] \boxdot f[1])$ and its corresponding calculus expression is expressed by $\odot^1_{e;f}$ which simply copies the input lists from the input port $e$ to the output port $e$ and the same for the input port $f$ and the output port $f$. Since this encodes the iteration values at nesting depth 1 the full representation of the processor semantics becomes $F_\mathsf{P} = \odot^1_{e,f} \rhd_{e,f} \odot^1_{e,f \to g}(\mathbf{sync}(\mathbf{fr}_g([\mathbf{Bounce}, \mathbf{Failover}, \mathbf{Retry}_4]([s_1]))))$.

A feature that was not discussed is the control link between two processors, which indicates that the sink processor cannot start execution before the source processor, also called the controlling processor, has produced all its output. This can be simulated by adding dummy input ports to the sink processor for each of the output ports of the source processor and connecting them. The expected types of these dummy input ports should be exactly the produced types by the corresponding output port, taking into account the offered types and iteration strategy for the source processor. Adding such dummy input ports can be done by nesting the processor in a nested workflow with an empty dispatch stack and that contains only this processor, and adding the dummy ports as extra unconnected workflow inputs. These unconnected input ports can be represented by a special dummy input port operator $\mathbf{dum}_a$ which defines a fragment with a single input port $a$ of port type $\tau$ and no output ports. Its behavior is simply that it consumes the input values from port $a$ and nothing else. Formally its syntax and semantics is defined by:

$$\frac{}{\mathbf{dum}_a : \langle a : \tau \rangle \twoheadrightarrow \langle\rangle} \qquad \frac{\mathbf{in}^*(\langle a = v \rangle) \Downarrow \alpha}{\mathbf{dum}_a \Downarrow \alpha}$$

Next to representing unconnected workflow inputs this operator also allows us to represent workflow graphs where certain output ports are not connected to any input port or workflow output. In this case the output port can be connected to the input port of a dummy input port operator.

## 8. Conclusion and further research

We have presented in this paper a calculus with trace semantics to define the formal semantics of Taverna 2 workflow graphs. This calculus consists of a limited set of operators that each capture a specific aspect of the execution of workflows represented in these workflow graphs. Specifically they allow the description of the iteration strategies, the pipe-lined execution of processors, and the dispatch stack mechanism that is used in Taverna 2 to configure the specific behavior of a processor. Although the formal definition of the semantics of the presented operators is far from trivial, we claim that their intuitive meaning is not hard to understand and can help to create insight into the exact meaning of Taverna 2 workflow graphs. Moreover, we think it can serve as a starting point for determining whether the behavior of two different workflow graphs is identical.

Although the formalisation of properties like parallel and pipelined execution is complete, aspects of the definition of workflow graphs remain to be formalized, in particular their complete translation into calculus expressions remains the subject of future research. This would include the complete algorithms for deriving the offered types for processor input ports, the produced types for processor output ports and the complete translation of an annotated iteration strategy, which were all only described informally here. In addition also the translation of control links was only described informally and will be formalized in more detail in future work.

## Acknowledgments

The authors would like to acknowledge the support of all members of the $^{my}$Grid team, and are especially grateful to Stian Soiland-Reyes and Stuart Owen for providing precious technical insight into the Taverna model.

## References

[1] W.M.P. van der Aalst, The application of Petri nets to workflow management, J. Circuits Systems Comput. 8 (1) (1998) 21–66.
[2] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Sterling, Dieter König, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, Alex Yiu, Web Services Business Process Execution Language Version 2.0, OASIS Committee Draft, May 2006.
[3] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, Sinisa Zimek, Web service choreography interface (wsci) 1.0, World Wide Web Consortium, Note NOTE-wsci10-20020808, August 2002.
[4] P. Fisher, C. Hedeler, K. Wolstencroft, H. Hulme, H. Noyes, S. Kemp, R. Stevens, A. Brass, A systematic strategy for large-scale analysis of genotype phenotype correlations: Identification of candidate genes involved in African trypanosomiasisafrican trypanosomiasis, Nucleic Acids Research 35 (16) (August 2007) 5625–5633.
[5] A. Goderis, C. Brooks, I. Altintas, E. Lee, C. Goble, Composing different models of computation in kepler and ptolemy ii, in: International Conference on Computational Science, in: Lecture Notes in Comput. Sci., Springer, Berlin/Heidelberg, 2007, pp. 182–190.
[6] Object Managment Group, Business Process Modeling Notation (BPMN), Version 1.2, OMG Document Number: Formal/2009-01-03, Standard document URL: http://www.omg.org/spec/BPMN/1.2, January 2009.
[7] J. Hidders, J. Sroka, Towards a calculus for collection-oriented scientific workflows with side effects, in: OTM Conferences (1), 2008, pp. 374–391.
[8] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, Jan Van den Bussche, DFL: A dataflow language based on petri nets and nested relational calculus, Inf. Syst. 33 (3) (2008) 261–284.
[9] D. Hull, K. Wolstencroft, R. Stevens, C.A. Goble, M.R. Pocock, P. Li, T. Oinn, Taverna: A tool for building and running workflows of services, Nucleic Acids Research, 34 (Web-Server-Issue), 2006, pp. 729–732.
[10] Nickolas Kavantzas, David Burdett, Greg Ritzinger, Tony Fletcher, Yves Lafon, Charlton Barreto, Web services choreography description language version 1.0, World Wide Web Consortium, Candidate Recommendation CR-ws-cdl-10-20051109, November 2005.
[11] Frank Leymann, Web services flow language (WSFL 1.0), Technical report, IBM, May 2001.
[12] P. Li, J. Castrillo, G. Velarde, I. Wassink, S. Soiland-Reyes, S. Owen, D. Withers, T. Oinn, M. Pocock, C. Goble, S. Oliver, D. Kell, Performing statistical analyses on quantitative data in taverna workflows: An example using R and maxdBrowse to identify differentially-expressed genes from microarray data, BMC Bioinformatics 9 (334) (2008), August.
[13] B. Ludäscher, I. Altintas, C. Berkley, Scientific workflow management and the Kepler system, in: Concurrency and Computation: Practice and Experience, Special Issue on Scientific Workflows, 2005.
[14] T. McPhillips, S. Bowers, B. Ludäscher, Collection-oriented scientific workflows for integrating and analyzing biological data, in: Proceedings 3rd International Conference on Data Integration for the Life Sciences (DILS), LNCS/LNBI, Springer, 2006.
[15] T. McPhillips, S. Bowers, D. Zinn, B. Ludäscher, Scientific workflow design for mere mortals, Future Generation Computer Systems 25 (5) (2009) 541–551.
[16] Robin Milner, Communicating and Mobile Systems: The Pi-Calculus, Cambridge University Press, 1999, June.
[17] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, P. Li, Taverna: A tool for the composition and enactment of bioinformatics workflows, Bioinformatics 20 (17) (2004) 3045–3054.
[18] Chun Ouyang, Eric Verbeek, Wil M.P. van der Aalst, Stephan Breutel, Marlon Dumas, Arthur H.M. ter Hofstede, Formal semantics and analysis of control flow in WS-BPEL, Sci. Comput. Program. 67 (2–3) (2007) 162–198.
[19] C. Pautasso, G. Alonso, Parallel computing patterns for grid workflows, in: Proc. of the HPDC2006 Workshop on Workflows in Support of Large-Scale Science (WORKS06), Paris, France, 2006.
[20] Benjamin C. Pierce, David N. Turner, Pict: A programming language based on the pi-calculus, in: G. Plotkin, C. Stirling, M. Tofte (Eds.), Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press, 2000.
[21] Wolfgang Reisig, Petri Nets: An Introduction, Springer-Verlag New York, NY, USA, 1985.
[22] D. Smedley, S. Haider, B. Ballester, R. Holland, D. London, G. Thorisson, A. Kasprzyk, Biomart – biological queries made easy, BMC Genomics 10 (22) (2009).
[23] D. Turi, P. Missier, D. De Roure, C. Goble, T. Oinn, Taverna workflows: Syntax and semantics, in: Proceedings of the 3rd e-Science Conference, Bangalore, India, December 2007.

[24] W.M.P. van der Aalst, Ter A.H.M. Hofstede, YAWL: Yet another workflow language, Infor. Syst. 30 (4) (2005) 245–275.
[25] Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, Bartek Kiepuszewski, Alistair P. Barros, Workflow patterns, Distrib. Parallel Databases 14 (1) (2003) 5–51.
[26] R. van Glabbeek, The linear time-branching time spectrum, in: Proceedings on Theories of Concurrency: Unification and Extension, Springer-Verlag, 1990, pp. 278–297.
[27] M. Weidlich, G. Decker, M. Weske, Efficient analysis of BPEL 2.0 processes using $\pi$-calculus, in: Asia-Pacific Service Computing Conference, The 2nd IEEE, December 2007, pp. 266–274.