Fundamenta Informaticae 92 (2009) 373–396
DOI 10.3233/FI-2009-80
IOS Press

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

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

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

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

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

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

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

This paper is the second of two. In the first one [13] we have defined, explained and discussed fundamental notions for describing Scufl graphs and their semantics. Here, in the second part, we use these notions to define the semantics and show that our definition can be used to prove properties of Scufl graphs.

Keywords: formal semantics, Scufl, workflows, Taverna workbench

1. Introduction

Taverna [11] is an easy to operate workbench for workflow development and enactment. It allows users to graphically construct workflows from libraries of available components and is intended for use in bioinformatics data analysis experiments. The most important virtues of Taverna are that it is very easy to use, has a specialized and expressive graphical specification language and integrates many data analysis tools. In [12] it is stated that the number of such tools exceeds 1000. It also includes additional useful features like service discovery, storing intermediate results and tracking of data provenance. The workbench is being constantly developed, but it is already considered stable and has been used in real life research, e.g., [14, 7].

This paper is the second of two. In the first part [13] we have defined, explained and discussed fundamental notions for describing Scufl graphs and their semantics. Here, in the second part, we use these notions to define the semantics and show that our definition can be used to prove properties of Scufl graphs. To account for side effects the semantics is defined as a transition system. Finally, this part also includes a more elaborate comparison with the work in [15].

For the convenience of the reader we briefly recall some of the formal notions that were introduced in the first part. The set $\mathcal{V}_{tav}$ contains all the *complex values* which are constructed from MIME values and recursively nested lists. We use $\perp$ as a special marker to indicate the absence of a value. The set $\mathcal{G}$ is the set of all *Scufl graphs* which are defined as tuples $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ where I is the set of workflow inputs, O is the set of workflow outputs, π_i and π_o are the sets of processor input ports and output ports, respectively, E_d is the set of dataflow edges, E_c is the set of control flow edges, λ is the function that assigns to each processor either a service name or a nested Scufl graph, ils is the function that assigns incoming links strategies to processor input ports and workfow outputs, ps is the function that assigns product strategies to processors and dv is the function that assigns default values to some processor input ports. Given a Scufl graph g we define *the nesting graph* $\mathcal{N}_g$ to be the graph over all the Scufl graphs in g, including g itself, that indicates which graph is nested in which other graph. The function $type_i$ gives for each processor p in a Scufl graph the tuple type that describes its input interface.

The set $\mathcal{V}_{ext}$ contains the extended complex values which are recursively nested lists that contain tuples of complex values. For each product strategy s a function $[\![s]\!]^\tau$ defines the semantics of s by mapping a tuple of complex values containing a field for each port label in s to an extended complex value. The result represents the value over which a processor with product strategy s will iterate. It contains tuples of the tuple type τ, which is the input value type expected by the processor, and each of these tuples represent the input of a single iteration step of the processor.

The set $\mathcal{I}$ contains all complex value indices, i.e., path expressions such as $1/3/2$ that indicate positions in complex values and extended complex values. Finally, there are the functions first, get, put and next for iterating over and constructing complex values. The function $\mathsf{first}(v)$ finds the first

position in the extended complex value v at which we find a tuple value. The function $\mathsf{get}(v, \beta)$ finds the value at position β in the extended complex value v. The function $\mathsf{put}(v, \alpha, w)$ inserts into the complex value v at position α the complex value w and returns the result. Finally, the function $\mathsf{next}(v, \alpha)$ finds in extended complex value v the first position after position α that contains a tuple.

2. Transition system semantics

In this section we define the semantics of Scufl graphs in terms of a transition system, i.e., we specify a set of possible states of the Scufl graph and which transitions are possible between these states. The following subsection discusses the states, it is followed by subsections on auxiliary notions for describing the transitions, then the transitions themselves are discussed, and the final subsection shows that the defined semantics can be used in proofs of properties of Scufl graphs.

2.1. Scufl graph state

The state of a Scufl graph is described in two levels. At the lowest level we describe the so-called *local state* of each of the subgraphs. This local state consists of a description of the states of the workflow inputs and outputs, the processor input and output ports, and the processors themselves, but only those that are directly part of the subgraph in question. At the highest level the *global state* of a Scufl graph g is described by giving the local states of all the Scufl graphs in $\mathcal{G}_g$, i.e., all subgraphs of g including g itself. In the following we first define the notion of local state, followed by a definition of the global state.

We start with an informal introduction of the components of a local state. Consider the Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$. The *workflow input value mapping* $Iv : I \rightarrow (\mathcal{V}_{tav} \cup \bot)$ stores the value associated with each workflow input. The $\bot$ represents the lack of value, which here means that it has not been inserted yet or has already been pushed to the connected processor input ports. Next, the *workflow output value mapping* $Ov : O \rightarrow (\mathcal{V}_{tav} \cup \bot)$, the *input port value mapping* $ipv : \pi_i \rightarrow (\mathcal{V}_{tav} \cup \bot)$ and the *output port value mapping* $opv : \pi_o \rightarrow (\mathcal{V}_{tav} \cup \bot)$ store the values associated with workflow outputs, processor input ports and processor output ports respectively. The stored values are constructed by the incoming-links strategy function (see Section 2.3 of [13]) in case of the workflow output value mapping and the input port value mapping, or by the put function (see Section 2.2 of [13]) in case of the output port value mapping. This means that even if they have already been defined, i.e., are not equal to $\bot$, they may still be extended with additional values arriving from further data edges or iteration steps, respectively. Next, each processor itself can be in several states like "scheduled" or "preparing", which is specified by the *execution state mapping* $es : P \rightarrow \{\text{"scheduled"}, \text{"preparing"}, \text{"waiting"}, \text{"finished"}, \text{"failed"}\}$. The state "scheduled" indicates that the processor has not yet been used. The state "preparing" indicates that execution of this processor has already started but the input value, or in case of iteration some of its subvalues, have still to be processed. The state "waiting" indicates that the processor is waiting for a nested Scufl graph or an external service to return a result[1]. The state "finished" indicates that it has finished with success. The final state "failed" indicates that it has finished with failure. Finally, since a processor might have to iterate

[1]In official Taverna terminology the states that we call "preparing" and "waiting" are divided into $executing$ and $iterating$ for when the processor is either processing a value of its expected type or a value that is more deeply nested, respectively.

over subvalues of the input value, the current position in the input value is stored by the *iteration index mapping* $ii : P \to \mathcal{I}$.

Definition 2.1. (Local state)
Given a Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$, a local state of g is a tuple $ls = (Iv, Ov, ipv, opv, es, ii)$ such that:

- $Iv : I \to (\mathcal{V}_{tav} \cup \bot)$ is the workflow input value mapping,
- $Ov : O \to (\mathcal{V}_{tav} \cup \bot)$ is the workflow output value mapping,
- $ipv : \pi_i \to (\mathcal{V}_{tav} \cup \bot)$ is the input port value mapping,
- $opv : \pi_i \to (\mathcal{V}_{tav} \cup \bot)$ is the output port value mapping,
- $es : P \to \{$"scheduled", "preparing", "waiting", "finished", "failed"$\}$ is the processor state mapping,
- $ii : P \to \mathcal{I}$ is the iteration index mapping.

We refer to the set of all local states for all Scufl graphs as LS. The input port value of an input port (p, l), normally denoted as $ipv((p, l))$, will also be written as $ipv(p, l)$. Likewise the output port value of an output port (p, l) will also be written as $opv(p, l)$.

Scufl graphs do not have stateful features, such as counters or data-stores, that can be read and updated during a run of the Scufl graph. So the definition of a local state does not contain anything that represents the state of such elements. Of course these can be simulated by defining a set of special basic processors that have as their semantics that they read or write certain data stores. However, also for such basic processors that represent calls to stateful services, we do not represent the state of the service in the local state. This is because we consider this state not a part of the Taverna system but a part of the environment with which it communicates. It is possible to reason about the behavior of Taverna while taking into account that a service it calls has certain stateful behavior, e.g., is a counter. For that a description of that behavior, ideally also in the form of a state transition system, has to be composed with Taverna's state transition system such that their mutual transitions, i.e., the service calls, are synchronized.

Definition 2.2. (Global state)
A global state of a Scufl graph g is a function $gs : \mathcal{G}_g \to LS$ that associates with each subgraph $g' \in \mathcal{G}_g$ a local state of g'.

Note that only one state is associated with each subgraph which means that it executes only one run at any moment. Since we restrict ourselves to hierarchically nested Scufl graphs (see Section 1.5 of [13]) this cannot lead to resource contention between different parts of the Scufl graph. Although in Taverna it is possible to choose whether the iteration steps are executed sequentially or in parallel, we will only describe here sequential execution. It is possible to describe a semantics that would allow parallelism, see for example [5, 4], but we have chosen not to do so in this paper because it would complicate the presentation of the main concepts of the semantics of Scufl.

2.2. Ready ports and enabled processors

The fundamental notion that determines the execution of a Scufl graph is the notion of *enabledness* of a processor, i.e., whether in a certain state a processor can start processing its input. One necessary condition for this is that all its input ports are *ready*, i.e., store a fully constructed input value. In the following we describe these two notions in more detail.

Informally, a processor input port is said to be ready, if the value assigned to it will not be further extended by the incoming-links strategy function (see Section 2.3 of [13]).

Definition 2.3. (Ready input port)
Given a Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ we say that input port $p_{in} \in \pi_i$ is *ready* in a local state $ls = (Iv, Ov, ipv, opv, es, ii)$ iff p_{in} either has no incoming data edges or if p_{in} has incoming data edges then it holds that :

(i) if $ils(p_{in}) = \mathsf{first}$, then the first value for p_{in} has already arrived, i.e., $ipv(p_{in}) \neq \perp$, and

(ii) if $ils(p_{in}) = \mathsf{merge}$, then all the values for p_{in} have already arrived, i.e., the $ipv(p_{in})$ is a list with length equal to the number of data edges ending in p_{in}.

Recall that input ports with no incoming edge must have a default value specified, and therefore are always ready.

Note that if a select-first incoming-links strategy is specified, the port does not wait for values from all incoming data edges, but is ready after receiving the first one. On the other hand, if the merge incoming-links strategy is specified, the port has to wait for a value from every incoming data edge. This way the merge setting can be viewed as a shortcut for an intermediary processor with a separate input port for each incoming data edge and one output port, that composes values from distinct ports into a list[2].

The notion of readiness is extended to workflow outputs, which is natural since the values stored there will also be constructed by the incoming-links strategy function (see Section 2.3 of [13]). There is a small exception to this in the behavior of Taverna 1.7.1, where a workflow output with the merge strategy may become ready even if only values from some of the incoming data edges arrived and it is certain that no more will since the processors that should produce them failed. However, this behavior seems to be idiosyncratic.

The notion of readiness now allows us to define the notion of enabledness. Informally, a processor is said to be enabled, when it can start processing its input. There are three conditions that have to hold for that to happen. First, it has to be scheduled, which means that in the current run of the Scufl graph it was not used yet. Second, all the processors that it synchronizes with through the control edges must have already finished without a failure. Finally, every one of its input ports has to be ready, i.e., a value has to be available to be consumed from it, either one that was produced during the computation or provided as default. Formally:

Definition 2.4. (Enabled processor)
Given a Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ and its local state $ls = (Iv, Ov, ipv, opv, es, ii)$, a processor $p \in P$ is said to be *enabled* iff it holds that:

[2] Although, in the intermediary processor case the ordering of the elements of the result list would be always the same and not correspond to the order in which the input values have arrived.

(i) $es(p) =$ "scheduled", and

(ii) for every control edge $(p', p) \in E_c$, $es(p') =$ "finished", and

(iii) each input port of p is ready in ls.

Notice that during one Scufl graph run each processor at the top level can start processing of the input at most once, so it can produce at most one result value and thus each data edge transports at most one value.

2.3. Finished Scufl graphs

Here we explain when a Scufl graph is considered to be *finished*. Informally, a Scufl graph is finished when all the workflow input values were propagated, all values on processor output ports were propagated and there are no more processors that can start preparing, are preparing or are waiting.

Definition 2.5. (Finished Scufl graph)
A Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ is said to be *finished* in a local state $ls = (Iv, Ov, ipv, opv, es, ii)$ iff it holds that:

(i) for every workflow input $i \in I$ it holds that $Iv(i) = \perp$,

(ii) for every processor output port $(p, l) \in \pi_o$ it holds that $opv(p, l) = \perp$,

(iii) none of the scheduled processors is enabled, and

(iv) there are no preparing or waiting processors.

Furthermore, we say that the Scufl graph *finished with a success*, when its every workflow output $o \in O$ is ready, otherwise we say it *finished with a failure*.

This definition of finishing with a success or failure is implied by the fact that a Scufl graph can be nested and thus must produce values for its every workflow output, so that the processor in which it is nested can produce values on its every output port. However, in the real Taverna two exceptions are present which we briefly discuss here. First, for a Scufl graph that is not nested, i.e., the top level Scufl graph, it is enough to have at least one of its workflow outputs ready so that it finishes with a success. Second, a nested Scufl graph that iterates, i.e., was executed for a nested value in the value computed by the product strategy, always finishes with a success, even if none of its workflow outputs are ready. In the result of such iteration the empty string is used to fill in the missing results for workflow outputs that were not ready, but only when in a subsequent iteration step this workflow output becomes ready. However, if during all iterations a nested Scufl graph has not produced any value on a certain port, then the associated nested processor will fail anyway. For example, assume a nested Scufl graph with one workflow input and one workflow output is defined such that it returns its input value when it is unequal to "x", and no value otherwise. Then, if it iterates over ["x", "y", "x", "y", "x"] it returns ["", "y", "", "y"]. However, iterating with such a nested Scufl graph over a list with just "x" elements causes a failure of the nested processor.

The inclusion of the extra empty values seems an attempt to save such iterating nested Scufl graph from failure. However, the empty values will probably be misinterpreted in the remaining part of the

Scufl graph in which the iteration over the nested Scufl graph occurred. Moreover, the absence of the extra empty values when they are not followed by ordinary results, may also confuse the user. For example, consider the Scufl graph in Fig. 1, where the nested Scufl graph is used to submit a paper to a PhD symposium and apply for a grant to visit it, and the "Declaration_of_expenses" processor has the dot product strategy specified. Let us assume, that this Scufl graph is started with a list of three PhD students {"X", "Y", "Z"} and during the iteration the papers written by "X" and "Z" are accepted, but for some formal reasons they do not get grants and the paper written by "Y" is rejected, but he gets a grant anyway since the money are available. That is the list {"", "gnY"}, where "gnY" is the grant number for "Y" is returned on the output port $grant\_number$, and the list {"idX", "", "idZ"}, where "idX" is the accepted paper identifier for "X" while "idZ" for "Z", is returned on the output port $conference\_name$. Now, if the "Declaration_of_expenses" processor is not prepared to handle empty values, "X" will have his expenses refunded even though he had no grant, "Y" will get money from his grant even though he did not go to the symposium and "Z" will not have his expenses refunded, despite the fact that he was in the same situation as "X". Although the first thing is not bad in this context, the remaining two probably are. Furthermore, if the symposium chair was used to running this Scufl graph for individual PhD students, he would probably be dissatisfied by the different behavior, i.e., running this Scufl graph separately for any of "X", "Y" and "Z" would alert him with an error.

Therefore we have chosen not to allow in our formal semantics Taverna's exception for nested Scufl graphs and define them also to be finished with failure if not all their workflow outputs have produced a value. For uniformity we also do not allow Taverna's exception for the top level Scufl graph, so also there we define the notion such that all workflow outputs must produce a result in order for it to finish with success.

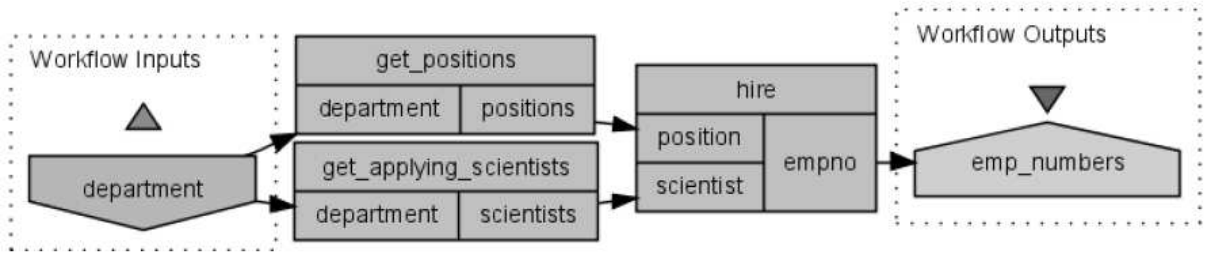

Figure 1. Iteration over a nested Scufl graph

2.4. Scufl graph initialization and result collection

When a Scufl graph starts execution its state needs to be reset such that any remaining state properties such as intermediate and final results of the previous execution are removed. Therefore we introduce the notion of an *initial state* in which we reset the workflow outputs, the processor input ports, the processor output ports, the processor states and the iteration indices. Note that the workflow inputs are not required to be empty. Formally the notion is defined as follows.

Definition 2.6. (Initial state)
A local state $ls = (Iv, Ov, ipv, opv, es, ii)$ of Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ is

said to be an *initial state* iff:

(i) $Ov = \{(o, \bot) \mid o \in O\}$,

(ii) $ipv = \{(p_{in}, \bot) \mid p_{in} \in \pi_i\}$,

(iii) $opv = \{(p_{out}, \bot) \mid p_{out} \in \pi_o\}$,

(iv) $es = \{(p, \text{“scheduled”}) \mid p \in P\}$, and

(v) $ii = \{(p, \epsilon) \mid p \in P\}$.

An initial state for which $Iv^{-1}(\{\bot\}) = \emptyset$, with $Iv^{-1}(X) := \{i \mid Iv(i) \in X\}$, is called *full*, if $Iv^{-1}(\{\bot\}) = I$ it is called *clean* and otherwise the initial state is called *partial*.

In addition we define the function init to return the initial local state of a given Scufl graph after initiating its workflow inputs with values stored on fields of a given tuple. Formally, the function $\mathsf{init} : \mathcal{G} \times \mathcal{V}_{tup} \rightarrow LS$ is defined as a partial function such that for a Scufl graph g and a tuple t where $\mathrm{dom}(t) \subseteq I$ it holds that $\mathsf{init}(g, t)$ is the initial state $(t \cup \bar{t}, Ov, ipv, opv, es, ii)$ of g, where $\bar{t} = \{(i, \bot) \mid i \in (I \setminus \mathrm{dom}(t))\}$. Note that the returned initial state is full if $\mathrm{dom}(t) = I$.

We also define the function result to return the tuple of values computed on the workflow outputs in a given local state of a given Scufl graph. Formally, the partial function $\mathsf{result} : LS \rightarrow \mathcal{V}_{tup}$ is defined such that $\mathsf{result}(ls) = Ov$ if $ls = (Iv, Ov, ipv, opv, es, ii)$ and $Ov \in \mathcal{V}_{tup}$. Observe that $\mathsf{result}(ls)$ is defined if a Scufl graph g finished with a success in local state ls.

2.5. State transitions

In this section we describe the possible transitions of the state of a Scufl graph. Recall that a system and its state is defined by a hierarchical Scufl graph g and a global state gs of g. For each type of transition we will specify a precondition over gs that must be satisfied and specify the new global state gs' such that the transition $gs \rightsquigarrow gs'$ is possible.

Before we proceed with the formal description of the transitions, we summarize them in a brief and informal overview:

Propagation of values from workflow inputs (PROPWI) The values in the workflow inputs are propagated to the processor input ports and workflow outputs to which they are connected by data edges. At their destination they are added to any value that is already present there according to the incoming-links strategy.

Initializing processor execution (INITPE) A scheduled and enabled processor is prepared for execution, i.e., the output port values are initialized and the iteration index is set to the first suitable value in the result computed by the product strategy.

Starting a service call by a basic processor (STARTSC) A call is made to the service associated with the basic processor, with the value indicated by the iteration index as a parameter.

Finishing successfully a service call by a basic processor (SUCFSC) A call to a service succeeds and returns a value. The value is distributed and inserted into the different output port values of the processor. The iteration index is moved to the next suitable value.

Failure of a service call by a basic processor (FAILSC) A call to a service fails and so the whole execution of the processor fails.

Starting a nested Scufl graph execution (STARTNSGE) The nested Scufl graph is initialized with the value indicated by the iteration index.

Finishing successfully a nested Scufl graph execution (SUCFNSGE) The nested Scufl graph finishes with success and returns a value. This value is distributed and inserted into the different output port values of the processor. The iteration index is moved to the next suitable value.

Failure of a nested Scufl graph execution (FAILSGE) The nested Scufl graph finishes with failure, and so the whole execution of the processor fails.

Finishing processor execution (FINPE) If the iteration index is undefined because there is no next suitable value, the executing of the processor finishes with a success.

Propagation of values from processor output ports (PROPOP) When a processor is finished, but not failed, the values of its output ports are propagated to the processor input ports and workflow outputs to which they are connected by data edges. At their destination they are added to any value that is already there according to the specified incoming-links strategy.

We now describe the transitions in full detail using the following notation. For a local state $ls = (Iv, Ov, ipv, opv, es, ii)$ we let $ls[\mathbf{Iv} := Iv']$ denote the local state $(Iv', Ov, ipv, opv, es, ii)$. In a similar fashion we define $ls[\mathbf{Ov} := Ov']$, $ls[\mathbf{ipv} := ipv']$, $ls[\mathbf{opv} := opv']$, $ls[\mathbf{es} := es']$ and $ls[\mathbf{ii} := ii']$, as the local states equal to ls but with the indicated tuple position replaced with the new value. For a function f and values x and y, we let $f[x \mapsto y]$ denote the function that is equal to f except that it maps x to y, i.e., the function $\{(x', y') \mid (x', y') \in f, x' \neq x\} \cup \{(x, y)\}$. For two functions f and h, we let $f[h]$ denote the function equal of f except for values x for which h is defined, which are mapped to $h(x)$, i.e., the function $\{(x', y') \in f \mid \neg\exists y'' : (x', y'') \in h\} \cup h$.

2.5.1. Propagation of values from workflow inputs (PROPWI)

Consider a workflow input $i \in I$. If the value of i is defined, i.e., $Iv(i) \neq \bot$, then this value is removed from the workflow input and added to the input ports and workflow outputs to which i is connected with a data edge. For each such input port and workflow output the data is added as specified by the corresponding incoming-links strategy. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$i \in I$, $Iv(i) \neq \bot$

transition: $gs \rightsquigarrow gs[g \mapsto ls']$ where
$ls' = ls[\mathbf{Iv} := Iv[i \mapsto \bot]][\mathbf{Ov} := Ov[Ov']][\mathbf{ipv} := ipv[ipv']]$, with
$Ov' = \{(o, [\![ils(o)]\!](Ov(o), Iv(i)) \mid o \in O, (i, o) \in E_d\}$ and
$ipv' = \{(p_{in}, [\![ils(p_{in})]\!](ipv(p_{in}), Iv(i))) \mid p_{in} \in \pi_i, (i, p_{in}) \in E_d\}$

Note that if a workflow input has no outgoing data edges, its value is anyway reset to $\bot$.

2.5.2. Initializing processor execution (INITPE)

Consider an enabled processor $p \in P$ in state "scheduled" and let v be the value computed by the product strategy of p from its available input port values and default values, i.e., $v = [\![ps(p)]\!]^{type_i(p)}(t_1 \cup t_2)$ where t_1 is the tuple constructed from the available values on the input ports and t_2 is the tuple constructed from the default values for the input ports for which no value is available, i.e., $t_1 = \{(l, ipv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) \in \mathcal{V}_{tav}\}$ and $t_2 = \{(l, dv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) = \bot\}$. The output port values of the output ports of the processor p are initialized with an empty list, the iteration index of p is set to the first iteration value in v, and the state of the processor is set to "preparing". Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, p is enabled in ls, $es(p) =$ "scheduled",
$t_1 = \{(l, ipv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) \in \mathcal{V}_{tav}\}$,
$t_2 = \{(l, dv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) = \bot\}$,
$v = [\![ps(p)]\!]^{type_i(p)}(t_1 \cup t_2)$

transition: $gs \rightsquigarrow gs[g \mapsto ls']$ where
$ls' = ls[\mathbf{opv} := opv[opv']][\mathbf{es} := es[p \mapsto \text{"preparing"}]][\mathbf{ii} := ii[p \mapsto \mathsf{first}(v)]]$, with $opv' = \{((p,l), []) \mid (p,l) \in \pi_o\}$

2.5.3. Starting a service call by a basic processor (STARTSC)

Consider preparing basic processor $p \in P$ and let v again be the value computed by the product strategy of p from its input port values. The precondition is that in v there is a next iteration element, i.e., $ii(p) \in \mathcal{I}$, and that the service was not yet called for this element, i.e., $es(p) =$ "preparing", then the execution state of p is set to "waiting". This models the real world event that the service $\lambda(p)$ is called with the parameters $\mathsf{get}(v, ii(p))$. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $\lambda(p) \in TS$, $es(p) =$ "preparing", $ii(p) \in \mathcal{I}$

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{es} := es[p \mapsto \text{"waiting"}]]]$

2.5.4. Finishing successfully a service call by a basic processor (SUCFSC)

Consider a basic processor $p \in P$ that is waiting for the result of a service call, i.e., $es(p) =$ "waiting", and let v again be the value computed by the product strategy of p from its input port values. For a possible result of such a service call the respective fields are inserted into the output port values at the position indicated by the iteration index, the execution state is set to "preparing" and the iteration index is advanced one position. This models the real world event that the previously made service call succeeds and returns a certain value. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,

$p \in P$, $\lambda(p) \in TS$, $es(p) =$ "waiting",
$t_1 = \{(l, ipv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) \in \mathcal{V}_{tav}\}$,
$t_2 = \{(l, dv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) = \bot\}$,
$v = [\![ps(p)]\!]^{type_i(p)}(t_1 \cup t_2)$, $(\mathsf{get}(v, ii(p)), t) \in [\![\lambda(p)]\!]$,

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{opv} := opv[opv']][\mathbf{es} := es'][\mathbf{ii} := ii']]$ where
$opv' = \{((p,l), \mathsf{put}(opv(p,l), ii(p), t(l))) \mid (p,l) \in \pi_o\}$,
$es' = es[p \mapsto$ "preparing"$]$ and
$ii' = ii[p \mapsto \mathsf{next}(v, ii(p))]$

2.5.5. Failure of a service call by a basic processor (FAILSC)

Consider a basic processor $p \in P$. If the processor is waiting for the result of a call, i.e., $es(p) =$ "waiting", then the call might fail and its execution state becomes "failed". This models the real world event that the call to the service $\lambda(p)$ failed. This leads to the following formal specification of the transition:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $\lambda(p) \in TS$, $es(p) =$ "waiting"

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{es} := es[p \mapsto$ "failed"$]]]$

2.5.6. Starting a nested Scufl graph execution (STARTNSGE)

This transition is very similar to the starting of a service call by a basic processor, except that the processor is not a basic processor but a nested Scufl graph and rather than starting a call to a service the nested Scufl graph $\lambda(p)$ is initialized for this iteration element. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $\lambda(p) \in \mathcal{G}$, $es(p) =$ "preparing", $ii(p) \in \mathcal{I}$,
$t_1 = \{(l, ipv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) \in \mathcal{V}_{tav}\}$,
$t_2 = \{(l, dv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) = \bot\}$,
$v = [\![ps(p)]\!]^{type_i(p)}(t_1 \cup t_2)$

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{es} := es']][\lambda(p) \mapsto \mathsf{init}(\lambda(p), \mathsf{get}(v, ii(p)))]$ where
$es' = es[p \mapsto$ "waiting"$]$

2.5.7. Finishing successfully a nested Scufl graph execution (SUCFSGE)

This transition is very similar to the finishing successfully of a service call by a basic processor, except that the processor is not a basic processor but a nested Scufl graph and it is required in the precondition that the nested Scufl graph $\lambda(p)$ must have finished with success in $gs(\lambda(p))$, and the result tuple is composed from the output ports of the nested Scufl graph, i.e., $t = \mathsf{result}(gs(\lambda(p)))$. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $\lambda(p) \in \mathcal{G}$, $es(p) =$ "waiting",
$\lambda(p)$ finished with a success in $gs(\lambda(p))$,
$t_1 = \{(l, ipv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) \in \mathcal{V}_{tav}\}$,
$t_2 = \{(l, dv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) = \bot\}$,
$v = [\![ps(p)]\!]^{type_i(p)}(t_1 \cup t_2)$, $t = \mathsf{result}(gs(\lambda(p)))$,

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{opv} := opv[opv']][\mathbf{es} := es'][\mathbf{ii} := ii']]$ where
$opv' = \{((p,l), \mathsf{put}(opv(p,l), ii(p), t(l))) \mid (p,l) \in \pi_o\}$,
$es' = es[p \mapsto$ "preparing"$]$ and
$ii' = ii[p \mapsto \mathsf{next}(v, ii(p))]$

2.5.8. Failure of a nested Scufl graph execution (FAILSGE)

This transition is very similar to the failure of a service call by a basic processor, except that the processor is not a basic processor but a nested Scufl graph and it is required in the precondition that the nested Scufl graph $\lambda(p)$ must have finished with a failure in its local state $gs(\lambda(p))$. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $\lambda(p) \in \mathcal{G}$, $es(p) =$ "waiting",
$\lambda(p)$ finished with a failure in $gs(\lambda(p))$

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{es} := es[p \mapsto$ "failed"$]]]$

2.5.9. Finishing processor execution (FINPE)

If the processor is preparing and there is no next iteration index, then the state of the processor becomes "finished". Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $es(p) =$ "preparing", $ii(p) = \bot$

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{es} := es[p \mapsto$ "finished"$]]]$

2.5.10. Propagation of values from processor output ports (PROPOP)

Consider a processor output port $(p,l) \in \pi_o$. If the value of (p,l) is defined, i.e., $opv(p,l) \neq \bot$ and the processor is finished, but not failed, then this value is removed from the processor output port and added to the input ports and workflow outputs to which output port (p,l) is connected with a data edge. For each such input port and workflow output the data is added as specified by the corresponding incoming-links strategy. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$(p, l) \in \pi_o$, $opv(p, l) \neq \bot$, $es(p) =$ "finished"

transition: $gs \rightsquigarrow gs[g \mapsto ls']$ where
$ls' = ls[\mathbf{Ov} := Ov[Ov']][\mathbf{ipv} := ipv[ipv']][\mathbf{opv} := opv[(p, l) \mapsto \bot]]$, with
$Ov' = \{(o, [\![ils(o)]\!](Ov(o), opv(p, l)) \mid o \in O, ((p, l), o) \in E_d\}$ and
$ipv' = \{(p_{in}, [\![ils(p_{in})]\!](ipv(p_{in}), opv(p, l))) \mid p_{in} \in \pi_i, ((p, l), p_{in}) \in E_d\}$

Note that if a processor output port has no outgoing edges, its value is anyway reset to $\bot$.

2.5.11. Scufl graph run

The specification of possible transitions defines a transition system that can be used to describe the semantics of Scufl graphs. An instance of a computation of a particular Scufl graph g, i.e., a sequence of successive global states reached during the computation, will be called a *run*. We will denote a run of global states $gs_1, \ldots, gs_n$ of g as $gs_1 \rightsquigarrow \ldots \rightsquigarrow gs_n$, by which we mean that $gs_i \rightsquigarrow gs_{i+1}$ for each $1 \leq i \leq n - 1$.

A run $gs_1 \rightsquigarrow \ldots \rightsquigarrow gs_n$ of g will be called a *cleanly initialized run* if it starts with an initial local state of g, i.e., $gs_1(g)$ is initial, and clean initial local states of all the nested graphs, i.e., for all $g' \in \mathcal{G}_g$ such that $g' \neq g$ the local state $gs_1(g')$ is a clean initial state.

2.6. Soundness of the transition system

To check the completeness of our semantics definition we are going to formally prove a property of Scufl graphs, which states that for every Scufl graph g all its cleanly initialized runs that start with g initialized with any input values of any type[3] and possibly missing input values eventually finish, either with success or with failure.

At the same time this exercise shows that the formal semantics as defined in this paper can be used in proofs of this kind.

Theorem 2.1. For every Scufl graph g and any of its cleanly initialized runs $gs_1 \rightsquigarrow \ldots \rightsquigarrow gs_n$:

(i) there is a maximum number of steps that this run can be extended with, i.e., such $m \in \mathbb{N}$ that for every run $gs_1 \rightsquigarrow \ldots \rightsquigarrow gs_n \rightsquigarrow gs_{n+1} \rightsquigarrow \ldots \rightsquigarrow gs_k$ of g it holds that $k \leq m$, and

(ii) if in gs_n none of the transitions is possible then g is finished.

Proof:
In the following we assume g to be a Scufl graph and gs its global state.

We first show that the runs are of finite length. The idea is to show that the global state of g in some sense decreases with each transition and this decreasing cannot proceed indefinitely. For that we define a *global state vector* which is a natural number vector. The composition of the vector is based on the properties of combined local states of the Scufl graphs that occupy the same level of the tree given by

[3] Under our liberal type semantics this includes heterogeneous values, and therefore all complex values.

the nesting graph $\mathcal{N}_g$ (in the following referred to as the nesting tree)[4], i.e., graphs that as the nodes of the tree have the same depth. Let $\mathcal{N}_g(k)$ be the set of graphs at depth k of the nesting tree $\mathcal{N}_g$, i.e., $\mathcal{N}_g(0) = \{g\}$ and $\mathcal{N}_g(k+1) = \{g' \mid (g,p,g') \in E, g \in \mathcal{N}_g(k)\}$. For each non-empty level k of the nesting tree the vector contains six subsequent properties: (1) the total number of workflow inputs of graphs $g_k \in \mathcal{N}_g(k)$ which in $gs(g_k)$ are not empty, (2) the total number of processors of graphs $g_k \in \mathcal{N}_g(k)$ that in $gs(g_k)$ are scheduled, (3) the total number of processors of graphs $g_k \in \mathcal{N}_g(k)$ that in $gs(g_k)$ are neither finished nor failed, (4) the total number of elements that still have to be iterated by processors of graphs $g_k \in \mathcal{N}_g(k)$ in $gs(g_k)$, (5) the total number of processors in graphs $g_k \in \mathcal{N}_g(k)$ that in $gs(g_k)$ are preparing (6) the total number of processor output ports in $g_k \in \mathcal{N}_g(k)$ that in $gs(g_k)$ are empty.

The following functions of signature $\mathbb{N} \to \mathbb{N}$ give the values of those properties. We assume that $g^k = (I^k, O^k, P^k, \pi_i^k, \pi_o^k, E_d^k, E_c^k, \lambda^k, ils^k, ps^k, dv^k)$ and $gs(g^k) = (Iv^k, Ov^k, ipv^k, opv^k, es^k, ii^k)$.

1. $\mathsf{notewfi}_g^{gs}(k) = \big|\{\, i \mid i \in I^k, Iv^k(i) \neq \bot, g^k \in \mathcal{N}_g(k)\,\}\big|$
2. $\mathsf{psched}_g^{gs}(k) = \big|\{\, p \mid p \in P^k, es^k(p) = \text{“scheduled”}, g^k \in \mathcal{N}_g(k)\,\}\big|$
3. $\mathsf{pnotff}_g^{gs}(k) = \big|\{\, p \mid p \in P^k, es^k(p) \neq \text{“finished”}, es^k(p) \neq \text{“failed”}, g^k \in \mathcal{N}_g(k)\,\}\big|$
4. $\mathsf{iterleft}_g^{gs}(k) = \sum\{\, \mathsf{togo}(v,i) \mid p \in P^k, ii^k(p) = i, g^k \in \mathcal{N}_g(k)\,\}$
5. $\mathsf{pprep}_g^{gs}(k) = \big|\{\, p \mid p \in P^k, es^k(p) = \text{“preparing”}, g^k \in \mathcal{N}_g(k)\,\}\big|$
6. $\mathsf{ewfo}_g^{gs}(k) = \big|\{\, o \mid o \in O^k, Ov^k(i) = \bot, g^k \in \mathcal{N}_g(k)\,\}\big|$

where $\mathsf{togo}(v,i) : \mathcal{V}_{ext} \times \mathcal{I} \to (\mathcal{V}_{ext} \cup \bot)$ is defined such that $\mathsf{togo}(v,i) = 0$ if $\mathsf{next}(v,i) = \bot$ and $\mathsf{togo}(v,i) = 1 + \mathsf{togo}(v, \mathsf{next}(v,i))$ if $\mathsf{next}(v,i) \neq \bot$. It is easy to see that the functions are well defined in any state of a cleanly initialized run.

In the vector the components corresponding to smaller depths in the nesting tree precede the ones for bigger depths. Formally the vector is defined as follows:

$$
\begin{aligned}
&(\mathsf{notewfi}_g^{gs}(0), \mathsf{psched}_g^{gs}(0), \mathsf{pnotff}_g^{gs}(0), \mathsf{iterleft}_g^{gs}(0), \mathsf{pprep}_g^{gs}(0), \mathsf{ewfo}_g^{gs}(0)\\
&\ldots\\
&\mathsf{notewfi}_g^{gs}(h_g), \mathsf{psched}_g^{gs}(h_g), \mathsf{pnotff}_g^{gs}(h_g), \mathsf{iterleft}_g^{gs}(h_g), \mathsf{pprep}_g^{gs}(h_g), \mathsf{ewfo}_g^{gs}(h_g))
\end{aligned}
$$

where h_g is the height of the nesting tree, i.e., the biggest number k such that $\mathcal{N}_g(k) \neq \emptyset$. Observe that the vector is thus of finite size determined by the height of the nesting tree, i.e., its size is equal six times the height of the nesting tree.

We are now going to show that under a lexicographical ordering each transition in a cleanly initialized run decreases the global state vector. For that we are going to list how each state transition changes the vector:

PROPWI does not increase any component and decreases the number of not empty workflow inputs (1),

[4]We assume Scufl graphs to be hierarchical and thus the nesting graph $\mathcal{N}_g$ is a tree (see Section 1.5 of [13]).

INITPE increases the number of remaining iterations (4) and the number preparing processors (5), but at the same time decreases the number of scheduled processors (2),

STARTSC does not increase any component and decreases the number of preparing processors (5),

SUCFSC increases the number of preparing processors (5), but at the same time decreases the number of remaining iterations (4),

FAILSC does not increase any component and decreases the number of processors that are neither finished nor failed (3),

STARTNSGE increases the part of the vector that corresponds to the nested Scufl graph, which is on a bigger depth thus less important in our ordering, and decreases the number of scheduled processors (2),

SUCFSGE similarly as SUCFSC increases the number of preparing processors (5), but at the same time decreases the number of remaining iterations (4),

FAILSGE does not increase any component and decreases the number of processors that are neither finished nor failed (3),

FINPE does not increase any component and decreases the number of processors that are neither finished nor failed (3),

PROPOP decreases the number of empty workflow outputs (6).

It is well known from set theory that the set of natural number vectors of a given length with lexicographical ordering is a well-founded partially ordered set and thus it does not contain an infinite descending chain. For the self containment of this work we show this formally in the Appendix A (see Corollary A.1). This proves that all runs are of finite length, since from the non-existence of an infinite descending chain it follows that there is a bound on the number of transitions by which a given run can be extended.

To complete the proof of Theorem 2.1 we are going to show that if a state has been reached in which no transitions are possible, i.e., none of the transitions has its preconditions satisfied, then g is finished. The proof will follow by induction on the height of the nesting tree $\mathcal{N}_g$

We first assume that the nesting tree $\mathcal{N}_g$ is of height 1 and that in a global state gs' of g none of the transitions has its preconditions satisfied. We will show that g is finished in $gs'(g)$. For that we look at the four conditions in definition 2.5. It is clear that (i) directly follows from the unfulfillment of the preconditions for transition PROPWI, (ii) directly follows from the unfulfillment of the preconditions for transition PROPOP and (iii) directly follows from the unfulfillment of the preconditions for transition INITPE. As for (iv) let us first notice that if the nesting tree $\mathcal{N}_g$ is of height 1, then g contains only basic processors, i.e., $\mathcal{N}_g = \{\{g\}, \emptyset\}$. If there would be any preparing processor, then either STARTSC or FINPE transitions would be possible depending on whether there is a next iteration element for that processor. Also, if there would be any waiting processor, then both the SUCFSC and FAILSC transitions would be possible[5]. Thus all the conditions for a finished Scufl graph are satisfied.

[5]Recall that we do not include the state of the external services in our formal model and thus there is no dependency on any such state in the preconditions for the transitions.

We now assume that the thesis holds for all the Scufl graphs with the nesting tree of height smaller or equal to n and we are going to show that it also holds for all graphs with the nesting tree of height $n+1$. Let the nesting tree $\mathcal{N}_g$ be of height $n+1$ and let gs' be a global state g such that none of the transitions has its preconditions satisfied. We will show that g is finished in $gs'(g)$. As before we look at the conditions in the definition 2.5. For conditions (i), (ii) and (iii) the reasoning follows. As for (iv) the argument for non-existence of preparing processors remains the same. Similarly for the non-existence of waiting basic processors. The only thing left to show is that there are no waiting processors that represent nested Scufl graphs. Let us assume by contradiction that a waiting processor p exists in g and represents a nested Scufl graph. Because preconditions for SUCFSGE and FAILSGE transitions are not satisfied, then the nested Scufl graph of p cannot be finished. Yet, the nesting tree of that nested Scufl graph is of height smaller or equal to n and since we have assumed that no transitions are possible, it follows from the induction assumption that the nested Scufl graph is finished. This completes the proof by contradiction and thus the proof by induction. □

It is easy to see that a cleanly initialized run of a Scufl graph may finish with failure if (1) not all input port values are available or if (2) a basic processor fails. In both cases some processors can never produce their output which may prevent some or all workflow outputs from becoming ready. It can also be observed that (1) and (2) are the only reasons for a Scufl graph not to succeed and thus for every Scufl graph g all its cleanly initialized runs that start with g in a fully initialized local state, i.e., with all the workflow input values present, eventually terminate with success if we exclude the failure of transitions. Although we do not give here a formal proof, this follows intuitively from the facts that processors without input ports are immediately enabled, all processor input ports have either incoming data edges or a default value specified and because Scufl graphs contain no cycles.

3. Dealing with heterogeneous values in Taverna

Until now in the discussion of the semantics of Scufl we focused our attention on homogeneous lists. Yet, heterogeneous values can be created in Taverna with the use of the merge incoming-links strategy or by an iteration on a processor that returns values with various nesting depths in its subsequent executions, e.g., sometimes lists and sometimes lists of list. Unfortunately, heterogeneous values are not processed consistently in the current implementation of Scufl. In Section 1.2 of [13] we showed that some services, such as the built-in flatten operation, do not handle such values consistently. Furthermore, the way the dot product is implemented in Taverna yields sometimes rather unexpected results for heterogeneous values. Informally, the dot product in Taverna is computed by iterating over the tuples in both arguments. During the iteration, the subsequent pairs of tuples are combined, i.e., first with first, second with second and so on. The combinations are placed in the result nested list on positions pointed by the longer of the indexes of the combined tuples. If both indexes have the same length, the left one is chosen. This is different from our definition of the $\cdot$ operator from Section 2.4 of [13] because we structure the result according to the argument with the higher nesting depth, which means that the indexes for tuples combinations are taken from an argument chosen in advance and not determined for each combination of tuples separately. Of course, for homogeneous values both methods produce the same results, since for homogeneous list all its tuple indexes have the same length. Yet, in Taverna the resulting indexing can contain gaps. For example, if a, b, c, x, y and z are tuples, and a dot product of $[[[a,b]],c]$ and $[[x,y,z]]$ is computed, then the result would contain: $a \cup x$ on index $1/1/1$, $b \cup y$ on index $1/1/2$ and $c \cup z$ on index $1/3$. The $1/2$ position

would have to be filled up by some kind of empty value. This problem does not occur if the definitions of dot product from Section 2.4 of [13] are taken, i.e., $[[[a,b]],c] \cdot [[x,y,z]] = [[[a \cup x, b \cup y]], c \cup z]$ and $[[[a,b]],c] \cdot_r [[x,y,z]] = [[[a \cup x]]]$.

Here we discuss two possible solutions to the heterogeneous values problem in Taverna. One, is to adopt the formal semantics from this paper which seems intuitive while at the same time allows heterogeneous values everywhere and deals with them consistently. We elaborate on this in Section 3.1. The other solution is to avoid heterogeneous values at all, which we discuss in further detail in Section 3.2.

3.1. Allowing heterogeneous lists

The semantics defined in this paper deals with heterogeneous lists intuitively and consistently, yet its adoption in the workbench may require additional effort for adjusting some of the services. For examples, for the built-in flatten operation a definition is possible that processes the heterogeneous values consistently, that is:

$$\mathsf{flat}(x) \quad = \quad \begin{cases} [] & \text{if } x = [] \\ list(x_1) + \ldots + list(x_n) & \text{if } x = [x_1, \ldots, x_n] \end{cases}$$

where $list(x) = x$ for list values and $list(x) = [x]$ for mime values. Note, that with this definition, flattening of $[[x],[[y]]]$ yields $[x,[y]]$ and not $[[x],[y]]$ as it is the case in Taverna.

It is also possible to extend the type coercion mechanism described in Section 1.2 of [13]. If a certain service that requires its input lists to be homogeneous and of a specific nesting depth, gets a value that is non homogeneous or is of lower nesting depth, then there is always an intuitive interpretation of subvalues in that value as more deeply nested ones, namely by nesting them in singleton lists. For this a homogenisation function $\mathsf{hom}_\tau : [\![\tau]\!] \rightarrow [\![\tau]\!]$ can be used, that maps all complex values of type τ to homogeneous complex value with the maximum nesting depth possible in τ. It is defined such that:

- $\mathsf{hom}_{\mathcal{M}}(x) = x$,
- $\mathsf{hom}_{[\tau]}(x) = [\mathsf{hom}_\tau(x)]$, if $x \in [\![\tau]\!]$, and
- $\mathsf{hom}_{[\tau]}([x_1, \ldots, x_n]) = [\mathsf{hom}_\tau(x_1), \ldots, \mathsf{hom}_\tau(x_n)]$, if $[x_1, \ldots, x_n] \notin [\![\tau]\!]$.

The function hom_τ packs values that do not have the maximum nesting depth allowed in τ into singleton lists, and if the value does have the right nesting depth and $\tau = [\sigma]$ then it applies itself to the elements of the list for the type σ. For example, $\mathsf{hom}_{[[[\mathcal{M}]]]}([1,[2]]) = [\mathsf{hom}_{[[\mathcal{M}]]}([1,[2]])] = [[\mathsf{hom}_{[\mathcal{M}]}(1), \mathsf{hom}_{[\mathcal{M}]}([2])]] = [[[\mathsf{hom}_{\mathcal{M}}(1)], [\mathsf{hom}_{\mathcal{M}}(2)]]] = [[[1],[2]]]$. Thanks to this function we can safely assume that all services can deal with all homogeneous and heterogeneous complex values that belong to their input type.

It is worth pointing out that the existence of such type coercion is consistent with Taverna's philosophy of trying to fix the type mismatches for the user.

3.2. Adapting the semantics to avoid heterogeneous lists

An alternative to allowing heterogeneous lists and dealing with them consistently is to disallow them completely and allow only homogeneous lists. This means that we have to make sure that, under the

assumption that values containing heterogeneous lists cannot be entered by the user, no computation can produce values with heterogeneous lists. New values appear in a Scufl graph in the following cases: (1) they are produced in a processor execution, (2) they are created in the incoming-links strategy computation, (3) they are created in the product strategy computation, or (4) they are produced in a processor iteration.

As for (1), a service that was provided with only homogeneous values as arguments could produce heterogeneous results. One possibility is to interpret this as a failure, another is to always adapt the result value with the homogenisation function, i.e., if the service call returns v, then use $\mathsf{hom}_\tau(v)$ as the result, where τ is the smallest type of v.

As for (2), the select-first strategy does not change the values, so it cannot cause a heterogeneous value to appear. Yet, the merge strategy can, if the subsequent values provided to it are of different nesting depth. Similarly as in (1), this can be remedied with the use of the homogenisation function to extend the merge function as follows:

$$[\![\mathsf{merge}_{hom}]\!](t, v) = \begin{cases} [v] & \text{if } t = \bot \\ \mathsf{hom}_{min(\{\tau \mid (t+[v]) \in [\![\tau]\!]\})}(t + [v]) & \text{otherwise} \end{cases}$$

where $min : \mathcal{P}(\mathcal{T}_{tav}) \rightarrow \mathcal{T}_{tav}$ returns the minimal type in a set of types.

As for (3), neither the dot nor the cross product can produce heterogeneous values from homogeneous arguments, regardless which definition is chosen, so no extra care is necessary.

Finally, as for (4), a heterogeneous value can be created, if the processor returns results with different nesting depths in the subsequent iteration steps. Again, this can be solved with the use of homogenisation function, this time to extend the put function such that:

$$\mathsf{put}_{hom}(v, \alpha, t) = \begin{cases} \mathsf{put}(v, \alpha, t) & \text{if } \mathsf{put}(v, \alpha, t) = \bot \\ \mathsf{hom}_{min(\{\tau \mid \mathsf{put}(v,\alpha,t) \in [\![\tau]\!]\})}(\mathsf{put}(v, \alpha, t)) & \text{otherwise} \end{cases}$$

4. Related work

We start with comparing the presented work with that of Turi et al. in [15]. In that work a calculus is defined to represent Scufl graphs and a semantics is defined for them in terms of function that map workflow input values to a workflow output value.

The most important difference is that in our work we assume that calls to services have side effects, or, in other words, are observable events that are part of the behavior of the system. This means that two computations that call services in a different order or a different number of times, are not considered as equivalent, even if they compute the same output value. Therefore we describe the semantics of the system not in terms of functions, but in terms of a transition system that describes which calls are made in which order, which arguments were passed, and which output values are produced in the workflow outputs as the result of the Scufl graph. A consequence of the side-effect assumption is that, contrary to what Turi et al. assume, nesting a Scufl graph is no longer a purely syntactic construct because it synchronizes the consumption of values on the input ports and the production of values on the output ports, and so changes the observable behavior of the transition system. The same holds for the control edge, which can only have meaning if the order of computations is an observable aspect of the system.

The second difference with the work by Turi et al. is that their syntax is defined by a statically typed calculus. Since a Scufl graph is polymorphic and can work on inputs of different types, its semantics cannot always be described by a single calculus expression and may require a different one for each possible type of input value. In addition, the coercion to more deeply nested list types by wrapping and the implicit iteration strategy have to be made explicit in calculus expressions. As a result the mapping of a real Scufl graph, as described in our work, given certain presumed input types for the workflow inputs, to a calculus expression is not simple. We list the main three reasons why this is not straightforward:

1. Values that contain no basic vales, such as $[]$, $[[]]$ and $[[], [[]]]$ belong to multiple types. This has sometimes unexpected consequences. For example, assume a processor with one input port that expects values of type $[\mathcal{M}]$ and one output port that produces values of type $[\mathcal{M}]$. Given the polymorphic iteration mechanism one would expect that when presented with input of type $[[\mathcal{M}]]$ this processor will produce a result of type $[[\mathcal{M}]]$. However, if the value $[]$ is offered, which is of type $[[\mathcal{M}]]$, it will not iterate since the value is also of the expected type $[\mathcal{M}]$, and therefore produce a value of type $[\mathcal{M}]$ and not of type $[[\mathcal{M}]]$. As a consequence it is not true that if the type of the input of the input values is known, there is a single type that describes all possible outputs. Hence there cannot be a calculus expression that describes the behavior of a Scufl graph for a certain input type.

2. During iteration at deeper nesting levels, certain empty lists are removed. If the identity processor that expects values of type $\mathcal{M}$ receives the value $[[[]], [[v, w], []], [[]], [[x]], [[]]]$ with $v, w, x \in \mathcal{V}_{\mathcal{M}}$, it returns $[[], [[v, w]], [], [[x]]]$. Simulating this in the calculus would require a test for empty lists.

3. Another problematic case is the dot product at deeper nesting levels. Assume a processor with two input ports that computes the function $F(x, y)$, expects $\mathcal{M}$ on both its ports, and has the dot product iteration strategy. If it receives $v = [[v_1, v_2], [v_3, v_4]]$ and $w = [w_1, w_2, w_3]$ with $v_1, v_2, v_3, v_4, w_1, w_2, w_3 \in \mathcal{V}_{\mathcal{M}}$, then the result is $[[F(v_1, w_1), F(v_2, w_2)], [F(v_3, w_3)]]$. It is possible to compute F for the listed combinations by first flattening v, but the result would then be the flat list $[F(v_1, w_1), F(v_2, w_2), F(v_3, w_3)]$. The difficulty lies in simulating that the result is nested according to the structure of v.

A more fundamental difference between the calculus and our semantics is that failure of processors and Scufl graphs is not taken into account in the calculus. It can be argued that this aspect should be dealt with at a lower abstraction level, and in Taverna 1 there are indeed other mechanisms such as the specification of the number of retries and alternative services to deal with this. Moreover, in Taverna 2 the concept is replaced by special error values that indicate that certain subvalues were not produced correctly. However, we maintain that it is an interesting and useful feature to have at the language level, for example for specifying powerful fall-back strategies in Scufl itself. It is also essential for understanding the semantics of Scufl in Taverna 1, if only because it is used to represent conditional branching as discussed in Section 1.4 of [13].

Other related work defines the semantics with the use of workflow models based on Petri net [1] formalism like the DFL language [5] which was designed by the authors of this work for describing scientific workflows. Yet, we have found that the combination of Scufl's select-first incoming-links strategy and failure mechanism makes the mapping to Petri net based formalisms, and in particular to DFL, involved and creating little insight into the semantics of Taverna. This is because for example

extensive additional constructs are necessary to clean the tokens that are left in the simulation of select-first strategy and the exact number of such tokens is not known in advance if failures are possible.

For an extensive overview of systems and different approaches in the young but very active area of scientific workflows, the reader is referred to [3]. In the remainder of this section we will focus on one of these systems that shares with Taverna the distinct possibility of defining polymorphic workflows that can work on inputs of different types.

Kepler [9] is a scientific workflow system similar to Taverna in scope and application area. It also is mainly applied in bioinformatics but has been used in other areas like ecology, oceanography and geology. It is based on the Ptolemy II [2] system, which is a modeling and simulation environment with a formal semantics based on the Tagged Signal Model [8, 6], and extends it with new features and components for scientific workflow design and for efficient workflow execution. A distinguishing feature of Kepler is its ability to enact one and the same workflow according to different computation models which the workflow author specifies with the so called *director*. Kepler includes directors that correspond to process network, synchronous dataflow, continuous time, discrete event, and finite state machine computation models. As in Taverna the execution in a process network workflow model is driven by input data availability, i.e., an actor can fire, if some input data tokens are available on all its input ports. On the output ports it produces tokens with the result values, which are usually immediately transfered to further actors.

In [10] an extension of Kepler is proposed that allows polymorphic workflows to be defined. This is based on a technique that is similar to that of Taverna, where an actor that expects input of a certain type can also operate on collections of other types by automatically identifying nested values of the right type and operating on them. This type of polymorphic actor is referred to by the authors as a *collection-aware actor* and the workflows that contain them *collection-oriented workflows*. A difference with Taverna is that the user can specify in more detail how such nested values are identified and how they are iterated over, where in Taverna this is completely transparent. Another difference with Taverna is that Kepler features an elaborate and refined type system which explicitly allows heterogeneous values and this type system is used in the specification of the aforementioned collection-aware actors.

5. Conclusion

In these two papers (for the first part see [13]) we have presented a formal definition of the syntax and semantics of Scufl, the workflow specification language of the Taverna environment. The syntax is based on hierarchically nested graphs and the semantics is given by a transition system whose state is defined by the local states of each graph in the hierarchy, which in turn is defined by the state of each component in that graph.

We maintain that an expressive and effective formalism for describing its semantics is important for Scufl. Not only do we think it helps to build a correct and consistent implementation that is easy to understand by users, but it is also a requirement for formal research on a wide variety of subjects such as provenance, optimization, transformation and verification of soundness properties. A nice case in point is the Ptolemy system where formal semantics was investigated in an early stage [6] and played an important role in its development [8]. The semantics of Scufl as defined by us is very thorough and consistent with the observed behavior of the real implementation, but at the same time very involved and giving too much attention to behavior that might be considered as irrelevant detail. Clearly a simpler and

more elegant formalism would be more effective in the mentioned applications. Yet, we believe that a careful examination of all the details is a necessary first step to guarantee that more elegant definition does not oversimplify and stays close enough to the behavior of the defined Scufl graph. In addition, we believe that for any language that describes non-trivial systems of which users must sometimes precisely understand the semantics, verifying the possibility of an elegant formalization is the best way to test if the language is well-designed. This work contributes to such verification by highlighting the following aspects of the current implementation:

- The type coercion and implicit iteration mechanisms do not always protect from construction of heterogeneous values and the version of Taverna that we have investigated (version 1.7.1) not always deals consistently with them. This can be easily fixed, but as we show a definition of semantics is possible and even simpler with a type system that explicitly allows heterogeneous values. Therefore two semantics were proposed, one in which heterogeneous lists are allowed and consistently dealt with, and one in which they are not allowed and their creation is prevented.
- The polymorphic semantics of Scufl make the mapping of Scufl graphs to calculi such as presented by Turi et al. problematic, and in some cases even impossible. As explained in Section 4 this is partially because of the subtle behavior on empty lists, and partially because of how the polymorphic behavior of the product strategies is defined.
- During iteration over a value that contains empty lists some of them can be omitted, e.g., an identity processor iterating on $[[[]], [[1,2],[]], [], [[3]], [[]], []]$ yields $[[], [[1,2]], [], [[3]]]$.

The purpose of the definitions in this paper and its previous part [13] is not only to provide a semantics of Scufl that gives a precise and formally analyzable description of the defined Scufl graph, but also to investigate the different design choices that were made in assigning semantics to the various constructs in Scufl. In particular two issues were addressed. The first, is the already mentioned inclusion of the heterogeneous values. The second, is the semantics of the product strategies such as the dot product and the cross product. Based on their formal definitions they are analyzed and alternatives are proposed and evaluated.

References

[1] van der Aalst, W.: The application of Petri Nets to Workflow Management, *The Journal of Circuits, Systems and Computers*, **8**(1), 1998, 21–66.

[2] Department of EECS, U. B.: Ptolemy II project and system, `http://ptolemy.eecs.berkeley.edu/ptolemyII`, 2008.

[3] Guan, Z., Hernandez, F., Bangalore, P., Gray, J., Skjellum, A., Velusamy, V., Liu, Y.: Grid-Flow: a Grid-enabled scientific workflow system with a Petri-net-based interface, *Concurrency and Computation: Practice and Experience*, **18**(10), 2006, 1115–1140.

[4] Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., Van den Bussche, J.: Petri net + nested relational calculus = dataflow, *Proc. 13th Int. Conf. on Cooperative Information Systems (CoopIS), LNCS 3760*, Springer, Agia Napa, 2005.

[5] Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., Van den Bussche, J.: DFL: A dataflow language based on Petri nets and nested relational calculus, *Information Systems*, **33**(3), 2008, 261–284, ISSN 0306-4379.

[6] Lee, E. A., Sangiovanni-Vincentelli, A.: Comparing models of computation, *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, IEEE Computer Society, Washington, DC, USA, 1996, ISBN 0-8186-7597-7.

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

[8] Liu, X., Lee, E. A.: *CPO Semantics of Timed Interactive Actor Networks*, Technical Report UCB/EECS-2007-131, EECS Department, University of California, Berkeley, Nov 2007.

[9] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., Zhao, Y.: Scientific workflow management and the Kepler system: Research Articles, *Concurr. Comput. : Pract. Exper.*, **18**(10), 2006, 1039–1065, ISSN 1532-0626.

[10] McPhillips, T. M., Bowers, S., Ludäscher, B.: Collection-Oriented Scientific Workflows for Integrating and Analyzing Biological Data, *DILS* (U. Leser, F. Naumann, B. A. Eckman, Eds.), 4075, Springer, 2006, ISBN 3-540-36593-1.

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

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

[13] Sroka, J., Hidders, J.: Towards a formal semantics for the process model of the Taverna workbench. Part I, *Fundamenta Informaticae*, 92(3) 2009, 279–299.

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

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

Appendix

A. Basic properties of lexicographical ordering of number vectors

Definition A.1. (Number vector)
A *number vector of dimension* $n \in \mathbb{N}$ is a tuple $\bar{c} = (c_1, \ldots, c_n)$ with $c_1, \ldots, c_n \in \mathbb{N}$. The set of all such vectors is denoted as $\mathbb{N}^n$.

If $\bar{c} = (c_1, \ldots, c_n)$, then we let $(c_0, \bar{c})$ denote the number vector $(c_0, c_1, \ldots, c_n)$.

Definition A.2. (Number vector ordering)
Over $\mathbb{N}^n$ we let $\leq^n$ denote the lexicographical ordering, i.e., it holds that:

(i) $() \leq^0 ()$, and

(ii) $(c_1, \bar{c}) \leq^{n+1} (d_1, \bar{d})$ iff (a) $c_1 < d_1$ or (b) $c_1 = d_1$ and $\bar{c} \leq^n \bar{d}$.

Proposition A.1. $(\mathbb{N}^n, \leq^n)$ is a partially ordered set for each $n \in \mathbb{N}$.

Proof:
We show this by induction on n. For $n = 0$ this is clear since $\mathbb{N}^0 = \{()\}$ and $() \leq^0 ()$. For $n+1$ we can show, using the induction assumption for n, the reflexivity, antisymmetry and transitivity as follows:

A0.0.1. Reflexivity Let $(c_1, \bar{c})$ be an arbitrary vector from $\mathbb{N}^{n+1}$. By induction we know that $\bar{c} \leq^n \bar{c}$ and by (2b) it then follows that $(c_1, \bar{c}) \leq^{n+1} (c_1, \bar{c})$.

A0.0.2. Antisymmetry Let $\bar{c}', \bar{d}' \in \mathbb{N}^{n+1}$ such that $\bar{c}' \leq^{n+1} \bar{d}'$ and $\bar{d}' \leq^{n+1} \bar{c}'$ where $\bar{c}' = (c_1, \bar{c})$ and $\bar{d}' = (d_1, \bar{d})$. Based on the definition of number vector ordering this is only possible if $c_1 \leq d_1$ and at the same time $d_1 \leq c_1$ which together imply $c_1 = d_1$. From this and the $\bar{c}' \leq^{n+1} \bar{d}'$ it follows that $\bar{c} \leq^n \bar{d}$. Similarly we get $\bar{d} \leq^n \bar{c}$. Now from the induction assumption we know that $\bar{c} = \bar{d}$ which completes the proof since we already showed that $c_1 = d_1$.

A0.0.3. Transitivity Assume that $(c_1, \bar{c}) \leq^{n+1} (d_1, \bar{d})$ and that $(d_1, \bar{d}) \leq^{n+1} (e_1, \bar{e})$. Then one of the following cases holds: (i) $c_1 < d_1$ and $d_1 < e_1$, (ii) $c_1 < d_1$ and $d_1 = e_1$, (iii) $c_1 = d_1$ and $d_1 < e_1$, and (iv) $c_1 = d_1 = e_1$, $\bar{c} \leq^n \bar{d}$ and $\bar{d} \leq^n \bar{e}$. In the first three cases (i), (ii) and (iii) it follows that $c_1 < e_1$, and therefore $(c_1, \bar{c}) \leq^{n+1} (e_1, \bar{e})$. In case (iv) it follows that $c_1 = e_1$ and by induction that $\bar{c} \leq^n \bar{e}$, and therefore $(c_1, \bar{c}) \leq^{n+1} (e_1, \bar{e})$. □

Definition A.3. (Well-founded)
A partially ordered set $(V, \leq_V)$ is said to be *well-founded* if it holds for every non-empty subset $V' \subseteq V$ that it contains at least one minimal element, i.e., an element $v \in V'$ such that for all $w \in V'$ if $w \leq_V v$ then $w = v$.

Proposition A.2. The partial order $(\mathbb{N}^n, \leq^n)$ is well-founded.

Proof:
We prove this with induction on n. For $n = 0$ it holds since there is only one element in $\mathbb{N}^0$. Next, we consider $n+1$. Let c'_1 be the smallest number in $\{c_1 \mid (c_1, \bar{c}) \in \mathbb{N}^{n+1}\}$. Then let $\bar{c}'$ be the minimal element in $\{\bar{c} \mid (c'_1, \bar{c}) \in \mathbb{N}^{n+1}\}$ w.r.t. $\leq^n$, which by induction exists. We now show that $(c'_1, \bar{c}')$ is a minimal element of $\mathbb{N}^{n+1}$. Assume that $(c_1, \bar{c}) \leq^{n+1} (c'_1, \bar{c}')$. By the definition of $\leq^{n+1}$ it holds that $c_1 \leq c'_1$ and by the definition of c'_1 that $c'_1 \leq c_1$, and so $c_1 = c'_1$. From this it follows that $\bar{c} \leq^n \bar{c}'$. By induction and the fact that $\bar{c}'$ is a minimal element of a set of which $\bar{c}$ is also an element, it follows that $\bar{c}' = \bar{c}$. It therefore holds that $(c_1, \bar{c}) = (c'_1, \bar{c}')$. □

Proposition A.3. $(V, \leq_V)$ is well-founded iff V contains no infinite descending chains, i.e., there exists no injective function $f : \mathbb{N} \to V$ such that for every $n \in \mathbb{N}$ it holds $f(n+1) \leq_V f(n)$.

Proof:
We will prove both implications by contradiction. Let $V' \subseteq V$ be a non-empty subset without a minimal element. The sequence $f : \mathbb{N} \to V'$ can be defined as follows. $f(0)$ is an arbitrary element of V'. If f is defined for all $k \leq n$ and for every $k < n$ it holds $f(k+1) \leq_V f(k)$ but $f(k+1) \neq f(k)$ then

as $f(n+1)$ we choose any other element from V' such that $f(n+1) \leq_V f(n)$. The existence of such element follows from the fact that $f(n)$ is not minimal in V'. The other way around, if there exists an infinite descending chain f, then the image $\{f(1), f(2), \ldots\}$ is a non-empty subset of V that has no minimal element. □

Corollary A.1. The partially ordered set $(\mathbb{N}^n, \leq^n)$ contains no infinite descending chains.

Proof:
This follows directly from Propositions A.2 and A.3. □