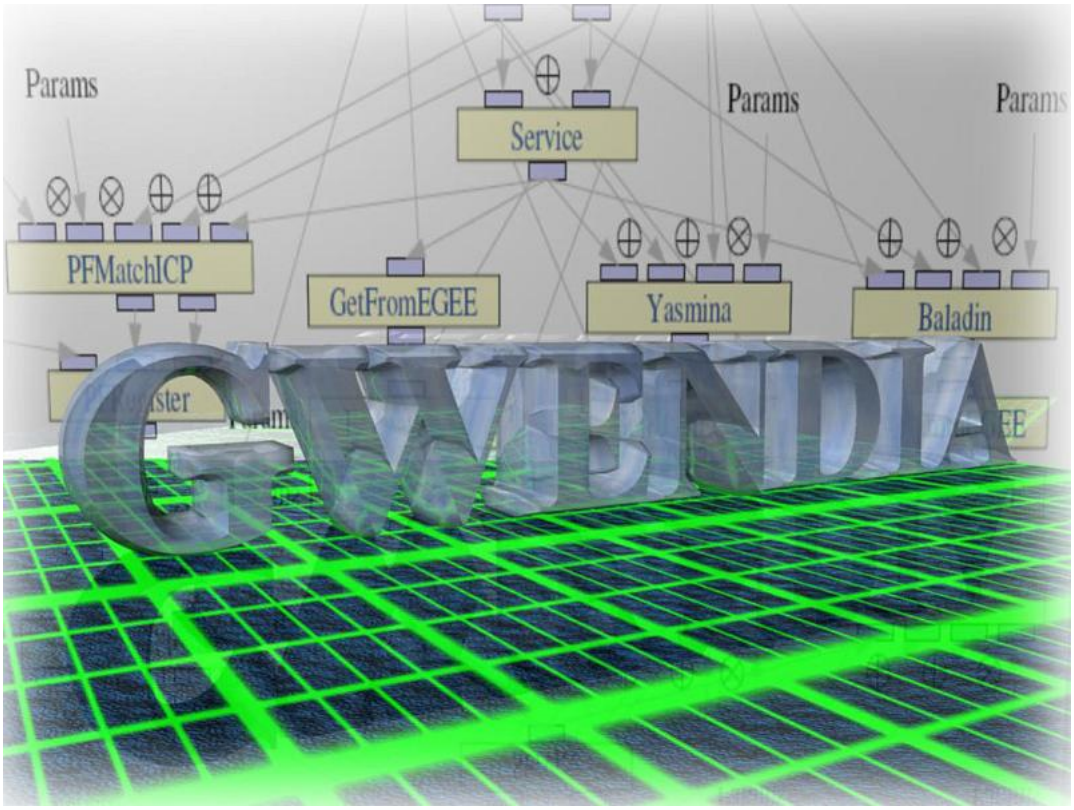
L1.2: Workflow language proposal



Johan Montagnat RAINBOW (I3S) johan@i3s.unice.fr
Benjamin Isnard GRAAL (LIP) benjamin.isnard@ens-lyon.fr
Ketan Maheshwari RAINBOW (I3S) ketan@polytech.unice.fr

Abstract

Workflow language proposal for the Gwendia project.

1 Introduction

1.1 Motivations

Workflow languages play an important role in the workflows design process given that they constrain the kind of computational pattern that can be represented. As a matter of fact, a very large number of workflow languages exist today, despite the very general adoption of C-like traditional programming languages, exhibiting the variety of needs for workflow-based applications. In the context of grid computing, this variety is exacerbated by the complexity of intrinsically parallel programming constructs. In deliverable L1.1 [9], we have proposed a classification of existing workflow languages emphasizing on expressiveness. This classification significantly differs from earlier taxonomies related either to the properties of workflow engine enactors [11] or the language abstraction level [2]. In our view, workflow languages and workflow enactors are more tightly bound than first appear. At the notable exception of BPEL, most languages are associated to a single enactor. Language have often be specialized for specific needs. They propose control structures that are not easily implementable in a different language and as a consequence, there exists few bridges between languages. Among workflow engines, Kepler is notorious for enabling different models of computations inside a single workflow [3, 1]. Different Kepler *directors* implement different workflow enactment strategies based on very heterogeneous languages (including data flows, process networks, discrete time event...). However, the different directors are completely independent and if they can be combined hierarchically using sub-workflows with different models of computation, each workflow level uses an homogeneous model of computation.

The GWENDIA project specifically focuses on coarse grain data-intensive scientific applications for which grid computing can improve performances dramatically. In this context, L1.1 [9] identifies two different approaches that are promising:

- data flows enable the transparent description of massively data-parallel applications;
- Direct Acyclic Graphs (DAGs) enable the low level description of the computing schedule.

While data flows are more geared towards language expressivity, DAGs are more effective for achieving high performance. In the former case, users are able to describe very complex data flows in a compact and service-oriented framework while in the later case parallelism and all inter-dependencies are statically described in a potentially very large size graph. Although both approaches are not *a priori* exclusive (DAGs are often automatically generated from a higher level description languages), DAGs impose some constraints (acyclic nature of graphs in particular) that make difficult or even impossible the automatic instantiation of every data flows into a DAG. In a service-oriented data flow approach, the computing agents (*processors*) are defined independently from the data sets to be processed. Conversely, a DAG is composed of *tasks* corresponding to the instantiation of processors and the the data set they process. In the context of GWENDIA, both a data flow oriented language (Scufl, see section 2.1) and a DAG-oriented language (MA-DAG, see section 2.2) are considered, aiming at achieving both abstract representation and high performance. In the reminder of the document we try to conciliate those partially conflicting approaches.

The objective of this technical document is to propose a GWENDIA workflow language. This proposition is based on an enrichment of existing languages. It aims at achieving two

different goals:

- to ease the description of the complex application data flow use cases addressed within the project from a user point of view; and
- to ensure good performances and grid resources usage.

As will be discussed further, compromises have to be adopted to reach both targets. The language that we propose is thus a high level abstraction that can be instantiated in different execution formats.

In the rest of this paper, the Scufl and the MA-DAG languages are first introduced (section 2). The application use case requirements are analyzed in light of these languages (section 3). Languages extensions are proposed to match the needs expressed (section 4). Finally, a formal representation of the GWENDIA language, integrating all elements is described in section 5.

2 Workflow languages considered

2.1 Scufl data flow-oriented language

Scufl was introduced within the myGrid project[1] to represent data flows enacted through the Taverna workflow engine [7]. Scufl is one of the first grid-oriented data flow language. It is represented as an XML document. One flaw of Scufl is that it is internally used for the needs of the Taverna workflow engine but that the language has never been properly defined nor documented. The basic semantic of Scufl as been recently formalized though [10]. As we will see in section 4.2, this formalization is still an early attempt to clarify Scufl and the current implementation of Taverna does not fully comply with it.

The future of Scufl is uncertain as it is currently undergoing massive and undocumented transformations in the context of Taverna2 development. It is too early to comment on the completely new Taverna2 data flow language as it has not been released yet and it is unclear whether it will ever be documented. However, the first information available indicate that it is totally uncorrelated to Scufl (only a Scufl converter will be provided for handling legacy workflows) and that it is fully user customizable (the invocation behavior of each workflow processor can be completely revised by the user). In this context, the semantic of the language can hardly ever be defined: each workflow instance will carry a specific invocation semantic depending on the plug-ins redefined by the user. We believe that in its desire to achieve maximal flexibility, Taverna2 has adopted a risky approach. In a short term, Taverna2 workflows can become unreadable to any user without analyzing the workflow-specific plug-ins code. We disagree with this approach that brings to much confusion on the sake of flexibility and we believe that a proper language, with a clear semantics need to be defined.

Scufl is a simple graph-oriented language than is defined through few XML tags: *processors*, *data links*, *coordination constraints* and *iteration strategies*.

Processors. In Scufl, computing activities are named *processors*. For convenience there exist different processor kinds. Without loss of generality we will focus on *java internal processors* which are executing predefined java-coded operations at the level of the workflow engine, *beanshell processors* which are interpreting java user code locally and *web*

[1] myGrid UK e-Science project: `www.mygrid.org`

service processors which are invoking standard web services. In a Scufl workflow, processors may fire multiple times depending on the data items that they receive to process. The processor XML tag can contain many different tags that specify the processor behavior. For example, beanshell processors have a specific tag to hold the java code to be interpreted, web service processor have another tag to define the service WSDL description document endpoint, etc. A commonality to all processors is that they define named input and output *ports*. Ports are buffers that hold either data items sent to the processor for computation or data items produced by the processor.

Data links. An output port of a processors P_0 and an input port of a subsequent processor P_1 are connected through links. *Data links* are by far the most widely used in Taverna data flows: a data link expresses a data dependency between P_0 and P_1. P_1 can only be enacted once it received one data item or more into all its input ports through data links. When exactly P_1 fires is determined by the processor iteration strategy defined below.

Coordination constraints. *Coordination constraints* are a specific kind of processor links that do not require any data to be exchanged between connected processors. The target processor of a coordination constraint can only fire once the source processor has completely executed, *i.e.* once it has fired for all data sets to process and it is certain that no further firing will be needed in the execution of this workflow. It is to be noted that cycles of linked processors may exist in Scufl. However, the behavior of the Taverna enactor in presence of cycles is hill defined. A clear semantic for data link cycles can be defined though, as discussed in section 4. Control links cannot appear within a cycle though given that the complete execution of a processor within a cycle cannot be determined.

Iteration strategies. Despite its apparent simplicity, the Scufl language provides a rich data flow semantic through *iteration strategies*. They define how many times a processor fires when it receives input data on two or more input ports. There are two basic iteration strategies when considering a pair of input ports p_0 and p_1 of processor P: they are known as *dot* and *cross* product respectively. In the case of a dot product, the processor will fire once for each pair of input data items received on p_0 and p_1. For example if p_0 receives data items a and b, and p_1 receives data items c and d, then the processor will fire two times, to produce $P(a, c)$ and $P(b, d)$. The dot product corresponds to a traditional *one-to-one* execution semantic. In case of parallel execution, the order of data items processed may be shuffled and care as to be taken by the workflow manager to respect the user expected semantics of computations. The cross product corresponds to an *all-to-all* execution semantics and in the former example, 4 data items would be produced: $P(a, c)$, $P(a, d)$, $P(b, c)$ and $P(b, d)$. By combining dot and cross produces in an arithmetic expression, complex iteration strategies can be defined for processors with more than two input ports.

List data sequences. An important aspect of the Taverna workflow engine is to support lists of consecutive data items and lists of embedded list semantics. Lists considerably enrich the semantic of Taverna and are important for implementing the GWENDIA use cases as described in section 3. However, lists are not clearly part of the Scufl workflow language but rather a consequence of the data flow strategy implemented in Taverna: only the

beanshell processor type properly handles lists although there is no reason why lists should be related to a specific processor kind. It is a flaw in Taverna design that we propose to fix in section 4 by extending the language to explicitly support lists. With lists, several data items can be logically considered as a single group of data. Some processors may process a complete list in a single invocation while others may process list items individually depending on the semantic of the processor. For instance, an arithmetic "square" operation may be invoked on individual integers while a statistical "mean operation" will be invoked on a list if integers. Embedded lists enable multiple level data sets management and provide support for synchronization of data items before processor invocation.

Example. The following XML extract gives an example of a Scufl workflow definition. The workflow is composed of 2 image sources, 1 beanshell processor and 1 output (sink). The processor in this example is an image registration processor which computes from a pair of input images the geometrical transformation needed to align the floating image onto the fixed image spacial frame. The 2 processor inputs are correlated through a dot product iteration strategy. Three data links define the workflow graph.

```
<s:scufl xmlns:s="http://org.embl.ebi.escience/xscufl/0.1alpha" version="0.2" log="0">

  <s:source name="FixedImage" />
  <s:source name="FloatingImage" />

  <s:processor name="registration">
    <s:beanshell>
      <s:scriptvalue>...</s:scriptvalue>
      <s:beanshellinputlist>
        <s:beanshellinput s:syntactictype="'text/plain'">ref</s:beanshellinput>
        <s:beanshellinput s:syntactictype="'text/plain'">floating</s:beanshellinput>
      </s:beanshellinputlist>
      <s:iterationstrategy>
         <i:dot>
            <i:iterator name="ref"/>
            <i:iterator name="floating"/>
         </i:dot>
      </s:iterationstrategy>
      <s:beanshelloutputlist>
        <s:beanshelloutput s:syntactictype="'text/plain'">matrix</s:beanshelloutput>
      </s:beanshelloutputlist>
      <s:dependencies s:classloader="iteration" />
    </s:beanshell>
  </s:processor>

  <s:sink name="Transformation" />

  <s:link source="FixedImage" sink="registration:ref" />
  <s:link source="FloatingImage" sink="registration:floating" />
  <s:link source="registration:res" sink="Transformation" />

</s:scufl>
```

2.2 MA-DAG task-oriented language

The MA-DAG language describes a directed acyclic graph of tasks represented as an XML document. This language is part of the DIET project[2] and is used to execute workflows on grids managed by the DIET grid middleware. The semantics of this language is simple as it only describes a set of tasks (called *nodes* in the language) and either data flows between them or simple dependency relationships. It does not contain any DIET-specific concept

[2]DIET Grid Middleware projecthttp://graal.ens-lyon.fr/~diet/

so the structure and syntax of the language is relatively generic. It does not currently manage lists of data items so the number of inputs and outputs of each task must be statically defined.

XML Document structure. The toplevel element is a `<dag>` element which contains `<node>` elements. Nodes correspond to executable tasks with defined input data which will be submitted to the grid middleware for mapping to a ressource and execution. The type of task to execute is specified by the `path` attribute of each node which contains the name of a service available on the grid. The precedence relationships between nodes in the DAGs are defined either by data flows (an input port of a node can have a source port linked to it, or an output port can have a sink port linked to it), or by simple control flow using the `<prec>` element that contains the id of another node.

Node description. A `<node>` element is composed of arguments, input ports, output ports, in-out ports and eventually preceding relationships. It must be uniquely identified within the DAG by its `id` attribute and contains the name of the service that will be used for execution on the grid (`path` attribute). Each port of the node (elements `<arg>`, `<in>`, `<inOut>` or `<out>`) must have a unique `name` attribute within the node and a defined `type` attribute which should contain one of the data types understood by the grid middleware. Some additionnal attributes are used only for the matrix type (base_type, nb_rows, nb_cols and matrix_order). The syntax of `source` or `sink` attributes is: `node_id#port_name`.

Document DTD. The current version of the DTD for the MA-DAG language is the following:

```
<!ELEMENT dag (node*)>
<!ELEMENT node (arg*, in*, inOut*, out*, prec*)>
<!ATTLIST node
   id ID #REQUIRED
   path CDATA #REQUIRED>
<!ELEMENT arg EMPTY>
<!ATTLIST arg
   name CDATA #REQUIRED
   type CDATA #REQUIRED
   value CDATA #REQUIRED
   base_type CDATA #IMPLIED
   nb_rows CDATA #IMPLIED
   nb_cols CDATA #IMPLIED
   matrix_order CDATA #IMPLIED>
<!ELEMENT in EMPTY>
<!ATTLIST in
   name CDATA #REQUIRED
   type CDATA #REQUIRED
   source CDATA #IMPLIED
   base_type CDATA #IMPLIED
   nb_rows CDATA #IMPLIED
   nb_cols CDATA #IMPLIED
   matrix_order CDATA #IMPLIED>
<!ELEMENT inOut EMPTY>
<!ATTLIST inOut
   name CDATA #REQUIRED
   type CDATA #REQUIRED
   source CDATA #IMPLIED
   base_type CDATA #IMPLIED
   nb_rows CDATA #IMPLIED
   nb_cols CDATA #IMPLIED
   matrix_order CDATA #IMPLIED>
<!ELEMENT out EMPTY>
```

```
<!ATTLIST out
   name CDATA #REQUIRED
   type CDATA #REQUIRED
   sink CDATA #IMPLIED
   base_type CDATA #IMPLIED
   nb_rows CDATA #IMPLIED
   nb_cols CDATA #IMPLIED
   matrix_order CDATA #IMPLIED>
<!ELEMENT prec EMPTY>
<!ATTLIST prec
   id CDATA #REQUIRED>
```

Example. The following XML document is an example of MA-DAG workflow. It is composed of 3 different types of tasks: first the "dockingparameters" task takes as input one parameters file and an additional integer value. It produces two files that are processed in parallel by two instances (ie two tasks) of the "docking" service. These tasks each produce one output value. The outputs are finally processed by a single "dockingmerge" task. The precedence relationships are here defined by the data flows so there is no `<prec>` element used.

```
<dag>
  <node id="A" path="dockingparameters">
    <arg name="arg1" type="DIET_LONGINT" value="2000" />
    <arg name="arg2" type="DIET_FILE" value="parameters.dat" />
    <out name="out1" type="DIET_FILE" />
    <out name="out2" type="DIET_FILE" />
  </node>
  <node id="B1" path="docking">
    <arg name="arg1" type="DIET_LONGINT" value="4000" />
    <in  name="arg2" source="task1#out1" type="DIET_FILE" />
    <out name="out1" type="DIET_LONGINT" />
  </node>
  <node id="B2" path="docking">
    <arg name="arg1" type="DIET_LONGINT" value="4100" />
    <in name="arg2" source="task1#out2" type="DIET_FILE" />
    <out name="out1" type="DIET_LONGINT" />
  </node>
  <node id="C" path="dockingmerge">
    <arg name="arg1" type="DIET_LONGINT" value="3000" />
    <in name="arg2" source="B1#out1" type="DIET_LONGINT" />
    <in name="arg3" source="B2#out1" type="DIET_LONGINT" />
    <out name="out1" type="DIET_LONGINT" />
  </node>
</dag>
```

3 Use cases requirement analysis

3.1 Cardiac data flow

The GWENDIA data flows exhibit complex data flows as described in detail in technical document [4]. As an illustration, a simplified data flow extracted from the cardiac application is given in figure 1. The data source is the `Patient ID` processor which provides patient identifiers. For the sake of simplicity, the data flow is illustrated for a single patient "P" but the application usually applies to a patients database. For each patient, a temporal sequence of 3D images covering the beating heart cycle is analyzed. Each 3D volume is composed of 2D slices. Depending on the processor, the data may be processed as independent 2D slices, as 3D volumes or even as a complete 4D sequence. The data is stored on disk as individual 2D slices in DICOM format. When the "DICOM reader"

processor is invoked, it returns the list of files associated to the input patient. In our example, the 4D image is composed of 2 volumes (green and red), each one being composed of 4 2D slices. Given that the processor is invoked only once in our example, it will return a single output containing all slices to be processed. The "Image crop" subsequent processor processed slices individually: it has to be invoked 8 times in this case (possibly in parallel), hence the compound output of "DICOM reader" has to be disassembled prior to the processor invocation. Note that nothing guarantees the ordered processing of the slices and the cropped slices resulting from the processor activity may be produced in a completely different order than the input slices. The subsequent "Interpolation" processor needs to process complete volumes. Hence, slices have to be reordered and grouped prior to the processor invocations (2 times in this case). "Interpolation" produce volumes that will be consumed by the "Pyramid decomposition" processor as is. This processing produce several resolution 3D images for each input volume however: as for the DICOM reader, the outputs are compound structures. Finally, the gradient computing processed the different resolution volumes individually.

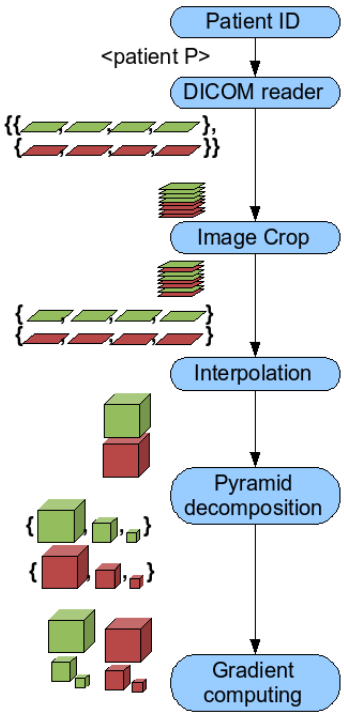


Figure 1: Simplified cardiac data flow

This example demonstrates that data flow transformations are needed between processors invocations: the processors have to expose the data sets that they provide and the workflow engine has to handle the data flow manipulation as required. There are two kinds of data flow transformations involved in this example:

- **Data sets depth**. Data links are transporting data items that may be grouped or conversely groups of data items than need to be flattened. For instance, the output of the "DICOM reader" has to be interpreted as a 2-dimensional list of slices, that needs to be flattened prior to "Image crop" invocation. Conversely, the slices resulting from "Image crop" processing have to be grouped in a correct order prior

to "Interpolation" invocation. These operations may involve any depth flattening or grouping.

- **Data formats**. The data is manipulated in different formats. For instance, the "Interpolation processor" interprets groups of 2D input slices (encoded in the 2D DICOM file format) as volumes and produces 3D images (encoded in the 3D HDR image format). Although semantically equivalent, these two formats completely differ and a transformation takes place within the processor.

The format transformation is completely domain dependent and is not handled at the workflow management level. The workflow engine is only exposed to the depth of the data set and make no assumption on the correlation between processors input and output formats.

It is to be noted that the data items grouping operation corresponds to a partial data synchronization operation that was identified in deliverable L1.1 [9].

3.2 Drug discovery data flow

The drug discovery data flow is much simpler but it causes a serious scalability problems: from hundreds of thousands to millions of computing tasks can be triggered almost instantaneously by this application. The workflow engine needs to regulate the amount of tasks simultaneously processed to avoid overloading the workflow host. This can be achieved either by an internal task submission policy of the workflow engine or a transformation of the workflow in a regulatory loop which maintains a constant number of tasks execution on the grid infrastructure.

Another particularity of the drug discovery workflow is to involve a thresholding operation. The threshold value can be either determined statically, from experience, or dynamically, as a percentage of the best results computed.

3.3 Language properties analysis

As illustrated in the cardiac application example, the workflow engine is exposed to the depth of the data flow items. In Scufl, there is a partial support for expressing multi-depth data items to be transferred on the data links: the beanshell processors define different cardinality input/output ports expressing the expected input/output depth of the items received/processed. The data items are mapped to java objects manipulated in the beanshells code (`ArrayList` objects are used to represent groups of data items). When the depths of an output port and an input port it is connected to through a data link do not match, the Taverna engine performs the necessary flattening or grouping operations. Unfortunately, this feature is restricted to beanshell processors. In particular it is not available for web services that are used for enacting application on the grid infrastructure. A hybrid solution could be implemented using web service grid processor and intermediate beanshell processors for handling the data flow. Although this solution provides the desired functionality for the cardiac application, it is heavy, counter-intuitive and it makes workflow maintenance and evolution unnecessarily tiresome.

With DAGs, the data flow has to be explicitly specified in the workflow graph: there are as many instantiation of a workflow processor into a DAG node as times the processor needs to be invoked. This strategy can only be implemented with static graph which size is completely known. However in the cardiac application example, the number of volumes in for each patient varies. A specific DAG would thus be necessary for each patient.

The drug discovery application introduces the extra need for control structures such as loops and conditionals. Loops can be represented in a data flow, as will be discussed in section 4.4. DAGs can only represent loops by unfolding the loop iterations sequentially. As a consequence, only bounded loops with a known number of iteration can be treated statically. Conditionals can be represented in data flows as filters. DAGs have no conditional expression.

4 Language extensions

This section introduces several workflow language extensions that are desirable for the implementation of the GWENDIA application use cases discussed above. The result is a data flow language with embedded control structure. The next section will formalize the GWENDIA language.

4.1 Typed data items

The workflow input/output ports need to be typed in order to ensure the workflow coherence. The MA-DAG support primitive types handling. List types will be added as discussed below. The Scufl language is not typed. The MOTEUR workflow engine can exploit web-services argument types defined in the WSDL document to validate the coherence of a flow of web-services. The type information will be integrated to the workflow language so that it applies to all processor kinds. Mapping between workflow-defined types and specific service types (*e.g.* web service XSD types) will be needed.

4.2 Lists management

As discussed previously, grouping data into different depth lists enables the processing of complex data flows. Moving from data items to lists of data items is very intuitive in a data flow framework. A stream of data items flowing on a data link from a data flow may be interpreted as a list. In fact, lists processing is completely integrated in *functional languages* such as Haskell[3]. As Ludäsher and Altintas outlined, data flows are very similar to functional languages [5]. The invocation semantic of a processor receiving an input list corresponds to a *map* of the processor function to the list items. An extension of Kepler was proposed, based on the map operator. Similarly, the Taverna workflow engine is adopting a fully functional approach, enriched by a multi-depths lists manipulation semantic when considering beanshell processors. However, it is to be noted that none of these earlier approach fully comply with the cardiac workflow requirements:

- The functional languages have been studied and formalized in depth. Although their execution order is not completely deterministic and some language interpreters support parallelism, the regular sequential flow of codes is hardly as expressive as workflow graphs to represent parallelism. In addition, iteration strategies are not natively supported and require complex implementations.
- Kepler can manipulate lists but it does not include iteration strategies which considerably reduces the its data flow director expressiveness.

[3] http://haskell.org

- Taverna only implements lists management for beanshell processor. This capability is not really part of the Scufl language. Moreover, Taverna 1 is not grid-interfaced and it has strong limitations concerning the data parallel execution of its data flows.

We aim at proposing a list operations semantic that is compatible with fully asynchronous execution of data flows and well integrated into the GWENDIA workflow language.

4.2.1 Lists management semantic

A data flow lists management semantic is defined as follows:

- Data links are transporting lists: all items sent from a same workflow data source or a processor output port are known to belong to a same list. It corresponds to a top level list that may contain items themselves represented as lists.
- Processors define the expected list depth on each of their input or output port.
- The depth m of an output and the depth n on an input port it is connected to may differ. If $m > n$ then the list is flattened by $m - n$ levels before the items are delivered to the target port. If $m < n$ then the items are collected by $n - m$ level lists before they are delivered. Note that if the items produced in the output port cannot be collected, because they are not originally belonging to a high level enough list, the data link is not valid.
- The workflow engine ensures the reordering of list items when they are collected in a list.

As outlined in [5], in a data flow the entities transferred through the data links remain individual data items. It is sufficient to transfer a special "end of list" data tag that is interpreted by the workflow engine to detect the complete availability of a list in an input port. In an asynchronous implementation, the list items also need to be numbered to recover their correct ordering when collecting items in a list.

The end of list tags are also convenient to detect computation termination. Each output port delivers a list of produced items. In a completely asynchronous execution, a processor can produce results while it is still expecting further inputs to process. It is only when all input end of list tags have been received that the processor can in turn deliver an end of list tag to the subsequent processors. Figure 2 illustrates the data flow involved around the "DICOM reader" processor of the cardiac application (see figure 1). In this example, the processor received two data items, `d[0]` and `d[1]`, corresponding to two patient identifiers. It will fire twice and for each patient it will produce a 2-dimensionnal list containing all volumes for each patient and all slices for each volume. The input patients list is terminated by an `<eol>` end of list identifier. Similarly, each volume is delimited by an `<eol>`, tagged with the patient number and the volume number, each sequence (list of volume) is delimited by an `<eol>` tagged with the patient number, and finally the whole output list is delimited by a higher level `<eol>` tag.

It is to be noted that the list collection capability provides a data synchronization functionality. In the example of figure 1, the "Interpolation" processor uses this capability to synchronize all image slices *from a same patient, from a same volume* prior to processing the coherent slices stack.

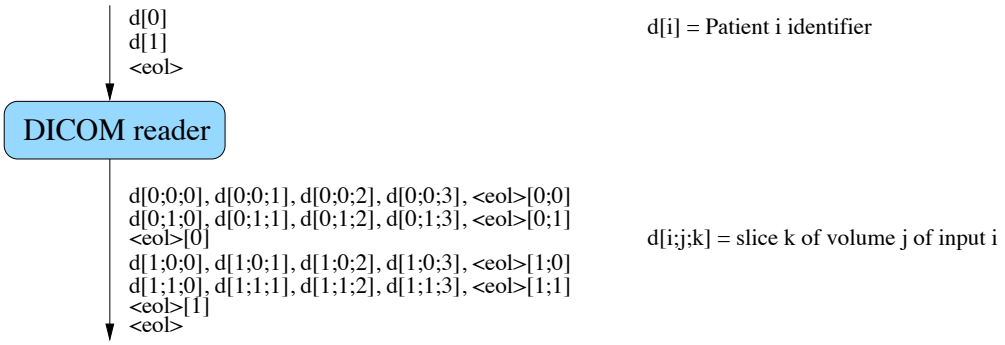


Figure 2: Data items and list structures involved in the "DICOM reader" processor example. This processor produces a 2-dimensionnal list for each input patient ID received.

4.2.2 Lists management in MOTEUR

Currently, MOTEUR does not handle lists. Beanshells with list input/output types are interpreted as receiving/returning a single item independently of the processor expected list depth. As a consequence, MOTEUR does not provide partial data synchronization and cannot enact the cardiac workflow application use case without heavy tricks implemented through service ad-hoc wrappers. MOTEUR does handle asynchronous data flows and is capable of identifying a data item history for the need of iteration strategies implementation in a concurrent context though. Provided that the processors are instrumented to declare expected lists depth, MOTEUR can be extended. List will be managed at the same level as iteration strategies as will be detailed in section 4.3.

4.2.3 Lists management in MA-DAG

Currently the data flows described in the MA-DAG language are limited to scalar, file or matrix data types. To manage lists of data items as described previously, the language should include new types of data flows, as well as a new syntax to be able to include references to items within a list. Therefore a new **container** data type must be implemented, as a generic list of items that can contain any other type of data including lists. The MA-DAG language specification will contain:

1. New possible values for the `type` attribute valid for the different kind of ports within a node (ie elements `<arg>`, `<in>`, `<inOut>` or `<out>`). For example, a list of integer items would be described by $type = "LIST(DIET\_INT)"$, and a list of lists of files would be described by $type = "LIST(LIST(DIET\_FILE))"$.

2. An extended syntax for the `source` attribute of `<in>` or `<inOut>` elements and for the `sink` attribute of `<out>` elements. For example, a node (B) taking as input the first element of the list generated by node A would be described as:

```
<node id="node_A" path="service_A">
  <out name="out" type="LIST(DIET_FILE)"/>
</node>
<node id="node_B" path="service_B">
  <in name="in" type="DIET_FILE" source="node_A#out[1]">
</node>
```

Another example for describing a node (C) that takes as input a list of lists of files composed of two elements that are the lists generated by nodes A and B:

```
<node id="node_A" path="service_A">
  <out name="out" type="LIST(DIET_FILE)"/>
</node>
<node id="node_B" path="service_B">
  <out name="out" type="LIST(DIET_FILE)"/>
</node>
<node id="node_C" path="service_C">
  <in name="in" type="LIST(LIST(DIET_FILE))" source="node_A#out;node_B#out">
</node>
```

These enhancements to the MA-DAG language would allow the processing by the grid middleware of tasks that have a dynamic number of inputs and outputs, such as the cardiac workflow described in section 3.1. Currently the DIET grid middleware does not handle such data structures but the specification of the API is being standardized by the OGF[4] and an implementation of this API will be done soon within the DIET project.

Remark: these enhancements do not affect the MA-DAG language DTD

4.3 Iteration strategies

4.3.1 Match iteration strategy

Scufl defined two basic iteration strategies to combine a pair of input ports: the *cross product* corresponding to an *all-to-all* composition semantic and the *dot product* corresponding to a *one-to-one* semantic. Most data flows only implement the one-to-one semantic (*e.g.* Kepler PN director). The addition of the all-to-all semantic and the ability to compose these basic strategies in a more complex expression considerably enrich the language semantic.

In WORKS'06 [6] we have proposed an alternative iteration strategy, close to the one-to-one semantic but which depends on the application semantic. The WORKS'06 strategy is based on the definition of groups of related workflow input data by the user. For instance, the user can define the data belonging to a same patient as members of a same group. The WORKS'06 operator will match two data items received on two input ports if they belong to (or one of their ancestor in the data production history tree belongs to) a same group (*e.g.* if they ultimately relate to a same patient). By opposition, the one-to-one semantic is dependent on the order of arrival of the data items. Furthermore, the WORKS'06 operator uses the provenance information on the data sets transformed within the workflow. It proved to be a powerful operator to ensure that only semantically related data items are processed altogether at any depth in the workflow processing.

In the GWENDIA language, we propose to extend the WORKS'06 operator semantic in a *match* iteration strategy, later on denoted $\oplus$, that complements the dot and cross products (denoted $\odot$ and $\otimes$ respectively). The match operator will relate to a specific data group, as opposed to the WORKS'06 operator which matches any pair belonging to any identical group. As a consequence, a data item may belong to several groups (*e.g.* $G1$ and $G2$) and be related in a processor P to other data items belonging either to $G1$ or to $G2$ depending on the semantic of P. For instance, let us consider multi-modality medical images: for each patient P_0 and P_1 are acquired $T1$, $T2$ and PD magnetic resonance image (denoted $T1_0$ and $T1_1$, etc). The images can be grouped by patient (*e.g.* it makes sense to analyze different images from a same patient if the processor is performing multi-modal

[4]Open Grid Forum http://www.ogf.org/

classification of the image content) or by modality (*e.g.* to register images of different patients in a same spatial frame). Figure 3 illustrates the different groups that can be defined by the user and the matching operator behavior.

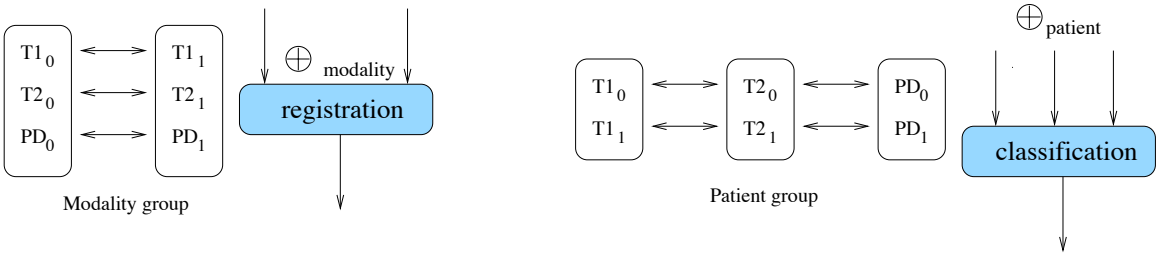


Figure 3: Matching operator.

The match operator has an impact to the input data sets file format in addition to the workflow language itself: the input data needs to be tagged.

4.3.2 Iteration strategy combined with lists

Iteration strategies are fully compatible with lists. In fact iteration strategies only have a meaning when dealing with lists of data items: the all-to-all semantic corresponds to a cartesian product between two lists of items while the one-to-one semantic refers to a matching of items from two different lists (using an order criterion or a group belonging criterion for matching). Although declared as properties of processors in Scufl, both iteration strategies and list manipulations correspond to data flow pre-processing applied prior to processor invocation. They could appear as specific data flow *adaptors* as illustrated in figure 4.

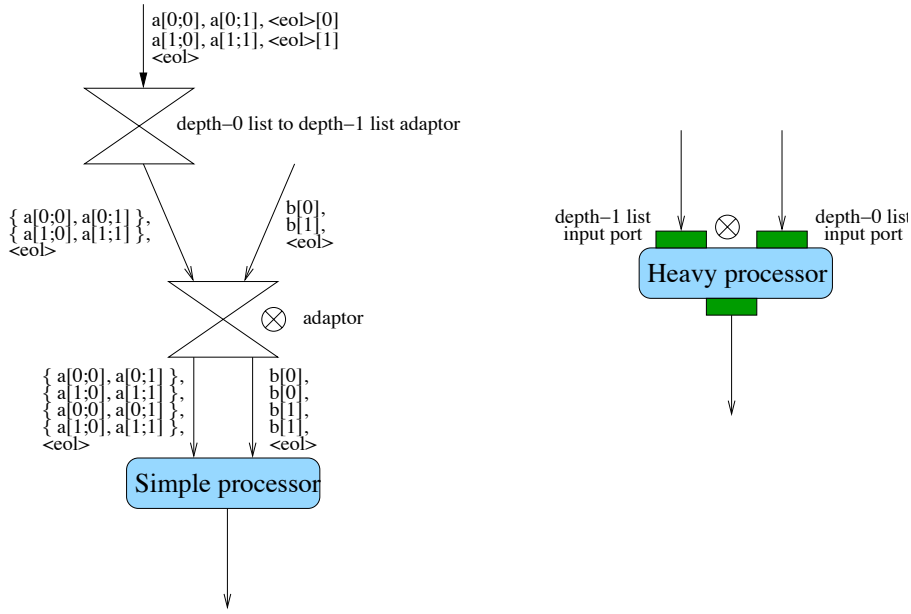


Figure 4: Explicit adaptors (left) versus heavy processor (right) view of data flows.

In the GWENDIA workflow language we prefer to declare lists adaptors and iteration strategies together with the processor to declare its invocation semantics internally. However, a logically equivalent expanded workflow view such as the one show in left of figure 4 can be proposed to end users for debugging purposes.

4.4 Control structures

Scufl does not define control structures but a special case of conditional (`fail-if-false/true` processors). In Taverna 2 design, control structures are not really integrated. There is only a recommendation to embed data flows into a control workflow using sub-workflows if control structures are needed, similarly to what is done in Kepler. This approach is sound but it has the severe drawback to imply data synchronization in all sub-workflows embedded, hence breaking data flow pipelines and loosing data parallelism. Some authors advocate separating data and control flow for the sake of simplicity indeed [3, 1], although they do not explicit the "complexity problem" arising when embedding control structures within the data flow.

In the GWENDIA language we prefer control structures completely integrated within the data flow to achieve maximal performance. Both loops and conditionals may be seamlessly integrated in a data flow.

Conditionals. Conditionals can be implemented in a data flow as data filters. Processes need to be able to return an empty output in some cases. This empty output will be interpreted by the workflow engine which will not trigger invocation of subsequent processor(s). A processor may evaluate a condition and send data items on its outputs depending on the test value. Such a conditional may either correspond to an *if-then-else* semantic (a processor with 2 outputs, and data items sent either on one or the other of the outputs) or a *switch-case* semantic (a processor with multiple outputs dispatching each input item to one of its outputs) or even a filter that transfers on its output(s) only a subset of the data items received (as exemplified in figure 5, right). The condition can be interpreted either at the business code level or as a local beanshell script. Business-level conditionals are not known from the workflow manager and they are handled as any processor. Their only particularity is to potentially return an empty result. Beanshell are convenient for evaluating conditions as they provide a mean to map workflow values into local variables and to perform any computation with these values. Beanshell conditionals are evaluated at the workflow engine level and they can be identified as specific processors. A specific support will be provided in the GWENDIA language.

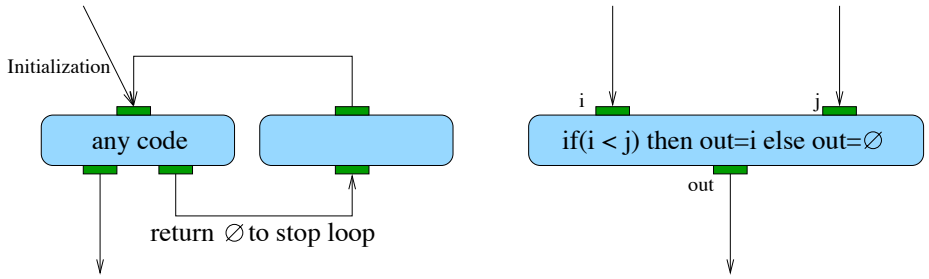


Figure 5: Control structures embedded in a data flow: loop (left) and conditionals (right).

Loops. As mentioned before, loops of processors may exist in a data flow through circular data links connectivity. Setting a loop require that two data links can be connected to the same input port of a processor: one link connected to an initialization source, the other link looping the data items (see left of figure 5). To stop the loop iterations, it is also needed that a processor can send no data item on one of its output ports as described in the case of conditionals. The loop condition is evaluated internally to one of the processors in the loop. It can be completely specific to the business domain and not be visible at the workflow level. Alternatively, one can use the beanshell processors to evaluate the loop condition, similarly to what is proposed in conditionals. Such a loop is identical to a conditional with a data link loop (see left of figure 6 for an example). No additional language construct is needed to express it. A special case it the `for` kind of loop for which the number of iterations is known in advance and the loop counter can be incremented in a beanshell processor. A specific support will be provided to handle `for` loops: the `for` processor will maintain a counter for each input data item traversing the loop. It will have to differentiate input data items and looping data items that are receive on a same port, as illustrated in right of figure 6. This can be done by analyzing the history of data items computations.

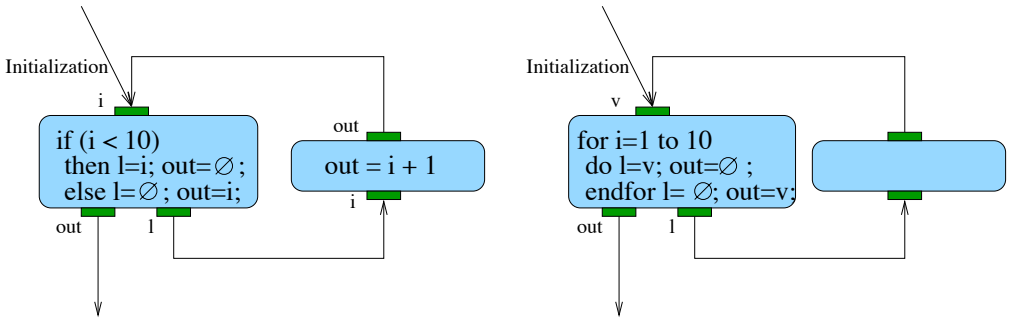


Figure 6: Implementing loops: `while` loops are implemented as conditionals (left) while `for` loops require specific language support (right).

4.4.1 Implementation of control structures in a data flow

The implementation of the loop and conditional control structures in a data flow are straight forward. The main extension needed is to authorize a processor to return an empty result upon invocation. Given that different kind of processors return different forms of output (*e.g.* a web service always returns a value, be it an empty string or an empty list), a special code has to be decided to be interpreted has the empty output.

4.4.2 Implementation of control structures in a DAG

Loops are naturally unfolded in DAGs. This prevents the use of loops for which the number of iterations is unknown in advance.

Filters cannot be implemented in DAGs. Both unbounded loops and filters require dynamic evaluation.

4.5 Optimization add-ons

The number of data items that a processor takes as input or produces as output can be left unspecified in the workflow language as this can be managed during execution of the workflow. However this limits the possibility to instantiate before execution all the tasks to be executed. As a consequence it limits the ability to schedule these tasks in an efficient way, particularly if the execution environment is shared with other workflow submitters. To overcome these limitations, we propose some additionnal XML attributes for the input and output elements of a processor node in order to specify the cardinality of these inputs/outputs.

If we take as an example a node that generates as output a list of data items (that could be a list of files, or a list of integers), with the hypothesis that all instances of this node will produce lists with the same number (5) of data items, then this number can be provided in the workflow language within the description of the output port, as the `card` attribute.

Syntax for this example:

```
<processor name="node_A">
    <in name="in"/>
    <out name="out" type="LIST(FILE)" card="5"/>
</processor>
```

In case the output has a higher depth of lists imbrication, the value of the `card` attribute contains a semicolumn-separated list of integers, one for each level of the structure. This restricts the usage to structures containing the same number of sub-items in each item of a given level. The example below shows the syntax for a node with an output that is a list of 5 sub-lists, each sub-list containing 3 files:

```
<processor name="node_B">
    <in name="in"/>
    <out name="out" type="LIST(LIST(FILE)" card="5;3"/>
</processor>
```

The number of values (separated by semi-column) for the `card` attribute must match the depth of the structure specified as value of the type attribute. In case only the number of items at a given level of the structure is known but not at all levels, then a special character '`x`' can be used to replace the integer value. It informs the parser that the cardinality at this level is not known before execution. Below is an example of such a node that has an output that is a list of 5 sub-lists, each sub-list containing a variable number of items known only at execution time:

```
<processor name="node_C">
    <in name="in"/>
    <out name="out" type="LIST(LIST(FILE)" card="5;x"/>
</processor>
```

5 Language formalization

5.1 Gwendia language

The GWENDIA language is represented in XML, using the syntax defined in this section and implementing the semantics described in the rest of this document.

Types. Values flowing through the workflow are typed. Basic types are `integer`, `double`, `string` and `file`. Files are string identifiers that are not interpreted by the workflow manager. All types may be embedded in any-depth `list` (*e.g.* `list(integer)`, `list(list(string))`...).

Processors. A `processor` is a data production unit. A regular processor invokes a service through a known interface. Defined processor types are `webservice`, `diet` and `beanshell`. Special processors are workflow `input` (a processor with no inbound connectivity, delivering a list of externally defined data values), `sink` (a processor with no outbound connectivity, receiving some workflow output) and `constant` (a processor delivering a single, constant value). To improve readability, the `input`, `sink` and `constant` processors are grouped in an `<interface>` tag within the document. Other example of processors are grouped in a `<processors>` tag. Web services define a `<wsdl>` tag pointing to their WSDL description and the operation to invoke. Beanshells define a `<script>` tag containing the java code to interpret. `DIET` services define a `<service>` tag describing the path to service to invoke.

Processor ports. Processor input and output ports are named and declared. A part may be an input (`<in>` tag), an output (`out` tag) or both and input/output value (`inout` tag). For each input/output, the type is specified. The input/output type also defines the port depth list. Iteration strategies are defined through a scufl-like operators tree expression. The tree nodes are one of the 3 basic data manipulation operators (`dot`, `cross` or `match` operator). The match operator defines the match tag. Few processor examples are given below:

```
<workflow>

  <interface>

    <constant name="parameter" type="integer">
      <value>50</value>
    </constant>

    <source name="reals" type="double" />

    <sink name="results" type="file" />

  </interface>


  </processors>

    <processor name="docking" type="webservice">
      <wsdl url="http://localhost/docking.wsdl" operation="dock" />
      <in name="param" type="integer" />
      <in name="input" type="file" />
      <out name="result" type="double" />
      <iterationstrategy>
        <cross>
          <port name="param" />
          <port name="input" />
        </cross>
      </iterationstrategy>
    </processor>

    <processor name="statisticaltest" type="diet">
      <service path="weightedaverage" />
      <in name="weights" type="double" />
      <in name="values" type="list(integer)" />
```

```
      <in name="coefficient" type="double" />
      <out name="result" type="file"/>
      <iterationstrategy>
        <cross>
          <port name="coefficient" />
          <match tag="patient">
            <port name="values" />
            <port name="weights" />
          </match>
        </cross>
      </iterationstrategy>
    </processor>

</processors>
```

Data links. A data link is a simple connection between a processor output port and a processor input port as exampled below:

```
<links>
    <link from="reals" to="statisticaltest:coefficient" />
    <link from="docking:result" to="statisticaltest:weights" />
    <link from="statisticaltest:result" to="results" />
</links>
```

Conditionals and `while` loops. Condition and `while` loops tests are implemented as special beanshell processors (see figure 5):

```
  <condition>
    <in name="i" type="integer" />
    <in name="j" type="integer" />
    <out name="out" type="integer" />
    <if>i < j</if>
    <then>out=i;</then>
    <else>out=VOID;</else>
  </condition>
```

`For` loops. These loops are similar in their syntax (see figure 6):

```
  <for>
    <in name="v" type="double" />
    <out name="l" type="double" />
    <out name="out" type="double" />
    <from>1</from>
    <to>10</to>
    <step>1</step>
    <do>l=v; out=VOID;</do>
    <endfor>l=VOID; out=v</endfor>
  </for>
```

5.2 DAG instantiation

The instantiation consists in converting **processors** from the Gwendia functional workflow language into **tasks** in the MA-DAG language. For each processor node in the workflow several tasks can be generated, one for each set of input data available to the node. Other types of nodes (eg conditions or loops) can trigger the generation of a whole set of tasks.

The complexity of a DAG instantiation depends firstly on the structure of the functional workflow (presence of loops or conditions) and secondly on the cardinality of inputs and outputs of the processor nodes.

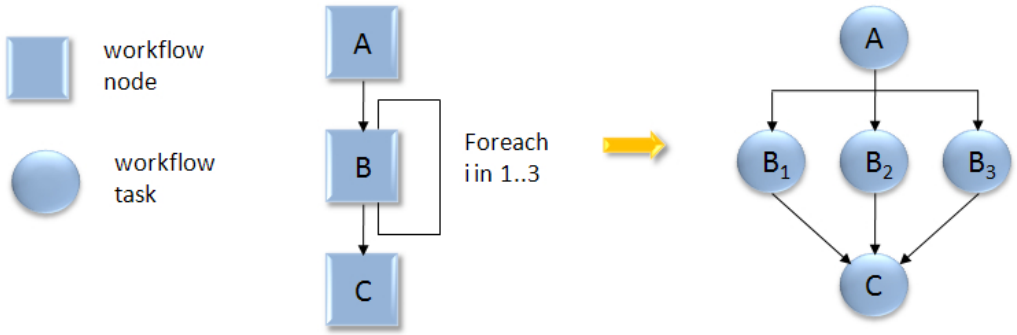


Figure 7: Loops

- Loops : to be able to instantiate a loop, the list of values for the loop parameter must be known so if this list depends on the execution of other nodes then the loop instantiation will be done after execution of these nodes. Then the nodes within the loop will simply be instantiated as a graph of tasks for each value of the loop parameter (see figure 7). Some optimization may be used at this stage to avoid generating too many tasks simultaneously in case the loop parameter has a large range of values.

- Conditions: as for loops, the condition variables must be known before the condition can be instantiated, so if one of the variables depends on the execution of other nodes then the condition instantiation will be done after execution of these nodes. Then the condition can be evaluated and the DAG instantiation can continue or not depending on the result.

- Cardinality of inputs and outputs: The cardinality of an input is the number of data items provided for this input in one instance of the node (i.e. one task), provided that these items are grouped in a list. This number can be static or dynamic. The same applies to the outputs of the node. It is important to note that when a list of several data items is produced by a task, it can either be processed as a whole by child tasks, or it can be splitted in several elements that are processed separately. The functional workflow language specifies the correct behaviour by the values of the `type` attribute for the input and output.

To generate the tasks for a given node, the functional workflow language parser will check for each input port of the node wether the data source is external or is an output port of another node of the workflow. If the source is external, a link to a data file (XML) should be provided in the workflow and the parser will create an iterator on the collection of items specified in this file. If the source is an output port of another node, the parser will create an iterator on the collection of items produced by all the tasks that are generated for that node.

When the node has several inputs, the operator applied on this set of inputs will determine how the data items are combined to produce input data sets. The operators can be “dot”, “cross”, “match” (see section 4.3). The operator will be applied to the different input iterators to produce a global input iterator for the current node. For each iteration of this iterator i.e. one data set a new task will be created in the DAG and the outputs of this task will serve as source for the iterators of the following nodes. The

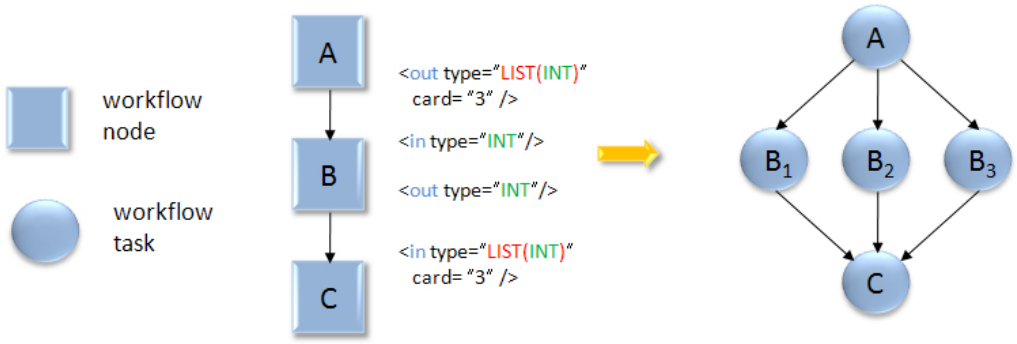


Figure 8: Iteration due to different list depths

number of tasks to be generated therefore depends on the number of items in the collection. This number can be known before execution by first looking at the number of data items in the external source file and by providing the cardinality of task outputs in the workflow language (see section 4.5).

In case the output (resp. input) is a list of data items, the iterator will take into account the depth of the list structure (which is statically defined in the wf language) and adapt the level of iteration to the depth specified for the input (resp. output) port (see figure 8). There are three possibilities:

1. output depth > input depth: for example if node A's output depth is 2 (*e.g.* output type is 'LIST(LIST(INT))' for integer items) and node B's input depth is 1, then one A task output will be connected to several B tasks input. Each B input will also be marked with a label that corresponds to the A output, as this may be used later in the workflow to group the outputs of tasks together (see below).

2. output depth = input depth: one output item will correspond to one input item

3. output depth < input depth: for example if node A's output depth is 0 (eg output is an integer) and node B's input depth is 1, the iterator will take items produced by several A tasks (A1, A2, etc...) and group them in a list to produce one item for a B task. This grouping will be done according to the labels attached to the A node outputs.

As DAG instantiation depends on knowing the cardinality of inputs/outputs, this process cannot be entirely done before the execution unless all required cardinalities are statically specified (idem if the loop conditions depend on task outputs). Therefore one functional workflow may be instantiated as several distinct DAGs with dependency constraints, which cannot be all generated at the same time. The instantiation is therefore a dynamic process that runs in parallel with the DAGs execution. A first DAG is generated using at least the entry nodes of the workflow (the nodes that do not depend on other nodes outputs). Then the generated outputs of this first DAG will trigger the instantiation of other nodes or of nodes that have been partially instantiated in the previous DAG. This process ends when all nodes of the workflow have been fully instantiated. Therefore the MA-DAG workflow language requires some other extensions to describe these DAG relationships. A DAG element will have a new attribute `id` that identifies the DAG, precedence relationships between DAGs will be described using new elements

`<prec id="[dag_id]">` within the `<dag>` element, and the `source` attribute of task inputs will eventually contain a reference to another DAG output, for example an input element of a task A in DAG X could have this form: `<IN name="taskA_in" type="DIET_FILE" source="dagY#taskC#out">`.

6 Conclusions

This document discussed the different approaches to implement data-intensive scientific data flows. In particular, it outlined the interest and differences between (functional) data flows and DAGs. A new workflow language, enriching several existing one among which Scufl and MA-DAG is then propose in the context of the GWENDIA project. This language fully enables the GWENDIA use cases and we believe will adapt to a very large spectrum of other use cases.

References

[1] Shawn Bowers, Bertram Ludäscher, Anne H.H. Ngu, and Terence Critchlow. Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow. In *IEEE Workshop on Workflow and Data Flow for Scientific Applications (SciFlow)*, Atlanta, USA, April 2006.

[2] Yolanda Gil. *Workflow Composition: Semantic Representation for Flexible Automation*, chapter 16, pages 244–257. In [8], 2007.

[3] Antoon Goderis, Christopher Brooks, Ikay Altintas, Edward A. Lee, and Carole Goble. Composing Different Models of Computation in Kepler and Ptolemy II. In *International Workshop on Workflow systems in e-Science (WSES'07)*, Beijing, China, May 2007.

[4] Montagnat Johan. Application workflows descriptions. Technical report, GWENDIA project, ANR-06-MDCA-009, I3S, Sophia-Antipolis, France, May 2007.

[5] Bertram Ludäscher and Ikay Altintas. On Providing Declarative Design and Programming Constructs for Scientific Worklows based on Process Networks. Technical Report SciDAC-SPA-TN-2003-01, San Diego Supercomputer Center, San Diego, USA, August 2003.

[6] Johan Montagnat, Tristan Glatard, and Diane Lingrand. Data composition patterns in service-based workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS'06)*, Paris, France, June 2006.

[7] Tom Oinn, Peter Li, Douglas B Kell, Carole Goble, Antoon Gooderis, Mark Greenwood, Duncan Hull, Robert Stevens, Daniele Turi, and Jun Zhao. *Taverna/myGrid: Aligning a Workflow System with the Life Sciences Community*, chapter 19, pages 300–319. In [8], 2007.

[8] Ian Taylor, Ewa Deelman, Dennis Gannon, and Matthew Shields. *Workflows for e-Science*. Springer-Verlag, 2007.

[9] Glatard Tristan, Montagnat Johan, Bolze Raphaël, and Desprez Frdric. L1.1: Bibliography on workflow representation languages. Technical report, GWENDIA project, ANR-06-MDCA-009, I3S, Sophia-Antipolis, France, June 2007.

[10] Daniele Turi, Paolo Missier, Carole Goble, David de Roure, and Tom Oinn. Taverna Workflows: Syntax and Semantics. In *IEEE International Conference on e-Science and Grid Computing (eScience'07)*, pages 441–448, Bangalore, India, December 2007.

[11] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for grid computing. *ACM SIGMOD records (SIGMOD)*, 34(3):44–49, September 2005.