A data-driven workflow language for grids based on array programming principles

Johan Montagnat
CNRS / Univ. of Nice
I3S laboratory
johan@i3s.unice.fr

Benjamin Isnard
INRIA
ENS Lyon, LIP
benjamin.isnard@ens-lyon.fr

Tristan Glatard
CNRS / INSERM / INSA
CREATIS laboratory
glatard@creatis.insa-lyon.fr

Ketan Maheshwari
University of Nice
CNRS, I3S laboratory
ketan@polytech.unice.fr

Mireille Blay Fornarino
University of Nice
CNRS, I3S laboratory
blay@polytech.unice.fr

ABSTRACT

Different scientific workflow languages have been developed to help programmers in designing complex data analysis procedures. However, little effort has been invested in comparing and finding a common root for existing approaches. This work is motivated by the search for a scientific workflow language which coherently integrates different aspects of distributed computing. The language proposed is data-driven for easing the expression of parallel flows. It leverages array programming principles to ease data-intensive applications design. It provides a rich set of control structures and iteration strategies while avoiding unnecessary programming constructs. It allows programmers to express a wide set of applications in a compact framework.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Data-flow languages*

General Terms

Languages, Design

1. ON WORKFLOW LANGUAGES AND GRID WORKFLOW ENACTORS

Workflows have become very popular in the eScience community to design, describe and enact parallel and data intensive scientific processes. Apart from a minority of applications for which a specific, and often expensive, parallelization work has been invested, many scientific data analysis procedures are exploiting generic workflow description tools and execution engines which lower development cost, deal with distribution-specific issues and enable the scientists to focus on their area of expertise rather than the gridification of the application.

Theoretically, workflow design tools and workflow enactors are independent components that have to be considered in the life cycle of a scientific application exploitation. The former relates to a language enabling the expression of the application logic while the latter relates to the distributed implementation and optimization of this logic, taking into account the grid interface and capability. In practice though, both aspects are often tightly coupled as there tends to be a specific language for each grid workflow enactor designed. This observation illustrates the fact that there is a wide variety of needs for different scientific applications. The definition of a language suitable for a category of applications is a trade-off between different aspects including the ease of use, the expertise of the workflow design community, the specificity of the workflows to be represented, performance, etc. This paper specifically focusses on the definition of a workflow language covering a wide variety of applications, keeping in mind that the practice often leads to the definition of workflow- and grid-specific enactors.

The confusion between workflow languages and enactors is such that the language is not always well defined nor documented. The language might be hidden behind the workflow execution engine, or the workflow designer when a high level graphical interface is used for visual programming. This lack of language specification makes it difficult for end-users to determine beforehand the workflow solution adapted to their specific needs. As a result, practical considerations (which middleware is supported, etc) are often prevailing over the expressiveness of the underlying language.

This statement does not apply to the BPEL language [23] which is a widely adopted specification as there exist many different implementations of BPEL enactors. However, the BPEL language was designed to describe business orchestrations of distributed web services. It targets distribution but not high throughput nor data intensive computing and it received only limited attention in the grid community [2]. Yu and Buyya [27] proposed a detailed taxonomy, studying many existing workflow management systems for grid computing. This taxonomy is based to a large extent on the capabilities of the workflow enactors and provides only limited insight to the workflow languages expressiveness. A more recent work by Goderis and co-authors [8] studies a topic more closely related to the workflow language expressiveness: it

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WORKS '09, November 15, 2009, Portland Oregon, USA.
Copyright 2009 ACM 978-1-60558-717-2/09/11 ...$10.00.

compares the models of computations of various workflow directors implemented in Kepler [17] with the perspective of enabling composition of different models in sub-workflows of a global orchestration. This works illustrates how different enactors designed for different application needs have different expressiveness power.

Many different languages have been considered within the grid community, from raw Directed Acyclic Graphs (DAGs) of computational processes (DIET MA-DAG [3], CONDOR DAGMan [15]) to abstractions for parallel computations such as Petri nets [1], meta-models [22], data-driven languages such as Scufl [25] and scripting languages such as SwiftScript [28]. Each of these approaches can be defended through some aspects well covered in their design: DAGs are convenient for scheduling [18, 9], Petri nets can be used to detect properties such as potential deadlocks, data-driven languages ease the description of applications logic for non-expert users and scripting is extensively used by programmers for prototyping, etc. However, among the plethora of grid workflow engines developed today, there has been relatively little effort invested in studying the underlying language expressiveness. Some exceptions should be mentioned though, such as the Scufl semantics definition [25] or the search for Turing completeness [6].

In this paper, we propose a workflow language dedicated to scientific applications design on a grid infrastructure. The language targets ease of use and high expressiveness as it is mostly intended to be manipulated by scientists who are not experts of distributed computing and it addresses a variety of applications. The languages targets the coherent integration of (1) a data-driven approach; (2) arrays manipulation; (3) control structures; and (4) maximum asynchronous execution. On the way, we analyze different existing approaches to motivate the language introduced and compare it to existing material. This specification is the result of a joint consortium working on applications from several scientific domains (medical image analysis, drug discovery, astronomy, etc) and different workflow enactors, in the context of the GWENDIA project [13]. Making a complete analysis of existing workflow languages is out of reach of this paper. More details on this topic can be found in chapter 2 of [5]. We defend the idea that data-driven languages are particularly appealing for designing scientific data analysis applications and execute them on a distributed system such as a grid. Therefore, our language is data-driven as motivated in section 2. It borrows from early principles and late extensions of array programming as detailed in section 3. The language and it operationalization are discussed in section 4.

2. GRID DATA-DRIVEN WORKFLOWS

2.1 Coarse-grain, implicit parallelism representation languages

Workflows are representing an application logic and in that sense, any programming language such as traditional C, java or scripting languages could be adapted to represent workflows. However, workflows act at a different scale than software composition systems: they deal with human-scale processes that are scheduled over time [4]. Detaching from traditional programming language is important to obtain "a workflow in which each step is explicit, no longer buried in Java or C code. Since the workflow is described in a unified manner, it is much easier to comprehend, providing the opportunity to verify or modify an experiment" [2].

Similarly, in the field of distributed applications, especially targeting grid infrastructures, workflows deal with coarse rather than fine-grain parallelism which is better described with traditional parallel programming approaches such as MPI or OpenMP. The ability to represent an orchestration of coarse grain software processes, without the additional complexity of a tightly coupled message passing interface, is therefore desirable. Scientific workflows are a typical examples of procedure manipulating heavy computation codes where lower level programming languages are used to tackle the fine-grain complexity of the data analysis. Scientific workflow languages are providing an extra level of data analysis procedure representation by describing the coarse-grain interaction of independent codes. In this context, the added-value of the workflow languages relies mostly on its ability to federate non-instrumented codes in a single procedure, to validate the integrity of the workflow and to provide higher level capabilities that where not necessarily available in the native code languages such as parallelism and data flow descriptions. In the case of compute intensive scientific workflows considered in this paper, the expression of parallelism is particularly important. By exploiting workflows, many users, non-experts in distributed computing, expect to benefit from a parallel implementation without explicitly writing parallel code. Any workflow graph of dependencies intrinsically represents some degree of parallelism [7]. In many scientific application though, the data parallelism is massive and is the primary source of performance gain expectation. Especially on large scale distributed systems such as grids where communications are costly, data parallelism is a coarse grain parallelism that can be efficiently exploited. As we will see in this section, data parallelism is represented differently depending on the approach adopted.

Platform-independence of the workflow definition is also an important aspect of workflow programming [19], even crucial in the grid computing area where applications are typically composed from heterogeneous codes and software, each of them having its own architecture or system requirements. It is also motivated by the emergence of component-based programming models that promote code reusability and platform-independence. While traditional scripts, that are often considered as the ancestors of workflows, are designed for execution in an homogeneous environment, workflows provide a representation of the logic of the application independently from the implementation. Built on top of service-oriented architectures, workflows foster code reusability, thus reducing applications development time. As a consequence, workflows are increasingly cited as a transparent way to deploy applications on grids and a large amount of applications rely on them for a successful gridification.

2.2 Most scientific workflow languages are data driven

As orchestrations of coarse-grain processes, workflows are well represented through graphs which nodes represent data analysis processes and arcs represent inter-dependencies. Inter-dependencies may be data dependencies (data exchange needed between subsequent processes) or pure control dependencies (the dependency only expresses a synchronization of processes execution in time). In practice though, there is a data transfer involved in a majority of cases encountered. There exist some exceptions (*e.g.* real time

simulation engines) but many scientific applications are described as data analysis pipelines: successive processes are inter-dependent through data elements, usually exchanged through files, that are produced and consumed during the analysis. It is especially true when dealing with independent codes without message passing interface.

Indeed, among the many existing scientific workflow languages, focus is often put on the data although it does not always appear explicitly. The Scufl language is a good example of a data-driven language. All inter-dependencies are data links, except for a specific control link used for the cases where there is no data transfer known from the workflow engine. It is the availability of data which drives the execution and the data flow plays a crucial role as will be illustrated in section 3. DAGs as represented in DAGMan or MADAG conversely, are typically pure control flows. Dependencies in DAGMan for instance only represent temporal synchronization barriers in the execution of subsequent processes. Yet, each process defines a set of input and output files. In a majority of cases, subsequent processes happen to refer to the same files (the output files of one process happen to be the same as the input files of the subsequent one). In that case, the control links are only hiding a data dependency. The same DAG could be expressed by making the data transfers explicit. With the language as it stands, there is a level of redundancy in the representation and preserving its coherence is important for the workflow semantics. Such languages are usually not intended for direct human use anyway and they are produced by higher-level tools that ensure the coherence.

An interesting example is the SwiftScript language, proposed as a workflow language founded on a scripting approach programmers are familiar with. To make the language adapted to grids, the inventors of SwiftScript generalized the use of *futures* [10] for every variables defined in the script. Futures are non-blocking assignment variables, using a proxy to ensure immediate execution of assignment and performing lazy blocking on variable value accesses only. The systematic use of futures for every variables in the language makes it completely data-driven: the execution progresses asynchronously as long as a data access is not blocking. The availability of data enables blocked thread to restart execution. The beauty of futures is to make this process completely hidden from the programmer who can use a scripting language he is familiar with without ever manipulating explicit parallel programming constructs.

Another interesting example is AGWL, the Askalon language [26], which defines a set of control structures with specific control ports distinguished from the data ports used for data exchanges. Despite the clear separation between data and control ports and links, a fixed pattern is defined for each language structure which imposes the coherence between the data exchanges and the associated control: control links cannot be defined independently from data links except for a specific pure control dependency structure.

Functional languages have also been proposed for expressing grid workflows given that they are data-centric programming languages well adapted to represent data-driven workflows [16]. Yet, there is no real grid workflow language based on functional principles as far as we know.

2.3 Data driven languages express parallelism implicitly

Data driven languages separate the definition of data to process from the graph of activities to be applied to the data. This separation of the scientific data to analyze and the processing logic is convenient in many experimental situations: an application is designed and implemented independently of the data sets to consider, and the same workflow can be reused for analyzing different data sets without any change. The data flow is produced at execution time by considering the data to be analyzed in a particular run and the set of inter-dependent activities the data is sent to. Depending on the complexity of the workflow and the expressiveness of the underlying language, this data flow computation may be a static operation that can be performed once when starting execution, or it may require a dynamic implementation.

Beyond the convenience of separating data and application logic, data driven language are particularly appealing to the grid community as they make it possible to express parallelism without any explicit construct. Indeed, the application graph expresses (independent) codes that may be enacted in parallel and data parallelism is expressed through the multi-entries input data set pushed in the workflow. In addition, pipelining can be implemented to optimize workflow execution performance on a distributed platform [7]. The futures adopted in SwiftScript language similarly make the language implicitly parallel.

The separation of data and applications logic has profound implications for the programmer and the execution engine. This makes a clear difference between a language such as BPEL, where variables are explicitly assigned (data to process is declared within the language) and explicit control structures are needed for exploiting parallelism (*foreach* kind of language constructs), and languages such as Scufl or SwiftScript where no additional parallel language structure is needed. (Although SwiftScript defines an explicit *foreach* structure to handle data parallelism, this could be avoided as will be shown in this paper). To declare data to be processed, Scufl manipulates data lists which items are mapped to activities input ports at run time. Similarly, although adopting a more traditional scripting approach SwiftScript defines mapping functions to match the workflow input (script variables) with their actual value read from files or other data sources.

2.4 Data and control flows

To wrap up this discussion, figure 1 shows a simple application workflow expressed with three different families of languages from left to right: pure data-driven language (*e.g.* Scufl), explicit variables assignments and parallel constructs (*e.g.* BPEL), and pure control flow (*e.g.* DAGMan). Red arrows represent data dependencies while blue connectors represent control dependencies between subsequent activities.

Independently from the language considered, the execution engine will interpret the workflow with its inputs data to produce the *data flow*: a directed acyclic graph of data segments produced with provenance linking. The data flow generated may be an extremely large DAG even for simple application workflows. Scufl for instance defines *iteration strategies* that enable the declaration of complex data flows in a compact framework. Note that some expressive language will accept application graphs with cycles while the data flow is always a DAG (where possible loops have been unfolded by the execution process).

As shown in this example, control structures defined in a

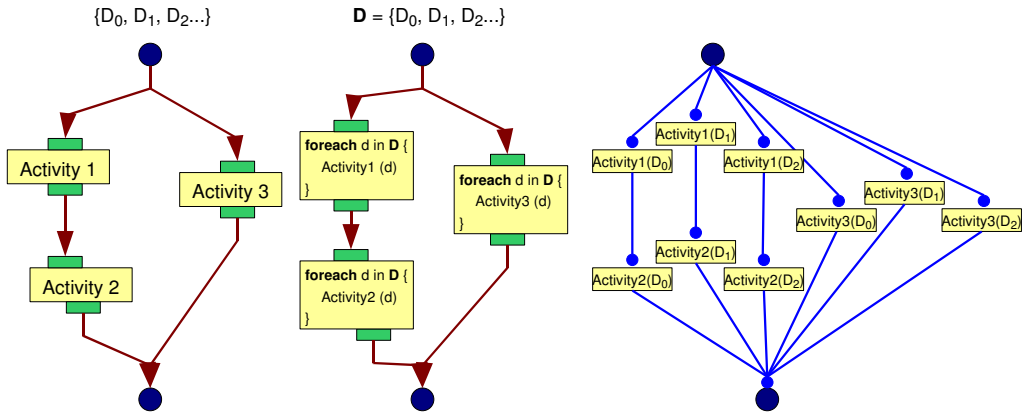


Figure 1: Data-driven language (left), explicit variables assignment and parallel constructs (center) and pure control language (right).

language may be needed or not, for declaring a given data flow, depending on the model of computation adopted. In existing languages, there appears to be a separation between control structures-less data-driven languages such as Scufl and imperative languages making extensive use of control structures such as BPEL. The boundary is not as clear between the languages though. The Scufl language introduced as special form of conditional (*fail-if-true / fail-if-false*) as it is useful to manipulate control structures in data-driven languages. Conversely, SwiftScript adopted principles of data-driven execution through futures. We claim that both approaches can be conciliated to produce an expressive language that proposes to the programmer both control structures of interest to express any application and a data-driven engine to make the implementation of parallel processes as transparent as possible. This proposal is founded on the principles of array programming introduced in section 3.

3. ARRAY PROGRAMMING PRINCIPLES

Array programming was introduced in the early sixties to ease the description of mathematical processes manipulating arrays [11]. It was initially thought as a simplified way for manipulating array data structures in the language and many implementations are sequential. However, it was also considered as a mean to take advantage of mainframes vectorial processors and exploit data parallelism. The principle is simple: arrays are considered as first-class entities within the language and traditional arithmetic operators (such as addition, etc) are defined natively to operate on arrays or combination of scalar values and arrays (*e.g.* if X and Y denote arrays of numerical values, $X + Y$ and $2 \times X$ are valid expressions). Array operations are seen as a convenience to avoid writing explicit loops for simple repetitive operations. They reduce the need for control structures use inside the language.

Array programming principle is not limited to arithmetic operations and can be generalized to any case of function application. In [21], Mougin and Ducasse propose an extension of the array programming concepts to object-oriented languages. They introduce the application of methods (referred to as message passing) on arrays of objects and/or the application of methods to arrays of parameters. An explicit syntax is used to denote expansion over an array: $X@+@Y$ and $2*@X$ are the explicit denotations for $X+Y$ and $2\times X$ respectively. Other array manipulation operators are also defined: *reduction* (*e.g.* scalar sum of all array members); *compression* as a form of test over all array components ($X\text{at}(X > 10)$ returns the array of components greater than 10); *sorting* (for numeric arrays); *joins* (searching for indices of some elements in an array); and *transposition*.

3.1 Arrays as first-class entities

In array programming, arrays are defined as indexed collections of data items with homogeneous type. An array of objects defines a new data type and therefore, arrays may be nested at any depth. To a data item is therefore always associated a type, and a possibly multi-dimensional integer index (one dimension per nesting level). For instance, $\mathbf{a} = \{\{\text{“foo”}, \text{“bar”}\}, \{\text{“foobar”}\}\}$ is a 2-nested levels array of strings and $\mathbf{a}_{0,1}$ designates the string “bar”.

SwiftScript defines first-class entities arrays which are compatible with this definition. The Scufl language refers to *lists* of data items that are in fact corresponding to the same concept: Scufl's list items are indexed and typed [25]. Lists as defined in functional languages can match this definition if they are limited to objects of same type. We prefer to refer to the term *array* as it is more usually referring to an ordered collection of values than *lists*.

3.2 Arrays and functional processes

As an operator, or any function, may be defined to be applied either to scalars or arrays in array programming languages, the data-driven language define computing activities independently from the data objects sent to these activities. An activity will fire one or more times depending on the exact input data set it receives. Consider the example given on the left of figure 1: activity 1 will fire 3 times as it receives an array with 3 scalar values during the workflow execution. Iterations over the array element is handled implicitly by the execution engine.

Similarly, functional language put the emphasis on the definition of functions, independently of the data items that are processed. It has been observed that the functional *map*

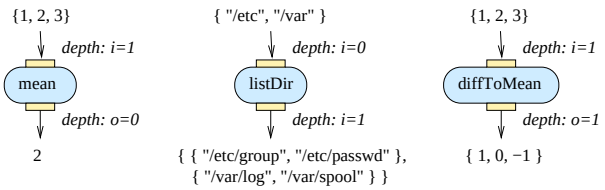


Figure 2: Impact of ports depth. Left: $i = 1, o = 0$. **Center:** $i = 0, o = 1$. **Right:** $i = 1, o = 1$.

operator defined as:

$$\begin{array}{lccccc} map: & (\alpha \to \beta) & \to & [\alpha] & \to & [\beta] \\ & f & & [x_1 \dots x_n] & \mapsto & [f(x_1) \dots f(x_n)] \end{array}$$

can be used to transform a scalar function into an array function [16]. If $f : \alpha \to \beta$ is a function operating over scalar values then $F : [\alpha] \to [\beta]$ such that $F = map(f)$ is the corresponding function operating over arrays with 1 nesting level. It should be noted that in this case, the mapping operator has to be applied explicitly though.

3.3 Activity ports depth

Activities may receive inputs with different nesting levels. The usual behaviour of an activity receiving a nested array is to fire once for each scalar value embedded in the nested structure. However, there are cases where the semantics of the activity is to process a complete array as a single item rather than each scalar value individually. For example, an arithmetic mean computation activity will consider all numerical values received in an input array at once and compute their mean. Such an activity reduces the depth of data items received: an array of values is reduced to a single scalar. Conversely, a number generator could take as input a scalar seed value and produce as output a list of random number, thus increasing the depth of the data processed.

To generalize this notion of input/output data structure depths transformation, we can consider that activities have input/output ports with defined *depth*. The ports are buffers for the values received and produced. The depth of a port determines the number of nesting levels the input port will collect or the output port will produce. It impacts the number of firings of the activity considered. Let us denote with n the nesting level of an array received by an activity input port of depth $i \leq n$. The activity will fire once for every nested structure of depth i received and therefore produce an output array with nesting level $n - i$ (see an example in figure 2, left). Similarly, if an output port has depth o, a nested array with o nesting levels is produced for each invocation and consequently, the global structure produced has $n - i + o$ nesting levels (see figure 2 center and right). The default behaviour is obtained for activities with depth 0 input and output ports: the processed structures nesting levels are unchanged.

An important property of activities invocation in an asynchronous execution is that multiple invocations of an activity on array items preserve the array indexing scheme: the j^{th} data item in the output port corresponds to the processing of the j^{th} data item in the input port, independently from the actual execution order of the activity multiple firings. An input port depth higher than 0 corresponds to an implicit synchronization barrier: in an asynchronous execution context, the port buffers input data and will fire the activity only when sufficient data items have been collected.

3.4 Iteration strategies

An important concept introduced in the Scufl language is the one of *iteration strategies*. An iteration strategy becomes useful when an activity has 2 or more input ports, to define how the data flow received on each port is combined and triggers activity invocations. There are two basic iteration strategies defined in Scufl, known as *cross product* and *dot product*. When a cross product is defined between a pair our input ports, all data values received on one port are combined with all values received on the second one. If arrays of size n and m are received on the ports, the activity fires $n \times m$ times. The dot product causes the activity to fire only once for each pair of data items received on its input ports. The data items are matched with their index rank: the activity will fire $\min(n, m)$ times and the output with rank $k \leq m, n$ corresponds to the processing of the pair of inputs with the same rank k in each input port.

Iteration strategies can be combined in a complete iteration expression tree when more input ports are involved, thus producing complex iteration patterns without requiring the programmer to write any explicit loop. They are expressive operators of a data-driven language. They can be extended with additional iteration patterns as illustrated in section 4.1.4.

It is to be noted that elaborated array programming language such as [21] provide a syntax to specify iteration strategies on multiple depths nested arrays. The "@" operator may be followed by an index rank and used to define more complex iteration strategies. For instance, $X@1 \times @2Y$ is equivalent to a cross product (it produces an array of arrays containing the produces of all components of X multiplied by all components of Y instead of a simple array of all $X_i \times Y_i$ values).

4. GWENDIA LANGUAGE

Beyond the analysis of some existing scientific workflow languages and the exhibition of some similarities between the approaches proposed, this paper addresses the definition of a workflow language, named GWENDIA, targeting the coherent integration of (1) a data-driven approach; (2) arrays manipulation; (3) control structures; and (4) maximum asynchronous execution capabilities. As such, it borrows ideas from data-driven languages (*e.g.* Scufl), array programming and imperative languages (*e.g.* SwiftScript). The objective is to obtain a compromise with maximum expressiveness (all workflows that can be expressed in the above mentioned languages should be expressible in the GWENDIA language), ease of use (the expression should be as compact as possible) and enable efficient execution. The major challenge is to obtain a coherent language as it bridges concepts from different languages.

4.1 Language definition

4.1.1 Data structures

The data manipulated in the language is composed from scalar typed data items (typically, the basic types `integer`, `double`, `string` and `file` are considered). Data structures, defined as fixed-size heterogeneous collections of data items can be defined.

Data with homogeneous types may be grouped in arrays. An array is an ordered collection of data items with the same type. A simple array is a collection of scalars (*e.g.* $\mathbf{a} = \{2, -3, 1\}$ is an array of integers). A one-dimension index designates each of its data item ($\mathbf{a}_0$ designates the integer 2). An array may be empty. An array may contain other arrays at any nesting level. An array of arrays is further referenced as a *2-nesting levels* array, etc. Note that a scalar s and a singleton $\{s\}$ are different data entities, with different types. A scalar data item corresponds to a *0-nesting level* array.

The special value Ø (*void*) represents the absence of data. It will be useful for special operations and insuring the consistency of the language as described below.

4.1.2 Workflows

A workflow is described as a graph of activities interconnected through dependency links. Each activity corresponds to either one construct of the language or the execution of an external application code. Activities are fired as soon as input data becomes available for processing. Activities may be fired an arbitrary number of times, or never fired at all, depending on the data flowing in the workflow. An important property of activities invocation in an asynchronous execution is that multiple invocations of an activity on array items preserve the array indexing scheme: the indices of produced data matches the indices of processed data as explained in section 3.3. The indexing scheme has no influence on the order of activity firings though.

The interface to activities is defined through typed input and output ports with known depth as defined in section 3.3. The output port types define the activity type. Upon firing, an activity may either execute successfully, thus producing an output of known type, or encounter an error and throw an exception. An exception causes the workflow engine to be notified of the error (user reporting) and produce the special value Ø as the result of the execution. An activity may produce Ø as a result of its processing, meaning that no value was considered (*e.g.* a data filter might forward data items that passed the test and return Ø for the others).

An activity which receives Ø as input does not fire and just pass the void value on to the subsequent activity(ies). This execution semantics guarantees that the process continues execution as much as possible, processing other data items that did not raise exception conditions.

Workflow inputs are special activities, with no input port and a single output port, that fire without pre-requesite when the workflow is started. Valid workflow inputs are (1) data *sources* receiving user defined data, (2) *constants* containing a single value, or (3) user defined activities with no input ports that will fire only once, when the workflow is started. Workflow outputs are special activities, with no output port and a single input port, that performs no processing and collect results received from other activities in the workflow.

4.1.3 Dependencies

A data link interconnects one activity output port with one activity input port. The port types have to match. The data link defines a data dependency between two activities. An input port can only receive a single data link to preserve the data indexing scheme during processing. Different outputs may be connected to different activities, causing the produced data items to be replicated to all subsequent activities connected.

In case there is no data dependency explicitly defined between two activities but an order of execution should be preserved, a control link interconnecting these activities may be used. A control link emits a signals only once the source activity completed all its executions (taking into account the possible iteration strategy applied to it). The target activity will start firing only once it has received the control link signal, processing the data items buffered in its input ports or to be received, as usual. In case several control links are connected to a target activity, it only starts firing when all control signals have been received. In an asynchronous execution environment, a control link thus introduces a complete data synchronization barrier.

4.1.4 Iteration strategies

Iteration strategies define how input data items received on several input ports of a same activity are combined together for processing. They specify how many times an activity fires and what is its exact input data sequence for each invocation. Iteration strategies are also responsible for defining an indexing scheme that describes how items from multiple input nested arrays are sorted in an output nested array.

dot product.

The dot product matches data items with exactly the same index in an arbitrary number of input ports. The activity fires once for each common index, and produces an output indexed with the same index. The nesting level of input data items, as received and transformed after port depth considerations (see section 3.3), in all ports of a dot product should be identical. The number of items in all input arrays should be the same. The ports of a dot product are associative and commutative. A Ø value received on a dot product port matches with the data item(s) with the same index(ices) received on the other port(s) and produces a Ø output without firing the activity.

cross product.

The cross product matches all possible data items combinations in an arbitrary number of input ports. The activity fires once for each possible combination, and produces an output indexed such that all indices of all inputs are concatenated into a multi-dimensionnal array (data items $\mathbf{a}_i$ and $\mathbf{b}_j$ received on two input ports produce a data item $\mathbf{c}_{i,j}$), thus increasing the input data nesting depth. The ports of a cross product are associative but not commutative. A Ø value received on a cross product port matches with all possible combinations of data items received in other ports and produces a Ø output without firing the activity.

flat cross product.

The flat cross-product matches inputs identically to a regular cross product. The difference is in the indexing scheme of the data items produced: it is computed as a unique index value by flattening the nested-array structure of regular cross produces ($\mathbf{a}_i$ and $\mathbf{b}_j$ received on two input ports produce a data item $\mathbf{c}_k$ with index $k = i \times m + j$ where m is the size of array $\mathbf{b}$), thus preserving the input data nesting depths. As a consequence, the flat cross product may be partially synchronous. As long as the input array dimension are not known, some indices cannot be computed. Similarly

as the cross product, the ports of a flat cross product are associative but not commutative. A Ø value received on a flat cross product port behaves as in the case of a regular cross product.

match product.

The match product matches data items carrying one or more identical user-defined tags, independently of their indexing scheme (see [20] for a complete definition and motivation of the match product). Similarly to a cross product, the output of a match is indexed in a multiple nesting levels array item which index is the concatenation of the input indices. A match product implicitly defines a boolean valued function match($\mathbf{u}_i, \mathbf{v}_j$) which evaluates to true when tags assigned to $\mathbf{u}_i$ and $\mathbf{v}_j$ match (*i.e.* the specified tags values are equal). The output array has a value at index i, j if match($\mathbf{u}_i, \mathbf{v}_j$) is true. It is completed with Ø values: if match($\mathbf{u}_i, \mathbf{v}_j$) is false then $\mathbf{w}_{ij} = \emptyset$. The ports of a match product are thus associative but not commutative. A Ø value received on a match product input does not match any other data item and does not cause activity firing.

4.1.5 Control structures

The data-driven and graph-based approached adopted in the GWENDIA language makes parallelism expression straight forward for the end users. Data parallelism is completely hidden through the use of arrays. Advanced data composition operators are available through activity port depth definitions and iteration strategies. Complex data parallelisation patterns and data synchronization can therefore be expressed without additional control structures. `foreach` kind of structures that are usually used for explicit data parallelization are not needed. Code parallelism is implicit in the description of the workflow graph. `fork` and `join` kind of structures are not needed either.

The only control structures considered for the GWENDIA language are therefore conditionals and loops. The semantics of contional and loops operating over array types need to be precisely defined. To our knowledge, existing array-based languages do not define such a semantic and the programmer needs to define the conditional and loop expressions on scalar values (consequently using `foreach` kind of structures to iterate on the content of arrays, refer to SwiftScript for an example). Special activities are defined to express conditionals and loops. These activities have a constrained format and input/output ports for enforcing the semantics defined in this document.

Conditional and loop expression computations are expressed using the java language and interpreted each time the activity is fired. The data received on the input ports of a control structure is mapped to java variables (basic types or java ArrayLists depending on the input port depths).

Conditionals.

A conditional activity represents an array-compliant *if then else* kind of structure. A conditional has:

1. an arbitrary number of input ports (possibly operating iteration strategies), each of them mapped to a variable of the test expression;
2. a test expression to evaluate for each data combination received from the input ports; and
3. an arbitrary number of special paired output ports. Each pair corresponds to a single output with the first pair element linking to the *then* branch and the second pair element linking to the *else* branch.

The test expression is evaluated each time the conditional activity fires. A user-defined result is assigned to the *then*, and optionally to the *else*, output port for each data sequence evaluated. A void result (Ø) is assigned to the opposite port automatically. Consequently, the *then* and the *else* output ports receive a nested array with the same structure and size, as defined by the input nesting levels, ports depths and iteration strategies used, but complementary in the sense that if a value is available in one of the ouput, the corresponding item is empty in the other output and vice versa. The indexing scheme used is coherent with the usual indices computed by iteration strategies. The void results are therefore indexed coherently. In case the *else* assignment is omitted by the user, a Ø output value is produced on the *else* outputs each time the condition is invoked. A Ø value received on the input ports cause the conditional activity to produce two Ø values in all its *then* and *else* outputs without evaluating the conditional.

Figure 3 shows examples of conditional activity and the result of the enactment over multiple-nesting level arrays. The left side example is a simple conditional without definition of the *else* condition. The output list contains one empty item for the value that did not pass the condition in the *then* branch and the *else* branch only receives Ø values. The center example is a complete conditional with both *then* and *else* branches. The two output arrays are complementary. This example also shows the use of two inputs. The 1 nesting level input arrays are transformed in 2 nesting levels output arrays by the iteration strategy applied between the inputs. The right side example is a complex example with the mixed use of multiple port depth values, iteration strategy and multiple output ports.

With partial (and complementary) arrays produced by conditionals, two additional list manipulation activities become useful as exemplified in figure 4.

The *filter* activity is a single input / single output ports activity that filters a nested array structure such that all empty items are removed from the array. This activity is useful to discard all results that have not passed the condition, if the indexing of resulting items does not need to be preserved. As a consequence, the items in the structure will be re-indexed. It is to be noted that this activity introduces a partial synchronization barrier: an item in an array cannot be re-indexed until all preceding items have been computed and determined as empty or not. The filtering operation can create unbalanced lists in terms of size.

The *merge* activity is a two input ports / one output port activity that merges the content of two complementary lists with the same structure into a single list. It can be used to merge the lists resulting from the *then* and the *else* branch of the conditional for instance. If the list structures differ or the lists are not complementary (an item at a given index is non empty in both lists) the merge activity raises an exception.

Loops.

A loop represents an array-compliant *while* kind of structure. A loop is composed by:

1. An expression used as stop condition.

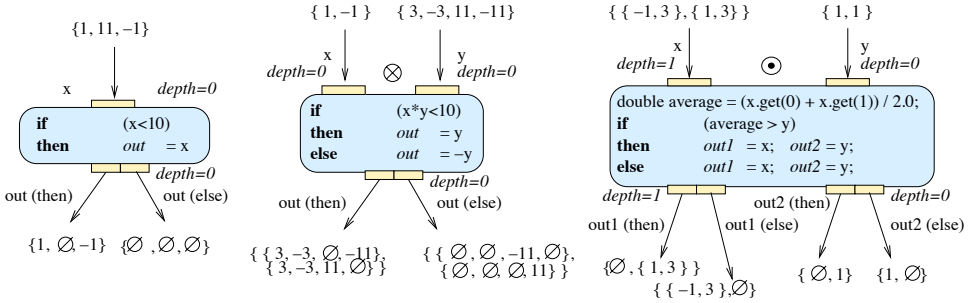


Figure 3: Three conditional examples. ⊙ denotes the dot iteration strategy and ⊗ denots the cross.

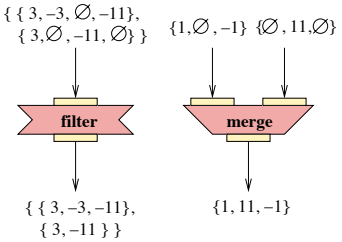


Figure 4: Filtering and merging lists with empty items.

2. One or more input ports. Loop input ports have a particular dual structure: they are composed of an outer part, receiving the loop initialization value from the outer part of the workflow, and an inner part, receiving the values that loop back to the activity after one or more iteration of the loop.

3. One output port bound to each input port. It receives the values sent to the corresponding input port and also has a dual structure: the outer part only receives a value when the loop condition becomes false (hence the loop stops iterating) while the inner part receives iteratively all values received either on the initialization (outer) or the looping (inner) part of the corresponding input port.

The inner input port of a loop can only receive a link that is connecting from an activity which one of its ancestors is connected to the inner output port (*i.e.* a loop has to exist). In addition, a loop activity has a specific indexing scheme on its inner port which increases the nesting level of input arrays by one: for each initialization value causing the activity to fire, a sub-array is created that will hold all the values generated by this initialization while the loop iterates. A Ø value received on the input ports cause a Ø value to be produced on the corresponding outer port without evaluation of the condition.

Figure 5 illustrates a simple loop and the data flowing through each port. This loop receives an array with two values (1 and 2) as initialization. As the condition passes for the first value, it is transferred to the inner part of the output port, causing a 2 nesting levels array to be created. The second initialization value also passes the condition and is transferred to a second 2 nesting levels sub-array. The first value will cause 2 iterations of the loop before the stop condition is met while the second value will only cause 1 iteration. As a consequence, the inner array as two sub-arrays with different lengths. The outer part of the output port only receives the stop condition values. The array transferred on the output port has the same nesting level as the input since both input and output ports have depth 0.

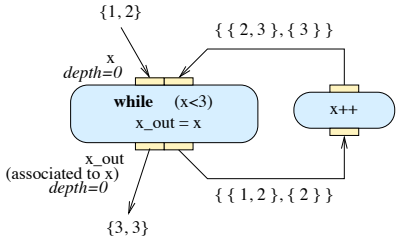


Figure 5: Simple loop example.

A *for* kind of control structure has exactly the same structure as the loop structure, except that the starting value, end value and step are fixed (hence the number of iterations is fixed). In the *for* case, the number of iterations is the same for all initialization values. Consequently, the inner sub-arrays will be of equal length.

Complete example.

Figure 6 illustrates a complete example including a loop, a conditional and a merge activity. The input value is the array $\{-1, 2\}$ received on the x port of the **while** activity. The output is the $\{-3, 3\}$ array produced on port x_{out}. The arrays defined on the workflow links show all data items and their indexed position as they have been transferred at the end of the workflow execution.

4.2 Implementation

As outlined in the introduction, defining a workflow language and building the engine supporting that language are different problems. However, the engine capability (performance, robustness, interfaces, etc) if of outmost importance for the end user. The workflow language has some impact on the engine(s) supporting as it (1) gives an upper bound of what kind of workflow is expressible and (2) may impact the performance that the engine can achieved depending on the flexibility provided by the language. This section discusses practical considerations when coming to an implementation of the GWENDIA language.

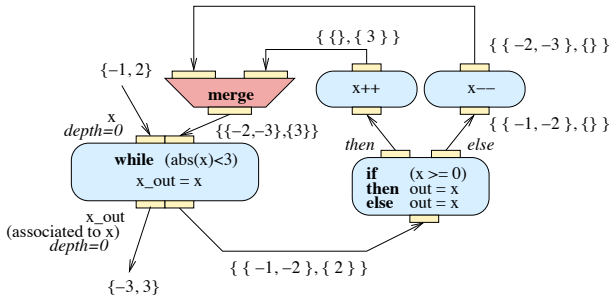


Figure 6: Complete example with loop, conditional, and array merging activities.

The definition of rich data flows including nested arrays and iteration strategies, requires the definition of a strict data indexing scheme and operational rules relating to each control structure and iteration strategy for indexing the produced data. All control structures and iteration strategies do not have the same properties regarding the implementation though. Some are *foreseeable* in the sense that given the workflow inputs, the data flow can be inferred statically prior to execution. Other are *unforeseeable* in the sense that it is only at execution time that the data flow can be dynamically computed. Pre-computing the complete data flow allows to transform the workflow in a DAG prior to its execution, enabling various optimizations such as DAG scheduling and DAG transformations for optimization purposes. Unforeseeable language constructs bring additional expressiveness to the language but prevent DAG generation.

Among the language constructs introduced above, the dot product, cross product, flat-cross product and for loops are foreseeable. Thanks to the knowledge of the fixed number of iterations, a for loop can be unfolded to compute the data flow DAG. There is a significant difference between the cross product and the flat-cross product iteration strategies though. While the cross product can be implemented completely asynchronously (when a new input data item is received on one of the ports, the resulting firing of the activity can be triggered without delay), the flat-cross product may be blocking due to the need to know one array dimension prior to the results indices computation (see section 4.1.4).

There are many constructs in the Gwendia language that lead to an unforeseeable behavior. A simple one is an activity that produces an output array of unknown size. The resulting data flow cannot be inferred unless the activity can guarantee a fixed size for the arrays it will produce. An optional array size attribute can be defined in Gwendia workflows in order to make this construct foreseeable whenever possible. However, the conditionals can only be resolved at execution time and loops cannot be statically unfolded as the number of iteration will depend on the execution results. The match iteration strategy is also completely dependent on user-defined tags. Workflows depending on these constructs are definitely unforeseeable (including the possibility for non terminaison due to an infinite loop).

Two workflow engines are currently being implemented to support the Gwendia language. Both are operational at the time of writing this paper but only cover a part of the language. The MOTEUR workflow engine is a dynamic interpretor of the data flow [14]. Initially designed to support the Scufl language and enable complete asynchronous execution, the engine is being extended. MOTEUR continues to support Scufl workflows as the Gwendia language covers it. The MA DAG [12] engine is a DAG generator engine built on top of the DIET middleware. MA DAG benefits from all the workflow scheduling mechanisms included in the middleware. Due to the impossibility to produce complete DAGs, MA DAG generate partial DAGs greedily: a sub-DAG, as complete as possible, is produced as soon as possible. Incomplete nodes are produced on the unforseeable constructs until the execution engine reaches these points and they can be resolved. More sub-DAGs are then produced until completion of the execution.

5. CONCLUSIONS

Workflow languages are important for the end user as they define the kind of workflows that can be expressed. There are many approaches adopted in the grid community today but more effort has been spent in the workflow engines than in the underlying languages. The Gwendia language targets the coherent integration of:

- a data-driven approach to achieve transparent parallelism;
- arrays manipulation to enable data parallel application in an expressive and compact framework;
- conditional and loop control structures to improve expressiveness; and
- asynchronous execution to optimize execution on a distributed infrastructure.

Workflows are enacted independently of the language, through two different engines adopting completely different approaches.

6. ACKNOWLEDGMENTS

This work is supported by the French ANR Gwendia project under contract number ANR-06-MDCA-009.

7. REFERENCES

[1] M. Alt and A. Hoheisel. A grid work. In *6th international conference on Parallel processing and applied mathematics (PPAM'05)*, page 715?722, Poznan, Poland, Sept. 2005.

[2] R. Barga and D. Gannon. *Scientific versus Business Workflows*, chapter 2, pages 9–16. In [24], 2007.

[3] E. Caron and F. Desprez. Diet: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[4] D. Gannon. *Component Architectures and Services: From Application Construction to Scientific Workflows*, chapter 12, pages 174–189. In [24], 2007.

[5] T. Glatard. *Description, deployment and optimization of medical image analysis workflows on production grids*. PhD thesis, Université de Nice Sophia-Antipolis, Sophia-Antipolis, Nov. 2007.

[6] T. Glatard and J. Montagnat. Implementation of Turing machines with the Scufl data-flow language. In *3rd International Workshop on Workflow Systems in e-Science (WSES'08)*, Lyon, France, May 2008.

[7] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec. Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR. *International Journal of High Performance Computing Applications (IJHPCA) IF=1.109*, 22(3):347–360, Aug. 2008.

[8] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, and C. Goble. Heterogeneous composition of models of computation. *Future Generation Computer Systems*, 25(5):552–560, May 2009.

[9] R. Hall, A. L. Rosenberg, and A. Venkataramani. A Comparison of Dag-Scheduling Strategies for Internet-Based Computing. In *International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–9, Long Beach, CA, USA, Mar. 2007. IEEE Computer Society.

[10] R. Halsted. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Apr. 1985.

[11] H. Hellerman. Experimental personalized array translator system. *Communications of the ACM (CACM)*, 7(7):433–438, 1964.

[12] http://graal.ens-lyon.fr/ diet/workflow.html. DIET MA-DAG workflow manager.

[13] http://gwendia.polytech.unice.fr. GWENDIA (Grid Workflow ENactment for Data Intensive Applications) contract ANR-06-MDCA-009.

[14] http://modalis.polytech.unice.fr/softwares/moteur/start. MOTEUR workflow manager.

[15] http://www.cs.wisc.edu/condor/dagman. CONDOR DAGMan DAG meta-scheduler.

[16] B. Ludäscher and I. 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, Aug. 2003.

[17] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience*, 18(10):1039 – 1065, Aug. 2005.

[18] G. Malewicz, I. Foster, A. L. Rosenberg, and M. Wilde. A tool for prioritizing DAGMan jobs and its evaluation. In *Proceedings of the 15th International Symposium on High Performance Distributed Computing (HPDC'06)*, pages 156–167, Paris, France, June 2006.

[19] A. Mayer, S. McGough, N. Furmento, W. Lee, M. Gulamali, S. Newhouse, and J. Darlington. Workflow Expression: Comparison of Spatial and Temporal Approaches. In *Workflow in Grid Systems Workshop, GGF-10*, Berlin, Mar. 2004.

[20] J. Montagnat, T. Glatard, and D. Lingrand. Data composition patterns in service-based workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS'06)*, Paris, France, June 2006.

[21] P. Mougin and S. Ducasse. Oopal: integrating array programming in object-oriented programming. In *18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA'03)*, pages 65–77, New-York, USA, 2003. ACM Press.

[22] C. Nemo, T. Glatard, M. Blay-Fornarino, and J. Montagnat. Merging overlapping orchestrations: an application to the Bronze Standard medical application. In *International Conference on Services Computing (SCC 2007)*, Salt Lake City, Utah, USA, July 2007. IEEE Computer Engineering.

[23] C. OASIS. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, 2007.

[24] I. Taylor, E. Deelman, D. Gannon, and M. Shields. *Workflows for e-Science*. Springer-Verlag, 2007.

[25] D. Turi, P. Missier, C. Goble, D. de Roure, and T. Oinn. Taverna Workflows: Syntax and Semantics. In *IEEE International Conference on e-Science and Grid Computing (eScience'07)*, pages 441–448, Bangalore, India, Dec. 2007.

[26] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the ASKALON grid environment. *ACM SIGMOD records (SIGMOD)*, 34(3):56–62, Sept. 2005.

[27] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing (JGC)*, 3(3-4):171 – 200, Sept. 2005.

[28] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In *IEEE International Workshop on Scientific Workflows*, Salt-Lake City, USA, July 2007.