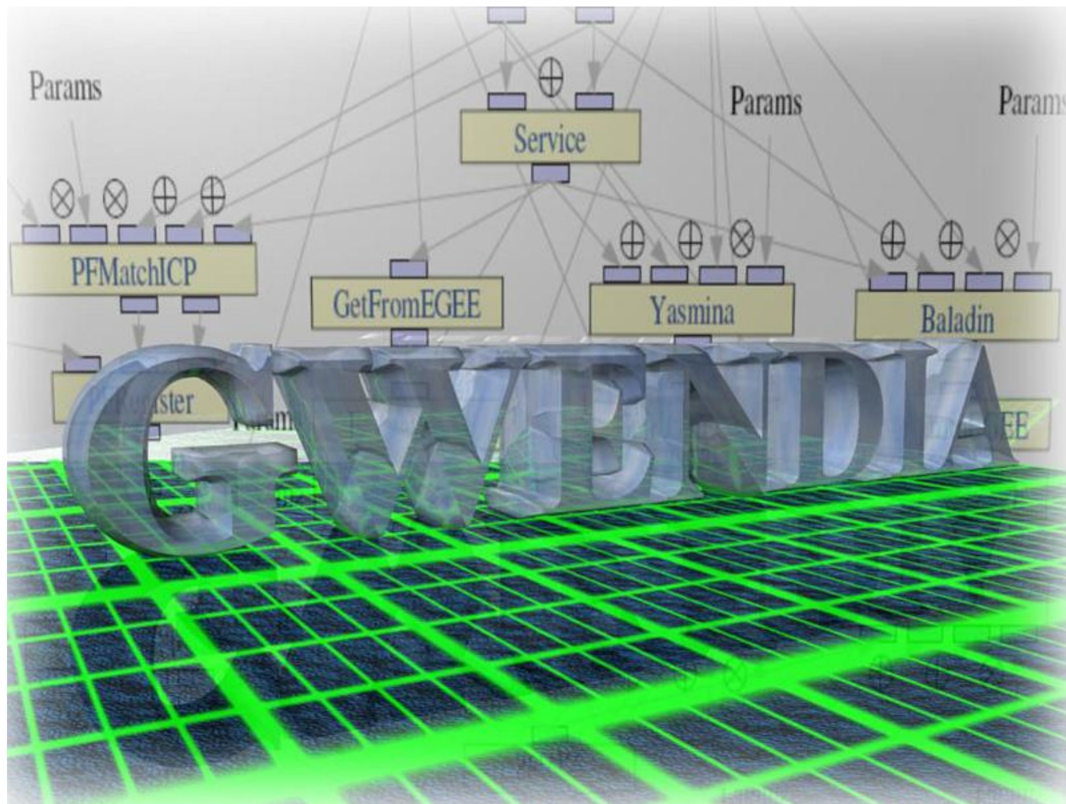
Gwendia ANR-06-MDCA-009
Grid Workflow Efficient Enactment for Data Intensive Applications

Financé par ANR

L1.1: Bibliography on workflow representation languages



Tristan Glatard	RAINBOW (I3S), ASCLEPIOS (INRIA Sophia)	glatard@i3s.unice.fr
Johan Montagnat	RAINBOW (I3S)	johan@i3s.unice.fr
Raphaël Bolze	GRAAL (LIP)	raphael.bolze@ens-lyon.fr
Frédéric Desprez	GRAAL (LIP)	frederic.desprez@ens-lyon.fr

Abstract

We propose a classification of workflow languages emphasizing on their expressiveness. Expressiveness is not a property that can easily be quantified. We explore different criteria for evaluating expressiveness, putting the focus on the expectations of the applications considered in the Gwendia project.

1 Introduction

The representation and enactment of workflows has been an area of investigation for a very long time. Originally developed for formally describing enterprise processes, they have been adopted in a broad context and in particular for the representation of scientific data analysis procedures. One of the main advantages of workflows is the simplicity and flexibility that they provide to the users. Although any business or scientific procedure could be represented through very expressive programming languages, workflow are preferred in many context due to their ability to:

- provide abstract representations simplifying the expression of complex procedures in a user-friendly manner;
- provide flexibility and dynamicity in the composition of business processes involved in a given procedure;
- provide a transparent code parallelization support;
- provide enactors that interface to different computing back-ends transparently.

Similarly to scripting languages that aim at easing prototyping by providing more flexibility than compiled programming languages (relaxed data types and declarations, on-the-fly interpretation, etc), workflow languages provide easy business processes composition in an environment that is high level (usually graphical) and flexible (often platform independent). As a consequence, from a user point of view the expected properties of workflow languages do not necessarily focus on very complex representations but also on simplicity, compactness and flexibility of the workflow.

1.1 Definitions

The workflow management coalition[1] proposes the following definition of workflow management:

> Workflow management is the automation of business procedures or "workflows" during which documents, information or tasks are passed from one participant to another in a way that is governed by rules and procedures.

This broad definition reflects the diversity of application domains where workflows are used. Indeed, before being studied for the description of distributed applications, workflows have been used for describing the organization of production processes in companies as well as the interaction between several business entities.

In figure 1 we depict a typical graphical representation of a workflow. The participants (yellow boxes) can be denoted as *tasks*, *services*, *processes*, *transitions*, *activities*, *functions* or *components* depending on the workflow approach. The enactment of a participant can be called *invocation* (mostly for services), *execution* (for tasks), *firing* (for transitions and activities) or simply *call* (for a function or component). Workflow is a particular type of software composition system, where the activities of the workflow are the components to be assembled. Activities may be interconnected by control links (blue doted link) and possibly data links. Control links are simple temporal dependencies between two activities: in figure 1, the third activity cannot fire before the second ended. In some cases, data

[1] http://www.wfmc.org

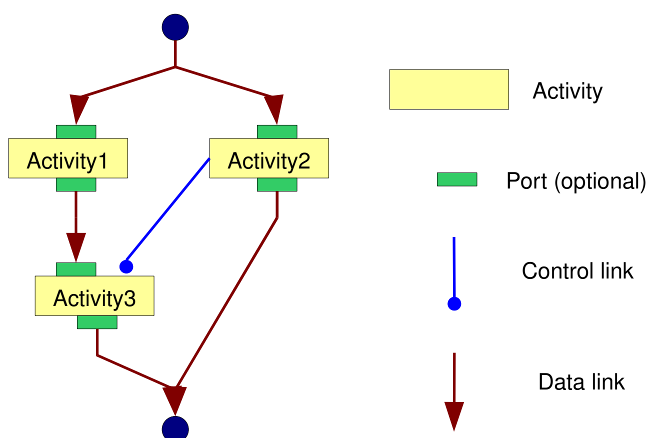


Figure 1: Workflow components and links

links (red arrows) express data exchanges between successive activities. They also imply time dependencies between the firing of two different components but in addition they involve an exchange of data between one activity input and one activity output. In cases of asynchronous workflow enactors, additional ports are required to buffer the data to be processed by different activities: data links may be connected to input and output ports. The graph of nodes and data links represent the *data flow* in a workflow. The graph of nodes and every links represent the *control flow* (often data flow links are also used as control flows as they imply dependencies).

In [12], Gannon notices that workflows act at a different scale than software composition systems: they deal with human-scale processes that are scheduled over time. Similarly, in the field of distributed applications, workflows deal with coarse rather than fine grain parallelism which is better described with traditional parallel programming approaches such as MPI or OpenMP. Gannon also underlines that workflows refer to a centralized execution in which a single engine is responsible for the control of the process. In a grid context, this property of workflows has pros and cons. On the one hand, it is true that a centralized perspective is necessary to have a coherent control of the execution of the application in order to be able to provide a representation of the status of the application to the user. Yet, on the other hand, centralization may lead to dramatic performance limitations, in particular when dealing with data-intensive applications that involve large numbers of activities.

In [22], the authors propose a definition highlighting the platform-independence of the workflow definition:

> We consider a workflow to be the organization of a structured application in an abstract fashion, such that the implementation of the atomic tasks being organized is independent from the organization itself.

This aspect of workflow programming is 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 tightly coupled to the platform and architecture, workflows provide a representation of the logic of the application independently from the implementation. It is particularly important for grid applications, where the heterogeneity of the resources and middlewares is critical. 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. Works related to service composition propose alternate definitions where the workflow itself is viewed as a service [42, 41].

Workflows may also be defined by the use of a simple graphical language for end-users, thus easing code understanding and application development. Workflows offer a unified and simple view of complex experiments that may gather heterogeneous codes from various developers and institutes. Barga and Gannon indeed noticed in [5] that:

> The result is 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.

1.2 Workflow representation languages

In this deliverable and in the context of the GWENDIA project we focus on the representation of scientific workflows with the perspective of their execution on distributed computing infrastructures. Scientific codes are a typical examples of 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 document, 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 rewriting parallel code. Any workflow graph of dependencies intrinsically represent some degree of parallelism. 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 section 2, data parallelism is represented very differently depending on the approach adopted.

In the literature there is also often a confusion between workflow languages and workflow enactors. Although both should be independent, except for a few standards a new workflow language and the corresponding enactor are developed all together and there is no alternative to enact a workflow written in a specific language. Most workflow languages are therefore high level representations and the work performed by the enactor to transform this abstract representation into an executable set of computational tasks is often hardly detailed. In some rare cases though, there exist different representations for the abstract and concrete executable workflows.

Many orthogonal classifications of workflow languages could be adopted depending on the interest of the workflow user. Yu and Buyya [45] propose an extensive taxonomy that is focusing on the properties of workflow enactors (load balancing, fault tolerance, etc). Gil [13] is focusing more on the abstraction level of the workflow representation (distinguishing templates that can be instantiated and executable workflows). In this deliverable we focus on the expressiveness and performance of compute intensive application. We therefore adopt a classification, outlined in figure 2, that emphasizes both on the abstraction level of the workflow language and its capability to express a broad spectrum of applications. We then consider the way workflow enactors can turn the abstract representation into a concrete workflow to execute it on a distributed grid infrastructure.

2 Classification: from models to effective execution

The following classification is based on the presence or absence of *functions*, *data* and *resources* in the workflow representation. Many other criteria could be considered but in the context of the Gwendia project, we consider them as transverse and we focus on the amount of information concerning the process execution that is provided inside the workflow description languages. The classification proposed in this section is summarized in figure 2: existing languages will be studied from completely formal models, that are used for the analysis of workflow properties, to concrete schedules of tasks graphs. If a workflow definition gathers functions, data and resources, then it is fully *executable*. On the opposite, if none of those three aspects are defined, the workflow is a *formal model*. Otherwise, the workflow representation may be a *functional workflow* if the functions are defined but neither the resources nor the data are available in the description. If resources are defined but data is not, then the workflow representation is a *service workflow*. Finally, in *tasks graphs*, functions as well as data are defined but the resources are missing.

Formal models are only used to study mathematical properties of workflows. They are not directly usable as a particular implementation. The highest level workflow engines enact functional workflows. Enactment of such workflows requires the independent definition of data sets and mapping onto grid resources. There are more specific languages defining either resources (the service workflows), either the data (the tasks graphs) or both (the concrete workflows). Workflows languages on the left hand side of the figure share the property that they do not define data sets: the users have to define data sets at run time and the workflow managers have to build data flows prior to execution of the workflows. A consequence is that often the number of tasks to execute is unknown prior to the execution. Conversely, workflow languages on the right hand side instantiate all computation tasks given that all data fragments to process are known prior to the execution. As a consequence they are less flexible from a user point of view but on the other hand they facilitate the scheduling problem.

Concrete workflow languages are usually not available to the users as such but they rather are the product of a workflow enactor processing a workflow description. All workflow engines have to produce explicitly or implicitly concrete workflows as their output.

2.1 Formal workflow models

Formal workflow models correspond to languages where no information is given about the nature of the implied activities, the amount and type of data processed and the resources used. Those models are suitable for workflow analysis because they offer an abstract

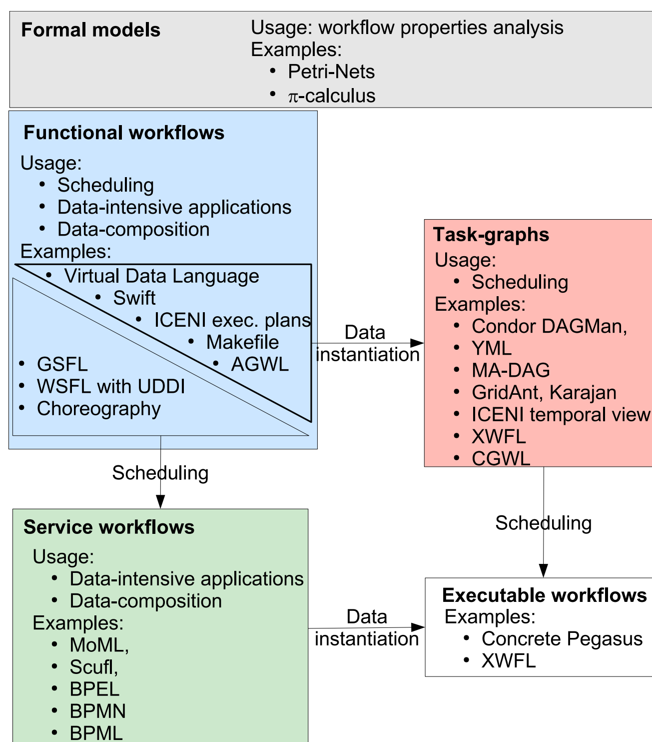


Figure 2: Classification of workflow languages

representation of the application. For instance, properties such as liveness (the absence of deadlocks) and boundedness (of the amount of generated data for instance) can be inferred from such models. Two broad classes of formal models have been proposed: *Petri nets* and *π-calculus*. There has been hot debates about the superiority of one model above the other [34, 37] and they both lead to the development of systems or standards. For instance, π-calculus is said to have inspired the development of choreographies (presented in section 2.2) whereas various workflow engines are based on Petri nets [38].

2.1.1 Petri Nets

Petri nets have been introduced in the thesis of C.A Petri, in 1962 [31]. It is a graphical modeling tool applicable to many systems and particularly suitable for parallel systems as they extend the notion of state machine with concurrency.

A Petri Net is a particular kind of directed bipartite graph, associated with a set of tokens. It is made of two kinds of nodes called *places* and *transitions*. Edges of the graph are either from a transition to a place of from a place to a transition. State machines are a subclass of Petri nets: in a state machine, each transition has exactly one input place and one output place. Tokens are located in places. Multiple edges linking the same transition to the same place or the same place to the same transition can be represented as a weighted edge whose label denotes the number of corresponding unitary edges. A transition is enabled when all of its input places contain at least the number of token of the corresponding edge label. It is then ready to *fire*. After a transition has fired, it produces for each output places the number of token of the corresponding edge label [30]. An illustration of the transition firing rule is given on figure 3. The top line of figure 4

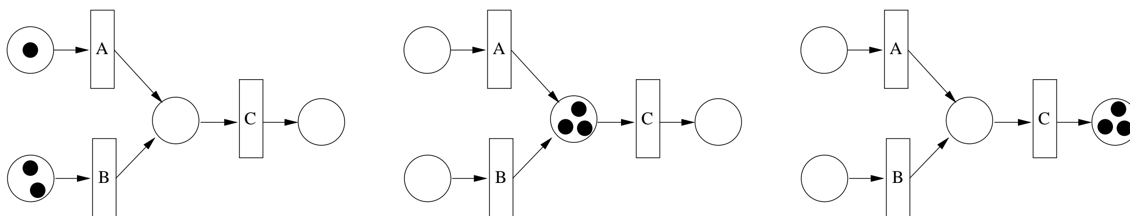


Figure 3: Evolution of the multi-merge workflow pattern implemented with Petri-Nets for a particular initial marking of p_1.

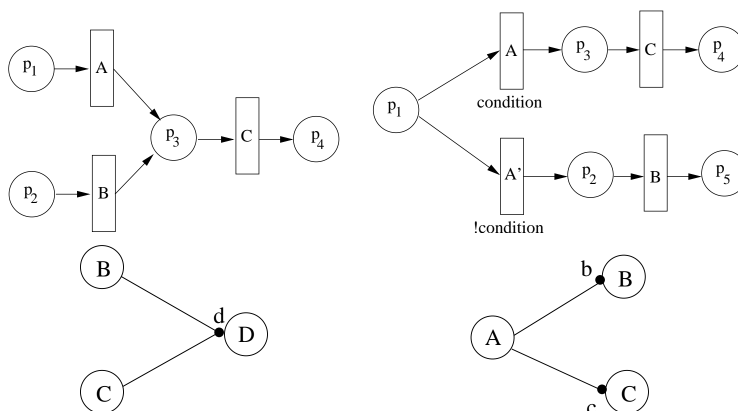


Figure 4: Implementation of workflow patterns. Left: Multi-merge ; Right: exclusive choice. Top: with Petri-Nets ; Bottom: with π-processes.

(adapted from [39]) displays an implementation of two classical workflow patterns using Petri-Nets: the multi-merge and the exclusive choice.

Several extensions of Petri Nets have been proposed and used for various applications. For instance, timed nets [21] introduce delays associated with transitions and/or places and stochastic Petri Nets associate a random variable to the time delays [2]. Inhibitor edges have also been introduced in extended Petri Nets: they disable the transition to which they are connected when their input place has a token [1]

Colored Petri Nets (CPN) were introduced to ease the manipulation of data values in Petri Nets [16]. They are particularly used for workflow modeling and are the basis of the YAWL workflow system [38]. CPN are a sub-class of High Level Petri Nets, which also include Hierarchical Petri Nets. In a colored Petri net, every token have a value. In a CPN, tokens are distinguishable: each of them is associated to a *color* which represents a data value. Places have an associated *color set* which represents the data type to which belong the colors of all their tokens. Edges are annotated with expressions that determine the exact data values removed and added by the firing of a transition.

2.1.2 π-calculus

In [34], the authors claim that some of the procedures used in business cannot be modeled using workflow engines that do not rely on π-calculus. They suggest to adopt the term *process* to denote workflows relying on the π-calculus formalism. A singular characteristic

of π-calculus is that it is able to exchange information among activities whose relationships evolve as a result. This feature is called *mobility*. Mobility is required to model processes where the exchange of information fosters the link between activities. The example of email exchanges is often cited to illustrate such a behavior: by receiving emails sent to multiple recipients, a participant becomes aware of addresses of other people, thus developing her communicating ability. Partisans of π-calculus advocate that static representation systems such as Petri nets cannot properly represent mobility [32].

Pi-calculus is an extension of process algebra aiming at handling concurrency. It has been proposed by Milner [28]. Pi-calculus is described in terms of *processes*, *channels* and *names*. Channels are used by processors to exchange messages. Both messages and processes are called names and thus cannot be distinguished. The sending of a message u over a channel x is written $\bar{x}(u)$, whereas receiving the message u over the channel x is denoted by $x(u)$. Channels themselves can also be sent and received, which make possible the description of mobile processes such as the email use-case described in the previous paragraph. The sending and receiving of a message u over any channel can be abbreviated respectively by $\bar{u}$ and u. Processes can be composed sequentially by the operator $''.''$ or in parallel, with the notation $''|''$. The choice operator $''+''$ is also available as well as the $''!''$ unary operator which is used to specify that a process can be iterated as many times as required. A condition about a particular name can be expressed by the $[x = y]$ notation. There are two particular processes: 0, which does not do anything and stops the process execution and τ, which corresponds to a hidden activity, that does not take part into the global process [43]. A τ process is an activity that corresponds to an effective activity of the workflow: for instance, it may model the computation of a service operation on some data, which is seen as a black box from the workflow point of view.

It is clear that π-calculus is able to model both control and data flows. Van der Aalst's workflow patterns [39] are expressed using π-calculus in [33]. Examples from this work are recalled here, to illustrate the π-calculus formalism. The bottom line of figure 4 presents two workflow patterns: the left of the figure presents shows the *multi-merge* whereas the right of the figure displays the *exclusive choice*. The π-calculus representation of the multi-merge is the following:

$$\begin{aligned} B &= \tau_B.\bar{d}.0 \\ C &= \tau_C.\bar{d}.0 \\ D &= !d.\tau_D.0 \end{aligned}$$

Each line of this equation models a particular process of the workflow. After their execution, processes B and C both send the same name d which is required by process D. The presence of a ! operator in front of process D indicates that it will be replicated as many times as needed. In this case, two copies of D will be done. The *exclusive choice* is represented by the following π-processes:

$$\begin{aligned} A &= \tau_A.(\bar{b}.0 + \bar{c}.0) \\ B &= b.\tau_B.0 \\ C &= c.\tau_C.0 \end{aligned}$$

For this process, the choice operator + is needed to distinguish the 3 invocation cases.

The π-calculus formalism has been extended to the case of Web-Services orchestrations in [23]. In this work, the authors add a transaction operator which is able to cope with

faults and to trigger a recovery process if the fault message is received. Based on the same `web`π_∞ extension of the π-calculus, another application to orchestrations is proposed in [20], where the authors detail a π-calculus based semantics for WS-BPEL. Based on their analysis, it is highlighted that the three different error handling mechanisms of WS-BPEL are not necessary and a novel orchestration language based on the idea of event notification as the unique error handling mechanism is proposed. In the context of choreographies, a formal model of WSCI (see section 2.2) using a process algebra approach (CCS) is proposed in [4] and applied to web service compatibility, replaceability and the automatic generation of adapters.

2.2 Functional workflows

Functional workflows are the class of workflows for which only the activities and their dependencies are defined. In this class, the size of the handled data sets is not represented and will only be known at run time. As a consequence, *it is not possible to determine the number of tasks generated by such workflows before the execution.* This property can be used to determine whether a workflow model belongs to this class or not. In particular, this class contains traditional script languages and workflow languages that have elaborated control constructs allowing to define unbounded loops, *i.e.* loops for which the number of iterations cannot be known before run time. Depending on the architecture of the application, the activities of the workflow can be described with their actual code (such as in scripting languages for instance) or only with their interface (*i.e.* in service-oriented architectures).

This workflow representation focuses on the chain of processings. In particular, this class covers pure data flow applications, also known as *pipelines*. To become executable, such workflows have to be instantiated on the data which is provided at run time. The workflow can then be iterated on the data. Defining how this iteration behaves is the role of *data composition* strategy that will be discussed in section 3.3.5. Resources are not defined either, thus making this representation suitable for scheduling the execution tasks.

This workflow class is particularly suitable for data-intensive applications. Indeed, it prevents the developer from an exhaustive description of the whole task set required by its application: she only has to describe the functional template of the application which is instantiated on the data by the workflow manager.

Virtual Data Language. The Virtual Data Language (VDL) [46] is a functional workflow language that derives from a former VDL [11]. It has control flow constructs such as `for each`, `if`, `switch` and `while`. It is based on the declaration of procedures written in a C-like syntax. Procedures can be atomic or made by other procedures. VDL does not make any assumption about the size of the input data sets. However, the underlying workflow manager (The Virtual Data System - VDS) expands the VDL definitions into a tasks graph (see section 2.4) and executes them. This is made possible by the fact that `foreach` nodes are expanded at run time thus enabling data sets to have a dynamically determined size. We guess that a similar late expansion system is used for the other control flow constructs that lead to the execution of tasks whose number is not known before run time. The data types representation is extensively described in VDL. It relies on an XML Data Set Typing and Mapping (XDTM) that allows the types of data sets and procedures to be defined abstractly in terms of XML schema. Separate mapping descriptors then define how such abstract data structures translate to physical representations. For

instance, XDTM provides mappings from file names to their absolute path in a file system. Yet, data is not instantiated inside the VDL representation. This is made at the VDS level. Both those arguments lead us to put this approach in the functional workflow class, even if it is tightly interfaced with tasks graphs: what is called a "high-level" workflow representation in Fig.17.8 of [46] is a functional workflow because data segments are not defined on this representation. The described implementation of the VDL prototype converts this workflow definition into a tasks graphs by expanding Directed Acyclic Graph (DAG) nodes (Fig. 17.9 of. [46]).

GSFL. The Grid Service Flow Language (GSFL) has been designed as an adaptation of the WSFL to grid services, which have different needs from standard Web-Services [42]. In particular, its authors underlines the fact that the workflow specification needs to be able to allow communication between the services to avoid the workflow manager to become a bottleneck centralizing the data transfers. Avoiding centralized enactment is not straightforward with web services, whereas OGSA introduced facilities for that. In particular, GSFL provides a mechanism to connect notification sources and sinks defined in the OGSA. GSFL is also able to handle OGSA registries and factories for creating grid services. A GSFL document defines services providers, the activity model, the composition model and the life-cycle model. Service providers are the list of services involved in the workflow. They can be located statically, by a hard specification of an endpoint or invoked using factories. In the latter case, resources are not defined in the workflow document, which leaves room for further scheduling. The so-called activity model identifies the particular operations of the services involved in the workflow. The composition model describes the data and control flow between the activities and the life-cycle model contains a list of precedence links describing the order in which the services execute.

ICENI / ICENI-II. ICENI's authors identify two different workflow representations: the spatial and the temporal ones [22, 26]. The user specifies the workflow in a spatial expression, which, in our terminology, corresponds to a functional representation. This user-defined workflow is also called an execution plan. At this stage, components are described in terms of meaning and behavior. ICENI then converts it to a temporal description, *i.e* a tasks graph. As underlined by the authors, problems appear when the functional description is not acyclic, as discussed in the next sections. As in GSFL, the components themselves talk to their partners, without any execution centralization. ICENI II is described in [25, 24]. Three steps are identified in the workflow generation: specification, realization and execution. Specification produces an abstract workflow whereas realization aims at validating the workflow and then map its elements to concrete resources. Execution deals with the monitoring of the application and functionalities to allow component migration.

AGWL. The Abstract Grid Workflow Language (AGWL) is the workflow language used by the ASKALON workflow manager [10] which offers two interfaces for generating large-scale scientific workflows in a compact and intuitive representation: graphical modeling using the UML standard and a programmatic XML based language. AGWL workflows can be either generated from a graphical UML description or directly written by the end-user. AGWL workflow descriptions are definitely independent from the execution resources. A dedicated scheduler is responsible for resource allocation and a resource manager handles

reservation. AGWL workflows include both control-flow and data-flow. Control-flow constructs include `sequences`, `dags`, `for`, `forEach`, `while` and `do-while` loops, `if` and `switch` construct and more advanced constructs such as `parallel activities`, `parallelFor` and `parallelForEach` loops and collection iterators. The user can also specify properties and constraints (such as memory requirements) for activities and data flow dependencies. An example from [10] underlines that dynamic loops can be defined, which lead us to put this language in the functional workflows category. ASKALON uses another language, CGWL in order to have a tasks graph representation of the workflows. Before the execution, the workflow manager performs a mapping from AGWL to CGWL.

Choreographies. The term *choreography* originates in a metaphor of a workflow which is viewed as an artistic work performed by actors, *i.e* the activities of the workflow. In that sense, choreography is opposed to *orchestration*: in a choreography, each actor is linked to other ones and the global process is obtained as a result of those local interactions. On the contrary, in an orchestration, actors are directed by a central conductor which manages the whole orchestra. Choreography and orchestration are terms that are tightly related to the Web-Services, as specified by the W3C. Choreography is thus often categorized as a decentralized approach whereas orchestration is centralized [22]. However, even if the workflow *description* is not centralized in a choreography as it is in an orchestration, the practical implementation of a workflow manager that would permit such a decentralized execution is not specified. Extensions of Web-Services such as WSRF and OGSA seem to be mandatory in order to have such a decentralized execution.

The initial choreography specification was the Web-Services Choreography Interface[2] (WSCI). WSCI allows a Web-Service to define *interfaces* that describe processes from its operations. Operations can be composed in sequential or parallel executions and loops and conditions can be defined. WSCI interfaces describe choreographies between the operations of a Web-Service. WSCI defines *global models* on top of operations. Global models describe choreographies between interfaces of several services. It provides a set of connections (mappings) between pairs of individual operations of communicating activities. In [4], authors formalize WSCI using π-calculus. WSCI set up the basis for the development of the Web-Services Choreography Description Language[3](WSCDL). In this language, *interactions* are defined among different *roles*. Roles can be played by different *behaviors* that may (optionally) be linked to particular WSDL interfaces. Indeed, the W3C candidate recommendation for WSCDL specifies that:

> A behavior without an interface describes a roleType that is not required to support a specific Web Service interface.

Thus, WSCDL choreographies are far from being executables: they only describe patterns for message exchanges among abstract activities. According to the W3C, a choreography language is not an executable business process description language or an implementation language. The role of specifying the execution logic of an application will be covered by these specifications.

Makefile. Makefiles are a particular kind of task workflow that completely relies on data flow. Activities of the workflow are defined by a command line that includes services

[2]http://www.w3.org/TR/wsci/
[3]http://www.w3.org/TR/ws-cdl-10/

(executables) and data (arguments of the command line). Tasks are linked by precedence constraints. When a task is ready to be executed, it is effectively fired if and only if one of its input files has been modified since the last invocation. Makefiles can include conditionals and loops. Thus, the number of tasks generated by the execution of a makefile may not be known prior the execution. Moreover, it is possible (with the `-j` option of the workflow engine `make`) to define a number of processes that may run concurrently, potentially on different CPUs, so that the resources are not defined inside the Makefile. Consequently, it has to be categorized as a functionnal workflow.

2.3 Services workflows

This section describes some workflows representations where both functions and resources are specified. Actually, those examples include resources in their description through their reference to Web-Services. A WSDL document indeed specifies the endpoint of the service. As a consequence the workflow manager cannot perform any scheduling. Yet, scheduling is still possible at the last minute, for instance using a particular submission service as will be discussed in section 2.6.

Scufl (Taverna) workflows. Scufl is a data-flow oriented language that basically describes the pipeline of an application. Many different kind of activities are specified by Scufl. For instance, string constants fire only once and return a single string value. Web-Services can also be enacted by specifying a WSDL document and a particular operation as well as compiled Java code or Beanshells activities[4] that embed a piece of Java code.

Sources and sinks correspond to the inputs and outputs of the workflow. Each of them may contain several data segments on which the workflow is iterated. Their content is not specified inside the Scufl document: it is independent from the workflow description and is only known at run time. In that sense, Scufl is a typical example of functional workflow. However, it is true that Web-Services activities are bound to a particular resource, included in their WSDL description. A Scufl workflow instantiated on some input data could thus be considered as an executable workflow rather than a tasks graph.

Scufl does not include control structures. However, the `FailIfFalse` and `FailIfTrue` activities are defined to implement conditional branching in a workflow, although no control operator such as *if* is defined in Scufl. Those activities fail or succeed depending on their Boolean input value, thus discarding or enabling the activities depending on them in the workflow.

Attached to each activity with at least 2 input ports is an iteration strategy. Iteration strategies are used to control how multiple items of the input ports are combined. Scufl iteration strategies are detailed in section 3.3.5.

Activities input and output *ports* that can contain several data items and are inter-connected with *data links*. An output port can be connected to several input ports. In this case, the data items are broadcast to all the connected input ports. Similarly, several output ports can be linked to a single input port. In this case, data items are buffered into the input port according to their order of arrival. *Control links* can also be specified in Scufl to provide elementary coordination constraints.

No control structures are available. Apart from the basic control link, the workflow is completely driven by the presence or absence of data in the input ports of an activity: it

[4] http://www.beanshell.org/

will fire if and only if all of its ports contain adequate data. It is not possible to define variables in Scufl. As a consequence, there is no expressions nor operators in the language.

MoML (Kepler) workflows. Activities of a MoML workflow are called *actors*. In MoML, each actor must define the type of each of its ports. Links (called *relations*) can only be defined between ports with compatible types. Ports participating in several relations have to be defined as *multi-ports*.

MoML defines no semantics for an interconnection of components. It instead provides a mechanism for attaching a “director” to a model. The director defines the semantics of the interconnection. MoML knows nothing about directors except that they are instances of classes that can be loaded by the class loader [17]. 4 directors can be defined in Kepler: Continuous Time (CT), Discrete-Event (DE), Synchronous Data Flow (SDF) and Process Networks. The CT director is used to model physical systems: the workflow is then directed by a clock. In the DE director, the workflow is also directed by a clock. Each actor communicates with the other ones by sending them timestamped signals. The director orders those signals and distributes them to their targets. In the PN director, each actor is executed in a dedicated thread. Relations between actors are waiting queues of finite capacity. Writing into a queue is never blocking whereas reading in an empty queue is blocking. The SDF director is used to simulate data flows.

Orchestrations: BPEL, BPML, BPMN, WSFL, XLANG. Orchestrations are workflows of Web-Services. This denomination also originates in a metaphor of a workflow which is viewed as a musical partition interpreted by the activities and directed by the workflow engine. Orchestrations differ from choreography by the point of view adopted by the developer. In an orchestration, a single workflow engine is responsible for the execution of the application. It centralizes the services invocations so that services do not communicate between each other [22]. Orchestration is also referred to as a concrete workflow whereas choreography is abstract. Indeed, in a choreography, resources are not mandatorily defined whereas orchestration precisely defines services WSDL and consequently endpoints.

The *de facto* orchestration standard is BPEL. It emerged from previous specifications: WSFL, XLANG, BPML and BPMN that do not survived the BPEL emergence. In [42], the authors provide a technology survey of workflow languages for Web-Services in 2002. In particular, a detailed analysis of WSFL is provided. WSFL includes both control and data links. From our classification point of view, a remarkable feature of this language is the identification of the services participating in the workflow by using a *locator* element which allows a service to be described by a static (hard reference to a WSDL), a local, a UDDI (the service is looked up using the UDDI API) or a mobility (the service provider is referenced in a message generated by some activity of the workflow) binding, which allows us to put this language in the “functional without resource specification” workflow class.

In its current 2.0 version, BPEL includes several control constructs: `switch`, `pick`, `while`, `for each`, `repeat until`, `wait`, `sequence` and `flow`. Activities may include Web-Service invocations, receive and reply and variable assignation. It proposes a fault handling mechanism through the `exit`, `throw`, `rethrow` and `compensate` constructs. Because of those control constructs, it is not possible to convert a BPEL workflow definition to a DAG. In particular, it is not possible to determine the number of service invocations, which may be dependent on the nature of the input data. That is why we put BPEL orchestrations in the “functional with resource specification” class.

YAWL. YAWL is built upon the Petri-Nets formalism. Its specification originates in an exhaustive study of workflow managers with respect to a set of workflow patterns [38]. Thus, the goal of YAWL is to overcome the expressiveness limitations of the contemporary workflow management systems. It is based on high-level Petri nets, to which extended constructs such as advanced synchronization, multiple instances and cancellation patterns are added, thus defining the extended workflow nets (EWF).

2.4 Tasks graphs

In tasks graphs, both functions and data are defined. A task is defined as the association of a treatment (*i.e* a function) with data items (*i.e* the parameters of the function). In this class of workflows, the tasks to be performed are completely defined: the workflow representation specifies their number as well as their nature.

Tasks graphs can be characterized by the fact that the amount of execution tasks of the workflow is known prior the execution. Consequently, tasks graphs have to be directed acyclic graphs (DAGs) or at least to contain only bounded loops (*i.e* the ones for which the number of iterations is known before run time). For the same reason, conditional operators are not allowed in tasks graphs. The case of exceptions, compensation handlers, retries allowed in case of failure and other fault-tolerance mechanisms has to be distinguished from conditional operators. Indeed, even if those constructs lead to the generation of a potentially unpredictable amount of tasks by the workflow, they only concern particular execution conditions that are not supposed to occur in normal operation mode.

This class of workflows is intensively used in the development of parallel applications. They are the most suitable representation for scheduling. Indeed, the only missing information to have the workflow completely defined is the mapping onto resources, which is the goal of scheduling. There is abundant literature about the scheduling of tasks graphs [18].

Condor DAGMan. Condor DAGMan [19] is one of the most used tools for tasks graphs. It allows the user to define precedence constraints between Condor jobs that are submitted to a pool of resources. So-called "pre" and "post" scripts may be defined to be executed respectively prior or after the job itself. Fault-tolerance facilities are also available, such as the ability to define a number of retry attempts in case of failure during the execution.

DIET MA-DAG. DIET is a grid middleware providing scalable scheduling facilities for grid servers [6]. MA-DAG, a workflow management system has been developed on top of it [3] and is based on a DAG model. This approach focus on scheduling, by offering the ability to use different advanced algorithms. Multi-workflow scheduling is also under investigation.

XWFL. The Workflow Enactment Engine (WFEE) uses the XML-based Workflow Language (XWFL) [44]. This language allows users to describe tasks and their dependencies. This language is made of three sections: parameter definitions, task definitions and data link definitions. This language supports both abstract and concrete workflows: resources can be specified so that we could also put this language in the executable class. Parameters can be used in order to define parametric tasks as described in the previous paragraph. Data links are then used to specify the tasks graph.

GridAnt. GridAnt is the workflow manager built on top of the Java CoG Kit. GridAnt is based on an existing commodity tool called Ant. Ant provides a flexible mechanism to express script dependencies n a project build process. Apart from expressing task dependencies, it also supports sequential and parallel execution constructs that allow sub-tasks to be executed in sequence or in parallel. However, to support arbitrary control flows through dependency graphs, one must support constructs that allow conditional execution, block iteration (looping), exception and error handling and several other workflow patterns defined by Van-der-Aalst [39]. Ant provides mechanisms only to direct the flow of control. It lacks the infrastructure to support workflow composition allowing the output of one activity to become the input to another. Through additional components in the GridAnt system, we overcome this restriction, enabling GridAnt to support workflow orchestration and composition [41].

GridAnt is included in Karajan. Karajan includes `if`,`while` and `for` constructs. Added support for flow control constructs such as conditions and loops. Supports parallel loops and parallel choice. Variables and operators. Lists and ranges [40].

Yvette ML. The YML framework defined YvetteML, a parallel programming language which is used to model workflows [9, 8]. YvetteML includes a component model and a graph description language. Components are defined as an encapsulation of task nodes of a directed acyclic graph representing a complex application. They represent a chunk of computation requiring no communication with the rest of the application. Components are made of a so-called abstract declaration, which specifies the type and mode (in, out or inout) of the parameters as well as a user-provided implementation that adds some decorations to a C/C++, Fortran or Java code in order to be able to compile it on different platforms. The YvetteML graph language is a control-flow language. Several control constructs dedicated to parallel applications are present such as *par do*, *seq do*, *wait* or *signal*. A typical example (extracted from [9]) of the YvetteML graph language is:

```
const problemSize := 10000;
event evt[2];
var MatrixReal vRes[1];
par(i:=1; problemSize) do
   compute fillMatrixReal(vRes[i],problemSize,i);
   signal(evt[i,1]);
end par do
```

The YML Framework interacts with the user using a compiler which translates components into binary applications. The model of the YML workflow framework can contain loops, iterations and branching: the compiler completely expands graphs to make them ready for scheduling. Loops are unrolled, condition evaluated, unvisited branches spread out of the graph and constants are propagated. The compiler translates applications described using the YvetteML language to a set of components calls. Regarding our workflow classification, the YvetteML compiler acts as a translator from a functional workflow instantiated on its input data to a tasks graph. However, the dynamicity of the functional workflow approach cannot be handled by YML and the number of tasks generated by the application is foreseeable. That is why we put it in the tasks graph class. Yet, the YvetteML workflow language seems very similar to the one of the Virtual Data Language [46].

2.5 Executable workflows

Executables workflows correspond to the mapping of a tasks graph onto resources or the instantiations of a flow of services onto data sets. This form is used for enactment only. It does not provide any level of abstraction and therefore it does not raise any interest from a user point of view.

2.6 Enacting workflows: from workflow "code" representation to executable workflows

A workflow engine is needed to transform any workflow source code into an executable list of tasks that can be submitted to a grid infrastructure. Depending on the workflow language considered, the work and the possibility of the workflow engine differ. Task-based workflows are graphs of computable tasks. The role of the enactor is then to schedule the execution of these tasks on grid resources, respecting the dependencies defined. Conversely, service workflows determine the resources to use. The role of the enactor is then to create the data flows starting from the workflow input data sets described independently and invoking the corresponding resources. Considering functional workflows, both data flow composition and scheduling have to be considered.

To produce a concrete workflow, some tools use an intermediate representation explicitly. For instance, ICENI transforms so called *execution plans* (functional workflows, considered as user views) into *temporal views* (tasks graphs). Askalon transforms AGWL (functional) into CGWL (task graphs). Pegasus uses an execution planer to transform VDL (functional) into concrete Pegasus (executable). Conversely, WS-based languages such as WSFL, GSFL and Choregraphy (functional) produce BPEL code (service workflows). Hence, functional workflow language enactors have to make a choice between first composing data flows (move from left to right column in figure 2) or first scheduling execution (move from top to bottom row). In figure 2, the functional workflows have been divided in two sub-sets depending on there enactment strategies: data composition first (tasks graph production) or service localization first (service workflows).

The production of tasks graphs prior to the execution is only possible when the exact data segments is known and all looping control structures are bounded. The workflow languages are then limited to a restricted number of control structures and the users have to produce explicitly data sets. If both conditions are not met, it is still possible to perform scheduling on the fly but only considering sub-parts of the workflow for which all tasks are known. This *last minute instantiation* of the data flows let more flexibility to the user (no limitations on the control structures and more dynamic data sets) but prevent many schedulers from operating optimally. It should also be noted that the service category of workflow is very restrictive in a grid environment as it does not let *a priori* any flexibility to schedule the resulting tasks on grid resources. This limitation is overcome by two different engines where *last minute location* is implemented: in the case of GSFL, the OGSA WS factory is used to instantiate services on the fly on grid resources while in the case of MOTEUR, a specific grid interface WS is invoked that will delegate the handling of the call to a grid scheduler.

Last minute data flows instantiation is needed to improve user-friendliness of tasks graphs approaches and last minute location is needed to improve performance of service approaches.

3 Workflow languages evaluation criteria

3.1 Workflow languages and workflow engines

It is not always easy to separate workflow languages from the workflow engines used for their enactment. It should be noted that in many cases, workflow languages are syntactically defined but the precise semantics of their execution is not. Therefore, two workflow engines for a same language can easily produce slightly different results especially when considering parallel execution.

Workflows execution performances heavily depend on the workflow engine. Although a language may represent parallelism, the enactor may or may not implement parallel execution. For instance, the Scufl workflow description language is inherently data parallel but the associated Taverna enactor was not designed to exploit grid resources and it limits the execution to 10 parallel threads even if the user provides more than 10 independent data fragments.

In section 3.3 we only focus on the languages, considering that different workflow enactors could be implemented regardless of the current tool limitations.

3.2 Workflow languages and external codes

The aim of workflow languages is to orchestrate the execution of external scientific codes. These codes are themselves written in highly expressive languages such as C++ or Java that we will call *native* language by opposition to the *workflow* language. The execution of a workflow causes the alternative execution of pieces of workflow and native code.

The workflow and native languages may have different levels of expressiveness: it may be possible to represent in a piece of external native code some operations that a workflow language could not express. For instance, the Scufl language does not provide any expressions and therefore cannot represent control structures such as `if` or `while` which test conditions should be represented as expressions. However, an external piece of code could be executed that will evaluate any expression and depending on the result produce some output that will modify the workflow execution procedure (decide on which branch of the `if` to enact or loop in the workflow). Some authors have even reported more indirect tricks to bypass the limitations of their workflow languages. For instance, a tasks graph cannot represent loops but using the `retry` on fail capability of DAGMan, a smart code can return errors on purpose to enforce the engine to retry and consequently loop on a node[5].

Such manipulations may enrich to some extend the workflow languages. However we do not consider these as contributing to the expressiveness of workflow languages as they depend on the writing of collaborative native code and they are usually reserved to expert users. When an application developer is reaching the expressiveness limitation of her workflow language, she should probably consider alternative languages.

3.3 Expressiveness

3.3.1 Turing completeness

In the case of programming languages, Turing completeness is often regarded has a reference in the language expressiveness as it implies the possibility to express any computable

[5] https://lists.cs.wisc.edu/archive/condor-users/2005-November/msg00000.shtml

algorithm. Turing completeness can be considered for workflow languages as well, although two points should be considered:

- As stated in the introduction, workflow languages are more focusing on expressing non functional features (e.g. parallelism) that might not be available in the native languages, and not necessarily any computable algorithm since a large part of the application complexity is embedded in native scientific codes.
- Given that native codes such as C, that are known to be Turing complete, can be embedded, Turing completeness can often be achieved by collaborative native code. For instance, any workflow language may represent a single processor executing the complete application written in C code. This ensures Turing completeness through external native code but this does not provide any information on the workflow language itself.

When not considering collaborative native code, most script-based language are probably Turing complete (they embed same kind of expressions and control structures than C). All tasks graphs-based approach cannot represent loops and therefore are not Turing complete. In the literature, there is little information on the Turing completeness of the various languages introduced in section 2.

3.3.2 Other formal evaluation methods

Schema relations and patterns may both be used to formally evaluate the expressiveness of workflow languages. While schema relations provide comparison elements between languages, patterns are more specific to the expressiveness of each language individually.

Schema relations. A concrete method to compare two workflow languages is to compare semantically their XML schemas. In [27], such a method is applied to compare BPML to BPEL4WS. The authors rely on previous works concerning the merging of heterogeneous data bases [7, 35] to compare different schemas. In [27], the authors conclude that BPML and BPEL4WS are neither symmetric nor equivalent but that several of their concepts overlap. This evaluation method is limited to XML-based workflow languages and require intensive human input to define the semantics of the language control structures.

Workflow patterns. A more application-oriented method to evaluate a workflow language is to test its ability to describe a set of *workflow patterns* [39]. In his thesis, Kiepuszewski investigates methods to analyze control flow aspects of workflow specifications [15]. Before entering into formal considerations, he uses very pragmatic methods that are the workflow test harness and workflow patterns. A mapping of workflows to Petri Nets is then presented. Workflow patterns have only been tested for a limited number of languages.

3.3.3 Expression of parallelism

The expression of parallelism, and especially data parallelism that is dominant in many scientific applications, is of high interest for grid-enabled workflows. Out of the grid community, many workflow languages do not provide parallel structures. Among the parallel workflow languages, the expression of parallelism significantly differs depending on the workflow representation adopted.

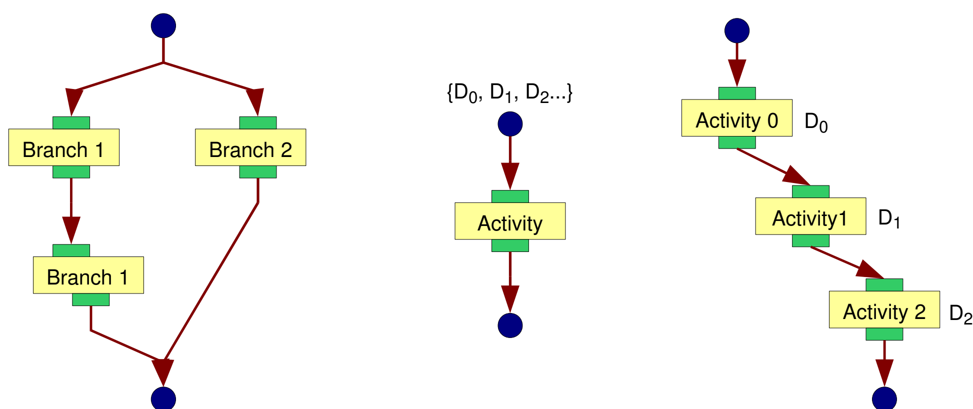


Figure 5: Three kinds of parallelism: workflow parallelism (left), data parallelism (center) and pipelining (right).

Any workflow graph of dependencies intrinsically represent some degree of parallelism: two workflow nodes that are not the ancestor of one another can be computed independently and therefore can possibly execute concurrently if they are available for processing at the same time. Depending on the workflow language considered, additional degrees of parallelism might be considered though. Figure 5 illustrates 3 kinds of parallelism that can be envisaged. On the left, workflow parallelism is depicted: branches 1 and 2 can be enacted in parallel. In the center, data parallelism is depicted: the activity can be fired concurrently as many times as there are input data segments. The last level of parallelism depicted on the right is a mix of code and data parallelism better known as pipelining: two different activities can be fired concurrently for processing two different data segments.

Figure 6 depicts the different approaches to represent data parallelism in different languages. In the tasks graph approach (right of figure 6), it is completely implicit in the workflow graph. Any independent branch in the DAG can be enacted concurrently. Therefore, there is no explicit parallel construct in the associated representation languages. There is no explicit data transfer either: each workflow node is defined with a specific data segment to process and two consecutive activities may or may not process the same data segment. Hence, only temporal synchronization links are needed. In this approach, the expression of data parallelism requires the replication of the DAG of processings over all data fragments considered. This approach quickly becomes humanly intractable when considering scientific application where tens to millions of data fragments may be involved. An upper layer DAG generator is then needed.

For non-tasks graphs approaches, the workflow graph only implicitly represent code parallelism. The expression of data parallelism depends on the constructs of the workflow language considered. A large family of workflow languages use explicit parallel operators (center of figure 6). For instance BPEL, which used to be a fundamentally sequential language, was recently enriched with a `foreach` operator. Other dedicated languages may include parallel control structures such as `foreach` or `dopar`. Among the languages available, one can distinguish between bounded parallel constructs, for which the number of tasks (the number of data fragments to process) is known in advance, and unbounded constructs for which the number of tasks will only be discovered dynamically at run time. In the first family, codes can be automatically translated in DAG representations

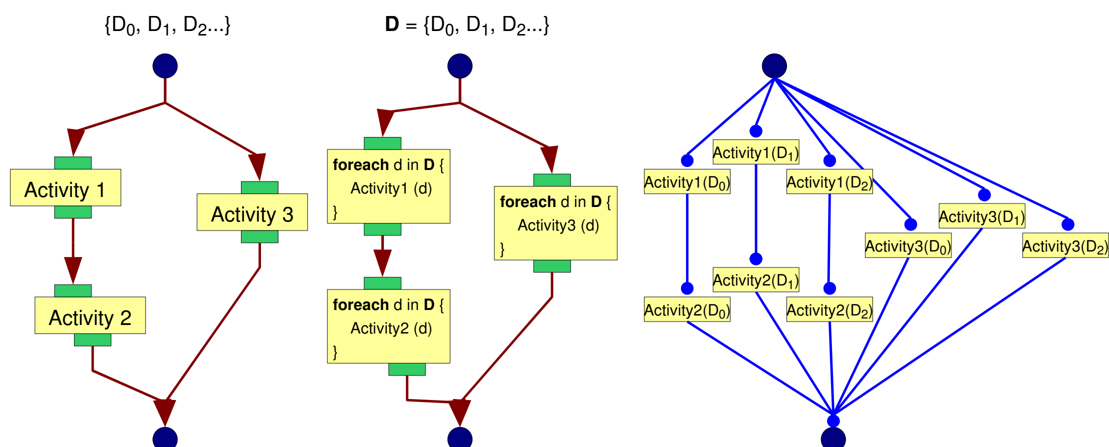


Figure 6: Data parallelism: completely asynchronous and transparent (left), explicit in the language (center) and DAG implicit (right).

prior to execution. The second family is more flexible as the number of data segments may depend on the computations but it prevents prior graph generation thus limiting scheduling strategies that can be applied. Last minute sub-DAGs can still be generated or the enactor has to deal with parallel execution by itself.

There exists an alternative approach to explicit language constructs, where data parallelism is implicit in the language (left of figure 6). In such languages, the data fragments are not explicitly represented through variables. Only the data flows are represented and data description is externalized. It is the combination of the description of the process and the data sets to process that will implicitly produce parallel data flows. In Scufl for instance, data fragments are not part of the language which only describes services to enact. At run time, the Scufl enactor can make as many service calls as needed to process all data segments that are dynamically discovered. Everything happens as if each service was embedded in an unbounded `foreach` kind of operator. A major difference though is related to the fact that `foreach` operators usually imply a data synchronization barrier as explained in section 3.3.4.

3.3.4 Synchronization over data sets

When considering parallel execution, an important feature is the ability to synchronize two concurrent flows. In workflows, code synchronization is obviously represented by the workflow graph dependencies. But when considering concurrent execution over different data sets, synchronization barriers concerning different data segments are not explicitly represented in the workflow graph for non-tasks graphs approaches. Data synchronization is differently handled depending on the workflow language considered. The semantics of `foreach` kind of control structure is usually such that the parallel execution only starts once all data fragments are available and the control flow will continue beyond the structure only once all data fragments considered have been processed: in other terms, it enforces data synchronization for each activity inside the workflow but it prevents the use of pipelining. This strategy may be valid for parallel architectures but it may become very penalizing in the case of tasks submitted to grid infrastructures with highly variable

execution times.

A major difference between a data parallel such as Scufl and parallel control structures is that Scufl fires a service asynchronously for each data segment to be processed at the time it becomes available. It thus exploits both data parallelism and pipelining. In Scufl data synchronization can still be obtained through the semantics of control links: a control link is blocking for a target activity as long as the source activity it depends on has not completed (*i.e.* it has not processed all possible data segments in the workflow).

In both case (data parallel control structure or data parallel language), the synchronization is concerning all the data fragments. One could imagine cases where synchronization is only concerning a sub-group of data: Gwendia applications exhibit such use cases. A more flexible data synchronization barrier expression mechanism is needed. It is possible to deal with it using embedded workflows in Scufl but partial synchronization is not really supported in the language itself.

In addition, it should be noted that synchronization over all data fragments may be hard to achieve in an unreliable grid context where very large data sets may be used: for instance in the drug discovery application of Gwendia, millions of concurrent processings are envisaged and it is unlikely that none of them will fail for various reasons even using retry on error policies. In many similar cases where it is more important to get a result over the *majority* of data fragments rather than *all* data fragments it is useful to express synchronization over *approximately* all data fragments. The fraction of admissible errors has to be let to the user. To our knowledge, no language provide *partial synchronization* nor *approximately all* synchronization capabilities.

3.3.5 Data composition strategies

In the discussion above and in figure 6, we have only considered services with single input ports. It is common though that a service accepts multiple input ports receiving multiple input data sets. The way data segments received on different ports are combined for processing is explicit when considering languages with parallel control structure. For instance for a service with to input data sets **A** and **B**, two embedded `foreach` loops on the elements of **A** and **B** such as:

```
foreach a in A
  foreach b in B
    fire activity(a, b)
```

will cause the explicit invocation of the service $|\mathbf{A}| \times |\mathbf{B}|$ times.

A fully data-oriented language has no control operators but alternative *iteration strategies* to deal with data flows composition. For instance, Scufl defines two basics iteration strategies known as *cross product* and *dot product* illustrated in figure 7.

The most commonly used iteration strategy is the dot product corresponding to a *one-to-one* composition of data segments: each data segment of the first set is composed with the matching data segment of the second set in their order of definition, thus producing $\min(|\mathbf{A}|, |\mathbf{B}|)$ results. This corresponds to the case where a sequence of pairs need to be processed. Another common composition strategy is the cross product corresponding to an *all-to-all* composition of data segments: all input data segments from the first set is combined with all input data segment from the second set, thus producing $|\mathbf{A}| \times |\mathbf{B}|$ results. The semantics of invocation for activities with more than two input ports is defined by building an expression made of a binary tree of dot and cross-product operators which leafs are the activity input ports.

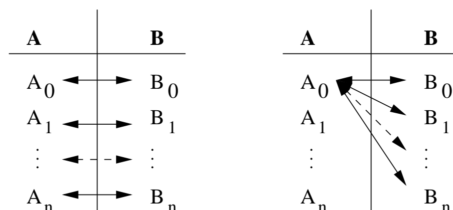


Figure 7: Scufl data composition operators. Left: dot product. Right: cross product.

It can be noted that the cross product semantics is almost equivalent to the embedded `foreach` loops given above as an example. There is a significant difference though related to data synchronization barriers: while the control structure will enforce synchronization, a fully data-oriented language will not. Activities are fired as soon as data segments become available and the iteration strategies are dynamically reevaluated each time new input data segments become available.

Moreover, most workflow languages provide only the dot-product operator for data composition. The semantics of this one-to-one composition is more difficult to define as it relies on an association between pairs of data segments. In case of sequential execution, the association is usually defined as the order of arrival of data segments in input ports. In case of parallel execution, the association needs to be clearly defined as the order of execution and data arrival is not guaranteed. In [29] we propose a clear semantics based on the definition of groups of data sets.

The data composition strategies are a compact and powerful mechanism to describe data parallelism without explicitly writing parallel code. To our knowledge, only Scufl does introduce both one-to-one and all-to-all strategies. It does not clearly define the semantics of the one-to-one composition in a parallel context though.

4 Summary

Table 1 summarizes the main criteria that are discussed in this document for a number of workflow languages that are representative of the different categories identified. The languages summarized here are all of interest for the GWENDIA applications in the sense that they provide some level of data parallelism.

4.1 Workflows expressiveness

The model line in table 1 refers to an existing formal description model. The rest of the table lines are grouped by kind of criteria: related to expressiveness, other properties of interest and implementation. The expressiveness is qualified depending on:

- Existing control structures: graphs of tasks do not include control structures while functional languages provide classical control structures (conditional and loops). Scufl is a particular language that is purely based on data flow definitions and for which only a particular implementation of the conditional is provided. YAWL does not provide traditional control structures but rely on patterns instead.

	workflow language	AGWL	CGWL	Scufl	MA DAG	Swift	GSFL	Makefile	YAWL	DAGMan
	classification (see figure 2)	func.	tasks	services	tasks	func.	func.	func.	?	tasks
	model	UML	DAG	no	DAG	no	no	no	Petri net	DAG
Expressiveness	ctrl. structures	yes	?	fail_if only	no	yes	yes	yes	patterns	no
	unbounded ops	yes	no	yes	no	yes	yes	yes	yes	no
	data parallelism	foreach	implicit	yes	implicit	foreach	yes	yes (-j)	yes (Petri)	implicit
	data links	yes	?	yes	yes	yes	yes	no	yes	no
	iter. startegies	no	no	yes	no	no	no	no	no	no
	patterns	?	?	?	?	?	?	?	yes	?
Properties	data description	no	no	WS only	yes	yes	WS only	no	no	no
	scheduling	last min.	last min.	last loc.	yes	last min	OGSA fact	no	?	yes
	Turing compl.	probably	?	native	prob. no	probably	?	probably	?	prob. no
Implementation	invocation	indep.	?	WS+local	GridRPC	?	OGSA/WS	local	WS	Condor
	enactor	Askalon	Askalon	Taverna MOTEUR	DIET	yes	yes	make	yes	Condor DAGMan

Table 1: Summary of workflow languages expressiveness

- Existing unbounded operations: only languages that are not instantiated in a concrete graph prior to the execution can represent unbounded control structures.
- Expression of data parallelism: data parallelism might be implicit (graphs of tasks), explicit (`foreach` kind of control structure) or inherent to the language (pure data flows).
- Existing data links: some languages only propose control links (or coordination constraints) while other include data transfer links.
- Existing iteration strategies: only Scufl implements different iteration strategies to our knowledge.
- Patterns that can be represented: YAWL has been designed to enact a broad variety of patterns. Some of these patterns are expressible in other languages but an exhaustive study of all patterns and their equivalence to other control structures for different languages should be performed.

The other properties of interest are:

- Data types description: in many cases, the workflow is defined independently from the data to process although in some cases the language include data types description and type checking is possible. For all workflow managers enacting Web-Services, the workflow language itself does not include data types but they can be found in the WSDL document describing the services enacted.
- The ability to provide a schedule: definitely true for tasks graphs, only last minute instantiation schedules can be created for the most expressive languages (with unbounded control structures or undetermined data sets). For service workflow languages, no schedule is usually possible although last minute location has been implemented in some cases.
- Turing completeness: this property is rarely demonstrated although highest level languages (with control structures) are probably Turing complete while tasks graphs are probably not. Scufl has been demonstrated to be Turing complete if the transition function is implemented in native code [14].

Often, there exists a single enactor for each workflow language although some exceptions can be found. Many remote invocation code strategies are envisaged including Web Services, GridRPC and Condor jobs submission.

4.2 Workflow languages in Gwendia

This study shows that a trade-off has to be found between workflow languages expressiveness and the efficient scheduling of the tasks generated by a workflow engine. Functional and service-based languages provide more flexibility to the users with the availability of loops and unbounded control structures. Conversely, tasks graphs are providing fixed topology graphs that can be scheduled more efficiently. In the context of the Gwendia project, we plan to explore both approaches.

From a user point of view, functional languages seem to be the most interesting approach. In the context of scientific applications, the expression of data parallelism is also mandatory. We are interested in the Scufl language for its capability to represent complex

data flows in a compact framework through its unique data composition operators. Scufl can indeed represent the applications of the Gwendia project and does not require from the user more than describing the data flows. *A priori* the parallel enactment is limited by the service approach adopted but last minute location proved to be an effective way of ensuring parallel enactment on a grid. It could be interesting to consider extensions of this language with loops and conditionals to obtain a language that is both providing unbounded control structures and iteration strategies. However, this would require significant changes in the language with addition of expressions.

From an optimization point of view, tasks graphs are preferable. In the case of large scale data parallel applications, the very large DAGs needed cannot be produced by hand though. In the context of the Gwendia project, we plan to generate the application DAGs automatically to provide a fixed set of tasks to the scheduler. This will be possible by performing some processings prior to the DAGs generation to determine the exact number of data fragments to consider. This will impose some limitations on the applications but the benefit in terms of performances has to be estimated.

References

[1] T Agerwala. A complete model for representing the coordination of asynchronous processes. Technical Report 32, John Hopkins University, Hopkins Computer Science Program, Baltimore, July 1974.

[2] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.

[3] Abdelkader Amar, Raphaël Bolze, Aurélien Bouteiller, Andréea Chis, Yves Caniou, Eddy Caron, Pushpinder Kaur Chouan, Gaël Le Mahec, Holly Dail, Benjamin Depardon, Frédéric Desprez, Jean-Sébastien Gay, and Alan Su. DIET: New Developments and Recent Results. Technical Report RR-6027, INRIA Rhône-Alpes, 2006.

[4] Brogi Antonio, Carlos Canal, Ernesto Pimentel, and Antonio Vallecillo. Formalizing Web Services Choreographies. *Electronic Notes in Theoretical Computer Science*, 105:73–94, 2004.

[5] Roger Barga and Dennis Gannon. *Scientific versus Business Workflows*, chapter 2, pages 9–16. In [36], 2007.

[6] Eddy Caron and Frédéric Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 2005.

[7] S Ceri and J Widom. Managing Semantic Heterogeneity with Production Rules and Persistent Queries. In *19th International Conference on Very Large Data Bases*, pages 108–119, San Francisco, USA, 1993.

[8] Olivier Delannoy, Nahid Emad, and Serge Petiton. Workflow Global Computing with YML. In *7th IEEE/ACM International Conference on Grid Computing*, pages 25–32, Barcelona, Spain, September 2006.

[9] Olivier Delannoy and Serge Petiton. A Peer to Peer Computing Framework ; Design and Performance Evaluation of YML. In *Third International Workshop on Algorithms, Models, and Tools for Parallel Computing on Heterogeneous Networks*, pages 362– 369, July 2004.

[10] Thomas Fahringer, Radu Prodan, Rubing Duan, Jürgen Hofer, Farrukh Nadeem, Francesco Nerieri, Stefan Podlipnig, Jun Qin, Mumtaz Siddiqui, Hong-Linh Truong, Alex Villazon, and Marek Wieczorek. *ASKALON: a development and grid computing environment for scientific workflows*, chapter 27, pages 450–471. In [36], 2007.

[11] Ian Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying and Automating Data Derivation. In *Scientific and Statistical Databases Management*, Edinburgh, UK, 2002.

[12] Dennis Gannon. *Component Architectures and Services: From Application Construction to Scientific Workflows*, chapter 12, pages 174–189. In [36], 2007.

[13] Yolanda Gil. *Workflow Composition: Semantic Representation for Flexible Automation*, chapter 16, pages 244–257. In [36], 2007.

[14] Tristan Glatard. *Description, optimization and exploitation of medical imaging applications on production grids.* Phd thesis, Nice-Sophia Antipolis University, Sophia Antipolis, France, 2007.

[15] Bartosz Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows.* PhD thesis, Queensland University of Technology, Brisbane, Australia, February 2003.

[16] Lars M Kristensen, Soren Christensen, and Kurt Jensen. The practitioner's guide to coloured Petri nets. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(2):98–132, December 1998.

[17] Edward A. Lee and Steve Neuendorffer. MoML - A Modeling Markup Language in XML, Version 0.4. Technical Report UCB/ERL M00/12, University of California, Berkeley, CA 94720, March 2000.

[18] Arnaud Legrand and Yves Robert. *Algorithmique parallèle.* Dunod edition, 2003.

[19] Miron Livny. Direct Acyclic Graph Manager (DAGMan). http://www.cs.wisc.edu/condor/dagman/.

[20] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, (70):96–118, 2007.

[21] J Magott. Performance evaluation of concurrent systems using Petri nets. *Information Processing Letters*, 18(1):7–13, 1984.

[22] A. Mayer, S. McGough, Nathalie Furmento, William Lee, Murtaza Gulamali, S. Newhouse, and J. Darlington. Workflow Expression: Comparison of Spatial and Temporal Approaches. In *Workflow in Grid Systems Workshop, GGF-10*, Berlin, March 2004.

[23] Manuel Mazzara and Sergio Govoni. A Case Study of Web Services Orchestration. In *Coordination Models and Languages*, LNCS 3454, pages 1–16. Springer Verlag, 2005.

[24] S. McGough, Jeremy Cohen, J. Darlington, Eleftheria Katsiri, William Lee, Sofia Panagiotidi, and Yash Patel. An End-to-end Workflow Pipeline for Large-scale Grid Computing. *Journal of Grid Computing (JGC)*, pages 1–23, February 2006.

[25] S. McGough, William Lee, Jeremy Cohen, Eleftheria Katsiri, and J. Darlington. *ICENI*, pages 395–415. In [36], 2007.

[26] S. McGough, L Young, A Afzal, S. Newhouse, and J. Darlington. Workflow Enactment in ICENI. In *UK e-Science All Hands Meeting*, pages 894–900, Nottingham, UK, September 2004.

[27] Jan Mendling and Martin Müller. A Comparison of BPML and BPEL4WS. In *1st Conference Berliner XML-Tage*, pages 305–316, Berlin, 2003.

[28] R Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.

[29] 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.

[30] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[31] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 3, Bonn, 1962.

[32] Frank Puhlmann. Why do we actually need the Pi-Calculus for Business Process Management? In *9th International Conference on Business Information Systems (BIS06)*, Klagenfurt, Austria, June 2006.

[33] Frank Puhlmann and Frank Puhlmann. Using the pi-Calculus for Formalizing Workflow Patterns. In *Third International Conference on Business Process Management (BPM'05)*, LNCS 3649, pages 153–168, Nancy, France, September 2005.

[34] Howard Smith and Peter Fingar. Workflow is jus a Pi process. 2003.

[35] S Spaccapietra, C Parent, and Y Dupont. Model Independent Assertions for Integration of Heterogeneous Schemas. *Very Large Data Bases Journal*, 1:81–126, 1992.

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

[37] Wil M.P Van Der Aalst. Why workflow is NOT just a Pi-process. 2004.

[38] Wil M.P Van Der Aalst and Arthur H.M. Ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[39] Wil M.P Van Der Aalst, Arthur H.M. Ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[40] Gregor von Laszewski and Mike Hategan. Workflow Concepts of the Java CoG Kit. *Journal of Grid Computing (JGC)*, 3(3-4):239 – 258, September 2005.

[41] Gregor von Laszewski, Amin Kaizar, Mihael Hategan, Nestor J. Zaluzec, Shawn Hampton, and Albert Rossi. GridAnt: A Client-Controllable Grid Workflow System. In *37th Hawai'i International Conference on System Science*, Island of Hawaii, Big Island, January 2004.

[42] P Wagstrom, S Krishnan, and Gregor von Laszewski. GSFL: A Workflow Framework for Grid Services. In Ivar Austvoll, editor, *Scandinavian Conference on Image Analysis*, Bergen, Norway, June 2002.

[43] Simon Woodman, Savas Parastatidis, and Jim Webber. *Protocol-Based Integration Using SSDL and pi-Calculus*, pages 227–243. In [36], 2007.

[44] Jia Yu and Rajkumar Buyya. A novel architecture for realizing grid workflow using tuple spaces. In *Proceedings. Fifth IEEE/ACM International Workshop on Grid Computing*, pages 119–128, November 2004.

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

[46] Yong Zhao, M. Wilde, and Ian Foster. *Virtual Data Language: A Typed Workflow Notation for Diversely Structured Scientific Data*, chapter 17, pages 258–275. In [36], 2007.