Three Fundamental Dimensions of Scientific Workflow Interoperability: Model of Computation, Language, and Execution Environment

Erik Elmroth Francisco Hernández Johan Tordsson

UMINF 09.05

Dept. of Computing Science and HPC2N
Umeå University, SE-901 87 Umeå, Sweden
{elmroth, hernandf, tordsson}@cs.umu.se

Abstract

We investigate interoperability aspects of Grid workflow systems with respect to *models of computation (MoC)*, *workflow languages*, and *workflow execution environments*. We focus on the problems that affect interoperability and illustrate how these problems are tackled by current scientific workflows as well as how similar problems have been addressed in related areas. Emphasis is given to the differences and similarities between local and Grid workflows and how their peculiarities have a positive or negative impact on interoperability. Our long term objective is to achieve (*logical*) interoperability between workflow systems operating under different MoCs, using distinct language features, and using different execution environments.

1 Introduction

To date, scientific workflow systems offer rich capabilities for designing, sharing, executing, monitoring, and overall managing of workflows. The increasing use of these systems correlates with the simplicity of the workflow paradigm that provides a clear-cut abstraction for coordinating stand-alone activities. With this paradigm scientists are able to concentrate on their research at the problem domain level without requiring deep knowledge of programming languages, operating systems, arcane use of libraries, or hardware infrastructure. In addition, the ease by which scientists can describe

experiments, share descriptions and results with colleagues, as well as automate the recording of vast amounts of data, e.g., provenance information and other data relevant for reproducing experiments, have made the workflow paradigm the fundamental instrument for current and future scientific collaboration.

Currently, there are many sophisticated environments for creating and managing scientific workflows that have lately also started to incorporate capabilities for using powerful Grid resources. Although similar in many respects, including domains of interest and offered capabilities, existing workflow systems are not yet interoperable. Rather than discussing if workflow systems are completely interoperable or not at all, here we argue that interoperability between workflow systems must be considered from three distinct dimensions: *model of computation (MoC)*, *workflow language*, and *workflow execution environment*. In previous work [20] we have demonstrated workflow execution environment interoperability by showing how a workflow tool can interoperate with multiple Grid middleware. Extending on this effort, we here discuss interoperability at the other two dimensions, i.e., MoC and workflow language.

With this paper our contributions are the following. We start a dialogue and argue that we must change the manner in which interoperability has been addressed and start concentrating on a three dimensional model that considers interoperability from the MoC, the language, and the execution environment dimensions. We present the reasons why dataflow networks have not been used in Grid settings and introduce some adaptations required in this MoC to be able to make use of it in Grid workflows. We further investigate the minimum language constructs required for a language to be expressive enough for supporting scientific workflows. We argue for a distinction between the languages used for describing complex workflows executing on the end-users desktop machine (i.e., *local workflows*) and possibly simpler languages only used for coordinating computationally intensive sub-workflows that run on the Grid. We support our discussion on results and algorithms from the areas of theory of computation, compiler optimization, and (visual) programming language research. With these results as a starting point we discuss language aspects relevant for local and Grid workflows, including iterations, conditions, exception handling, and implications of having a type system associated to the workflow language. We conclude our work in workflow languages by studying the repercussions of choosing between *control-driven* and *data-driven* style of representing workflows, including methods for converting between both representations.

The rest of the paper is organized as follows. Section 2 discusses the

current state of the art in scientific workflows, investigates the reasons why interoperability in workflows is desired, and introduces the three dimensions that must be considered when discussing interoperability of scientific workflows. Section 3 introduces some fundamental workflow concepts and definitions that are used throughout the paper. Section 4 discusses issues related to MoCs for Grid workflows including a description of how Petri nets and Coloured Petri nets have been used as MoCs for Grid workflows and a discussion on the reasons why dataflow networks, being the most common MoC for local workflows, has not yet been employed for Grid workflows. Section 5 focuses on workflow language related issues including differences between data-driven and control-driven representations as well as the use and implementation of language constructs such as conditions and iterations in light of their programming languages counterparts. Finally, in Section 6 we discuss how our findings, collectively, have an impact on Grid workflow interoperability and present our concluding remarks.

2 Workflow Interoperability

There are many scientific workflow systems currently in use, e.g., [9, 15, 43, 51]. Several of these have been developed successfully within interdisciplinary collaborations between domain scientists, the *end-users*, and computer scientists, the *workflow engineers*. Some of these systems target a particular scientific domain (e.g., Taverna [51]) while others cover a range of fields (e.g., Triana [9] and Kepler [43]).

The existence of such a wide range of workflow systems is comparable, although to a lesser degree, to the large number of programming languages available. In both cases solutions can be general purpose or tailored for specific domains, and the choice of one over the others depends not only on the problem at hand but also on personal preferences. Moreover, it is not possible to have one solution suitable for all problems and preferred by all users, and it is also not likely for a new solution to emerge and replace all existing ones. Yet, unlike programming languages in which interoperability is achieved at the binary or byte code level and the voices suggesting source-to-source interoperability long faded away, achieving interoperability between workflow systems is a venture of high priority.

In this section we examine motivations for workflow interoperability, introduce a point of view that tackles the problem at multiple dimensions, and investigate various ways in which the topic has been addressed in the literature.

2.1 A case for interoperability

It has recently been suggested [53] that end-users are not really pressing for interoperability among workflow systems. The rationale behind this suggestion is that these systems are developed in tight coordination between end-users and workflow engineers. Hence, instead of using other systems that already offer the required functionalities new features are added when needed. The outcome of this rationale is time consuming as it leads to duplication of efforts, it mis-utilizes resources that could otherwise be employed in more productive endeavors, and it is mainly beneficial for researchers who are involved in this development loop. Yet, it is also the case that users may not be interested in *full interoperability* between workflow systems. Full interoperability is commonly defined as the ability for systems (be them human users or software components) to *seamlessly* use the full functionality provided by the (others) interoperable systems in a totally transparent manner [24]. Deelman et. al. [14] notice that users may want to invoke one workflow system from another one or reuse all or part of a workflow description in a different system. Another motivation for interoperability is due to portability aspects, e.g., due to infrastructure changes. User preferences can also be taken into consideration. Once users become accustomed to a particular system it becomes a costly process to migrate to another one. Although it is possible to enumerate a long list of use cases in which interoperability is of value, we prefer to identify the following two categories that cover several of the use cases according to the purpose for seeking interoperability:

Collaborative and interdisciplinary research. Science is a collaborative endeavor. The importance of current scientific problems have made crucial for these collaborations to become large interdisciplinary enterprises in which scientists from different fields contribute to the final solution. These significant efforts require the sharing of research knowledge, knowledge that is often expressed in workflows.

Several workflow systems have been developed for operation within a specific scientific domain, thus, it is expected that users from different scientific fields, or in some cases from different research groups, use different workflow systems. It is very difficult to change systems just to enable these collaborations. Users are more comfortable working in environments familiar to them and adapting to a different one may involve steep learning curves [31]. For these cases it becomes imperative to coordinate multiple workflow systems within one workflow execution. A typical scenario for this type of interoperability is a workflow

system invoking another one for executing a functionality represented as a sub-workflow.

Lack of capabilities in a workflow system. Adaptation of a system initially designed for one scientific field to fulfill the requirements of another is typically done by extending the set of activities (or capabilities) offered to users, rather than changing the way in which the workflow system itself operates. Such extensions are commonly added by implementing the new functionalities using libraries offered by the systems themselves. However, these functionalities are typically locked-in and can only be used inside the targeted environment making it impossible to share them with users of other systems. It is then important to unlock the functionalities so that they can be used by other systems. The capabilities need not be computations, they can also be support for different hardware or software platforms. An extreme case is illustrated by technology obsolescence. If a system becomes obsolete and needs to be replaced by a newer system, it is of paramount importance to be able to reuse the workflows developed for the older system. This obsolescence is not limited to the workflow system itself but the execution environment, including middleware, as well.

2.2 Multiple dimensions for workflow interoperability

From the two categories enunciated above, we identify three dimensions relevant for scientific workflow interoperability. These dimensions are in line with a previous classification [53], but we argue that some aspects presented in that work, e.g., meta-data and provenance, although very important in practice, are not essential for workflow enactment coordinated by a workflow engine. In practice, a workflow engine can cooperate with a meta-data or provenance manager to achieve other types of interoperability. In this work we focus exclusively in the enactment process, that is, selecting and executing workflow activities free of dependencies (either control or data dependencies). Below we briefly introduce the three dimensions of interoperability and we present further details in the following sections.

2.2.1 Model of computation

The model of computation provides the semantic framework that governs the execution of the workflow, i.e., a MoC provides the rules on how to execute the activities and consume the dependencies. There are many MoCs that

have been considered as central abstractions for coordinating workflows, including Petri nets, dataflow networks, and finite state machines. Some problems are better suited for one MoC and in many cases a single workflow may be required to use abstractions from multiple MoCs.

Strong interoperability in this respect requires the transparent execution of workflows developed for one MoC by another one. A weaker notion is to be able to compose workflows with parts governed by different MoCs. A solution proposed by Zhao et al. [70] uses a bus in which workflow systems are considered as black boxes that can be invoked from other systems. However, for this compositions to be possible the MoCs must be compatible. Compatibility between MoCs has been studied by Goderis et al. [29]. Their work explores when and how different MoCs can be combined. Their contribution is significant but it is focused only on local workflow systems and does not address Grid workflows. Many of the MoCs described by Goderis et al. [29] are not functional in Grid settings as the assumptions of globally controlling coordinators and fine grained token-based interactions between concurrently executing activities are not possible to achieve (yet) in Grid environments. The basis of their work is a hierarchical approach, based on Eker et al. [18], in which workflows from different MoCs can be combined according to the level of the hierarchy. Sub-workflows are then seen as black boxes and their internal MoC is of no importance when working one level above in the hierarchy. Petri nets and DAGs (employed as the structure for specifying dependencies between activities) are the most common MoCs in Grid environments. As such, in Section 4 we look into how Petri nets and Colour Petri nets have been used for Grid workflows. In that section we also investigate the reasons why dataflow networks, the most common MoC for local workflow systems, has not been employed in Grid environments.

2.2.2 Workflow language aspects

One important workflow interoperability aspect is given by the set of supported language constructs. The constructs of interest in this study are iterations and conditions, the latter used both for execution flow control and exception handling. Another related topic is how, if at all, state is modeled in workflows. Our previous work has demonstrated the possibility of completely decoupling the workflow coordination language from the language used to describe individual activities [20]. Others propose to describe workflows with a high-level meta-language, that is not dependent on a particular workflow language [53]. Fernando et al. [23] suggest an intermediate workflow description format, and outline how the languages of Taverna and

Kepler could be represented in such a format. In Section 5 we discuss the trade-offs of having a full featured, Turing complete workflow language versus a simplistic activity coordination language. Furthermore, we investigate the consequences of having a type system attached to a workflow language and discuss the difficulties that the environments of current Grid infrastructures cause in this regard.

2.2.3 Workflow execution environments

In traditional workflow systems, the activities that form the workflow all execute locally on the desktop computer of the user, making it a *local workflow*. The emergence of powerful parallel machines and Grids has opened up the potential for utilizing applications that execute on remote machines and possibly are developed by other scientists or organizations. Accordingly, it is essential for a workflow system to support distributivity at some level. Since many of the current projects are pre-Grid, they, naturally, are not focused on Grid workflows, and are not able to optimize the capabilities that systems designed for Grid usage offer. On the other hand, several of the Grid-only workflow systems can appropriately use Grid resources but they lack the ease of use and facilities offered by the local systems. The necessities of current research demand for a balance between the local and the distributed, so in typical scenarios the local machine is used for menial tasks while Grid resources are used for activities that require extensive use of resources (e.g., computation and storage). In this setting, the benefits of workflow systems is that they abstract the communication complexities required to interact with the Grid.

One issue in the design of toolkits for real-life scientific workflows is the suitable level of granularity for interacting with Grid resources. Some projects, e.g., Kepler [43], Taverna [52], and Karajan [67] use Grid resources on a per-activity basis. Others, e.g., Pegasus [15], GWEE [20], and P-Grade [38] use Grid resources to enact workflows, that typically constitute a computationally intensive subset of a larger workflow. We refer to such sub-workflows as *Grid workflows*. For the rest of this paper, we assume that a workflow is a local workflow that makes use of one or more Grid workflows, the latter ones being our focus.

3 Concepts and Definitions

In this section we introduce some concepts and definitions that are used throughout the paper.

A *workflow* is represented by a Directed Graph (DG) W = {Nodes, Links}, where: 1) Nodes is the set of *workflow activities*, and 2) Links is the set of *dependencies* between workflow activities.

W is a static element that specifies the structure and possible orchestration of workflow activities and is commonly specified by a workflow language. The workflow language is usually represented in a textual manner, although graphical interfaces have been employed for facilitating the interaction with the workflow system. The size and complexity of workflows varies, and while a graphical representation may be optimal for simple workflows, this type of representation is not feasible when scaling the number of workflow activities. In such situations, a textual representation is better suited. Furthermore, even when graphical interfaces are employed the graphical language is associated with a textual representation for storing and managing workflow instances in files [30]. In many cases XML is used for the textual representation but scripting languages can also be employed.

Ports serve as containers of data associated with workflow activities. Ports also state communication channels between activities providing entry (input ports) and exit (output ports) points to the workflow activities. For Grid workflows it is common to assume that ports have no associated type information and that workflow activities internally distinguish the correct semantics of the data in ports. In Section 5.1 we consider the implications for the cases where the ports are typed (typical in local workflows). Communication to and from nodes is performed by specifying links between ports associated to different activities. The links represent dependencies whose nature, i.e., control or data flow, is unimportant from a representation point of view.

A *workflow activity* represents a unit of execution in a workflow. An activity can be an indivisible entity or it can be a sub-workflow containing other activities. Associated input ports provide the required input for the activity while the output is produced through output ports. As such, activities can be treated as functions whose domain is given by the cross product between input ports and whose range is given by the cross product of the output ports. A distinctiveness of Grid workflows is that workflow activities are stand-alone command line applications that require a mapping between the expected command-line arguments and the input and output ports used for communicating with other activities. There are different manners in which this mapping can be achieved, e.g., see [8] and [69]. For the rest of the paper we assume that this mapping has been performed so that workflow activities are enacted with the appropriate parameters and the required information is moved to and from activity ports.

A *Model of Computation (MoC)* is an abstraction that defines the semantics in which the execution of a workflow W is to be carried out. A *workflow engine* is a software module that given a workflow description (W) and a MoC (M) as input, executes the activities in W following the rules specified by M. Thus, a workflow engine provides a concrete implementation of a MoC and it is responsible for enacting workflows. The enactment is performed by selecting activities to execute. The manner in which the activities are selected and how the communication between activities is carried out is defined in the MoC. The engine can execute in a local machine or it can be exposed as a permanently available service accessible by many users. The latter useful when executing long processes, as tools can reconnect to the engine for monitoring and managing purposes without requiring permanent connections.

4 Model of Computation (MoC)

A model of computation is a formal abstraction that provides rules to govern the execution, in this case, of workflows. Programming and workflow languages have traditionally been designed with an underlying abstraction whose formal semantics is given by a MoC [35]. Similarly, workflow engines instantiate a MoC when enacting workflows. While it is common for workflow systems not to reveal the MoC used by the engine, there are some systems in which the explicit selection of MoC is required[1].

Different MoCs have different strengths and weaknesses. Selecting a MoC often depends on, among other things, how well the model abstracts the problem at hand and how efficient the model can be implemented. A too abstract specification, for the model, is not only inefficient but is also unfeasible to implement while too much detail in the specification over-constraints the model making it inefficient and more costly to implement [35]. In essence, for a MoC to be efficacious there should be a balance between the generality offered by an abstract specification and the particularities of a detailed one. Such a MoC is useful not only for a range of scenarios but it is also possible to model and analyze interesting aspects of individual models.

Several MoCs have been used for the general workflow case, e.g., Petri nets, dataflow process networks, and UML activity diagrams. Some of these MoCs are not suitable for scientific workflows, e.g., even though BPEL is widely used for workflows in a business context, it is not as popular in the scientific community. This, despite several attempts at adapting BPEL for

[1] e.g., in Kepler [43] MoCs are exchangeable and are called *Directors*.

the scientific workflow peculiarities [22, 42]. Instead, dataflow approaches have predominantly been used for scientific workflows [43, 51, 64]. According to McPhillips et al. [48] this adoption is due to the inherent focus on data in the dataflow MoC, a characteristic that resembles the scientific process.

Still, a straight forward[2] adoption of dataflow for Grid workflows is not suitable. This is due to the characteristics of current, and in the foreseeable future, Grid execution environments, such as the typical lack of control over the internal states of workflow activities and the impossibility of continuously streaming tokens between activities. As presented in Section 3, a distinctiveness of Grid workflows is that they typically consist of a number of independent command line applications that are configured by environment variables or command line arguments. In this setting, activities are considered black boxes and it is impossible for the workflow MoC to control their internal states. For example, to the Grid workflow MoC, activities are considered to be executing once they are scheduled for execution on a Grid resource, even though they in practice may be stalled in a batch queue. Another distinctive characteristic of Grid workflow MoCs is that, as opposed to the continuous streaming of tokens found in dataflow networks (e.g., as in [43]), activities execute only once and communicate with other activities at the end of this execution. Thus, it is important for the Grid workflow MoC to support asynchronous communication and to carry out all communication only when activities finish executing, disabling those activities that finish executing.

Because of the previous restrictions, MoCs that have been typically used for Grid workflows are limited to Petri nets or some type of control flow MoC specified either by DAGs [11] or by specialized imperative languages [61, 67]. Below we present the manner in which Petri nets have been used as a MoC for Grid workflows. We also make observations on the reasons why a dataflow approach is not commonly employed in Grid workflows.

4.1 Petri nets

Petri nets are a mathematical and graphical tool specially useful for modeling and analyzing systems that are concurrent, asynchronous, and nondeterministic. Based on Murata [50], a Petri net is a bipartite graph in which nodes called *places* are linked to nodes called *transitions* by directed edges. There are no elements of the same node type connected to each other, e.g., places are only connected to transitions and not to other places. Places

[2]By straight forward we mean using the same MoC , without changes, as in local workflow systems.

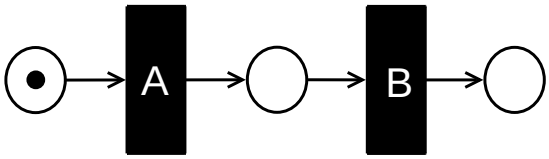

Figure 1: Petri net illustrating the control flow between activities A and B.

directed to transitions are called input places while places coming out of transitions are called output places. Places contain tokens that are required for enabling (initiation) the firing of transitions. Places also have an associated capacity that indicates the maximum amount of tokens they can hold. Edges have an associated weight that indicates how many tokens are required to enable a transition as well as how many tokes are produced when a transition is fired. A transition is fired only when each input place has the necessary tokens, specified by the edge weight, to enable that transition and if the output places have not yet reached full capacity. Once fired, an amount equal to the edge weight is set on each output place. The *marking* of the net describes its state and is given by the distribution of tokens in the places. The *initial marking* describes the state of the net before any transition has fired and a new marking is created after each firing of the net.

Petri nets have traditionally been used for representing the control flow of workflows [30, 33, 60, 65]. The manner in which this flow is represented is illustrated in Figure 1. In this network, two activities, A and B, are executed in sequence. Activity A is enabled (i.e., ready to fire) as there is a token in its input place (represented by the black dot). Conceptually, the firing of A symbolizes the execution of some activity in a Grid resource. When A completes execution a token is placed in the output place of A, which in this case is also the input place of B, thus enabling B. It is important to notice that B is not able to execute until A has finished. For this net the tokens not only symbolize the passing of control between activities, but they also maintain the state of the net. There is however no explicit information about the data created or consumed by the activities.

The limited support for combining control and data flow within the same model has been addressed by the introduction of specialized high-level nets, in particular *Coloured Petri nets (CPN)* [36]. In CPNs places have an associated data type (color set) and hold tokens that contain data values (colors) of that type. Arc expressions indicate the number and type of tokens that can flow through an arc. Tokens of the specific data types

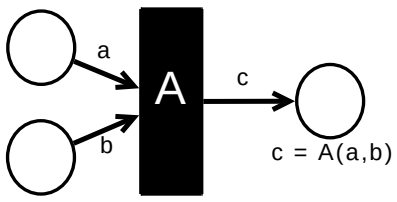

Figure 2: Representation of workflow activities using Coloured Petri nets. Based on the work in[34].

need to be present in its input places for a transition to fire. Transitions can also have an associated *guard*, a boolean expression that enables the firing of the transition whenever the guard evaluates to true. There is no ordering in how the tokens are consumed by a transition with respect to how they arrived to an input place. A queue can be associated with a place if ordering is desired. A more detailed discussion about this type of networks can be found in [36, 50]. Jensen [36] presents a more informal and practical introduction to CPNs whereas Murata [50] briefly touches upon the relationship between High-level nets, a group to which coloured nets belong to, and logic programs.

In the Grid workflow context Petri nets and CPNs have been used both as a graphical specification languages and as a workflow engine MoCs. Guan et al. [30] employ simple Petri nets as graphical language for defining workflows in the Grid-Flow system. Workflows defined with Petri nets are translated to the Grid-Flow Description Language (GFDL). Workflows in GFDL are then fed to the Grid-Flow engine. Language constructs[3] such as *OR-Split*, *AND-Split*, and *AND-Join* are used to generate instances of choice, loops, and parallel structures offered by GFDL. Hoheisel and Alt [34] employ CPNs both as specification language and as a MoC. In the latter case, transitions are used as processing elements (i.e., workflow activities) in which data tokens are distinguishable. Thus, transitions operate as functions whose parameters are obtained from the input places and the results are stored in the output places.

Figure 2 illustrates this process, where the result of applying the function in A to the parameters a and b is stored in the output place c. This Petri net models the data flow generated by the data files produced and consumed by the transition (representing a workflow activity) A.

While there are many characteristics that make Petri nets a sound choice

[3]In some settings called Workflow Patterns [66].

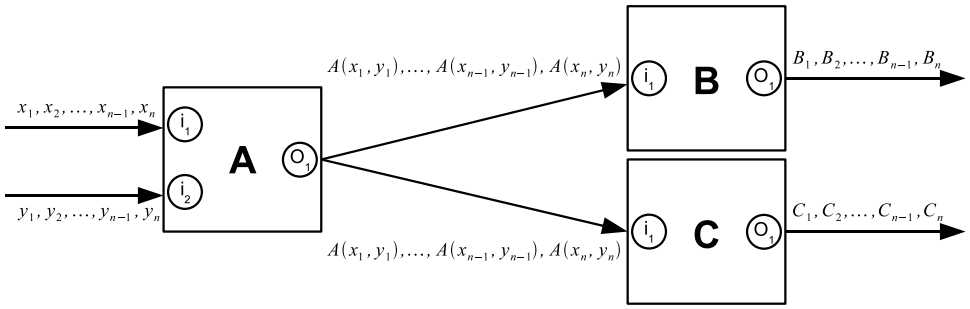

Figure 3: A dataflow network that instantiates concurrent execution of activities B and C.

for a Grid workflow MoC, there are some issues to be resolved. For example, Murata [50] identifies that Petri nets can become quite complex even for modest-size systems. A weakness of Petri nets when compared to a dataflow approach is the necessity to define parallelism explicitly e.g., using *AND-Split* and *AND-Join* [66].

4.2 Using dataflow networks on Grid workflows

Dataflow networks are the preferred MoC for local scientific workflows. For example, Triana [9], Kepler [43], and Taverna [52] offer capabilities that, one way or another, resemble the dataflow style of computation. In the original dataflow approach the focus was on fine-grained flows at the CPU instruction level. In those cases nodes represent instructions and edges represent data paths. When this metaphor is moved to the workflow paradigm, nodes no longer represent instructions but coarse-grained processing operations while edges represent dependencies between workflow activities.

Figure 3 illustrates a dataflow network in which activity A sends tokens concurrently to activities B and C. The figure presents a simplification of the actual process been carried out, nevertheless it helps us present the problems found when attempting to apply a dataflow approach to Grid workflows. The Figure shows a pipeline dataflow in which initial tokens are processed by A, and then B and C concurrently process the tokens generated by A. In the original dataflow process networks (e.g., as presented in [59]) tokens are continually streamed through the pipeline so that activities A, B, and C are all concurrently processing although operating on different tokens. The

circles inside the rectangles represent ports that serve as containers of data (see Section 3) and also serve as interfaces for establishing communication channels between workflow activities. In local workflows these ports have an associated data type that indicates the type of tokens that they can hold. These data types need not be restricted to simple types (e.g., integer, float, or string) as they can also be complex data structures [48]. On the contrary, in Grid workflows tokens only represent associations with data files and are otherwise untyped. Further discussions about type systems in workflows is presented in Section 5.1.

We identify the continuous streaming of tokens between activities and the lack of control of the workflow MoC over the internals of the activities as the main impediments for adopting dataflow style of computation in Grid workflows. Below we present a brief discussion on how these issues can be addressed.

Streaming of tokens between activities. In dataflow nets parallelism is achieved through concurrent processing of tokens by different activities. This e.g., can be seen in Figure 3 when A is processing token (x_i, y_i) while B and C are processing tokens produced by $A(x_{i-1}, y_{i-1})$, a previous execution of A.

In local workflows, this process is easily accomplished by e.g., interprocess communication or message passing. The nature of the Grid, however, impedes an easy solution if attempting to implement the same functionality on Grid resources. For Grid workflows, resources where activities A, B, and C are to be run must be *guaranteed* to start executing at the same time, a process known as *co-allocation* [21]. All resources must also be able to synchronize with each other to establish direct lines of communication between themselves.

The process of co-allocation of Grid resources is difficult for multiple reasons. At a technical level activity A must have access to the network addresses of the resources where B and C are running. However, this information is often not distributed outside the site in which B and C are running, making such a synchronization impossible. Another technical issue is that since many Grid applications operate on very large data files, transmitting only small bursts of data is not efficient. Furthermore, streaming tokens between activities requires adaptation of applications that normally communicate through files.

Several solutions for the problem of co-allocation of Grid resources have been proposed. These solutions usually depend on advance reservations

to ensure that all resources are available at the same time. However, the use of advance reservations introduces a problem at a managerial level, as reservations are known to degrade the performance of a system [45, 62, 63].

Lacking globally controlling coordinators. In local workflow systems the MoC has control over the internal processing of the activities. This means that any changes in the internal states of the activities are exposed to the MoC. For example, for the case illustrated in Figure 3, the MoC can recognize the state that activity A reaches after processing token (x_i, y_i).

This is not the case for Grid workflows. In this setting the MoC can only recognize that an activity is ready to execute, that an activity has been submitted to a Grid resource but for practical reasons can be considered to be executing, and that an activity has finished executing either successfully or with an error. All other changes in state are transparent to the MoC.

The nature of this obstacle is the use of command line applications that operate on *un-structured*[4] data files. However, the use of command line applications also simplifies the use of Grid resources by end-users as they are not required to modify their software. Thus, there is a trade-off between having simple coordinating MoCs in which applications can be easily included, and having more complex MoCs that require modifications to applications (even complete re-implementations) prior inclusion in the model.

While Petri nets have previously been used to model dataflow [40, 68], the use of CPNs facilitates this process. The CPN in Figure 2 can be adapted to model a processing unit from a dataflow network, i.e., the input and output places have similar functionality as input and output ports. A difficulty when modeling dataflow with CPNs is how to describe the implicit parallelism found in dataflow networks. A naïve approach produces conflicts among the concurrent activities. This can be seen in the CPN in Figure 4 that attempts to model the dataflow network of Figure 3. The conflict occurs after A fires and sets a single token in its output place. At this point, both B and C are enabled but only one can fire as there is only one token to consume.

Figures 5 and 6 illustrate two manners in which this problem can be resolved. The first approach is to add one output place for each transition that depends on the output place whereas the second approach adds a subnet. With the second approach there is a complete separation between the activities as opposed to the first approach in which the output and input

[4] As opposed to structured data files such as XML documents defined through XML schema or basic data types such as integer or string.

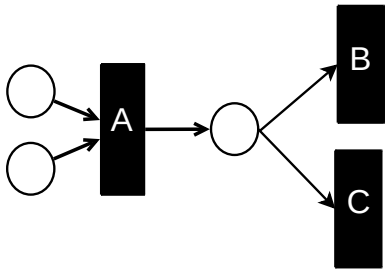

Figure 4: CPN that simulates the functionality of Figure 3. In this CPN activities B and C are in conflict.

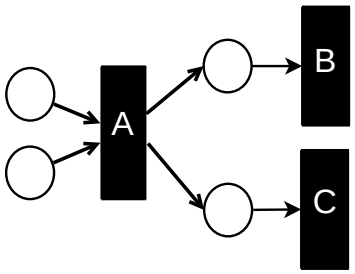

Figure 5: A CPN that requires one output place for each dependent activity. In this case one for each B and C.

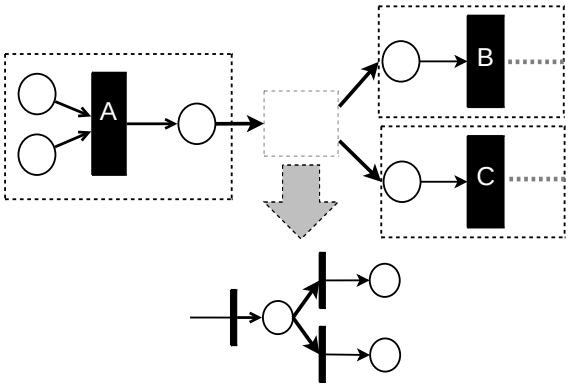

Figure 6: A CPN that uses a sub-net to connect dependent activities. The output place of A is not the same as the input places of B and C, thus each can have an associated buffer.

places are shared between the activities. This second approach provides a more accurate representation of dataflow processing units.

A MoC inspired by dataflow networks but adapted to Grid environments is presented in our previous work [20]. In this approach, activities execute once and are then disabled from further execution. Input and output ports are used to establish dependencies between activities. The only data type used is string and data values represent Grid storage locations where the data files are to be found. Tokens carry this information from one activity to the other. Files are transferred to the resources where the activities are to execute from the locations where previous activities stored their outputs.

While this discussion has focused on Petri nets and dataflow, it is important to mention that by far the most common approach for controlling dependencies in Grid workflows is through DAGs. Currently, there are several projects [11, 15, 38] that offer higher level interfaces for specifying workflows that are rewritten in order to reduce execution time. The output of these rewrites is often produced as DAGs. In these cases the DAGs represent schedules for execution of workflow activities on Grid resources.

5 Workflow Representation

In this section we address several factors relevant to workflow languages. We begin by exploring the use of *type systems* in workflow languages (Section 5.1). We explore the differences in capabilities offered by type systems depending on whether the language is for local or Grid workflows. As in the MoC case, the nature of these differences arises from the differences present in Grid environments.

A common differentiation in workflow languages is that between control-driven (control flow) and data-driven (data flow) styles for representing workflows. In control-driven workflows the complete order of execution is specified explicitly whereas in data-driven workflows only the data dependencies between activities are given and the execution order of the activities is inferred from the manner in which the data dependencies are satisfied. Thus the data-driven style specifies a partial order for the workflow activities and the exact order of execution is not known until run time. The choice of style is mostly driven by the selected MoC. However, several languages support mix-models, based on one MoC but with basic support for the other. In Section 5.2 we illustrate mechanisms to translate control-driven workflows into data-driven ones and vice-versa.

All modern general-purpose programming languages are Turing com-

plete, with the caveat that their run time environments as provided by today's computer hardware have a finite memory size as opposed to the infinite tape length of the universal Turing machine. Being Turing complete implies that a system is computationally equivalent to other Turing complete systems. Within the workflow community there are arguments for and against having Turing complete workflow representations. For example, according to Deelman [13], "one have to be careful not to take workflow languages to the extreme and turn them into full-featured programming or scripting languages". On the other hand Bahsi et al. [5] argue that workflows without conditions and iterations are not sufficient for describing complex scientific applications. Nevertheless, when examining current workflow systems [5, 14, 56] it becomes apparent that most systems are Turing complete[5]. In light of this finding, we discuss in Section 5.3 the manner in which the prerequisites for Turing completeness, namely *state management*, *conditions*, and *iterations* are implemented by different workflow languages.

5.1 Workflow languages and type systems

One aspect to consider about workflow languages is the choice of type system offered by the language. Which data types are present, whether data types must be explicitly specified or if implicit specification is supported, and whether the language must provide mechanisms for describing new types are all issues that vary among languages, specially between languages for local and Grid workflows.

For local workflows, it is relatively easy to wrap applications in an embedding model that ensures compliance with the type system. For this case activities are often developed from scratch, using APIs provided by the workflow system and are thus completely capable for operating within the framework provided by the workflow system. This is the case in e.g., Triana [9] and Kepler [43]. However, due to the use of command line applications, applying this functionality to Grid workflows is not simple. Data type information is not required for command line applications and thus it is not included in any one of the several job description languages (e.g., JSDL [2]) currently in use. This is the case also for the languages in e.g., DAGMan [11] and GWEE [20]. Nevertheless, there are some Grid languages that use different methods for including type information in the workflows, this is the case in e.g., BPEL [37], ICENI [47], and Karajan [67].

[5] As a side note, non-Turing complete languages have many uses [7], although they are typically used in specialized areas.

Exposing Grid applications through type interfaces, e.g., Web services or component models such as CCA [3] or GCM [10], can be a substantial effort as it requires e.g., software installations on remote machines [16, 46]. A further complicating factor when adding a type system to Grid workflow languages is that such languages most often lack support for defining custom data types. It is thus hard to express the structure of data files that are used by a given application. An exception is the Virtual Data Language (VDL) [69] that provides primitives for describing data sets, associating data sets with physical representations, and defining operations over the data sets. Furthermore, whereas it is relatively simple to verify that input values adhere to certain basic types (e.g., integer or float) it is more complex to verify that a data file is of a specific format or follows a predetermined structure.

In summary, using type information in Grid systems simplifies workflow design and error handling, but it also adds overhead as each application that is used must be exposed through a type interface. On the other hand, not using a type system (e.g., supporting a single data type) increases flexibility as any (command line) application can be embedded in a workflow, but this at the expense of higher difficulty in detecting data incompatibilities.

5.2 Control-driven and data-driven workflows

Control-driven and data-driven MoCs differ in the semantics of consuming dependencies between activities. In control-driven workflows, consuming a dependency results in the transfer of execution control from the preceding activity to the succeeding one, whereas in data-driven a data token is sent from the first activity to the next one and an activity is only able to execute after all data dependencies are cleared (i.e., all tokens are received). The choice between styles depends, in part, on the applications to be described, e.g., some areas such as image or signal processing have traditionally been represented with data-driven workflows as this model provides a natural representation of the problems studied within these fields [54, 64].

Recent results advocate for a simple hybrid model based on a data-driven style but extended with limited control constructs [14]. The validity of this model is attested by support of both styles of flows by several contemporary systems [15, 38, 43, 55, 61]. In part, such a hybrid model can be realized because, despite their differences in operation, converting from one MoC to the other is not a complicated process. However, it is possible that performing such conversions, control to data and data to control, requires simulating missing functionality with the primitives available in either style.

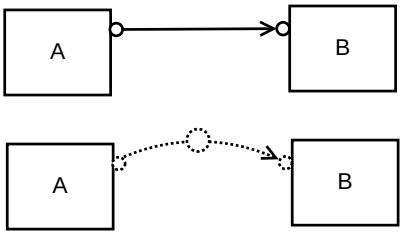

Figure 7: A control-driven dependency (top) denoted by solid lines and the corresponding data-dependency (bottom) illustrated by dashed lines. In both cases activity B executes after activity A.

This results in more complex workflows and increases the risk of introducing errors. Nevertheless, control- and data-driven workflows are interoperable at the workflow language level. Below we present a manner in which such conversions can be attained for the case of Grid workflows.

5.2.1 Conversions between control-driven and data-driven workflows

As presented in Section 3, communication between activities in Grid workflows is performed by the transfer of untyped data files. For this case control dependencies can be converted to data dependencies by using tokens that carry no data values (i.e., *dummy data tokens*) and in practice only represent the transfer of control from one activity to another. This case is illustrated in Figure 7 where equivalent control- and data-driven versions of a control dependency between activities A and B are presented. The circle in the bottom workflow being the dummy data token introduced to simulate the transfer of control from A to B. The case with actual data tokens is presented in Figure 8. In this case a file transfer (Activity B) that is represented explicitly in the top workflow is converted to an implicit transfer embedded in a data dependency as shown in the bottom workflow.

The reverse process can be applied for converting from data- to control-driven workflows. Data dependencies are converted by inserting an intermediate activity that performs an explicit file transfer from the location where the source activity was executed to the machine where the target activity is to be executed. The top workflow in Figure 8 illustrates the result of this process, converting from the bottom workflow in the figure. Notably, it is possible to eliminate the explicit file transfer (B) if such transfer is per-

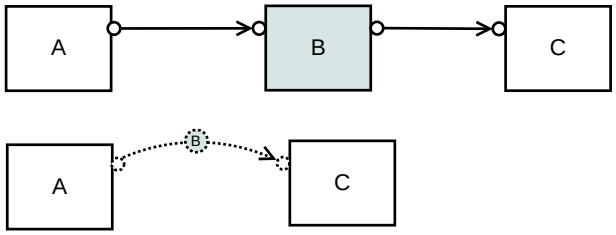

Figure 8: File transfers are workflow activities in control-driven workflows (Activity B in the top workflow) and data dependencies in data-driven workflows (data dependency with token labeled B in the bottom workflow).

formed as part of the job execution, a mechanism supported by most Grid middlewares (e.g., Globus [25]). For abstract workflows[6], as the resources where activities are executed are not know until enactment, the conversion must be performed after executing the source activity as the actual location of the data files is not known before. Otherwise the workflow must specify the resources where all activities are to be executed (i.e., it must be a concrete workflow).

An important step when converting from data- to control-driven is to perform a *topological sorting* of the data dependencies. Consequently, the resultant control-driven workflow specifies only one among many possible execution orders. As a result the precise execution order may differ between the data and the derived control-driven versions. This is however of no practical concern, as the respective execution order of all dependent activities is maintained.

In practice these conversions have been performed a number of times. For example, Mair et al. [44] describe how to convert both styles of workflows, control- and data-driven, to an intermediate representation based on DAGs. A concrete implementation between the Karajan language (control-driven) and the internal representation of GWEE (data-driven) is presented in our earlier work [20].

5.3 Essential language constructs

Turing completeness, the ability to simulate a universal Turing machine, is important when analyzing the computing capabilities of systems. The ability

[6]In abstract workflows only the structure of the workflow is specified. The physical resources where the activities are to be executed are not specified.

to manage state, e.g., by been able to define, update, and read variables, is one criteria for Turing completeness. In addition to *state handling*, *condition* (branching) and *repetition* (typically recursion or iteration) functionalities are also required mechanisms for Turing completeness. Here, we look at workflow languages in light of these mechanisms. Motivating use cases are presented for each mechanism as well as the manner in which the mechanisms are implemented by different workflow languages. We also take a look at when and how *Collections* are useful.

5.3.1 Workflow state management

In modern imperative and object-oriented programming languages state is managed by defining and updating variables that represent an abstraction of memory locations. Some workflow languages, such as Karajan [67] and BPEL [37], support state management through a variable construct similar to that of modern programming languages. Other workflow systems, e.g., DAGMan [11] and Taverna [52], have no built-in language mechanism to manage state. The only state in those systems is the run time state of the workflow activities (e.g., completed, running, waiting). Not having variables creates difficulties when using general condition and iteration constructs as these make branching decisions mostly based on state. A different approach for implementing a state-like mechanism is to use system parameters for defining properties that hold similar functionality as environment variables. This mechanism is used by ASKALON [61] and JOpera [55].

5.3.2 Conditions

By far the most common use for conditions in workflows is for *workflow steering*, a functionality that carries similar semantics as the well known *if* and *switch* constructs. The idea is that the flow is *dynamic* and the output is *non-deterministic*. The most frequent use case for workflow steering is classical flow of control where the branch to enact is decided at run time based on the outcome of previously executed activities. Changing the enactment of a workflow in reaction to external events is an alternative steering use case suggested by Gil et al. [27].

Another use case for conditions is *iterative refinement* scenarios, where some activity needs to be repeatedly executed until a condition is met. In addition to conditions, iterative refinement scenarios requires iteration constructs and a testing mechanism, both issues discussed in more detail later in this section.

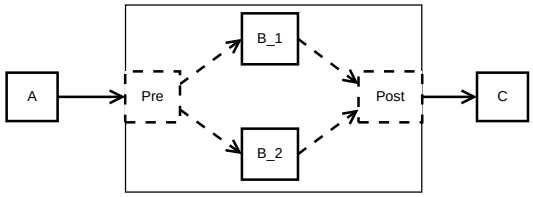

Figure 9: The black box approach where conditions are hidden in a special activity type. In this case, the condition is transparent to the rest of the workflow.

A final use case in which conditions have been employed is *fault tolerance*. A survey of fault tolerance mechanisms used in various Grid workflow systems is found in Plankensteiner et al. [57]. In this use case conditions are used for defining alternative actions for situations in which a workflow activity fails. In contrast with programming languages that typically have special language constructs for catching generated runtime exceptions, in distributed Grid workflow environments, errors in remotely executing activities do not generate such exceptions but rather result in *failed activities*. Conditional statements are typically sufficient for many cases of fault tolerance.

There are basically two abstractions for implementing conditions in a DAG or dataflow based workflow representation. The first one considers conditions as a special type of workflow activity. In this approach, illustrated in Figure 9, the condition is viewed as a black box with the branches hidden inside the activity. This form of condition gives rise to so-called "*structured workflows*" [41] which are analogous to conditions and iterations found in structured programming. This type of constructs have only one entry (pre) and one exit (post) point into and out of the workflow. An example of this approach is Triana [9]. The other alternative is to have the condition as an activity that selects a branch of execution but with all possible branches exposed in the main workflow. In this case, care must be taken as deadlock may arise if the branches are not well synchronized. This second approach is how conditions are implemented in JOpera [55] and Karajan [67].

After a branch is selected, data must be sent to the initial activity of the branch in order to trigger execution. However, unless special care is taken, not-selected branches may end up in a dead-lock state, waiting for input forever. One solution to this undesired effect is to prune the workflow graph by removing not-selected subgraphs from the workflow. Such a solution is

akin to the elimination of dead code, a well-studied problem in compiler theory [1]. The introduction of conditions can also introduce problems with synchronization with previous branches of the workflow. More specifically, combination of primitives such as OR-Split and AND-Join in BPEL [37] may result in dead-locks unless care is taken.

Detecting workflow termination becomes more complicated when the workflow contains branches that do not execute. With conditions implemented using the black-box approach, this problem can be solved by marking conditional activities as completed once one of its branches finish executing. A different approach is to mark branches along the non-selected paths with a terminal state that indicates that they are not to run. It is also possible to tag certain activities (such as activity C in Figure 9) as terminal ones. Once such an activity completes, the workflow enactment engine is assured that the workflow has finished executing.

In a typical, non-typed Grid workflow, condition evaluation (often referred to as testing) is hard to achieve as the workflow system has no control over the evaluation of the condition. Generality in the testing capabilities is also difficult to achieve unless the system limits what can be tested. As activities in Grid workflows usually communicate via files instead of typed variables, ordinary boolean testing is tedious in a Grid environment. To complicate things further, it is typically hard to distinguish between errors in the application execution and faults related to the Grid infrastructure. One possible solution is to offer a subset of predefined testing capabilities,as it is done in e.g., UNICORE [5, 17]. In this work, three sets of tests are defined: (1) ReturnCodeTest, indicating successful or failed task execution; (2) FileTest, for checking whether files exist, are readable, writable, etc.; and (3) TimeTest, that tests if a certain time has elapsed. These evaluation capabilities are based on information similar to what is known about process execution in shell scripting languages. An alternative method is to implement testing by an external agent that has the domain-specific knowledge required to perform comparisons [14].

5.3.3 Iterations

We distinguish between three types of iterations:

1. *Counting loops without dependencies* between iterations are often referred to as parameter sweeps or horizontal parallel iterations. This type of iteration generates parallel independent branches and is akin to applying a function to each element from a set. In many program-

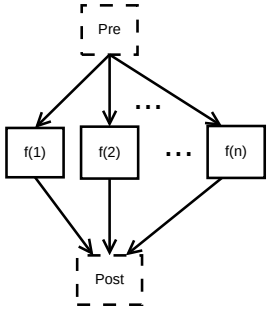

Figure 10: A counting loop without dependencies expressed as a parameter sweep.

ming and workflow languages, such loops are often expressed using language constructs such as *parallel-for* or *for-each*.

2. *Counting loops with dependencies* between iterations where the results from one iteration is used in the next one. This type of loop can hence not be independently executed in parallel. Typical syntax for these iterations is *do-n* and *for-n*.

3. *Conditional loops* are also referred to as non-counting iterations, temporally dependent iterations, or sequential iterations. This type of loop stops only when a certain condition is met. *While*, and *do-while* are used to express conditional loops in most programming languages.

Algorithm 1 Counting loop without dependencies

```
1: for I ← 1 ... N do
2:     f(I);
```

As demonstrated by Ludäscher et al. [43] counting loops without dependencies can be expressed using the *map* function from functional programming, that is, $f(x_1, x_2, \ldots, x_n) \Rightarrow (f(x_1), f(x_2), \ldots, f(x_n))$. Algorithm 1 illustrates a typical loop of this type and Figure 10 illustrates the equivalent workflow construct after applying the map function. This type of loop give rise to a high degree of concurrency as the threads of execution are completely independent. The same concurrency is impossible to achieve for counting loops with dependencies. The reason is that an iteration depends on results from a previous iteration. However, as illustrated in Algorithm 2

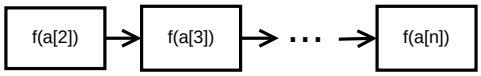

Figure 11: A counting loop with dependencies between iterations rolled out.

and Figure 11, this type of iteration can be rolled out [1, 12] and equivalent functionality can be provided without using iterations. Contrary to the two types of counting loops, conditional loops cannot be expressed by rewriting the workflow graph. Furthermore, this type of loop requires support for conditions to test when the terminal condition is met. We remark that it is trivial to rewrite a counting iteration as a conditional loop, whereas the opposite is not possible in the general case.

The mechanisms to support iteration constructs by workflow languages are similar to the ones used for conditions. In the first case, the black box approach, the iteration is a special workflow activity with one entry and one exit point. This is the approach taken by the extensible actor construct in Kepler [43]. The second alternative is to have an expression-like construct that allows the flow of control to iterate over selected activities in the workflow. This second approach introduces cycles to the workflow graph and creates a more complex enactment since care must be taken to avoid infinite loops.

Iterations, essentially being a flow of control construct, are easy to support by control-driven languages, whereas the semantics of iterations are unclear for pure data-driven languages. Mosconi et al. [49] investigates the minimal set of control flow constructs required to support iterations in a pure dataflow language and surveys existing implementations of iterations for visual programming languages.

Algorithm 2 Counting loop with dependencies

1: $a[1] \leftarrow initial\_value$;
2: **for** I $\leftarrow 2 \ldots N$ **do**
3: $\quad a[I] \leftarrow f(a[I-1])$;

5.3.4 Collections

Some programming languages, e.g., LISP and Perl, have built-in *for-each* operations that treat a collection of elements as a single entity. Similar ideas are used in vectorizing and parallelizing compilers [4, 32]. These type

of data-collection operations are also supported by some workflow systems, e.g., ASKALON [58] and the COMAD [48] implementation in Kepler. Data-collection mechanisms are data-centric and hence can simplify the use of dataflow style languages. However, collections do not bring additional functionality beyond what is offered by parameter sweeps or iteration constructs except the aforementioned simplification.

6 Discussion and Concluding Remarks

Here we discuss the topics covered in the previous sections with a comprehensive outlook. As such, while topics are ordered as they are introduced in those sections (Sections 2–5), there are cases in which the topics overlap section crossings.

6.1 Model of computation

The model of computation is the central concept of a workflow engine. It can even be said that the workflow engine is merely an implementation of a MoC. Previous results [13, 48] suggest that a dataflow approach suits the scientific process best. This is supported by the number of solutions that use the dataflow MoC, and the manner that these solutions can be trivially adapted to operate in disparate scientific fields. Common to these solutions is the use of local machines as execution environment, which appears natural as the environment that local machines offer is well adapted for the dataflow MoC.

This is not the case for Grid workflows. Limitations such as the lack of control over activities, lack of support for streaming tokens between activities, and the unavoidable requirement of executing activities in a batch processing fashion, make the use of a pure dataflow MoC unfeasible to achieve. Control-driven approaches appear to be better adapted for this type of environment. Nevertheless, there are several projects that attempt to use dataflow style of coordination for Grid workflows.

One way to achieve functionality similar to what the dataflow MoC offers is to use higher level representations that are later refined to concrete activity specifications. This is the case presented in, e.g., [8], [15], and [26]. In [15] and [26] a concrete workflow DAG with the correct order of execution for the activities is generated from more abstract representations, whereas in [8] a data-driven representation is concretized into dataflow-like workflows that are enacted on the Grid [20]. As presented in Section 4.2, CPNs can

also be used for representing dataflow style coordination but care must be taken to avoid (firing) conflicts.

Interoperability between MoCs can be achieved under certain circumstances. The manner in which the workflow activities are implemented, in particular the MoC underlying their design, is the key aspect that enables the activity to operate under different MoCs. On the local side Goderis et al. [29] provide insight on which combinations of MoCs are valid and useful. This work offers a hierarchical approach in which MoCs, called *directors*, require certain properties from the activities, called *actors*, that they coordinate. Directors also export properties to the actors in which they are included. The set of properties offered and required establishes a contract and depending on how well the contract is respected it assures the compatibility of actors and directors, and thus the potential compatibility among different MoCs. Actors that completely adhere to the contract are called *domain polymorphic* [18] and can be used by any director. Thus, when seeking interoperability, it is important to develop the workflow activities in a manner in which they can be controlled by different MoCs. However, this is not always possible.

The core of the difference between local and Grid workflows is the execution environment. Interoperability between local and Grid MoCs is thus possible only in a few cases and directly depends on the manner in which the activities are executed on the Grid. Nevertheless, in practice this type of interoperability is not often requested, instead, what is commonly expected is for local workflow systems to be able to submit work to the Grid on either a per activity or per sub-workflow basis. The latter case with aid from a Grid workflow system.

6.2 Language issues

Section 5.1 discusses the issue of type systems and workflow languages. As presented there, it is common for local workflows to support type information while it is much less common for Grid workflows to support this functionality. In the reminder of this section we revisit the various language constructs introduced in Section 5 and investigate the extent to which the respective mechanisms are required. For the various constructs, we look at typical use cases and discuss whether equivalent functionality can be achieved by different means or if the use case motivates the particular language construct used.

6.2.1 State management

Variables have traditionally been used to manage state in programming languages. This is also the case for the workflow language of Karajan [67]. However, lack of a variable construct need not imply that a language is not Turing complete. For example, Glatard et al. demonstrate how to implement a Turing machine in the Taverna Scufl language [28]. In this work, the limitation of not being able to define variables (and hence manage state) in Scufl is circumvented by performing state management inside one of the workflow activities (implemented in Java). For Petri nets it is also possible to handle state. For this case the state is given by the marking of tokens in the places of the net.

6.2.2 Conditions and iterations

It appears that in order to describe and execute anything but the most trivial process, workflow steering, and hence conditions, are required. However, the survey by Bahsi et al. [5] shows that not all workflow systems support conditions as part of their workflow language. Equivalent functionality can be achieved by other mechanisms, as illustrated e.g., by the pre and post scripts that are used to steer the path of execution in DAGMan [11]. This suggests that although conditions are required for workflow steering they need not necessarily be part of the workflow language as they can be expressed using alternative mechanisms. Instead, at least for Grid workflows, a mechanism to implement the testing required for conditions is more important.

The iterative refinement use case can be implemented in two ways. One alternative is a fine-grained workflow that iterates over individual activities until some condition is satisfied. In addition to handling conditions and testing, this approach requires the workflow language to expose a loop mechanism. Alternatively, it is possible to have a coarse-grained workflow in which the activities as well as the testing mechanism are all abstracted and hidden inside a single workflow activity. In this latter approach, which is taken e.g., by Kepler [43], conditions are not necessary for specifying the workflow. The support for the iterative refinement use case is hence a trade-off between (potentially too large) granularity, and thus possible limited parallelism, and added complexity of the workflow language.

There are two types of failures occurring frequently in workflows systems. In the first type, infrastructure problems such as network failures, power outages, temporarily unavailable storages or databases, insufficient disk space, incorrect hardware, etc. cause an activity or a file transfer to

fail. For this case a lower level tool would ideally ensures fault tolerance, e.g., by restarting interrupted file transfers, resubmitting failed jobs to alternative machines, etc. These types of mechanisms to recover from infrastructure problems are known as *implicit fault management* [33]. Such a recovering infrastructure removes the need for the user to manually, through conditions, encode alternative execution paths for the workflow to follow upon failures. In contrast the manual alternative quickly becomes unfeasible due to the large number of potential error sources. In the second type of failure the workflow enactment fails due to errors in the workflow itself. These errors can be faulty descriptions of activities, incompatible messages exchanged between activities, unintended deadlocks in the workflow graph (e.g., circular data dependencies), etc. For this case conditions are of limited use as the errors in the workflow are detected only during enactment whereas conditions must be added at design time. Manual inspection and modification of the workflow is typically required to solve this type of problems.

Similar to the case of conditions, a repetition mechanism is often required to express complex workflows. The mechanism need not be an iteration construct in the language, as it is commonly known that recursion offers the same functionality. The latter approach is taken by e.g., Condor DAGMan [11] and JOpera [55]. Another example of a repetition with no explicit construct is to use parameter sweeps for implementing loops without dependencies. For this case, a mechanism for distributing data, e.g., data collections, to the different threads of execution (each operating on a different iteration from the loop) often facilitates this process. As far as an iteration mechanism is required, there is always the possibility of using a single application that hides the iterative structure of the workflow. However, care must be taken in order not to limit potential concurrency and thus reduce the performance of the workflow.

6.3 Execution environment interoperability

It is difficult to address the issue of execution interoperability for local workflows as workflow activities are developed specifically for a particular system. These activities are dependent on libraries that offer a common execution and communication environment for operating within a particular system. It is not easy to decouple the functionality of the activity from the operational framework. Instead, in many cases, the targeted functionality must be re-implemented if it is required by other systems.

Web services are sometimes presented as the silver bullet of interoperability for distributed computing use cases. PGrade [39] and Taverna [52]

are well known examples of service-based solutions. Yet, using Web services only partially solves the interoperability problem, namely how to in a protocol and programming language independent manner invoke a capability (an operation) offered by a remote entity (a service). Issues related to the coordination of these activities, i.e., to the workflows, including workflow MoC and workflow language are not addressed. Standardization efforts for web service coordination languages, e.g., BPEL [37], have been found unsuitable for scientific workflows [6].

In the Grid, execution level interoperability often means being able to execute activities in resources that use different Grid middlewares. This type of interoperability is a well studied problem with several solutions. For example, in previous work we have show how this can be achieved by decoupling the submission of activities from the control of dependencies in a Grid workflow engine [20]. Then, by using a chain-of-responsibility pattern the correct middleware for executing each activity is selected at run time. A similar solution but at the activity level is presented in [21]. A more specialized solution that also operates with different middlewares and can work with groups of activities while offering fault tolerance is provided by our Grid Job Management Framework (GJMF) [19]. Similar approaches to workflow execution interoperability are proposed by P-Grade [39].

6.4 Granularity concerns

As we have seen from the discussions of conditions and iterations, the granularity of workflow activities affects the performance as well as the complexity of workflows. Too fine granularity can limit the performance due to a higher overhead in Grid interactions. Conversely, having too coarse-grained activities can also reduce workflow performance, in this case due to a reduction in concurrency as the problem can no longer be partitioned into smaller chunks that can operate independently without synchronization. Other issues in which granularity is of concern include the Grid interaction style and whether sub-workflows or individual activities are the basic means of submitting work to the Grid. In general, there is a trade-off between granularity on one hand and complexity and performance on the other. Nevertheless, varying the granularity of activities can be beneficial when striving for interoperability between systems.

6.5 Concluding remarks

In this work we give a comprehensive presentation of the different problems that directly affect interoperability among scientific workflows. Part of our results is the introduction of three dimensions for addressing interoperability issues. The degree of coupling between these dimensions (MoC, language, and execution environment) has interesting consequences. For example, an important lesson learned in our work with a middleware independent Grid workflow engine [20] is that a complete decoupling between execution environment (i.e., Grid middleware and job description language) and workflow language improves portability and interoperability of the engine, but also makes workflow design more tedious and error prone, as the workflow activities, viewed as black boxes by the enactment engine, are completely untyped. There hence exists a trade-off between usability and interoperability.

A similar trade-off also exists between execution environments and MoCs. For example, in essence, the goals of local and Grid workflow MoCs differ significantly. For local workflows, users are better able to express their solutions using MoCs closer to the problem space, as illustrated by the many different dataflow style solutions found for local workflows. However, in these solutions activities are tightly coupled to a particular workflow system and it is not easy to reuse those activities in a different one. On the other hand, in Grid workflows it is simple to provide interoperability at the middleware level. Yet it is harder to specify Grid workflows as, e.g., conditions and iterations are not always available. From our previous discussion we can argue that in local workflows it is better to interoperate at the *workflow level* whereas in Grid workflows is preferred to do so at the *activity level*. Furthermore, from the execution environment dimension, our findings support the use of hierarchical approaches that consider sub-workflows (and all activities) as black boxes.

At the workflow language level a more important trade-off is that between usability and complexity on one hand and potential concurrency (and thus performance) on the other. This trade-off appears in any decision for varying the granularity of activities. The most illustrative case is the different ways in which conditions and iterations are implemented by different systems. While it is possible to translate between languages, differences in implementation details may lead to a tedious processes, performed in ad-hoc ways, and not prone to automation.

Acknowledgments

We are grateful to Frank Drewes and Johanna Högberg for valuable feedback on theoretical aspects of computation. We are also grateful to P-O Östberg for fruitful discussions on general aspects of workflow systems and on workflow language constructs. We thank Ken Klingenstein and Dennis Gannon, organizers of the 2007 NSF/Mellon Workshop on Scientific and Scholarly Workflow, as well as the participants of this important meeting that gave us the opportunity for discussing relevant aspects of interoperability. This research was conducted using the resources of the High Performance Computing Center North (HPC2N). Financial support has been provided by The Swedish Research Council (VR) under contract 621-2005-3667.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[2] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, A. S. McGough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) specification, version 1.0. http://www.ogf.org/documents/GFD.136.pdf, February 2009.

[3] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *The Eighth International Symposium on High Performance Distributed Computing*, pages 115–124, 1999.

[4] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[5] E.M. Bahsi, E. Ceyhan, and T. Kosar. Conditional workflow management: A survey and analysis. *Scientific Programming*, 15(4):283–297, 2007.

[6] R. Barga and D. Gannon. Scientific versus business workflows. In I. Taylor et al., editors, *Workflows for e-Science*, pages 9–18. Springer-Verlag, 2007.

[7] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[8] A-C Berglund, E. Elmroth, F. Hernández, B. Sandman, and J. Tordsson. Combining local and Grid resources in scientific workflows (for Bioinformatics). In *9th International Workshop, PARA 2008 (accepted)*. Lecture Notes in Computer Science, Springer-Verlag, 2009.

[9] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming scientific and distributed workflow with Triana services. *Concurrency Computat.: Pract. Exper.*, 18(10):1021–1037, 2006.

[10] CoreGRID. Deliverable D.PM.04 basic features of the grid component model. Beta working paper series, wp 47, CoreGRID - Network of Excellence, 2007. Available at: www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf.

[11] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. Workflow management in Condor. In I. Taylor et al., editors, *Workflows for e-Science*, pages 357–375. Springer-Verlag, 2007.

[12] W.R. Cowell and C.P. Thompson. Transforming FORTRAN DO loops to improve performance on vector architectures. *ACM Transactions on Mathematical Software (TOMS)*, 12(4):324–353, 1986.

[13] E. Deelman. Looking into the future of workflows: the challenges ahead. In I. Taylor et al., editors, *Workflows for e-Science*, pages 475–481. Springer-Verlag, 2007.

[14] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, 2009.

[15] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, and D.S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[16] T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, and P. Kacsuk. GEMLCA: Running legacy code applications as grid services. *J. Grid Computing*, 3(1–2):75–90, 2005.

[17] D. Erwin (editor). UNICORE plus final report. www.unicore.eu/documentation/files/erwin-2003-UPF.pdf, visited December 2008.

[18] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[19] E. Elmroth, P. Gardfjäll, A. Norberg, J. Tordsson, and P-O. Östberg. Designing general, composable, and middleware-independent Grid infrastructure tools for multi-tiered job management. In T. Priol and M. Vaneschi, editors, *Towards Next Generation Grids*, pages 175–184. Springer-Verlag, 2007.

[20] E. Elmroth, F. Hernández, and J. Tordsson. A light-weight Grid workflow execution engine enabling client and middleware independence. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics. 7th Int. Conference, PPAM 2007*, pages 259–270. Lecture notes in Computer Science 4967, Springer-Verlag, 2008.

[21] E. Elmroth and J. Tordsson. A standards-based Grid resource brokering service supporting advance reservations, coallocation and cross-Grid interoperability. *Concurrency Computat.: Pract. Exper. (accepted)*, 2009.

[22] W. Emmerich, B. Butchart, and L. Chen. Grid service orchestration using the business process execution language (BPEL). *J. Grid Computing*, 3(3–4):238–304, 2005.

[23] S.D.I. Fernando, D.A. Creager, and A.C. Simpson. Towards build-time interoperability of workflow definition languages. In V. Negru et al., editors, *SYNASC 2007, 9th international symposium on symbolic and numberic algorithms for scientific computing*, pages 525–532, 2007.

[24] International Organization for Standardization. ISO/IEC 2382-1 information technology - vocabulary - part 1: Fundamental terms, 1993.

[25] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In H. Jin et al., editors, *IFIP International Conference on Network and Parallel Computing*, pages 2–13. Lecture notes in Computer Science 3779, Springer-Verlag, 2005.

[26] Y. Gil. Workflow composition: semantic representations for flexible automation. In I. Taylor et al., editors, *Workflows for e-Science*, pages 244–257. Springer-Verlag, 2007.

[27] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *IEEE Computer*, 40(12):24–31, 2007.

[28] T. Glatard and J. Montagnat. Implementation of Turing machines with the Scufl data-flow language. In *Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 663–668. IEEE, 2008.

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

[30] Z. Guan, F. Hernández, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy, and Y. Liu. Grid-Flow: a Grid-enabled scientific workflow system with a petri-net-based interface. *Concurrency Computat.: Pract. Exper.*, 18(10):1115–1140, 2006.

[31] F. Hernández, P. Bangalore, J. Gray, Z. Guan, and K. Reilly. GAUGE: Grid Automation and Generative Environment. *Concurrency Computat.: Pract. Exper.*, 18(10):1293–1316, 2006.

[32] W.D. Hillis and G.L Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[33] A. Hoheisel. User tools and languages for graph-based Grid workflows. *Concurrency Computat. Pract. Exper.*, 18(10):1001–1013, 2006.

[34] A. Hoheisel and M. Alt. Petri nets. In I. Taylor et al., editors, *Workflows for e-Science*, pages 190–207. Springer-Verlag, 2007.

[35] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEEE Proc.-Comput. Digit. Tech.*, 152(2):114–129, March 2005.

[36] K. Jensen. An introduction to the practical use of coloured Petri nets. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets II: Applications*, pages 237–292. Lecture Notes in Computer Science 1492, Springer-Verlag, 1998.

[37] D. Jordan and J. Evdemon (chairs). Web Services Business Process Execution Language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, September 2008.

[38] P. Kacsuk, G. Dozsa, J. Kovcs, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombas. P-GRADE: a grid programming environment. *J. Grid Computing*, 1(2):171–197, 2003.

[39] P. Kacsuk and G. Sipos. Multi-grid and multi-user workflows in the P-GRADE Grid portal. *J. Grid Computing*, 3(3-4):221–238, 2006.

[40] K.M. Kavi, B.P. Buckles, and U.N. Bhat. Isomorphisms between petri nets and dataflow graphs. *IEEE Trans. on Software Engineering*, 13(10):1127–1134, 1987.

[41] B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. In B. Wangler and L. Bergman, editors, *Advanced Information Systems Engineering, Proceedings of the 12th International Conference, CAiSE 2000*, pages 431–445. Lecture Notes in Computer Science 1789, Springer-Verlag, 2000.

[42] F. Leymann. Choreography for the grid: towards fitting bpel to the resource framework. *Concurrency Computat.: Pract. Exper.*, 18(10):1201–1217, 2006.

[43] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Leen, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency Computat.: Pract. Exper.*, 18(10):1039–1065, 2006.

[44] Michael Mair, Jun Qin, Marek Wieczorek, and Thomas Fahringer. Workflow conversion and processing in the ASKALON grid environment. In *2nd Austrian Grid Symposium*, pages 67–80. Österreichische Computer Gesellschaft, 2006.

[45] M.W. Margo, K. Yoshimoto, P. Kovatch, and P. Andrews. Impact of reservations on production job scheduling. In E. Frachtenberg and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 116–131. Lecture Notes in Computer Science 4942, Springer-Verlag, 2008.

[46] C. Mateos, A. Zunino, and M. Campo. A survey on approaches to gridification. *Softw. Pract. Exper.*, 38(5):523–556, 2008.

[47] A.S. McGough, W. Lee, J. Cohen, E. Katsiri, and J. Darlington. ICENI. In I. Taylor et al., editors, *Workflows for e-Science*, pages 395–415. Springer-Verlag, 2007.

[48] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäescher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541–551, 2009.

[49] M. Mosconi and M. Porta. Iteration constucts in data-flow visual programming languages. *Computer languages*, 26:67–104, 2000.

[50] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[51] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[52] T. Oinn, M. Greenwood, M. Addis, M.N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M.R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency Computat.: Pract. Exper.*, 18(10):1067–1100, 2006.

[53] NSF/Mellon Workshop on Scientific and Scholarly Workflow. Improving interoperability, sustainability and platform convergence in scientific and scholarly workflow. https://spaces.internet2.edu/display/SciSchWorkflow/Home, Visited August 2008.

[54] S.G. Parker, D.M. Weinstein, and C.R. Johnson. The SCIRun computational steering software system. In E. Arge et al., editors, *Modern Software Tools in Scientific Computing*, pages 1–40. Birkhauser press, 1997.

[55] C. Pautasso and G. Alonso. The JOpera visual composition language. *Journal of visual languages and computing*, 16(1–2):119–152, 2005.

[56] C. Pautasso and G. Alonso. Parallel computing patterns for grid workflows. In *Proc. of the HPDC2006 Workshop on Workflows in Support of Large-Scale Science (WORKS06) Paris France*, June 2006.

[57] K. Plankensteiner, R. Prodan, T. Fahringer Attila Kertész, and P. Kacsuk. Fault-tolerant behavior in state-of-the-art Grid workflow management systems. Technical report, CoreGRID, 2007. http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0091.pdf, August 2008.

[58] J. Qin and T. Fahringer. Advanced data flow support for scientific grid workflow applications. In *Proceedings of the ACM/IEEE Conference on Supercomputing SC 2007*, pages 1–12. ACM, 2007.

[59] H. Reekie. *Realtime signal processing: dataflow, visual, and functional programming*. PhD thesis, University of Thechnology at Sidney in the School of Electrical Engineering, September 1995.

[60] N. Russell, W.M.P. van der Aalst, and A.H.M. ter Hofstede. newYAWL:designing a workflow system using coloured Petri nets. In N. Sidorova et al., editors, *Proceedings of the International Workshop on Petri Nets and Distributed Systems (PNDS'08)*, pages 67–84, 2008.

[61] M. Siddiqui, A. Villazon, and T. Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized qos. In *Proceedings of the ACM/IEEE Conference on Supercomputing SC 2006*. IEEE, 2006.

[62] W. Smith, I. Foster, and V. Taylor. Scheduling with advance reservations. In *14th International Parallel and Distributed Processing Symposium*, pages 127–132. IEEE, 2000.

[63] Q. Snell, M. Clement, D. Jackson, and C. Gregory. The performance impact of advance reservation meta-scheduling. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing: IPDPS 2000 Workshop, JSSPP 2000*, pages 137–153. Lecture Notes in Computer Science 1911, Springer-Verlag, 2000.

[64] I. Taylor, M. Shields, I. Wang, and A. Harrison. The Triana workflow environment: architecture and applications. In I. Taylor et al., editors, *Workflows for e-Science*, pages 320–339. Springer-Verlag, 2007.

[65] W.M.P. van der Aalst and A.H.M. Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[66] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[67] G. von Laszewski and M. Hategan. Workflow concepts of the Java CoG Kit. *J. Grid Computing*, 3(3–4):239–258, 2005.

[68] C.Y. Wong, T.S. Dillon, and K.E. Forward. Analysis of dataflow program graphs. In *IEEE International Symposium on Circuits and Systems, ISCAS '98*, volume 2, pages 1045–1048. IEEE, 1988.

[69] Y. Zhao, M. Wilde, and I. Foster. Virtual data language: A typed workflow notation for diversely structured scientific data. In I. Taylor et al., editors, *Workflows for e-Science*, pages 258–275. Springer-Verlag, 2007.

[70] Z. Zhao, S. Booms, A. Belloum, C. de Laat, and B. Hertzberger. VLE-WFBus: a scientific workflow bus for multi e-science domains. In *2nd IEEE international conference on e-Science and Grid computing*, pages 11–19, 2006.