EXTENDING GWORKFLOWDL: A MULTI-PURPOSE LANGUAGE FOR WORKFLOW ENACTMENT

Simone Pellegrini, Francesco Giacomini
INFN Cnaf
Viale Berti Pichat, 6/2 - 40127 Bologna, Italy
simone.pellegrini@cnaf.infn.it
francesco.giacomini@cnaf.infn.it

Abstract Scientific workflows are becoming increasingly important as a vehicle for enabling science at a large scale. Lately, many *Workflow Management Systems* (WfMSs) have been introduced in order to support the needs of several scientific fields, such as *bioinformatics* and *cheminformatics*. However, no platform is practically capable to address the computational power and storage capacity provided by production Grids needed by complex scientific processes. In this paper, we introduce a *novel* design for a WfMS which has been proposed inside the CoreGRID project. Based on the *micro-kernel* design pattern, we have developed a *lightweight* Petri Net engine where complex functionalities – such as the interaction towards Grid middlewares – are provided at higher-level and described by means of Petri Net-based workflows. As the interaction with resources is described by workflows, it is possible to extend our platform by *programming* it using the *Grid Workflow Description Language* (GWorkflowDL).

Keywords: Grid Workflow, Workflow Enactment, Workflow Description Languages, Language Conversions, Petri Nets, GWorkflowDL, gLite, Sub-Workflows, Web Services

1. Introduction

Workflows are a modern phenomenon which, in the 1920, have been applied to the manufacturing industry for *rationalizing* the organization of work. Less than a decade ago, those concepts have been applied to the *Information Technology* (IT) world. Commonly, workflows are used to describe *business processes*, which consist of the flow or progression of *activities* – each of which represents the work of a *person*, an *internal system*, or the *process of a partner company* – toward some *business goal*. A *Workflow Management System* (WfMS) is a software component that takes as input a formal description of processes and maintains the state of processes executions, thereby delegat-

ing activities amongst people and applications. Recently, workflows have also emerged as a paradigm for representing and managing complex distributed *scientific* computation, specially in the fields of *bioinformatics* and *cheminformatics*.

Actually, many of the motivations of scientific workflows are also typical in business workflows. However, according to [1], it is possible to identify scientific workflow specific requirements such as *data* and *computation intensity* and dynamic resource allocation, scheduling and mapping to underlying distributed infrastructure such as Grid computing environments. While in business workflows the attention is indeed mainly placed in defining the *control-flow* (*control-driven*), in the scientific environment the *data-flow* definitely covers the principal role (*data-driven*) of the process design. This aspect also reflects in the workflow description languages and the underlying modeling formalisms. While scientific workflows are mainly based on *Directed Acyclic Graphs* (DAGs) the business workflows are modeled by means of the π*-Calculus* [2] and Activity Diagrams [3] formalisms.

Unfortunately, today, most of scientific WfMSs are not practically able to deal with complex and highly demanding processes. Many of them can address just a small set of computational resources and therefore they are not able to exploit real *production* Grids, which provide the computational power and storage capacity needed by complex scientific processes. Furthermore, a *standard* for workflow description has not been established yet and the variety of existing workflow languages intend to be specific to a platform limiting the *interoperability* of the workflow descriptions.

For these reasons, interoperability is becoming one of the main issues in the next-generation WfMSs and the CoreGRID project is taking large efforts in this direction. Inside the CoreGRID project, the FIRST Fraunhofer research group has proposed a workflow language, called the *Grid Workflow Description Language* (GWorkflowDL) which is based on the *High Level Petri Nets* (HLPNs) formalism [4]. The Petri Nets formalism has been chosen for several reasons [5] [6], and its semantics fits well with the majority of scientific processes. Furthermore, the expressivity of Petri Nets makes interoperability possible by *theorically* allowing the translation of the majority of scientific workflow descriptions – usually expressed in terms of DAGs – into a Petri Net-based description such as GWorkflowDL.

In this paper a *novel* WfMS architecture developed within the CoreGRID project is presented [7]. The characteristic of the system relies on its unique design which is based on the *micro-kernel* design pattern [8] [10] [9]. The purpose is to develop a *lightweight*, *fast* and *reliable* Petri Net-based engine with the ability to perform just few types of operations: *local method* calls, remote *Web Service* and *sub-workflows* invocation [11]. The main idea behind the WfMS is that '*everything is a workflow*' and therefore complex processes –

e.g. the execution of a job on a Grid environment – can be modeled by means of a workflow whose *atomic* tasks can be executed by the engine. The GWorkflowDL language has been improved in order to make it more expressive and deal with the *sub-workflows invocation mechanism* which, as we will see in Section 3, covers an important role in the *workflow enactment* process. Unlike other approaches, the presented design guarantees the neutrality towards the underlying mechanisms for task execution, in order not to compromise interoperability with multiple infrastructures and resources. In order to evaluate the capabilities of our design, tests have been done with workflows accessing resources available on the Grid provided by the EGEE project, a large and relatively mature infrastructure.

Section 2 introduces the GWorkflowDL language and the extensions introduced in order to make it more expressive and suitable for the internal representation of sub-workflows in our WfMS. In Section 3 an overview of the system will be presented. In Section 4 the language interoperability problem will be faced by introducing language translators. Section 5 presents how this work will progress in the future.

2. The Grid Workflow Description Language

The power of workflows relies in their *graph-oriented* programming model which is appealing for average users. In these years many formalisms have been considered for workflow modeling (DAGs, UML Activity Diagrams) and today, we see Petri Nets and π-Calculus as the main formalisms respectively for *data-driven* and *control-driven* workflows. Actually, π-Calculus is the formalism on which the *Business Process Execution Language for Web Services* (BPEL4WS) is based. BPEL4WS is a *de facto* standard for business workflows created by merging two workflow languages previously proposed by IBM (WSFL) and Microsoft (XLANG) [12]. On the other side, Petri Net-based workflow languages have not been yet standardized, but projects such as *Yet Another Workflow Language* (YAWL) [13] and the *Grid Workflow Execution Service* (GWES) [14] are demonstrating that Petri Nets can be used either to address the needs of business workflows as well as scientific ones.

The GWorkflowDL has been proposed by the FIRST Fraunhofer research group inside the CoreGRID project. Its main purpose is to define a *standard* for the description of Petri Net-based scientific workflows. It is *formally* based on the HLPN formalism and used in the *Knowledge-based Workflow System for Grid Applications* K-Wf and actually in its core component represented by the GWES. It is an XML-based language for representing Grid workflows which consists of two parts: (*i*) a *generic* part, used to define the structure of the workflow, reflecting the data and control flow in the application, and (*ii*) a

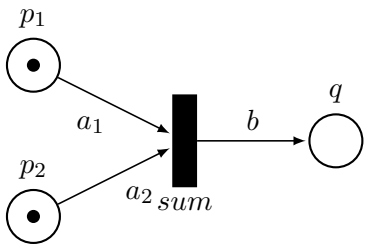

Figure 1. A two-operands `sum` operation modelled as a HLPN-based workflow

```
<workflow xmlns:op="http://www.gridworkflow.org/gworkflowdl/operation">
  <place ID="p1">
    <token><data><t1 xsd:type="xsd:int">3</t1></data></token>
  </place>
  <place ID="p2">
    <token><data><t2 xsd:type="xsd:int">2</t2></data></token>
  </place>
  <place ID="q" />
  <transition ID="T">
    <inputPlace placeID="p1" edgeExpression="a1"/>
    <inputPlace placeID="p2" edgeExpression="a2"/>
    <outputPlace placeID="q" edgeExpression="b"/>
    <op:operation>
      <op:operationClass name="plus"/>
    </op:operation>
  </transition>
</workflow>
```

Figure 2. GWorkflowDL abstract description of the Petri Net in Figure 1

middleware-specific part (*extensions*) that defines how the workflow should be executed in the context of a specific Grid computing middleware.

Recently [15], the language has been extended in order to represent *platform-independent* operations and therefore to make the workflow descriptions portable upon different WfMSs. Considering the workflow, expressed in HLPN, in figure 1, the *abstract workflow* can be expressed in GWorkflowDL as depicted in figure 2. At the abstract level, the structure of the Petri Net is *fully* described while operations – which are associated with transitions – are not specified. In figure 3, two *concrete*, *platform-specific*, implementation of the `sum` operation are provided. The `sum` operation is mapped (by the WfMS) into: (*i*) the invocation of a Web Service method and (*ii*) the execution of a local script.

In order to make the GWorkflowDL language compliant with our purposes and with the formal definition of HLPNs, it has been extended with several

```
...
  <op:operationClass name="plus">
    <op:wsOperation wsdl="http://localhost/math?wsdl"
                    operationName="plus" quality="0.6"/>
    <op:pyOperation operation="b = a1 + a2"
                    selected="true" quality="0.3"/>
  </op:operationClass>
...
```

Figure 3. Two *concrete*, *platform-specific*, implementations of the `sum` operation.

features. In the following sections, *three* extensions will be introduced with the main purpose to increase the GWorkflowDL language modeling capabilities and to introduce the mechanisms (e.g. the sub-workflows invocation) used by the WfMS during the workflow *enactment process*.

Sub-workflows

Real workflows have the tendency to become too large and complex, for this reason the *hierarchy* construct has been provided in order to reduce the overall complexity by structuring into sub-workflows. The sub-workflow concept allows for *hierarchical modelling*, i.e. it is possible to decompose complex systems into smaller systems allowing workflow reuse. A sub-workflow (also called *system*) is an aggregate of places, transitions and (possibly) subsystems. A sub-system, as happens with software components, defines an interface which is composed by a set of places, also called *connectors*. These connectors are divided into two categories, the *input connectors* (where tokens may enter into the system) and *output connectors* (where tokens may leave the system). The principle is very similar to the method invocation mechanism provided by common programming languages, where a piece of code can be packed into a method which exposes a signature.

In order to add support for sub-workflows in the GWorkflowDL language, we have introduced the `swOperation` element. For example, consider the workflow in Figure 4, its semantics is to sum the three numbers coming from the input places in_1, in_2, in_3 (which are the input connectors) and returns the result in the output place out (which is the output connector). The mapping between incoming/outgoing edges and input/output connectors can be *implicit* or *explicit*. The former uses a *positional semantics*, the N input places of the transition are mapped with the first N ($[0, N]$) places defined in the sub-workflow, while the M output places are mapped with the next M places ($[N+1, N+M]$). The latter one uses the optional `in` and `out` XML elements in order to make the mapping explicit as discussed in [15].

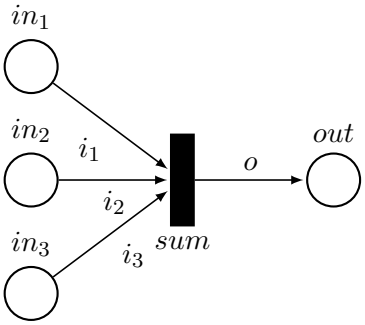

Figure 4. An abstract workflow which sums three *integer* numbers.

```
...
<transition ID="T">
  <inputPlace placeID="p1" edgeExpression="a1"/>
  <inputPlace placeID="p2" edgeExpression="a2"/>
  <outputPlace placeID="q" edgeExpression="b"/>
  <operation>
    <op:operationClass name="plus">
      <op:swOperation wsdl="http://localhost/math?wsdl"
                  operationName="plus" quality="0.6">
        <in name="in1">a1</in>
        <in name="in2">a2</in>
        <in name="in3">0</in>
        <out name="out">q</out>
      </op:swOperation>
    </op:operationClass>
  </operation>
</transition>
...
```

Figure 5. Explicit invocation of the sub-workflow depicted in Figure 4.

The invocation of such workflow is performed by using the `swOperation`. In the Figure 5 a concrete implementation of the sum operation via a sub-workflow is showed. In this case the optional XML elements `in` and `out` are used in order to map incoming edge variables to input connectors and outgoing edges variables to output connectors. The invocation of the sub-workflow is then managed by the engine in a platform specific way.

Place with Type

According to the *Coloured* Petri Nets formalism [16], tokens and also places have a *type* (also called *color*). Currently, the GWorkflowDL language allows

to specify type constraints for the tokens but not for the places. Add typing information to the places allows to check, even at *compile-time*, type *safety* of a Petri Net. For this reason, we have introduced an XML attribute – `type` – to the place element schema.

Typing information are also useful in order to check the *compatibility* of a transition operation with the relative incoming and outgoing places. For example consider the workflow in Figure 1 where places $p1$, $p2$ and q are of *integer* type; these constraints make the *signature* of the sum operation clear, i.e. $int\ sum(int,\ int)$. This feature avoids possible type *casting* errors which could raise at runtime and also helps the *mapping* from abstract to concrete workflow process by providing information of the operation signature.

Timed Transitions

In *Timed* Petri Nets, a time duration can be assigned to the transitions; tokens are meant to spend that time as reserved in the input places of the corresponding transitions. The GWorkflowDL language is currently not able to represent timed transitions and we think this feature is useful in order to make workflows deal with several design patterns which involve a delay, such as *polling*. Timed transitions can be supported at language level by introducing an *optional* XML attribute – `delay` – in the `transition` element. The delay is an integer value expressed in seconds.

3. The WfMS Overview

In order to evaluate the proposed language extensions and to demonstrate the capabilities of our design, we have developed a WfMS from the scratch. The project focuses the attention on different aspects of workflow management: (*i*) the *execution* of Petri Net-based workflows, the (*ii*) *mapping* from an abstract to a concrete workflow and the (*iii*) *conversion* between workflow description languages. The enactment of a workflow is performed by the engine which is the core of a WfMS and responsible of executing the workflow *tasks* respecting their dependencies. The second aspect is pursued by a *refinement process* which is based on the sub-workflow invocation mechanism. As stated, abstract workflows simply define the dependencies and the flow of data among macro-activities; the way in which tasks are performed is transparent to users and is managed by the WfMS. In our implementation, these macro-activities are implemented by concrete workflows which use primitives provided by the underlying platform. Interoperability is made possible by means of *translators*. Workflow language conversion will be discussed in more detail in the next section.

Tests have done with workflows accessing resources available on the Grid provided by the EGEE project, a large and relatively mature infrastructure.

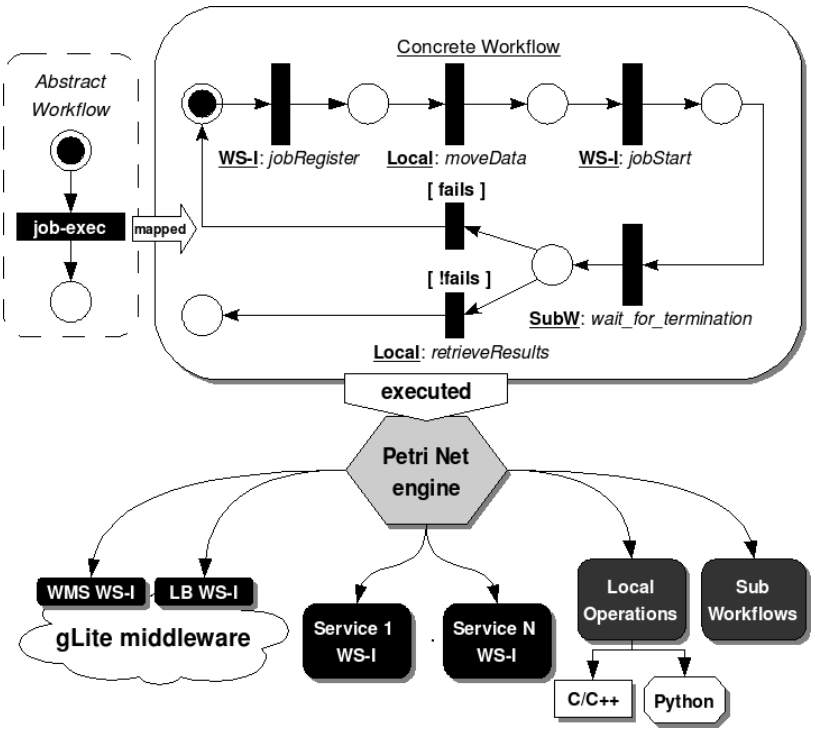

Figure 6. The WfMS architecture overview.

In particular, the execution of Grid jobs is performed by relying on the gLite *Workload Management System* (WMS) [17] through its Web Service interface (WMProxy). The WMS takes care of the resource management in gLite by finding the best available resources considering a set of users requirements and preferences (such as CPU architecture, OS, current load).

3.1 The Workflow Engine

The engine of a WfMS is the component which has, among others, the responsibility to interact with the underlying resources. Commonly, the support toward a specific resource or a Grid middleware is *hard-coded* into the workflow engine itself making the interaction with new platforms very difficult. Many of the existing WfMSs are designed to address just specific resources or Grid infrastructure and this choice also reflects in the associated workflow language which – instead to be unaware of the implementation details – risks to become too bound to a platform. This kind of design limits both the capa-

bilities of the WfMSs and the *portability* of the workflow descriptions making their reuse impossible.

An alternative approach is to design a *lightweight* engine based on the *micro-kernel* design pattern [8] [10]. The engine is able to execute efficiently Petri Nets where the operations associated with transitions are of few, well defined, types. As depicted in Figure 6, the interaction with arbitrary *Web Services*, the execution of *local methods* and the invocation of *sub-workflows* are the only concrete operations supported by the engine and also provided at GWorkflowDL language level (see previous Section). New functionalities are provided at *higher* level through sub-workflows. This makes the GWorkflowDL, and thus Petri Nets, the main *programming language* of our platform.

The engine internally keeps the HLPN model of a workflow and executes it according to the Petri Nets *semantics*. The implementation of such semantics, and in particular the *non-determinism*, faces with the *imperative paradigm* provided by the mainstream programming languages such as C/C++ and Java. In [11], the engine design details are discussed together with the problems of sub-workflow invocations, non-determinism, parallelism, and transition firing which are practically faced and efficiently solved.

3.2 The Refinement Process

As happens with programming languages, where new functionalities are provided through *libraries*, in our WfMS new functionalities can be provided by defining new workflows which are accessed using the sub-workflow invocation mechanism. The mapping from abstract operations onto a concrete implementation is currently established by the correspondence of the operation name and the sub-workflow name. The overview of the refinement process is depicted in Figure 6 where the execution of a job in a Grid environment is implemented by a sub-workflow able to interact with the EGEE/gLite middleware services. Interaction with a different Grid platform, e.g. UNICORE, can be provided – even at *runtime* – simply by defining a new sub-workflow which describes the job submission and job monitoring processes.

Despite its simplicity, this strategy cannot be considered to be *multi-purpose* as far as a name-to-name mapping could fail in many situations. As stated, the mapping of an abstract operation f onto a concrete workflow implementing f is done by the WfMS using an associative map which binds the f's name into a workflow definition. In those situations, where an operation has several implementations, this strategy cannot be exploited. For the future, in order to improve the refinement process and make it more flexible, we are investigating the possibility to use *ontologies*.

Furthermore, thanks to the adopted design the engine has no information about the state of the workflow. As far as all the macro-operations of a work-

flow are decomposed into sub-workflows which use *atomic* operations (i.e. a local method or a remote Web Service invocation), it is possible to represent the overall workflow *state* simply by using the Petri Net *marking*. In this way, it is possible to provide a *checkpointing* mechanism able to restore the execution of a workflow, after a system failure, just re-establishing the last marking of the net.

4. Language Translators

One of the main interests of the CoreGRID project is the interoperability among heterogeneous systems. The growing number of WfMSs and workflow languages is making interoperability one of the main issues in the workflow environment. In a WfMS, interoperability can be achieved at different levels of complexity. The simpler one is about the description languages and the possibility to run legacy workflows on different platforms; it can be achieved – under certain circumstances – by means of language *translators*. On the other side, the interoperability among WfMSs is more complex to achieve as far as no recognized standard exists and no platform exposes a clear interface.

In our activity, we have focused on the language conversion problem enabling our WfMS to execute legacy workflows written in the JDL and the SCUFL languages. The *Job Description Language* (JDL) is used in the gLite middleware for job description. It is based on Condor's *Classified Advertisements* (ClassAds) and allows the description of DAG-based workflows which are executed by DAGMan [22]. The *Simple Conceptual Unified Flow Language* (SCUFL) is the underlying language of the Taverna WfMS [23]. Scufl is widely used in bioinformatics and several *primitive* operations are provided at language level in order to interact with *biological-specific* services such as `biomoby` [20].

In this section we introduce two language translators able to convert DAGMan and Taverna workflows into GWorkflowDL descriptions. Our efforts proves the power of Petri Nets in describing workflows. Recent studies have indeed demonstrated that the modeling capabilities of Petri Nets outperform other formalisms in describing workflows [18]. Actually, it is also possible to convert workflows based on several formalisms in terms of Petri Nets making language interoperability possible. We choose GWorkflowDL as description language because it has been proposed inside the CoreGRID project and additionally we think it has the characteristics to become a standard for scientific processes description.

4.1 JDL to GWorkflowDL

DAGMan [22] acts as a meta-scheduler for Condor jobs submitting job respecting their inter-dependencies which are expressed as a *Directed Acyclic*

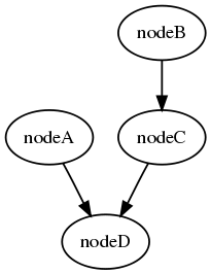

Figure 7. A simple DAG workflow.

Graph. In case of job failure, DAGMan continues until it can no longer make progress. The example of a DAG-based workflow is showed in the Figure 7. While `nodeA` and `nodeB` can be executed concurrently, `nodeC` must wait the termination of `nodeB` and `nodeD` can be executed only when `nodeA` and `nodeC` have been completed. However, DAGMan totally lacks control structures (such as *branches* and *loops*) and a customizable error handling; the default strategy after a node failure is the *re-execution* of the node up to a configurable number of times.

The *Job Description Language* (JDL) is an extension of the Condor's *Classified Advertisements* (ClassAds): a record-like structure composed of a finite number of attributes separated by semi-colons (`;`). It is used in the gLite middleware for describing Grid jobs. A workflow in JDL is defined by a set of *nodes* and a set of *dependencies* which define *precedence relations* between nodes. The conversion of a DAG to a Petri Net is quite *trivial*: a DAG node can be modeled by a Petri Net transition and the flow of data among nodes by using tokens. However, a DAG node represents the execution of a Grid job and that practically means (*i*) the submission of the job description, (*ii*) the transfer of its input files (also called the input sandbox), (*iii*) waiting for the job termination and (*iv*) the retrievement of the results (the output sandbox). As stated in the previous section, the sequence of these operations can be modeled via a workflow and each DAG node can be substituted by a sub-workflow invocation. The Figure 8 depicts the Petri Net-based *concrete* workflow resulting from the conversion of the DAG represented in Figure 7. In the workflow are visible some implementation details (`certificate` and `delegationId`) which are specific to the gLite Grid middleware.

4.2 Scufl to GWorkflowDL

Scufl is an XML-based language which allows to model DAG-based workflows. As stated, Scufl is the description language used by the Taverna WfMS

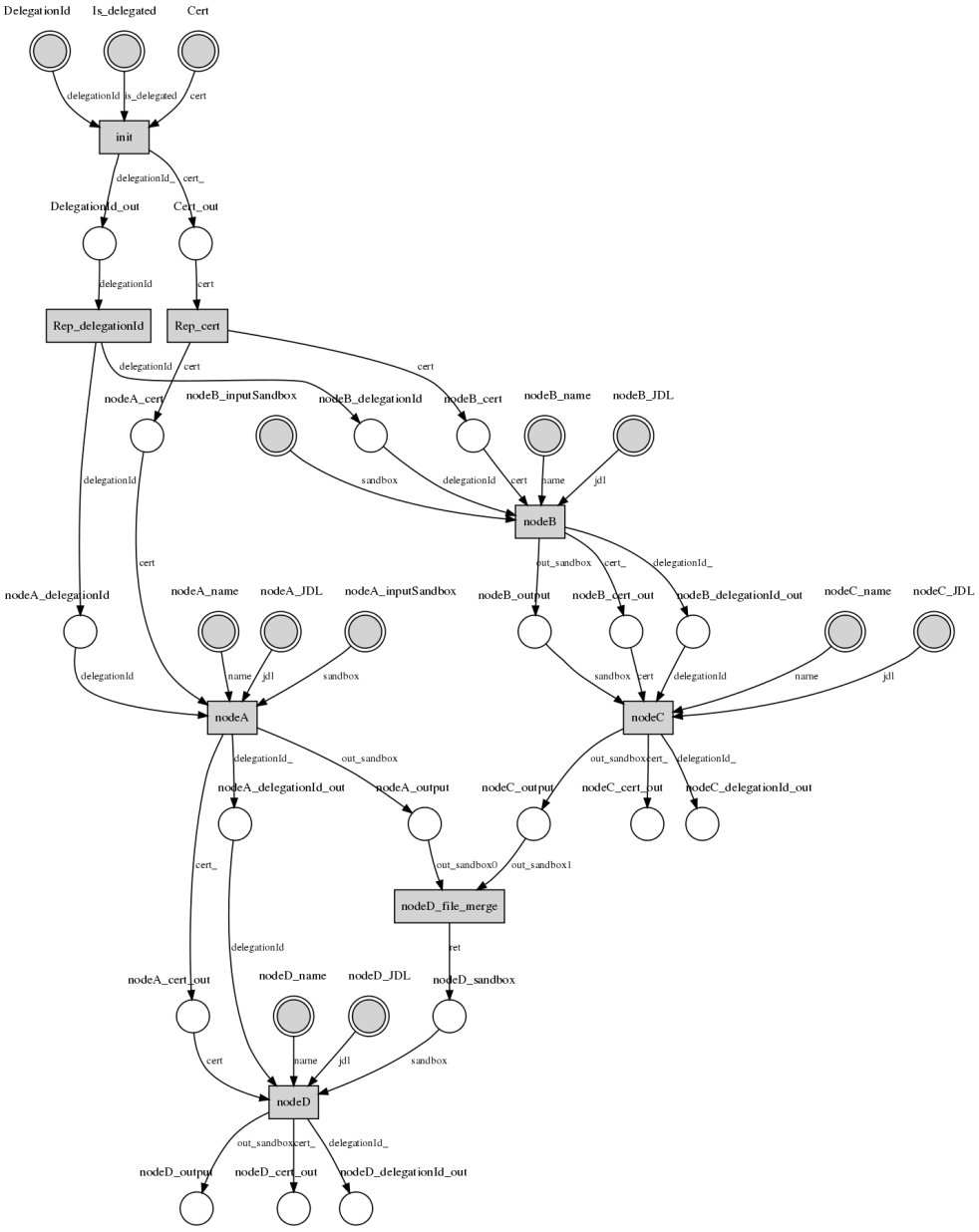

Figure 8. The result of the conversion of the DAG in Figure 7 into a Petri Net-based description.

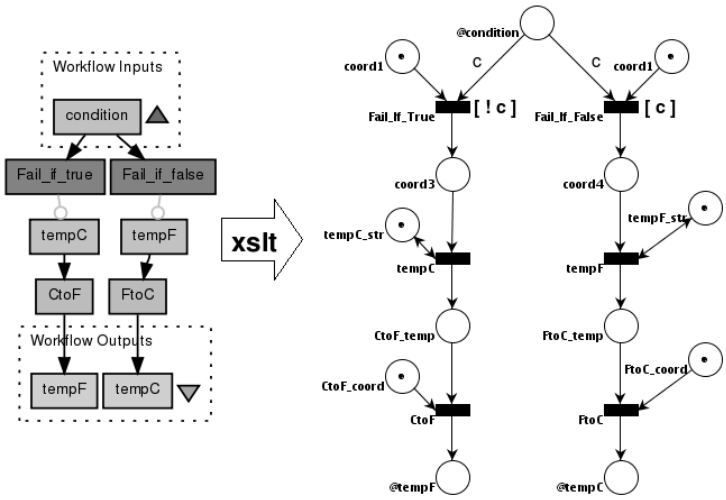

Figure 9. Conversion of a Scufl workflow (on the left) to a GWorkflowDL Petri Net-based workflow (on the right) by means of a XSL Translator

and in particular by the FreeFluo engine. Unlike JDL, Scufl allows to define several control structures which make the language more expressive. In Scufl several types of nodes exist: *processors*, *sources* and *sinks*. Sources and sinks represent respectively the input and output nodes of the process, the concept of computational node is represented by the processor. A processor can be either (and not limited to) a *local operation*, a *string constant*, a *generic Web Service invocation* or a *sub-workflow execution*. Nodes are connected via edges (or *links*) which define precedence relations and the data-flow.

Additionally, Scufl allows to model the control-flow by using the *coordination* element. Coordination constraints are used to prevent a processor transitioning between states until some constraint condition has been satisfied. An example is depicted in Figure 9 where the execution of processors `tempC` and `tempF` are respectively subordinated to the result of the `Fail_if_True` and `Fail_if_False` processors. Processors can be also decorated by several *properties* which allow to define: the *retry behaviour*, the *alternate* processor and the *iteration* strategy. The iteration strategy is one of the most powerful feature in Taverna, and it defines the implicit iteration over the incoming data sets. The complete Scufl language reference can be found in [21].

As Scufl and GWorkflowDL are XML-based the conversion is made possible by means of an *Extensible Stylesheet Language Transformations* (XSLT). We do not want to go deep into the translation process because this is a work

in progress and the converter is still under development. However, we are currently able to translate simple Scufl workflows which interact with arbitrary Web Services as depicted in Figure 9. The workflow shows an example of a *conditional execution* where the input value of the source node `condition` determines the execution of the left branch – when the input value is `false` – or the right one, otherwise. The left branch performs a temperature conversion from Fahrenheit to Celsius using a Web Service, the right one the inverse operation. The result of the conversion is stored into the corresponding sink node, `tempF` or `tempC`.

The Figure 9 shows how the workflow can be converted into a Petri Net-based description which keeps the same semantics. However, the majority of Scufl workflows uses local bindings to the *Java platform* making the conversion process even more difficult.

5. Conclusions and Future Work

This paper presents an overview of the design of a WfMS developed in the context of the CoreGRID project. Interaction with multiple resources and Grid infrastructures is made possible by adopting a novel design based on the micro-kernel design pattern, which makes the platform extensible by using sub-workflows. We have also investigated the language interoperability issues which are made possible by the use, and improvement, of the GWorkflowDL language. The *Turing-complete* semantics of GWorkflowDL makes the conversion from other formalisms (such as DAGs and π-calculus) possible by means of language translators.

The WfMS is nevertheless still under development and further work is needed in order to improve it and make it usable in production environments. As stated, the mapping from abstract to concrete workflow should be improved by using *ontologies* and other features – such as *planning* and *advance reservation* – should be investigated in order to make the system more compliant to the needs of scientific processes.

References

[1] On Scientific Workflow. *TCSC Newsletter, IEEE Technical Committee on Scalable Computing*, 9(1), 2007.

[2] F. Puhlmann1 and M. Weske1. Using the π-Calculus for Formalizing Workflow Patterns. *Lecture Notes in Computer Science*, pages 153-168, 2005.

[3] M. Dumas and A. H. M. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. *Lecture Notes in Computer Science*, pages 76–90, 2001.

[4] A. Hoheisel and U. Der. An XML-based framework for loosely coupled applications on grid environments. In P. Sloot, editor, *ICCS 2003*, number 2657 in Lecture Notes in Computer Science, pages 245–254. Springer-Verlag, 2003.

[5] M. Alt et al. Using High Level Petri-Nets for Describing and Analysing Hierarchical Grid Workflows. In *Proceedings of the CoreGRID Integration Workshop 2005, Pisa*, 2005.

[6] W. Aalst. Three Good reasons for Using a Petri-net-based Workflow Management System. In *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96)*, pages 179–201, Camebridge, Massachusetts, 1996.

[7] CppWfMS, http://wfms.forge.cnaf.infn.it/

[8] Douglas Schmidt et al.: *Pattern-Oriented Software Architecture*, Siemens AG, pages 171-192, 2000.

[9] S. Pellegrini et al. A Practical Approach to a Workflow Management System. *Proceedings of the CoreGRID Workshop 2007*, Dresden, Germany, 2007

[10] Dragos A. Manolescu: *An extensible Workflow Architecture with Object and Patterns*, TOOLSEE 2001.

[11] S. Pellegrini, F. Giacomini Design of a Petri Net-Based Workflow Engine. In *Proceedings of the 3rd International Workshop on Workflow Management and Applications in Grid Environments (WaGe08)*, Kunming, China, 2008.

[12] T. Andrews et al. Business process execution language for web services version 1.1. Technical report, BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems, 2003.

[13] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.

[14] A. Hoheisel. Grid Workflow Execution Service – Dynamic and Interactive Execution and Visualization of Distributed Workflows. In *Proceedings of the Cracow Grid Workshop 2006*, Cracow, Poland, 2007

[15] S. Pellegrini, A. Hoheisel et al. Using GWorkflowDL for Middleware-Independent Modeling and Enactment of Workflows. In *Proceedings of the CoreGRID Integration Workshop 2008, Crete*, 2008.

[16] Kurt Jensen: *An Introduction to the Theoretical Aspects of Colored Petri Nets*, Lecture Notes in Computer Science (Springer), 1994.

[17] P. Andreetto et al. Practical approaches to grid workload and resource management in the egee project. In *Proceedings of the Conference for Computing in High-Energy and Nuclear Physics (CHEP 04)*, Interlaken, Switzerland, 2004.

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

[19] W. van der Aalst. Pi calculus versus petri nets: Let us eat humble pie rather than further inflate the pi hype. *unpublished discussion paper*, 2003.

[20] BioMoby, http://biomoby.org/

[21] T. Oinn. XScufl Language Reference.
http://www.ebi.ac.uk/ tmo/mygrid/XScuflSpecification.html, 2004.

[22] Condor DAGMan, http://www.cs.wisc.edu/condor/dagman/

[23] Tom Oinn et al.. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, June 2004.

[24] A. Hoheisel and M. Alt. Petri Nets. In I.J. Taylor, D. Gannon, E. Deelman, and M.S. Shields, editors, *Workflows for e-Science – Scientific Workflows for Grids*, Springer, 2006.