IWIR: A Language Enabling Portability Across Grid Workflow Systems

Kassian Plankensteiner, Johan Montagnat, Radu Prodan

IWIR: A Language Enabling Portability Across Grid Workflow Systems

Kassian Plankensteiner
Institute for Computer Science
University of Innsbruck
Innsbruck, Austria
kassian@dps.uibk.ac.at

Johan Montagnat
I3S laboratory
CNRS
Sophia Antipolis, France
johan@i3s.unice.fr

Radu Prodan
Institute for Computer Science
University of Innsbruck
Innsbruck, Austria
radu@dps.uibk.ac.at

ABSTRACT

Today there are many different scientific Grid workflow management systems using a wide array of custom workflow languages. Some of them are geared towards a data-based view, some are geared towards a control-flow based view and others try to be as generic, and therefore often complex, as possible. All of these languages and custom workflow management system front-ends fulfill special needs and workflow creation paradigms for their respective user communities. The problem is that once a workflow application has been created in one of these systems, it becomes very hard to share the workflow with users working with different systems. Portability and interoperability between current systems barely exists. In this work, we present a common workflow language for use as an intermediate exchange representation by multiple workflow systems. It comprises atomic tasks, compound tasks including conditionals, sequential and parallel loops as well as an expressive set of data types and data flow constructs.

Keywords

Workflow Languages, Portability, Interoperability, Grid Computing

1. INTRODUCTION AND RELATED WORK

Currently, each workflow system comes with its own input language designed to satisfy the needs of its specific target community. Workflows are specified in different systems at various levels of detail, sometimes hiding the underlying infrastructure, and sometimes exposing at least part of the system. In most cases, however, workflows are hard-coded to the workflow system within which they have been developed. Existing language specifications range from simple and pragmatic scripting languages (e.g. shell, Python), to custom DAG-based representations (e.g. DagMan), or more modern XML-based descriptions (e.g. AGWL [5], GWENDIA [8], P-GRADE [6], SCUFL [7], Triana [10]). The control flow-based abstractions range from pure DAG specifications to more comprehensive imperative constructs such as conditional or loop statements (sequential and parallel). Other approaches based on data flow specifications include advanced collection or array-based data distributions and computations, or even data streaming constructs.

It is widely believed that imposing a single standard for the specification of scientific workflows is a difficult task that is likely not to succeed in being adopted by all communities given the heterogeneous nature of their fields and problems to solve. However, an agreement on an *Interoperable Workflow Intermediate Representation (IWIR)* sufficient for describing a large majority of existing workflow constructs at a lower level of abstraction that is only processed by workflow systems could be more successful as it is not directly exposed to a human developer. Since IWIR is being developed as part of the EU FP7 SHIWA project [1], it will gain the advantage of being implemented as intermediate language for many of the most successful scientific workflow systems that are part of the project: ASKALON, Moteur, P-GRADE, Triana and Pegasus.

Important design properties of IWIR are simplicity, being oriented towards the Distributed Computing Infrastructure (DCI) execution platform (analogous to a machine language) and not towards the user source code, and containing only a small set of constructs required for execution and optimization for the target platform. The idea of a single intermediate language is not unique and has been explored in other domains, for example by the UNiversal Computer Oriented Language (UNCOL) [4] proposed in 1958 by Melvin E. Conway as a solution for making compiler development economically viable.

A simple and portable intermediate workflow representation has a number of advantages for the application developers relative to the current practice of proprietary workflow languages:

1. It enables application developers to program applications using their favorite high-level workflow language and execute it on every DCI with an IWIR-enabled enactment engine;

2. It enables the application scientists to flexibly select the *best* enactment engine deployed on the *best* DCI infrastructure for running their workflows. This is usually a subjective decision that can only be answered by the scientists themselves, depending in part on the nature of the experiments and the scientists objectives (e.g. performance, reliability, cost);

WORKS11 2011 Seattle, Washington USA

3. It enables runtime interoperability between different workflow systems. Sub-workflows, specified either by the end-user or selected dynamically by the workflow scheduler, can be dynamically scheduled and transferred across different workflow systems in the form of a common intermediate representation, which creates numerous optimization opportunities;

4. It is a generic solution, open to integration of new languages and workflow systems. Integrating a new workflow language able to execute on n DCI infrastructures requires the development of one IWIR front-end ($O(1)$ complexity), while language-to-language translators require n front-ends ($O(n)$). Similarly, porting m interoperable workflows to a new DCI platform requires the development of one single IWIR-compliant back-end ($O(1)$), while language-to-language translations would require m back-ends, one for each workflow system. Therefore the IWIR solution reduces the effort of porting m workflow systems onto n distributed platforms from $m * n$ to $m + n$. This is an important step to make the development of new workflow systems for multiple existing DCI infrastructures economically viable.

To qualify for interoperability using IWIR, each workflow system will need to adjust its front-end to translate its source input language into the IWIR workflow representation. Once translated into this intermediate representation, the interoperability with the other systems is implicitly enabled. For this reason, we also call this interoperability scenario *front-end* workflow interoperability.

The paper is organized as follows. Section 2 gives an overview of the design decisions taken when specifying the IWIR language, while Section 3 defines the IWIR language and its constructs. We provide examples of how common data distribution strategies are expressed in IWIR and show an example of language translation on a real workflow application in Section 4. We introduce IWIRtool, a Java-based library for creation and manipulation of IWIR workflows in Section 5 and conclude in Section 6.

2. DESIGN AND TERMINOLOGY

When talking about workflow applications, we distinguish two parts, corresponding to two levels of abstraction: the *concrete* part and the *abstract* part of the workflow.

The *concrete* part of a workflow applicaton contains information about its computational tasks. This can mean a wide range of things – it can be information on how to execute a certain application on a certain machine, information on where and how to call a certain web service, an explicit program given in a scripting language or even an executable binary file representing the computational task itself. The type and form of information contained in the concrete part of the workflow is often specific to a certain workflow system and distributed computing infrastructure (DCI).

The *abstract* part of a workflow application deals with the orchestration of the computational tasks. It defines precedence relations between the computational entities described in the concrete part as well as the data flow between them. The abstract part of a workflow therefore deals with issues on a level of abstraction above the concrete part, which makes it independent of the underlying DCI infrastructure.

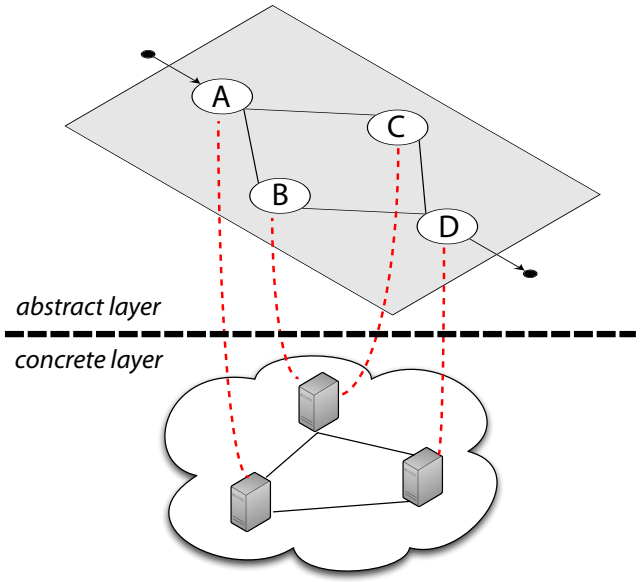

Figure 1: Two levels of a workflow: abstract vs. concrete

Both parts combined, abstract and concrete, make the workflow a fully specified, executable application. Figure 1 shows a graphical representation of the two layers. The mapping of tasks from the abstract part of the workflow to the concrete computational entities on the target DCI (concrete part of the workflow) can either be done at the time of workflow creation, or be handled by a scheduler component at workflow runtime. The IWIR language deals with the abstract part of the workflow and provides a mechanism to enable a one-to-many mapping from the abstract tasks to the concrete computational tasks.

In the course of developing the specification for the IWIR language, we extensively studied many scientific workflow systems and their workflow description languages, most notably AGWL [5], GWENDIA [8], P-GRADE [6], SCUFL [7] and Triana [10], to be able to specify a common ground that would cover all of the commonly used constructs.

IWIR as a language is designed to enable portability of workflows across different specification languages, different workflow systems and different Distributed Computing Infrastructures (DCIs). The IWIR language itself is a language enabling the portability of the *abstract part* of a workflow and it therefore decouples itself from the *concrete level* by abstracting from specific implementations or installations of computational entities through a concept called *Task Type*. IWIR avoids the use of constructs for *data manipulation*, therefore it does not define ways to change data directly (such as integer operations, concatenation of strings, etc.) but rather provides means to powerfully *distribute* data to computational entities, so-called tasks, that do data manipulation. IWIR focuses on the description of the workflow logic independently from the data sets to be processed. Our study of current workflow description languages led to the decision of creating a graph-based structure, mixing dataflows and an expressive set of sequential and parallel control structures, specified as an XML representation. These decisions allow for a straightforward transformation from most

of todays scientific workflow languages into a common IWIR-based representation.

Task Type.
A Task Type is a signature of a computational task. It is composed of a *type name* and a set of *input* and *output ports* with corresponding *data types* (see Section 3.1.1). A Task Type is used as an abstract placeholder to represent a set of *Task Deployments* implementing the given Task Type. Task Types are a concept belonging to the *abstract* level of the workflow.

Task Deployment.
A Task Deployment is a specific implementation of a Task Type. It can refer either to another IWIR (sub-)workflow (see Section 3.2) or to a single computational entity which can be an executable, a web service operation, a scripting language fragment etc. Information about a Task Deployment is usually stored in a Deployment Registry, which contains all the necessary information for execution (Task Deployment Descriptors). Task Deployments are not visible in IWIR, they are dynamically chosen from a set of Task Deployments implementing a Task Type at the time of workflow enactment. Task Deployments are a concept belonging to the *concrete* level of the workflow.

3. IWIR SPECIFICATION

An IWIR workflow description document defines a workflow in the IWIR language. A workflow consists of one top-level task (compound or atomic), which (if compound) may contain an arbitrary number of other tasks as well as data- and control-flow links. This top-level task forms the data entry and exit point of a workflow application and therefore also defines the signature of the application. Figure 2 shows the IWIR document structure.

```
<IWIR version="version" wfname="name"
 xmlns="http://shiwa-workflow.eu/IWIR">
  <task...>
</IWIR>
```

Figure 2: IWIR Document Structure

IWIR version.
The version attribute of the IWIR tag defines the version of the IWIR language specification that the workflow is defined in. This attribute is provided to make sure that future extensions of the IWIR specification do not interfere with existing workflow definitions. The current version of IWIR is version 1.1.

IWIR xmlns.
All XML tags and concepts defined for IWIR are defined in the xml namespace `http://shiwa-workflow.eu/IWIR`. To be able to concentrate on the concepts rather than the notation, we use a global namespace declaration here.

IWIR wfname.
The workflow name; serves as an identifier for the workflow.

task.
A workflow in IWIR has exactly one top-level task element. This element can be a compound task or an atomic task and its signature defines the required input ports as well as the output ports of the workflow and their data types. See Section 3.2 and 3.3 for a list of possible compound and atomic task constructs.

3.1 Data Constructs

3.1.1 Data Types

IWIR defines a set of data types that can be used in an IWIR document. An IWIR data type identifier is formed according to the following BNF grammar:

```
            <type>::= <simple-type> |
                      <collection-type>
<collection-type>::= "collection/"<type>
    <simple-type>::= "string" | "integer" |
                     "double" | "file" |
                     "boolean"
```

A *collection* is an ordered, indexed list of data elements of the same data type. The number of elements in a collection can be dynamic. One data item in a collection is always associated with a type and a, possibly multi-dimensional, integer index (indexing starts from 0, one dimension per nesting-level). The nesting level n of a collection can be determined by its data type, by counting the number of occurrences of the string `collection` in the type definition.

3.1.2 Data Flow

Data ports are connected to each other using the `link` construct (see Figure 3). Every composite task, and therefore every scope, has a `links` block containing all of the data flow links in its scope, making every composite task self contained with respect to its data flow. It is not allowed to cross scopes (see Section 3.1.4) using data flow links.

```
<links>
  <link from="from" to="to" />*
</links>
```

Figure 3: Defining data flow using the `link` construct

link from.
The `from` attribute of a link defines the source of the data flow connection. In IWIR, this attribute is specified in the form of `task/port`, where *task* is the name of the task and *port* is the name of the data port providing the data. We call the data port referred to by the `from` attribute the *source port* of the link.

link to.
The `to` attribute defines the destination of the data flow connection. In IWIR, this attribute is specified in the form of `task/port`, where *task* is the name of the task and *port* is the name of the data port consuming the data. We call the data port referred to by the `to` attribute the *target port* of the link.

The general rule is that the data type of the data port specified in the `from` attribute has to match the data type of the port referred to in the `to` attribute. There are a few exceptions to this rule to account for the semantics of

compound tasks such as *(parallel)ForEach* splitting data collections into single elements. For a full specification of the particulars of these exceptions, please refer to [9]. Additionally, IWIR allows the following implicit type-cast operations when connecting data ports using the `link` construct:

- `boolean` → `string`, `integer` → `string`, `double` → `string` and `integer` → `double`
- any type `A` → `collection/A` yields a collection containing only one entry
- `file` → `string` yields a URI to the file

Furthermore, IWIR mandates that a data port may only be the *target port* of a single link construct (in other words: one *target port* may only be linked to a single *source port*), except in cases where the specification explicitly says otherwise. Generally, building a cyclic data dependency using link constructs is not allowed in an IWIR workflow.

3.1.3 Control Flow

Sometimes it is required to define a pure control flow dependency between two tasks that does not involve any data dependency. Such a dependency can be expressed in IWIR using only the task names (as opposed to `task/port`) in the `from` and `to` attributes of the *link* construct (see Section 3.1.2, Figure 3).

A pure control flow link fires after the given source of the link successfully finished execution. In the case of the source being a sequential loop task, the control flow link therefore fires after successful execution of the final iteration. For parallel loop tasks, the control flow link fires only after every parallel iteration has successfully completed. If a task depends on more than one incoming control link, it is executed only after all incoming control links have fired.

As in the case of data flow links, building a cyclic control dependency using link constructs is not allowed in an IWIR workflow.

3.1.4 Scopes

In IWIR, data ports and tasks can only be referred to in certain regions of the workflow document. This area is called the *scope* of them. IWIR only allows a data `link` to refer to data ports and tasks within the current scope. Every scope has a single `links` block. The scope that a `links` block (see Figure 3) can see and access is called the *current scope* of the `links` block. It consists of the following elements:

Current Task The name and all data ports (input Ports, output Ports, loop Ports, loop counter ports, loop element ports, output ports, etc.) of the task containing the `links` block itself is an element of the current scope

Enclosed Tasks The names and all data ports of all first-level subtasks are elements of the current scope. The current scope does **not** extend to tasks embedded into the direct subtasks themselves.

These strict scoping rules establish an important principle in IWIR, the principle of *self-contained* tasks. In IWIR, every task is self-contained - providing a single point of entry (the input ports) and exit (the output ports). This establishes strong reusability and makes sure that every single task, atomic and compound, is a fully specified abstract workflow in itself. This allows systems to utilize the concept of sub-workflows and opens up the possibility to easily replace workflow parts.

```
<task name="name" tasktype="tasktype">
  <inputPorts>
    <inputPort name="name" type="type"/>*
    ...
  </inputPorts>
  <outputPorts>
    <outputPort name="name" type="type"/>*
    ...
  </outputPorts>
</task>
```

Figure 4: The `task` construct

3.2 Atomic Tasks

An Atomic Task is a task which is implemented by a single computational entity (an executable, a web service, a script, etc.). An atomic task can be described using the task construct, shown in Figure 4.

Task name.

The task name serves as an identifier for the task. Tasks must be organized in an IWIR workflow or a Compound Task which define a scope (see Section 3.1.4) for them. In the scope, the name of each task must be unique.

Task type.

In IWIR, the functional behavior and the interfaces of tasks are described by *Task Types*. A Task Type is an abstract description which can be implemented by different Task Deployments (concrete implementations of computational entities) deployed in a DCI or given as an executable entity like a binary executable or a script fragment, depending on the DCI and workflow system executing the workflow application. A Task Type can also refer to a sub-workflow. The Task Type must be defined in a Type Registry before enactment. Task Types shield the implementation details of Task Deployments from the IWIR programmer and help to enable workflow interoperability across different DCIs. Locating and invoking of Task Deployments are done by an underlying runtime environment.

inputPorts/outputPorts.

All data ports of the task are enclosed in the inputPorts/outputPorts sections. The number and types of the input and output ports are determined by the chosen task type. The `link` construct (see Figure 3) is used to define the data flow between input and output ports of different tasks.

3.3 Compound Tasks

A Compound Task is a task which encloses some Atomic Tasks and/or other Compound Tasks as well as their data- and control-flow links. The compound task and its links are self contained in the sense that data- and control flow links may not cross the boundaries of a Compound Task. Because of this, Compound Tasks are able to form separate self-contained scopes. We classify the compound tasks into two groups, *Basic Compound Tasks* and *Parallel Compound Tasks*. Basic Compound Tasks are sequential constructs similar to well known constructs in high-level languages such as `blockScope`, `if`, `while`, `blockscope`, `for` and

```
<if name="name">
  <inputPorts>
    <inputPort name="name" type="type"/>*
  </inputPorts>
  <condition> condition </condition>
  <then>
    <task .../>+
  </then>
  <else>?
    <task .../>+
  </else>
  <outputPorts>
    <outputPort name="name" type="type"/>*
  </outputPorts>
  <links>
    <link from="from" to="to" />*
  </links>
</if>
```

Figure 5: The `if` Task

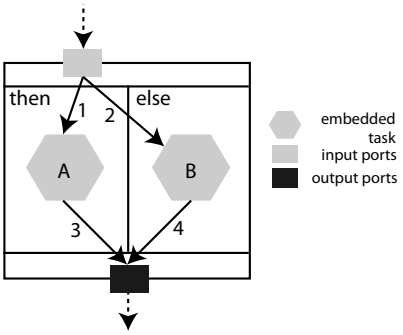

Figure 6: Data Flow in the `if` task

`forEach`. Parallel Compound Tasks are constructs that express parallel loops (`parallelFor` and `parallelForEach`).

3.3.1 *The* `blockScope` *task*

The `blockScope` compound task (not shown due to space limitations) enables the grouping of the contained tasks in one scope. This helps to avoid naming conflicts and enables to build DAG-like structures even at the top-level of the workflow.

3.3.2 *The* `if` *task*

The `if` compound task enables the conditional execution of the inner tasks. The definition of the `if` task can be seen in Figure 5.

Condition.

Section 3.4 explains how to formulate the condition expression that controls whether the `then` or the `else` branch is executed at runtime.

Output Ports.

It is generally unknown at compile time which branch of the `if` task is executed. Therefore, for each output port declared in the `if` task, it is necessary that both the output of one task from the `then` branch and the output of one task from the `else` branch are connected to this port using link constructs (if the `else` branch is omitted, a link from an input port of the if task to the output port needs to

```
<while name="name">
  <inputPorts>
    <inputPort name="name" type="type"/>*
    <loopPorts>
      <loopPort name="name" type="type">*
    </loopPorts>
  </inputPorts>
  <condition>
    condition
  </condition>
  <body>
    <task .../>+
  </body>
  <outputPorts>
    <outputPort name="name" type="type"/>*
    <unionPorts>
      <unionPort name="name"
                 type="collection"/>*
    </unionPorts>?
  </outputPorts>
  <links>
    <link from="from" to="to" />*
  </links>
</while>
```

Figure 7: The `while` Task

be created). Since for a given instance of the `if` task only one branch, either the `then` or the `else`, is executed, links connecting ports of tasks belonging to different branches are **not** allowed.

Figure 6 illustrates the usual data flow through the `if` construct. Data arrives at the input port. Depending on the condition evaluation, either the `then` or the `else` branch is executed, therefore either link 1 or link 2 are used to transfer data to the contained tasks A or B. After completion of the embedded task, the generated data is written to the output port using either link 3 or link 4.

3.3.3 *The* `while` *and* `for` *tasks*

The `while` and `for` tasks are provided to execute the loop body zero or more times sequentially. The definition of `while` can be found in Figure 7.

Loop Ports.

In sequential loop activities optional *loop ports* can be used to express cyclic data flow between consecutive iterations of the loop. See Figure 8 for an example. At the beginning of every iteration, data is flowing from the input port (through link 1) to the embedded task A. Additionally, data from the loop ports flows to task A over link 2. After all of the embedded tasks finished, one iteration is complete and link 3 is used to overwrite the contents of the loop port with data produced in task A. This data will be used in the next iteration (via link 2). If there are links to union ports (link 5 in this example), the data produced in A is appended to the collection at the linked union port. This data flow is repeated for every iteration. After the final iteration finished, link 4 is used to transfer data produced by A in the final iteration to the output port.

Number of iterations.

While the `while` task has a *condition* that controls how often the body is executed (see Section 3.4), the `for` task (not shown due to space limitations) has a special input port

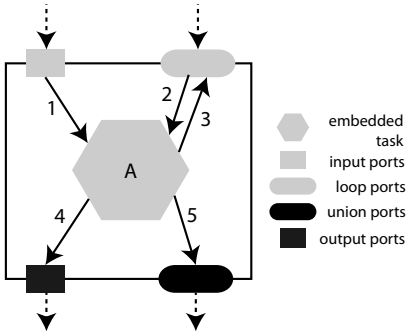

Figure 8: Data Flow in sequential loop tasks (`while`,`for`,`forEach`)

called a *loopCounter*. The loopCounter attributes (from, to, step) can either be set to fixed integer values or receive values produced by previously executed tasks. The value of the loop counter is initially assigned to the value specified at the *from* attribute and is increased by the value of *step* until it reaches the value of *to* or larger. The values of *to*, *from* and *step* are only evaluated once at the beginning of an invocation of the `for` task.

Output Ports.

An output port of a `while` or `for` task is assigned data (via a link) from a contained task. After completion of the `while` or `for` compound task, the output ports will be assigned the values specified by a link coming from task executions in the final iteration. Therefore, subsequent tasks can access only data produced in the last iteration through these output ports. If subsequent tasks need to access data produced by intermediate iterations, union ports have to be used. A union port can aggregate any data produced during iterations of the loop in a data collection, all that is needed is a data link (see Section 3.1.2) from an output port of a contained task to the union port.

3.3.4 The `forEach` task

The `forEach` compound task is similar to the `for` task except that in the `forEach` task there is an additional type of data input port, called `loopElement` port which receives a data collection over which the loop iterates sequentially. The `forEach` task is not shown here due to space limitations, but it operates very similar to the `parallelForEach` task shown later in this paper.

3.3.5 The `parallelFor` task

The `parallelFor` compound task (not shown due to lack of space) is similar to the sequential `for` task except that the `parallelFor` task can execute all of its iterations in parallel. This implies that there may not exist any data dependencies between different iterations of the body, therefore the `parallelFor` task does not provide any *loop ports*. Additionally, every output port of the `parallelFor` task has to be of a collection type (see Section 3.1.1) to accommodate the parallel production of data in the tasks iterations.

3.3.6 The `parallelForEach` task

The `parallelForEach` task (Figure 9) is similar to the `forEach` task with the difference that in `parallelForEach` all loop iterations can be executed simultaneously.

```
<parallelForEach name="name">
  <inputPorts>
    <inputPort name="name" type="type"/>*
    <loopElements>
      <loopElement name="name"
                   type="collection">+
    </loopElements>
  </inputPorts>
  <body>
    <task .../>+
  </body>
  <outputPorts>
    <outputPort name="name" type="type"/>*
  </outputPorts>
  <links>
    <link from="from" to="to" />*
  </links>
</parallelForEach>
```

Figure 9: The `parallelForEach` Task

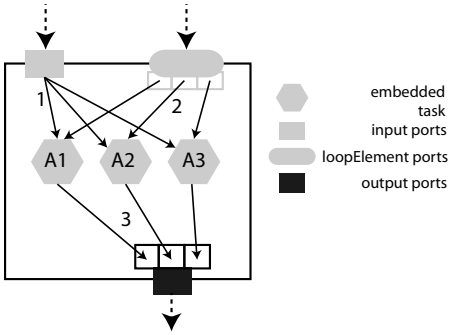

Figure 10: Data Flow in the `parallelForEach` tasks

It is assumed that the data input of any iteration is independent of data produced by other iterations of the same task. As in the `parallelFor` compound task, the `parallelForEach` compound task construct does not require the underlying workflow execution system to wait for the completion of every iteration before continuing the execution flow in every case; synchronization is only required if the correct execution of the data flow requires it, i.e. if a subsequent task requires all of the produced data to be available.

Loop Element Ports.

The `loopElements`-block in the `parallelForEach` task controls how often the loop body is executed. It encloses one or more loop element ports. In the case it contains one loop element port the behavior is the following: The `parallelForEach` loop concurrently iterates over each element of the collection linked to the loop element port. Linking the port to a task inside of the loop body results in a value based on the data type of the loop element port (without the first `collection/` identifier,see Section 3.1.2) and the iteration number at run time. In the case of more than one loop element port, the body is concurrently executed once per common element index of the collections referenced by the links connected to the ports. If the sizes of the collections do not match, the extra elements in the larger collections are ignored. This allows for *dot product* iteration strategies.

Output Ports.

The type of the output ports of a `parallelForEach` task must be a `collection` type (see Section 3.1.1). Each iteration writes some of its values determined by the data link connected to the port. The resulting collection is ordered, its j-th element is the data coming from the j-th iteration of the loop.

Figure 10 shows the usual data flow in a `parallelForEach` task. `A1`, `A2` and `A3` illustrate a single embedded task A in three parallel iteration instances, defined by a collection of size 3 given to the loopElement port of the `parallelForEach` task. Every iteration instance gets (via link 1) the same data coming from the input port. On the other hand, the collection in the loopElement port is split up, and every iteration i gets only the i-th element of the collection (via link 2). Link 3 then sets the j-th element of the collection produced in the output port to be the data produced by task A in iteration j.

3.4 Condition Expression in Compound Tasks

Operator	IWIR
=	=
! =	! =
>	>
>=	>=
<	<
<=	<=
and	and
or	or
not	!

Table 1: Conditional Operators in IWIR

To be able to evaluate the condition, as used in `if` and `while` tasks, a Boolean expression is used. Table 1 lists the Boolean operators that are allowed in IWIR workflows. For simplification reasons, IWIR limits the operands of a condition expression to values of type `boolean`, `double`, `integer` and `string`. The evaluation of the condition expression in IWIR is based on the XPath 1.0 specification [3], restricted to the parts applicable to conditional expressions in the IWIR language. To enable more straightforward and logical use of string values in IWIR conditions, we also added two exceptions to the `string`→`boolean` conversion, which we took from XPath 2.0 [2]. For the full specification of the condition expression evaluation in IWIR, refer to [9].

3.5 Properties and Constraints

Properties provide hints about the behavior of tasks, e.g. the expected size of the input data, the estimated computational complexity, the problem size, etc. Properties are referring to concepts that the underlying enactment system is *not forced* to take into account when executing a workflow.

Constraints *must be complied with* by the underlying workflow runtime environment, e.g. to use only a certain subset of a data collection, to flatten a nested collection, to minimize execution time, to provide a certain minimum amount of memory, to run on a certain specific host, architecture or DCI.

```
<properties>
  <property name="name" value="value" />*
</properties>
<constraints>
  <constraint name="name" value="value" />*
</constraints>
```

Figure 11: Properties and Constraints

In IWIR, properties and constraints are simple name-value pairs that can be defined by the user to provide additional information for the workflow runtime environment to optimize and steer the execution of workflow applications. Figure 11 shows how the `property` and `constraint` elements are specified. IWIR allows Properties and Constraints to be added to *data ports*, *atomic tasks* and *composite tasks*. Additionally, IWIR provides several built-in properties and constraints such as the `element-index` constraint that cuts down a data collection to a subset, the `flatten-collection` constraint that is able to flatten nested data collections and others. For a full explanation of these refer to [9].

4. EXAMPLES

In this section, we show examples of how to use IWIR to express common data distribution strategies featured in many different workflow languages as well as an example of workflow portability across two Grid Workflow Systems using language translators and IWIR as an intermediate language.

4.1 Dot Product

A dot product (one-to-one) data iteration strategy of data from two collections flowing into task `A` can be implemented in IWIR in the way seen in Figure 12. In this example, we have two data collections `collA` and `collB` as input to a *parallelForEach* task called *forEach1*. It contains an atomic task A which will be invoked $min(l(collA), l(collB))$ times, where $l(X)$ is the number of elements in the collection X. The i-th invocation of task A will be executed with the i-th

```
<parallelForEach name="forEach1">
  <inputPorts>
    <loopElements>
      <loopElement name="collA" type="collection/file" />
      <loopElement name="collB" type="collection/file" />
    </loopElements>
  </inputPorts>
  <body>
    <task name="A" tasktype="consumer">
      <inputPorts>
        <inputPort name="elementA" type="file" />
        <inputPort name="elementB" type="file" />
      </inputPorts>
      <outputPorts>
        <outputPort name="res" type="file" />
      </outputPorts>
    </task>
  </body>
  <outputPorts>
    <outputPort name="res" type="collection/file" />
  </outputPorts>
  <links>
    <link from="forEach1/collA" to="A/elementA" />
    <link from="forEach1/collB" to="A/elementB" />
    <link from="A/res" to="forEach1/res" />
  </links>
</parallelForEach>
```

Figure 12: An example for a dot product iteration strategy

```
<parallelForEach name="forEach1">
  <inputPorts>
    <inputPort name="collB" type="collection/file"/>
    <loopElements>
      <loopElement name="collA" type="collection/file"/>
    </loopElements>
  </inputPorts>
  <body>
    <parallelForEach name="forEach2">
      <inputPorts>
        <inputPort name="elementA" type="file"/>
        <loopElements>
          <loopElement name="collB" type="collection/file"/>
        <loopElements>
      </inputPorts>
      <body>
        <task name="A" tasktype="consumer">
          <inputPorts>
            <inputPort name="elementA" type="file"/>
            <inputPort name="elementB" type="file"/>
          </inputPorts>
          <outputPorts>
            <outputPort name="res" type="file"/>
          </outputPorts>
        </task>
      </body>
      <outputPorts>
        <outputPort name="res" type="collection/file"/>
      </outputPorts>
      <links>
        <link from="forEach2/elementA" to="A/elementA"/>
        <link from="forEach2/collB" to="A/elementB"/>
        <link from="A/res" to="forEach2/res"/>
      </links>
    </parallelForEach>
  </body>
  <outputPorts>
    <outputPort name="res"
                type="collection/collection/file"/>
  </outputPorts>
  <links>
    <link from="forEach1/collA" to="forEach2/elementA"/>
    <link from="forEach1/collB" to="forEach2/collB"/>
    <link from="forEach2/res" to="forEach1/res"/>
  </links>
</parallelForEach>
```

Figure 13: An example for a cross product iteration strategy

data element of both *collA* and *collB* as input.

4.2 Cross Product

A cross product (all-to-all) data iteration strategy of two collections flowing into task `A` can be implemented in IWIR in the way seen in Figure 13. In this example, we have two data collections `collA` and `collB` as input to a *parallelForEach* task called *forEach1*. It contains another *parallelForEach* task which in turn contains an atomic task A which will be invoked $l(collA) \times l(collB)$ times, where $l(X)$ is the number of elements in the collection X. The task *forEach1* is responsible for breaking up *collA* into its elements one by one, while *forEach2* breaks up *collB* into each of its elements and combines it with each element coming from *forEach1*. Subsequently, the atomic task A is invoked once for every combination of elements of *collA* and *collB*. The resulting data collection flowing out of *forEach1* is a collection of nesting level 2.

4.3 Case Study

To show the current state of the workflow transformation process using IWIR, we present a workflow known as *Image Registration*, its graphical representation in MOTEUR can be seen in Figure 14. Image Registration is a common medical image spatial alignment procedure.

The input contains images (scans) $\{I_0, I_1, I_2, \ldots\}$ of a patient acquired at different times. Because it is impossible to orient the patient precisely in the same position for each scan, the images are mis-aligned in space. The workflow automatically re-aligns the images by executing two alignment steps:

1. **Register to first**: aligns all images ($\{I_0, I_1, I_2, \ldots\}$) to the first one ($I_0$).
2. **Register to average**: aligns all resulting images to an average model to avoid any bias related to using the first image (I_0) as reference.

The **First** and **Average** activities are utility activities; *First* extracts the first image from the list, *Average* computes the mean of the list.

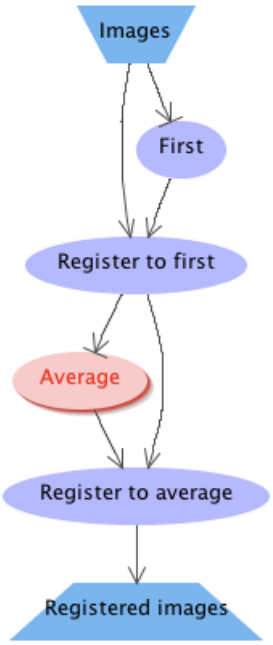

Figure 14: The Image Registration workflow in the graphical notation of MOTEUR

Figure 15 shows an excerpt from the Image Registration workflow in GWENDIA [8], the native workflow description language used by the MOTEUR system. The excerpt shows the activities, called *processors* in GWENDIA terminology, *First* (lines 1-6) and *Register to first* (lines 17-19). We can see that *First* has one input port of type string called `in` (line 2) and an output port of type string called `out` (line 3). In this case, the strings represent URLs of data files.

Activities in GWENDIA may receive inputs with different nesting levels; this is expressed using the concept of *port depth*. The depth of a port determines the number of nesting levels the input port will collect or the output port will produce before/after firing the activity. An input port depth of 0 denotes that the activity will fire for each received scalar value individually. An input port depth of n means that the activity will fire once for every nested structure of depth n received on the port.

We can see in Figure 15 (line 2) that the input port of the activity *First* has a port depth of "1" and will therefore consume a complete 1-dimensional array of strings (references to image files) and fire the activity once for each array of

```
...
 1:<processor name="First" >
 2: <in name="in" type="string" depth="1"/>
 3: <out name="out" type="string" depth="0"/>
 4: <beanshell>
 5: </beanshell>
 6:</processor>
 7:<processor name="Register to first">
 8: <in name="ref" type="string" depth="0"/>
 9: <in name="float" type="string" depth="0"/>
10: <out name="out" type="string" depth="0"/>
11: <iterationstrategy>
12:  <cross>
13:   <port name="ref"/>
14:   <port name="float"/>
15:  </cross>
16: </iterationstrategy>
17: <beanshell>
18: </beanshell>
19:</processor>
...
```

Figure 15: Excerpt from the Image Registration workflow in GWENDIA

strings flowing into the port. The output port of *First* (line 3) has a depth of 0, resulting in one string (file reference) per execution of *First*.

The second activity shown in Figure 15 shows that the input and output ports of *Register to first* all have a depth of 0 (line 8-10). Furthermore, we can see in the *iterationstrategy* block (lines 11-16) that the input ports *ref* and *float* are specified as being in a *cross* relationship. This means that the activity fires for every possible combination of items received on the input ports *ref* and *float*. In this concrete example workflow, *Register to first* receives all the images $\{I_0, I_1, I_2, \ldots\}$ on port *float* and the first image I_0 on port *ref*. This results in an execution of *Register to first* for the cross product combination of the two inputs: $\{(I_0, I_0), (I_1, I_0), (I_2, I_0), \ldots\}$, leading to the creation of a set of images (given as references to their locations). The rest of the workflow follows the same structure to execute the second alignment step.

Figure 16 shows the same portion of the workflow translated to IWIR. We can see that the input port of the task *First* was translated to the type `collection/file` (line 3). Since the GWENDIA workflow defined the port depth was as 1, we had to explicitly specify that this input port expects a collection of files to start the execution of the task in IWIR.

We can see that the GWENDIA task *Register to first* resulted in two tasks (*Register-to-first:cross* and *Register-to-first*) after the conversion to IWIR. As mentioned in Section 4.2, the cross product iteration strategy which is used in the *Register to first* activity can be expressed in IWIR using *ParallelForEach* tasks to split the incoming data collections (lines 10-41). From the port depths, the iteration strategy and the workflow structure in the GWENDIA workflow we can derive that one of the input ports will receive a collection of files. The other input port will receive just a single file. A cross product relation between these two ports can then be translated to IWIR by using a single *ParallelForEach* to split the incoming collection into its single entries. Each of these entries is then used (together with the one single file on the second port) for every execution of the *Register-to-first* task.

Finally, we can load the resulting IWIR workflow into the graphical user interface of the ASKALON workflow environment. This automatically triggers a conversion to its native

```
...
 1:<task name="First" tasktype="First">
 2: <inputPorts>
 3:   <inputPort name="in" type="collection/file"/>
 4: </inputPorts>
 5: <outputPorts>
 6:  <outputPort name="out" type="file"/>
 7: </outputPorts>
 8:</task>
 9:
10:<parallelForEach name="Register_to_first:cross">
11: <inputPorts>
12:  <inputPort name="ref" type="file"/>
13:  <loopElements>
14:   <loopElement name="float"
15:                type="collection/file"/>
16:  </loopElements>
17: </inputPorts>
18: <body>
19:  <task name="Register_to_first"
20:        tasktype="Register_to_first">
21:    <inputPorts>
22:     <inputPort name="ref" type="file"/>
23:     <inputPort name="float" type="file"/>
24:    </inputPorts>
25:    <outputPorts>
26:     <outputPort name="out" type="file"/>
27:    </outputPorts>
28:  </task>
29: </body>
30: <outputPorts>
31:  <outputPort name="out" type="collection/file"/>
32: </outputPorts>
33: <links>
34:  <link from="Register_to_first:cross/ref"
35:        to="Register_to_first/ref"/>
36:  <link from="Register_to_first:cross/float"
37:        to="Register_to_first/float"/>
38:  <link from="Register_to_first/out"
39:        to="Register_to_first:cross/out"/>
40: </links>
41:</parallelForEach>
...
```

Figure 16: Excerpt from the Image Registration workflow in IWIR

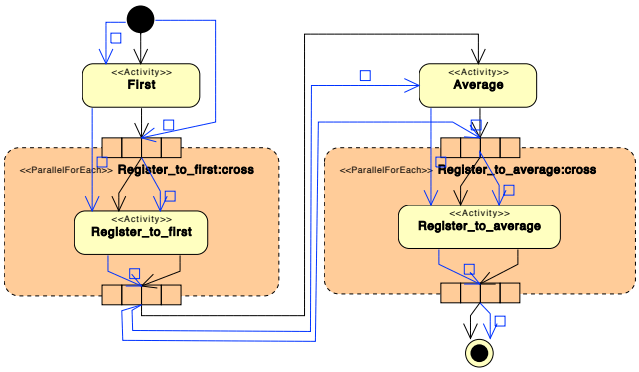

Figure 17: The workflow in the graphical UML-based notation of ASKALON

AGWL language and renders a graphical view in its UML-based interface, as can be seen in Figure 17.

5. IWIR IMPLEMENTATION

We have created an XML Schema specification for IWIR and developed IWIRtool, a Java-based implementation of an IWIR toolset for workflow system developers. IWIRtool is able to parse IWIR XML files and provides a Java Object representation enabling traversal and manipulation of the workflow. Additionally, it provides a simple Java API to enable the construction of IWIR workflows and the serialisation of IWIR Workflows as XML documents compliant

to the IWIR XML Schema. The IWIRtool is able to parse and evaluate the IWIR condition expressions and will validate IWIR documents for correctness in their control flow, data flow, data types and syntax. The IWIR XML Schema as well as the current version of IWIRtool, version 1.1.4, can be downloaded at `http://www.dps.uibk.ac.at/~kassian/shiwa/iwir/`.

IWIRtool is the basis for the development of language translators currently being implemented by the developers of five different well-known Grid workflow systems part of the SHIWA project: ASKALON, Moteur, P-GRADE, Triana and Pegasus. These language translators will eventually enable workflow portability across all of these workflow systems using IWIR.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented IWIR, an Interoperable Workflow Intermediate Representation designed to enable portability of workflows across numerous Grid workflow systems and originally written in different languages. The common IWIR representation enables the translation of workflows among n systems with $O(n)$ complexity and facilitates the integration of a new language into an IWIR-based interoperable environment with constant $O(1)$ complexity.

We have specified the IWIR language comprising atomic tasks, compound tasks including if, while, sequential for and parallel for statements, as well as different data types and data flow constructs to cover the abstract part of workflow applications. We provide an IWIR library, called IWIRtool, comprising a scanner, parser, and manipulation API. The IWIRtool is currently used to implement corresponding front-end and back-end translation solutions for many different well-known scientific workflow systems in the context of the EU FP7 SHIWA project [1]. Finally, we presented an example of how the current state of the translator solutions convert an existing workflow from one system to another using IWIR as intermediate representation. Future work will look into portability solutions for the concrete parts of workflow applications.

Acknowledgments

This work is partially funded by the European Union under grant agreement number 261585/SHIWA Project.

The authors would like to thank Andrew Harrison, Tristan Glatard, Miklos Kozlovsky, Gabor Hermann and Thomas Fahringer for their valuable input.

7. REFERENCES

[1] SHIWA: SHaring Interoperable Workflows for large-scale scientific simulation on Available DCIs. `http://www.shiwa-workflow.eu`, 2011.

[2] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0 (W3C Recommendation). Technical report, World Wide Web Consortium, January 2007.

[3] J. Clark and S. DeRose. XML Path Language (XPath) 1.0 (W3C Recommendation). Technical report, World Wide Web Consortium, 1999.

[4] W. B. Dobrusky and T. B. Steel. Universal computer-oriented language. *Commun. ACM*, 4:138–, March 1961.

[5] T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid workflow applications with AGWL: An abstract Grid workflow language. In *International Symposium on Cluster Computing and the Grid*. IEEE Computer Society Press, 2005.

[6] P. Kacsuk and G. Sipos. Multi-grid, multi-user workflows in the p-grade grid portal. *Journal of Grid Computing*, 3:221–238, 2005. 10.1007/s10723-005-9012-6.

[7] P. Missier, D. Turi, C. Goble, and et al. Taverna workflows: Syntax and semantics. In *IEEE International Conference on e-Science and Grid Computing*, Dec 2007.

[8] J. Montagnat, B. Isnard, T. Glatard, K. Maheshwari, and M. B. Fornarino. A data-driven workflow language for grids based on array programming principles. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, pages 7:1–7:10, New York, NY, USA, 2009. ACM.

[9] K. Plankensteiner, R. Prodan, T. Fahringer, J. Montagnat, and et al. Interoperable workflow intermediate representation. SHIWA Deliverable D6.1, December 2010.

[10] I. Taylor, M. Shields, I. Wang, and R. Rana. Triana applications within Grid computing and peer to peer environments. *Journal of Grid Computing*, 1(2), 2003.