Fine-Grain Interoperability of Scientific Workflows in Distributed Computing Infrastructures

Kassian Plankensteiner, Radu Prodan, Matthias Janetschek, Thomas Fahringer, Johan Montagnat, David Rogers, Ian Harvey, Ian Taylor, Ákos Balaskó, Péter Kacsuk

Journal of Grid Computing manuscript No.
(will be inserted by the editor)

Fine-Grain Interoperability of Scientific Workflows in Distributed Computing Infrastructures

Kassian Plankensteiner · Radu Prodan · Matthias Janetschek · Thomas Fahringer · Johan Montagnat · David Rogers · Ian Harvey · Ian Taylor · Ákos Balaskó · Péter Kacsuk

Abstract Today there exist a wide variety of scientific workflow management systems, each designed to fulfill the needs of a certain scientific community. Unfortunately, once a workflow application has been designed in one particular system it becomes very hard to share it with users working with different systems. Portability of workflows and interoperability between current systems barely exists. In this work, we present the fine-grained interoperability solution proposed in the SHIWA European project that brings together four representative European workflow systems: ASKALON, MOTEUR, WS-PGRADE, and Triana. The proposed interoperability is realised at two levels of abstraction: abstract and concrete. At the abstract level, we propose a generic Interoperable Workflow Intermediate Representation (IWIR) that can be used as a common bridge for translating workflows between different languages independent of the underlying distributed computing infrastructure. At the concrete level, we propose a bundling technique that aggregates the abstract IWIR representation and concrete task representations to enable workflow instantiation, execution and scheduling. We illustrate case studies using two real-workflow applications designed in a native environment and then translated and executed by a foreign workflow system in a foreign distributed computing infrastructure.

Keywords scientific workflow; interoperability; portability; intermediate representation; distributed computing; Grid computing

K. Plankensteiner · R. Prodan · M. Janetschek · T. Fahringer
Institute of Computer Science, University of Innsbruck, Technikerstr. 21a, 6020 Innsbruck, Austria, E-mail: {kassian.plankensteiner,radu.prodan,matthias.janetschek,thomas.fahringer}@uibk.ac.at

J. Montagnat
I3S lab, CNRS, Sophia Antipolis, France, E-mail: johan@i3s.unice.fr

D. Rogers · I. Harvey · I. Taylor
Cardiff University, Cardiff, UK, E-mail: {d.m.rogers,i.harvey,ian.j.taylor}@cs.cardiff.ac.uk

Á. Balaskó · P. Kacsuk
MTA SZTAKI, Budapest, Hungary, E-mail: {balasko,kacsuk}@sztaki.hu

1 Introduction

Currently, almost every scientific workflow development and execution system provides its own native input language designed to satisfy the needs of its specific target community. Workflow applications are specified in different systems at various levels of detail, sometimes hiding the underlying infrastructure, and sometimes exposing at least part of it. In most cases, however, workflow applications are hard-coded and therefore bound to the workflow system within which they have been developed. Running an existing workflow application in another system than the one in which it has been originally developed requires re-engineering and rebuilding it almost from scratch which is tedious and inefficient. This unfortunate situation makes the entire e-science workflow community fragmented, since sharing of existing workflow applications within and among domain-specific sciences to enhance synergies and reduce the time-to-solution is impossible.

Scientific workflows represent the experiments conducted by scientists. These experiments are usually data and computation intensive and often contain relatively simple control-flows and rules. Scientific workflow languages are therefore mostly targeted at modelling the data-flow between the individual workflow activities with a strong focus on efficient data processing and scheduling of computational units [30]. We therefore call these languages *data-flow oriented*. Languages like AGWL [15], GWENDIA [25], gUSE [19], SCUFL [23] and Triana Taskgraph [31] belong to the category of data-flow oriented scientific workflow languages.

Business workflows, on the other hand, are usually targeted at modelling the control-flow between individual business activities and describe the processes inside and between enterprises with a strong focus on implementing complex business rules. In contrast to the requirements of most scientific workflows, business workflows usually require the support for process integrity which includes transactions, rollback mechanisms and audits [18]. Because of their focus on modeling the control-flow, we call these languages *control-flow oriented*.

While the business process community has generally accepted the standardised, control-flow oriented Web Services Business Execution Language WS-BPEL [18] as their workflow language of choice, it has not been adopted by the scientific workflow community The creation of a single standard language for all users of scientific workflow systems is a difficult undertaking that will probably not succeed in being adopted by all communities given the heterogeneous nature of their fields and problems to solve.

To address this difficult problem, we present in this paper the *fine-grained interoperability (FGI)* framework developed as part of the SHIWA (SHaring Interoperable Workflows for large-scale scientific simulations on Available DCIs) European project [4] that brings together four representative European workflow systems: ASKALON from the University of Innsbruck, MOTEUR from the French National Center for Scientific Research (CNRS), WS-PGRADE from the Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SZTAKI), and Triana from the Cardiff University. The proposed interoperability is realised at two levels of abstraction: abstract and concrete. At the abstract level, we propose a generic *Interoperable Workflow Intermediate Representation (IWIR)* sufficient for describing workflows at a lower level of abstraction that is only processed by the existing workflow systems and not directly exposed to the human developer. Such a common representation shall be used as a common bridge for translating workflows between different languages independent of the underlying Distributed Computing Infrastructure (DCI). At the concrete level, we propose a bundling technique that aggregates the abstract IWIR representation and concrete task representations to enable workflow instantiation, execution and scheduling.

Such a solution based on 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. First, 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. Second, 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 experiment and the scientist's objectives (e.g. performance, reliability, cost). As an example, a workflow running with sensitive patient data may need to stay in a locally administered DCI to satisfy data privacy laws. Transferring the data to an external DCI would be against the law, therefore the workflow execution needs to be moved towards the data. Third, 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 transferred to, scheduled and executed in a different workflow system on-demand in the form of a common intermediate representation, which creates numerous optimization opportunities. Fourth, 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 front-end converter, while direct language-to-language translators require n front-end converters. Similarly, porting m interoperable workflows to a new DCI platform requires the development of one single IWIR-compliant back-end converter, while direct language-to-language translations would require again m back-ends, one for each workflow system. Therefore a solution based on an intermediate language reduces the effort of porting m workflow systems onto n distributed platforms from $m \cdot n$ to $m+n$. This is an important step to make the development of new workflow systems for multiple existing DCI infrastructures economically viable.

The paper is organised as follows. The next section discussed the related work. Section 3 presents the general FGI architecture based on the abstract and concrete separation of concerns described in Sections 4 and 5. Section 6 presents the IWIR bundle technology for packaging the interoperable abstract and concrete workflow and task representations. Section 7 presents information about implementing the proposed framework in the four pilot workflow systems: ASKALON, MOTEUR, WS-PGRADE, and Triana. Section 8 discusses the differences between BPEL and IWIR, and shows why IWIR is a more promising choice for enabling portability and interoperability between scientific workflow systems. Section 9 validates the interoperability framework using two real-world applications translated and run across multiple workflow systems. Section 10 concludes the paper and presents ideas for future work.

2 Related Work

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) [12] proposed in 1958 by Melvin E. Conway as a solution for making compiler development economically viable.

The GNU Compiler Collections GIMPLE [22] is a very successful example of a single intermediate representation. The GNU Compiler Collection (gcc) provides front-ends for converting C, C++, Objective C, Fortran, Java, Ada and Go to the GIMPLE representation. Additionally, there are already frontends for Cobol, Pascal, Mercury, Modula-2, Modula-3, VHDL, PL/I and UPC. In the back-end, gcc provides support for over 30 different target

architectures. Another example is Java bytecode [21], which can be generated from a wide range of source languages including C and Java. Software components called Java Virtual Machines can then execute the Java Bytecode on a large number of different target architectures. Intermediate Languages have also been researched in the domain of languages targeting parallel computing, e.g. HSSM Intermediate Language HIL [33], a universal intermediate representation for massively parallel software development.

Elmroth et al. present in [13] their investigations on interoperability aspects of scientific workflows, where they identify three different dimensions of the problem: *model of computation*, *workflow language* and *workflow execution environment*. They discuss the problems and challenges of interoperability of scientific workflow based on these dimensions, mainly focusing on the differences between local workflows and Grid workflows. Their work is mostly a survey of the current situation of workflow interoperability and an analysis of the concepts and semantics of different kinds of workflow languages. Their long-term objective is to achieve logical workflow interoperability in all dimensions. While their work investigates problems related to direct conversion and compatibility between the end-user languages, we provide a solution based on an intermediate language, reducing the integration effort significantly.

The XML Process Definition Language (XPDL) [34] is an XML language used to interchange business process definitions between different business workflow systems and tools. It has been designed to exchange both graphical representation and semantics of a business process, with the focus on the graphical representation and human interactions. XPDL has been standardised in 1998 by the Workflow Management Coalition (WfMC). It is often used as a serialisation format for Business Process Model and Notation (BPMN) [24] and has been especially designed to represent all concepts present in a BPMN diagram. In contrast to XPDL, we focus on scientific workflows.

The idea of separating the abstract from the concrete part of a workflow, as we do in our work, is rather common in scientific workflow systems, and therefore often already familiar to our target audience. One example is Pegasus [10], which is a framework for mapping scientific workflows onto distributed systems. It provides APIs for Java, Perl and Python for creating a DAX which is a description of an abstract workflow graph in XML format. In the DAX logical identifiers are used to reference files and jobs. On a DCI which utilises Pegasus the logical file identifiers are mapped to physical files using a Replica Catalog. A Transformation Catalog maps the logical task identifiers to executables also storing additional meta-information, while a Site Catalog tracks the compute resources of the DCI. Pegasus takes a DAX as input and uses the catalogs to create a concrete workflow that is executable on the DCI. Pegasus tries to achieve interoperability between different DCIs all running Pegasus by allowing workflow developers to specify abstract workflows which are then concretised on a given DCI. In contrast, our approach achieves interoperability between different workflow systems by providing means for the exchange of complete and partial workflows based on a commonly understood intermediate representation.

The interoperability scenario proposed in this paper is being researched and developed as part of the European FP7 SHIWA (SHaring Interoperable Workflows for large-scale scientific simulations on Available DCIs) [4] under the name of *fine-grained workflow interoperability*. The SHIWA project brings together four representative workflow systems: ASKALON from the University of Innsbruck, Moteur from the French National Center for Scientific Research (CNRS), WS-PGRADE from the Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SZTAKI), and Triana from the Cardiff University. The pilot applications come from the biomedical sciences and are provided by Charité – Universitätsmedizin Berlin and the Academic Medical Center of the University

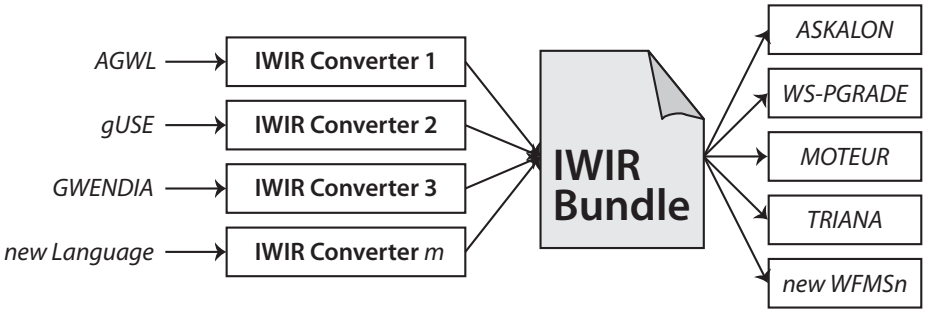

Fig. 1 Schematic fine-grained interoperability framework architecture.

of Amsterdam. Additional pilot applications are provided by four subcontractors. The second interoperability scenario researched by the SHIWA project is the coarse-grain workflow interoperability referring to the capability of nesting existing off-the-shelf workflow applications as black-box sub-workflows to be included as part of larger meta-workflows.

3 Architecture

Figure 1 displays the schematic architecture of the targeted fine-grained interoperability framework. To qualify for fine-grained interoperability and be part of the proposed open interoperable framework, 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, interoperability with the other systems is implicitly enabled.

We distinguish two parts of a workflow, corresponding to two levels of abstraction; abstract and concrete:

- The *abstract* part describes two generic aspects of a workflow that decouple its definition from the underlying implementation technology (e.g. legacy codes, Web services) and makes it portable across DCI platforms (e.g. gLite, Globus):
  1. the abstract input/output functionality of each workflow task (in terms of task type);
  2. the workflow-based orchestration of the computational tasks by defining the precedence relations in terms of control and data flow dependencies;
- The *concrete* part of a workflow application contains low-level information about its computational tasks' implementation technologies. This can mean a wide range of things such as how to execute a certain application on a certain machine, 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 DCI.

Figure 2 shows a graphical representation of the two layers that make the workflow a fully-specified executable application. The mapping of tasks from the abstract part of the workflow to the concrete computational entities on the target DCI can either be done at the time of workflow creation, or be handled by a scheduling 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

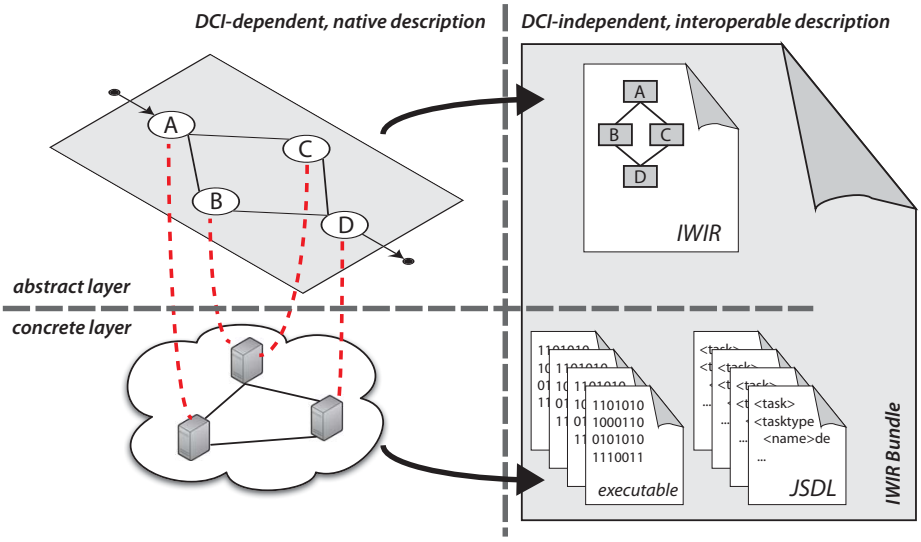

Fig. 2 Abstract and concrete layers in the fine-grained interoperability framework architecture. (color online)

our proposed architecture, the abstract and concrete part of an IWIR-compatible workflow are packaged in a single archive file called the *IWIR bundle* (see Section 6).

We designed IWIR to enable portability of workflows across different specification languages, workflow systems and DCIs. IWIR is a language enabling the portability of the abstract part of a workflow and 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 *data manipulation* constructs and therefore does not define direct ways to change data (such as integer operations or concatenation of strings), but rather provides means to powerfully *distribute* data to computational tasks that do the 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 us to the decision of creating an XML-based graph-based representation, mixing data-flows and an expressive set of sequential and parallel control structures.

The act of transforming a native workflow application to an IWIR workflow bundle can be broken down into the following steps (see Figure 2):

1. Convert the abstract part of the workflow to an *IWIR abstract workflow graph* representing the workflow logic expressed as an IWIR workflow document (see Section 4);
2. For each task type referenced in the IWIR-based graph representation, convert the concrete task implementation into a DCI-independent *concrete task representation (CTR)*, consisting of binary representations for the task as well as an explanation of how to invoke the task (see Section 5);
3. Create an IWIR bundle containing both the IWIR-based graph representation and all the CTRs, including appropriate meta-data information (see Section 6).

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

Fig. 3 IWIR document structure.

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

Fig. 4 Data flow `links` definition.

4 Abstract workflow interoperability

In this section we overview the main elements of the IWIR language, while a complete specification is provided in [28].

At the abstract level, each native workflow description document is translated into the intermediate *abstract IWIR workflow* representation. 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 3 shows the IWIR document structure consisting of the following constructs:

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

`IWIR xmlns` is the namespace of all IWIR XML tags and concepts. To be able to concentrate on the concepts rather than the notation, we use a global namespace declaration of `http://shiwa-workflow.eu/IWIR` here;

`IWIR wfname` is the workflow name which serves as an identifier;

`task` is the top-level task element of an IWIR workflow. This element can be a compound task or an atomic task and its signature defines the required input and output ports of the workflow and their data types. We present in Sections 4.5 and 4.6 a list of possible compound and atomic task constructs.

4.1 Data types

IWIR defines a set of simple data types modeled after the set of primitive data types of common programming languages, and a composite collection data type modeled after well-known array data structures. An IWIR data type identifier is based on XML schema data types and can be 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 (starting from 0 with 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.

4.2 Data flow

Data ports are connected to each other using the `link` construct (see Figure 4). Every composite task, and therefore every scope, has a `links` block containing all the data flow links in its scope defined using two attributes:

`links from` 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;

`links to` defines the destination of the data flow connection. This attribute is also 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.

Data flow links are not allowed to cross scopes (see Section 4.4) making every composite task self-contained with respect to its data flow.

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 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. A full specification of the particulars of these exceptions is given in [28]. 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 states otherwise. Generally, building a cyclic data dependency using link constructs is not allowed in an IWIR workflow, except in very specific cases.

4.3 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 Figure 4). A pure control flow link triggers after the given source of the link successfully finished its execution. In case of the source being a sequential loop task, the control flow link triggers therefore after the successful execution of the final iteration. For parallel loop tasks, the control flow link triggers 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 triggered. As in the case of data flow links, building a cyclic control dependency using link constructs is not allowed.

4.4 Scopes

In IWIR, data ports and tasks can only be referenced in certain regions of the workflow document called *scopes*. Every scope has a single `links` block. IWIR only allows a data

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

Fig. 5 `task` definition.

`link` to refer to data ports and tasks within the *current scope* consisting of the following elements (see Figure 4):

Current task represented by its name and all data ports (i.e. input, output, loop, loop counter, loop element, output) of the task containing the `links` block itself is an element of the current scope

Enclosed tasks represented by the names and all data ports of all first-level subtasks. The current scope does not extend to second-level tasks embedded into the first-level ones.

These strict scoping rules define an important IWIR principle of *self-contained* tasks providing a single point of entry (the input ports) and exit (the output ports). This ensures strong reusability since 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.

4.5 Atomic tasks

An atomic task is implemented by a single computational entity (e.g. executable, Web service, script), described using the task construct shown in Figure 5 and containing the following attributes:

`name` serves as an identifier for the task. Tasks must be organized in an IWIR workflow or a compound task which define a their scope (see Section 4.4). In the scope, the name of each task must be unique.

`type` describes the functional behavior and the interface of the task. A task type is an abstract description which can be implemented by different *task deployments* representing concrete implementations of computational entities deployed in a DCI (e.g. binary executable, script file, Web service instance). A task type can also refer to a sub-workflow and must be defined in a *type registry* before enactment. Task types shield the implementation details of task deployments from the IWIR representation and help enabling workflow interoperability across different DCIs. Locating and invoking of task deployments is done by an underlying runtime environment.

`inputPorts/outputPorts` enclose all the data ports of the task. The number and types of the input and output ports are determined by the chosen task type. The `link` construct (see Figure 4) is used to define the data flow between input and output ports of different tasks.

```
<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>
```

Fig. 6 `if` task definition.

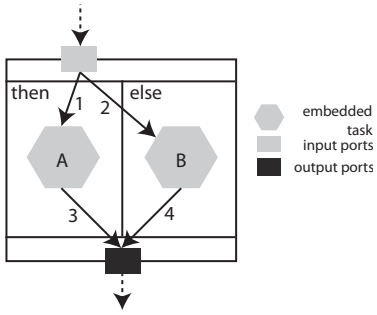

Fig. 7 Data flow definition in `if` task.

4.6 Compound tasks

A compound task encloses several atomic tasks and/or other compound tasks, including 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 compound task. Therefore, compound tasks are able to form separate self-contained scopes. We classify the compound tasks into two groups:

Basic compound tasks are sequential constructs similar to well known constructs in high-level languages such as `blockScope`, `if`, `while`, `blockscope`, `for` and `forEach`;

Parallel compound tasks are constructs that express parallel loops, i.e. loops in which there are no loop-carried data dependencies (`parallelFor` and `parallelForEach`).

4.6.1 `blockScope` task

The `blockScope` compound task (not shown here 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.

4.6.2 `if` task

The `if` compound task enables the conditional execution of the inner tasks, as described by its attributes (see Figure 6):

`condition` controls whether the `then` or the `else` branch is executed at runtime. For simplicity, IWIR limits the condition operands `boolean`, `double`, `integer` and `string` type values and supports most boolean operators. The expression evaluation is based on the XPath 1.0 specification [9]. To enable more straightforward and logical use of string values, we also added two exceptions to the `string`→`boolean` conversion inspired from XPath 2.0 [6]. For the full specification of the condition expression evaluation in IWIR, refer to [28].

`outputPorts` gathers using `link` constructs the output of tasks from both the `then` and from the `else` branch of the `if` task. This is necessary because it is generally unknown

```
<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>
```

Fig. 8 `while` task.

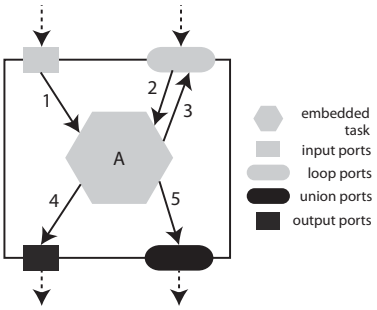

Fig. 9 Data flow definition in sequential loops.

at compile time which branch of the `if` task is going to be executed. If the `else` branch is omitted, a link from an input port of the if task to the output port needs to be created. Since for a given `if` task instance only one of the `then` or the `else` branches is executed, the links connecting task ports belonging to different branches are not allowed.

Figure 7 illustrates the usual data flow through the `if` construct. The data arrives at the input port and depending on the condition evaluation, either the `then` or the `else` branch is executed. Therefore, either link `1` or link `2` is 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`.

4.6.3 Sequential loops

The `while` and `for` tasks are provided to sequentially execute the loop body zero or more times. The definition of `while` is displayed in Figure 8 and consists of the following attributes:

`condition` is similar to the one defined by the `if` task and controls how often the `while` loop body is executed;

`loopPorts` are optional and are used to express cyclic data flow between consecutive in sequential loop iterations. Figure 9 shows an example in which 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 have 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 is used in the next iteration via link `2`. If there are links to *union ports* such as 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;

`loopCounter` is specific to the `for` task (not shown here due to space limitations) and not present in the `while` task that controls the repetition through the previously described `condition`. The `loopCounter` is initially assigned to the value specified at the `from` attribute and is increased by the value of `step` until it reaches the `to` attribute or larger. The `from`, `to`, and `step` attributes can either be set to fixed integer values or receive values produced by previously executed tasks and are only evaluated once at the beginning of an invocation of the `for` task;

`outputPorts` are assigned via a link from the embedded tasks after the last iteration of the `while` or `for` compound task. 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 that aggregate any data produced during iterations of the loop in a data collection need to be used. This is specified using a data link (see Section 4.2) from an output port of a contained task to the union port.

The `forEach` compound task is similar to the `for` task except that there is an additional type of data input port called `loopElement` port which receives a data collection over which the loop sequentially iterates. The `forEach` task operates similarly to the `parallelForEach` task shown later on in this paper.

4.6.4 Parallel loops

The `parallelFor` compound task is similar to the sequential `for` task except that it can execute all 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 4.1) to accommodate the parallel production of data in the tasks iterations.

The `parallelForEach` task is similar to the `forEach` task with the difference that all loop iterations can be simultaneously executed. As for `parallelFor`, `parallelForEach` 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, for example if a subsequent task requires all of the produced data to be available. A `parallelForEach` task is described by the following attributes (see Figure 10):

`loopElements` block encloses one or more loop element ports and controls how often the loop body is executed. In the case it contains one loop element, 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 runtime value based on the data type of the loop element port without the first `collection/` identifier (see Section 4.2) and the iteration number. In the case of more loop element ports, the body is concurrently executed once per common element index of the collections referenced by the links connected to these ports. If the collections sizes do not match, the extra elements in the larger collections are ignored which allows for *dot product* iteration strategies;

`outputPort` must be of `collection` type (see Section 4.1), where each iteration writes its output determined by the data link connected to the port. The resulting collection is ordered such that the `j`-th element represents the data coming from the `j`-th loop iteration.

```
<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>
```

Fig. 10 `parallelForEach` task.

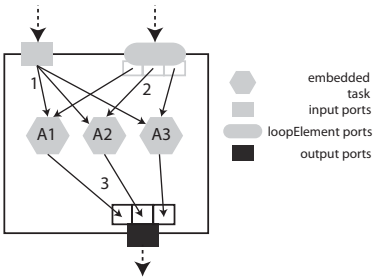

Fig. 11 Data flow definition in the `parallelForEach` task.

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

Fig. 12 Properties and Constraints

Figure 11 shows the usual data flow in a `parallelForEach` task, where `A1`, `A2` and `A3` are embedded tasks representing three parallel iteration instances defined by a collection of size three in `loopElement` port. Every iteration instance gets via link `1` the same data coming from the input port. Afterwards, the collection in the `loopElement` port is split up and every iteration `i` receives the `i`-th collection element via link 2. Finally, link 3 sets the `j`-th element of the collection produced in the output port to be the data produced by task `A` in iteration `j`.

4.7 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, on the other hand, must be complied with by the underlying workflow runtime environment, for example 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, or to run on a certain specific host, architecture or DCI.

In IWIR, properties and constraints are simple name-value pairs defined by the user to provide additional information to the workflow runtime environment for optimizing and steering the execution of workflow applications. As shown in Figure 12, IWIR allows `property` and `constraint` elements 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 the `producedAs/consumedAs` constraints that cover data pipelining and streaming.

```
<task name="Task_A" tasktype="org.example.exampleTask">
  <inputPorts>
    <inputPort name="in1" type="file"/>
  </inputPorts>
  <outputPorts>
    <outputPort name="out1" type="file"/>
  </outputPorts>
</task>
```

Fig. 13 An example task called Task_A and its signature in IWIR

5 Concrete workflow interoperability

An IWIR abstract workflow graph contains information about the precedence relations between computational tasks, their input/output signature, and the data flow between them. Being concerned only with the abstract part of a workflow, it does not contain information about the computational tasks themselves. To become a fully defined executable application, an FGI-compatible workflow needs a description of the concrete part of the workflow (alongside the abstract one) too. In this section, we summarise the most important aspects of the concrete workflow interoperability solution, while a detailed specification is given in [27].

The concrete part of an FGI-compatible workflow is given by a set of DCI-independent *concrete task representations* (CTR) for each task type contained in its IWIR abstract workflow graph. A CTR consists of two parts:

1. a set of files representing the computational task and its dependencies;
2. a JSDL template file describing how to invoke the computational task.

The first part can be fulfilled by providing for each task one or more executable files for each platform together with the library dependencies. While in the future we will endorse the usage of a wider range of technologies for representing concrete tasks, such as Web Archives (war) for Web Services, Virtual Machine images equipped with all necessary tools and libraries as well as executables for multiple architectures, we settled for statically linked Linux x86 binaries and shell scripts as a first proof-of-concept. Using wrapper scripts we can already invoke almost any type of concrete task this way.

The second part is based on the *Job Submission Description Language (JSDL)* [5], which is an extensible XML specification standardized by the Global Grid Forum in 2005. JSDL standardizes among others ways to describe job names, descriptions, resource requirements, execution limits, file staging, execution command, command-line arguments and environment variables, thus making it a good match to describe the invocation and resource requirements of a CTR in our architecture. A fully specified JSDL document contains concrete instantiated values for all of these fields (e.g. URLs pointing to intermediate data only existing during runtime). Therefore, using fully defined JSDL documents as a generic way to describe how to invoke CTRs in our proposed framework is not directly possible. For this reason, we abstract from fully defined JSDL documents by using placeholder tokens wherever there is information that can only be known, and therefore also only be instantiated, at runtime of a CTR. We call such a JSDL document containing placeholders a *JSDL template* document.

To be able to associate the placeholders to required data, we need to define a clear connection between the abstract signature of a CTR (IWIR task) and the placeholders contained in the JSDL Template. For example, the task `Task_A` shown in Figure 13 has one input port `in1` and one output port `out1`, both of the data type `file`. To execute a CTR implementing

this task, we could use a JSDL description such as the one given in Listing 1. Lines 3 – 16 define the application to be executed and the binary file to be staged in for invocation. Lines 17 – 22 define the data staging for input port `in1` and lines 23 – 28 define the data staging for output port `out1`. The input data staging description contains a concrete fixed URL in line 14. This URL references the location of the file provided to the input port of the concrete task invocation. To be able to use this JSDL description as a general description of how to invoke our CTR, we need to replace this concrete data with placeholders, such as in line 26, thereby creating a JSDL Template. By using the name of the corresponding ports in the placeholder (i.e. `<PLACEHOLDER_FILESERVER_in1/>` instead of the URL in line 14) we are able to establish the required logical connection to the information contained in the IWIR abstract task type signature.

Listing 1 JSDL description for the task in Figure 13

```
1  </JobDefinition> ...
2    <JobDescription>
3      <JobIdentification><JobName>exampleTask</JobName></JobIdentification>
4      <Application>
5        <ApplicationName>org/example/exampleTask</ApplicationName>
6        <jsdl-posix:POSIXApplication_Type>
7          <jsdl-posix:Executable>exampleTask</jsdl-posix:Executable>
8          <jsdl-posix:Output>std.out</jsdl-posix:Output>
9        </jsdl-posix:POSIXApplication_Type>
10     </Application>
11     <DataStaging>
12       <FileName>exampleTask</FileName>
13       <Source>
14         <URI>http://source.host:8080/getFile?path=1722/exampleTask</URI>
15       </Source>
16     </DataStaging>
17     <DataStaging>
18       <FileName>inputFile1.txt</FileName>
19       <Source>
20         <URI>http://source.host:8080/getFile?path=1722/inputs/inp1.txt</URI>
21       </Source>
22     </DataStaging>
23     <DataStaging>
24       <FileName>outputFile.txt</FileName>
25       <Target>
26         <URI><PLACEHOLDER_FILESERVER_out1/></URI>
27       </Target>
28     </DataStaging>
29   </JobDescription>
30 </JobDefinition>
```

We identify three different ways to create a JSDL template document during the conversion of a native workflow to the IWIR bundle format:

- *Automatic creation* for workflow systems whose native language contains invocation description in a form that is expressive enough to be converted to a JSDL document using an automatic converter;
- *Semi-automatic creation* for workflow systems that use RSL, xRSL or JDL as submission language for which a JSDL translator [27] can be used to semi-automatically create a JSDL template job description;
- *Manual creation* of the JSDL description by the workflow developer based on a given generic JSDL template using any text editor.

6 IWIR bundles

An IWIR bundle is a package containing both the IWIR abstract workflow graph and at least one CTR for each task type used. Additionally, the bundle contains metadata information describing the workflow and a mapping from abstract task types to CTRs. In other words, an IWIR bundle can be understood as a self-contained interoperable workflow, described in a common representation and containing all of the information and data required to execute the contained workflow on any FGI-compatible workflow execution system.

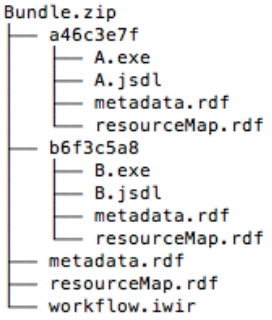

Fig. 14 Example IWIR bundle file structure.

IWIR bundles use the SHIWA CGI bundle file format and metadata framework, as specified in [17]. This open format reuses well-supported and widely deployed specifications based on the Resource Description Framework [2] (RDF), specifically the Simple Knowledge Organization System [3] (SKOS) and the Open Archive Institute's Object Reuse and Exchange [1] (ORE) vocabularies that simplify interoperability and integration with third party applications and projects. Exploiting RDF, ORE and SKOS provides a coherent framework for future-facing workflow reuse through the ability to aggregate, describe and infer relationships between resources.

Technically, an IWIR bundle is a compressed file (ZIP) containing the required files in a directory hierarchy. The IWIR workflow document is contained at the top-level directory, while each CTR is contained in a separate subdirectory named using a generated universally unique identifier (UUID). As required by the SHIWA CGI bundle format specification, each of these directories additionally contains the following two files:

`resourceMap.rdf` aggregates together a collection of files relating to the concept element, ranging from descriptive metadata files to raw data. In the scope of ORE, these files are known as aggregated resources and together with the resource map form an aggregation of the concept element. In the context of IWIR bundles, this file provides a list of all of the contents in a CTR (if in a subdirectory of the bundle), and/or provides a list of all CTRs and the IWIR workflow contained in the bundle (if on the top level of the bundle);

`metadata.rdf` is referenced by the resource map and contains all the key metadata information relating to the aggregation described in the resource map. In the context of IWIR bundles, the most important information contained here is the CTRs IWIR task type and its signature (if in a subdirectory), and/or the workflow signature (if on the top level).

Figure 14 shows the structure of a simple IWIR bundle describing a workflow containing two IWIR task types: `shiwa.fgi.example.A` and `shiwa.fgi.example.B`. The hierarchical nature of the bundle can easily be seen, each CTR being located in a subfolder based on a generated UUID. The workflow is located in the root of the bundle stored using its IWIR definition file (`workflow.iwir`). Nested below are the CTRs for the two IWIR task types, each containing both the executable file and the JSDL template file. Listing 2 shows part of the `resourceMap.rdf` file of the CTR in subdirectory `a46c3e7f`. Lines $3-6$ show the ORE aggregation that lists the contents of the CTR, including the metadata definition file `metadata.rdf`. The two most important functions of this file are the mapping of the CTR to a given IWIR task type as referenced in the IWIR workflow (`shiwa.fgi.example.A`, as

shown in Listing 3, line 5) and the reference to the JSDL template file describing the CTR invocation (see line 4 in Listing 3).

Listing 2 Excerpt from resourceMap.rdf describing the CTR of Task A

```
<rdf:Description rdf:about="aggr/">
  <ore:aggregates rdf:resource="A.jsdl"/>
  <ore:aggregates rdf:resource="A.exe"/>
  <ore:aggregates rdf:resource="metadata.rdf"/>
  <rdf:type rdf:resource="http://www.openarchives.org/ore/terms/Aggregation"/>
</rdf:Description>
```

Listing 3 Excerpt from metadata.rdf describing the CTR of Task A

```
...
<rdf:Description rdf:about="urn:uuid:a46c3e7f">
  ...
  <shiwa:definition rdf:resource="A.jsdl"/>
  <shiwa:tasktype>shiwa.fgi.example.A</shiwa:tasktype>
  ...
</rdf:Description>
...
```

7 Implementation

We have implemented a number of tools to help developers to integrate their respective workflow systems into our proposed architecture and avoid duplicated efforts:

IWIR tool parses IWIR XML files and provides a Java object representation enabling traversal and manipulation of the workflow. We have created an XML schema for IWIR and implemented a Java-based toolset to support workflow system developers in generating and manipulating IWIR documents as required by their language translators. Additionally, the tool provides a simple API that enables easy and correct construction and serialisation of IWIR workflows as XML documents compliant to the schema. Parsing and evaluation of the IWIR conditional expressions is supported too. The tool is able to validate IWIR documents for correctness in their control flow, data flow, data types and syntax when parsing or creating IWIR documents;

JSDL template creation tool takes as input a JSDL document and a corresponding IWIR task signature, analyses them and, with the help of the user, creates a JSDL template ready for inclusion in an IWIR bundle;

IWIR bundle tool parses an IWIR bundle and provides a simple API to access all the contained data;

IWIR bundle creation wizard guides the user through the manual creation of a workflow bundle, if full automation is not already provided by the workflow system integration. To achieve this, it will first ask the user for the IWIR document and parse it to obtain the contained IWIR task types. Then, it requests for every task type a JSDL template and the binaries to build the required CTRs. Finally, it requests for meta-data information like workflow name, description and dependencies, before creating a complete workflow IWIR bundle.

More information about these tools is presented in [27]. These tools have been employed by the four pilot workflow systems [15, 16, 19, 31] to create IWIR translation tools and interoperability plugins that fully integrate them (and their native languages) into our FGI architecture.

8 Discussion on BPEL

The Web Services Business Process Execution Language (WS-BPEL, or BPEL in short) [18] is a widely accepted, standardised language based on XML. It is designed for specifying the behaviour of executable as well as abstract business processes whose activities are web services. BPEL processes are exposed as web services themselves. The language incorporates standards like WSDL [8] for the specification of messages and web service endpoints, and XML schema types for the definition of variable types.

BPEL was introduced in 2002 by IBM, BEA Systems and Microsoft and standardised by the Organization for the Advancement of Structured Information Standards (OASIS) in 2004. Today there exist a lot of commercial (e.g. Oracle BPEL Process Manager, IBM WebSphere Process Server and Microsoft BizTalk Server) and open-source (e.g. ActiveBPEL and Apache ODE) business process execution engines which comply with the BPEL language specification, but also extend BPEL with proprietary extensions. Designed as a language for the description of business processes, BPEL is targeted at modelling the control-flow between individual business activities with a strong focus on implementing complex business rules. BPEL provides business process related features such as the support for process integrity including transactions, rollback mechanisms and audits [18]. BPEL is an imperative, control-flow oriented, Turing-complete language. Data exchange is based on globally shared variables, managed by a central entity. Variables in BPEL are mutable meaning that any task in between the definition and the use of a variable can potentially manipulate the value of the variable.

Since BPEL is the only workflow language in wide use today that was standardised by a standards body, there is a need to take a closer look at the potential of utilising BPEL to help in achieving interoperability and portability between existing scientific workflow systems before proposing a new language like our proposed Interoperable Workflow Intermediate Representation (IWIR).

8.1 Using BPEL as intermediate language for portable workflow exchange

Investigations on using BPEL as intermediate representation to enable fine-grained scientific workflow interoperability led us to the conclusion that there are several reasons why BPEL is not a good candidate:

- In BPEL the abstract part and the concrete part of a workflow are tightly coupled,
- BPEL is control-flow oriented whereas the majority of scientific workflow languages are data-flow oriented,
- BPEL needs to be extended to meet the requirements.

8.1.1 Tight coupling of the abstract part and the concrete part of a workflow

In BPEL the *abstract* and the *concrete* part of a workflow are tightly coupled since the specification of the process logic directly refers to WSDL-operations and also the message types are usually defined directly in WSDL. Moreover, WSDL only supports web services and the web service endpoints are usually hard-coded in WSDL. To be able to flexibly use BPEL as intermediate language fo portable workflows we would want to separate the *abstract* and the *concrete* parts and make each part replaceable.

One solution to this problem would be to use *abstract BPEL* [18] which would allow us to omit WSDL specific details during design time. However, this means that we also omit

information about the message types and the operation that is referenced by a workflow task. For our intended use this is not practicable since this information is required to match suitable concrete task representations to a given abstract workflow task.

Another possible solution would be *BPEL light* [26], which addresses the tight coupling of the abstract and concrete part. BPEL light completely disposes of WSDL and aims at only specifying message exchange patterns, which then need to be matched to arbitrary interface descriptions at runtime. Here we also have the problem of missing message types and operations in the abstract part and therefore a sub-optimal solution to our problem.

Furthermore, both of these solutions would additionally eliminate the advantage of being standardised languages overseen by a standards body.

8.1.2 Mismatch between control-flow oriented and data-flow oriented languages

IWIR as an intermediate language is targeted at scientific workflows, the majority of which are data-flow oriented. A data-flow oriented workflow [30] is modelled by a graph. Its nodes represent activities, the majority of its edges represent data-flow between activities. Each task has input and output ports where the input ports consume data and the output ports produce data. Data produced by an output port is forwarded through outgoing edges to the input ports of subsequent activities. In other words, variables in a data-flow oriented workflow are only locally visible and are immutable. This guarantees that no variable is referenced unexpectedly and there is no access conflict in parallel execution. Furthermore, this also allows to embody a functional programming style, which assumes the side-effect-freeness of the workflow activities. With this assumption failures can be handled in a simple and greedy way by re-executing the failed task. Whether a task can be executed is mainly dependent on the availability of the data represented by the incoming edges. Parallel execution of activities is mostly managed implicitly by the scheduler based on data-dependencies between activities.

Using BPEL as intermediate language would force most scientific workflow system developers to transform a data-flow oriented language to a control-flow oriented language and vice-versa. A control-flow oriented workflow [30] is also modelled by a graph with nodes representing activities. However, in control-flow oriented workflows, edges usually represent the explicit control-flow between activities. Whether a task can be executed or not is explicitly specified by the control-flow edges. Parallel execution of independent tasks always has to be specified explicitly and data is transferred between activities using explicitly defined shared variables. These variables are usually global and mutable. If users are not extremely careful, this can lead to access conflicts and race conditions in parallel execution and requires initialisation before the first use. Using globally shared variables requires additional effort during workflow creation and renders the handling of failures much more complex because the values of variables need to be considered when compensating the impact of the failure. This requires the use of sophisticated compensation mechanism.

As we can see, control-flow oriented languages exhibit different syntax and semantics than data-flow oriented languages. This leads to a syntactic and semantic mismatch. Elmroth et al. in [13] argue that functionality present in one style but missing in the other style requires simulation of functionality with availabe primitives resulting in increased complexity and a greatly increased potential for errors. Implementation of a two-way conversion would be a complex and cumbersome task due to this mismatch. To demonstrate the mismatch between BPEL and data-flow oriented scientific workflow languages, we want to give a simple example. A feature often found with scientific and other data-intensive workflow languages is data pipelining and streaming. In data-flow oriented languages this feature is aimed at

improving efficient execution over large data collections. Without data pipelining a data collection needs to be completely generated before it can be passed to subsequent consuming activities. With data pipelining individual elements of the data collection can already be passed to subsequent consuming activities before the whole data collection is complete. In data-flow oriented workflow languages, data pipelining can be achieved by simply tagging the relevant data link with a particular property (e.g. `producedAs/consumedAs` in IWIR). BPEL does not have an explicit support for this feature, it can therefore only be achieved by adding consecutively nested `forEach` constructs (see [30]), simulating the pipeline. When only converting from a scientific workflow language to BPEL this would still be acceptable, but for full integration we also need a conversion in the opposite direction. In this case we would be required to apply some form of pattern matching to figure out if a given set of nested `forEach` constructs implement data pipelining or if they just represent nested loops. In our opinion such a disadvantage would be detrimental to the adoption of the intermediate workflow language and render the implementation of language converters unnecessarily complex for a large majority of the targeted user base.

Additionally, BPEL was never designed as an intermediate language but to support the *programming in the large* paradigm [11]. For this purpose BPEL incorporates a large set of constructs which makes it Turing-complete. The problem with such a large feature set is that the implementation of a converter supporting it (especially a backward converter) is a complex and cumbersome task. In this respect XPDL [34] may be a better candidate since it was designed as an intermediate language for business workflows right from the start in contrast to BPEL, but XPDL is still a control-flow oriented language designed for business processes and therefore it suffers from the same disadvantages as depicted above. Moreover, it is more focused on graphical representation and human interactions.

We want to encourage scientific workflow communities to integrate their systems into our proposed fine-grained interoperability landscape by creating forward and backward converters. Therefore the intermediate language needs to be as simple and familiar and as closely related to the majority of scientific workflow languages as possible.

8.1.3 Proprietary extensions

Using BPEL as intermediate language would require us to find, implement and combine BPEL extensions to cover all of the requirements and peculiarities associated with scientific workflows. The BPEL standard does provide extension constructs that allow for extensibility, and every BPEL workflow using these constructs will still be a valid and standard-compliant workflow. However, the syntax and semantics of extensions are, by definition, not part of the BPEL specification and therefore the syntax and semantics of such workflows are also no longer purely defined by the BPEL standard. Furthermore, extensions add to the complexity of a BPEL workflow that uses them. It is therefore rather obvious that most of the advantages of BPEL being a standard are lost when adding multiple extensions. Moreover, extensions add complexity which counteracts with our goal of providing a simple intermediate workflow language that has a chance of being adopted in practice.

8.2 BPEL in the context of IWIR

As mentioned in Section 8.1.2, BPEL was never intended as an intermediate language but as an execution language. Therefore we see BPEL, when it is used as a scientific workflow language, as yet another language that should be translated to and from IWIR. Many of the

BPEL constructs can be explicitly converted to IWIR. The constructs that can not currently be explicitly converted are mostly business-process-related such as exceptions and out-of-band messages and global variables. Nevertheless, we are still able to incorporate these constructs in an IWIR workflow as black boxes wrapped into concrete tasks. In fact, the creation of IWIR converters for a suitable BPEL sub-set for scientific workflow applications is already planned as future work.

9 Experimental case studies

In this section, we first show examples of how to use the IWIR language to express common data distribution strategies featured in many different workflow languages. Then, we illustrate two case studies on how the FGI solution proposed in this paper enables workflow portability across multiple systems using IWIR language translators at the abstract level and the IWIR bundles at the concrete level.

9.1 Dot and cross products

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 Listing 4. In this example, we have two data collections `collA` and `collB` as input (lines $4-5$) to a `parallelForEach` task called `forEach1`. This compound task contains an atomic task `A` (lines $9-17$) which will be invoked $\min(l(\texttt{collA}), l(\texttt{collB}))$ times, where $l(\texttt{X})$ is the number of elements in the collection `X`. The *i*-th invocation of task `A` will be executed with the *i*-th data element of both `collA` and `collB` as input (lines $23-24$).

Listing 4 Dot product iteration example.

```
1  <parallelForEach name="forEach1">
2    <inputPorts>
3      <loopElements>
4        <loopElement name="collA" type="collection/file" />
5        <loopElement name="collB" type="collection/file" />
6      </loopElements>
7    </inputPorts>
8    <body>
9      <task name="A" tasktype="consumer">
10       <inputPorts>
11         <inputPort name="elementA" type="file" />
12         <inputPort name="elementB" type="file" />
13       </inputPorts>
14       <outputPorts>
15         <outputPort name="res" type="file" />
16       </outputPorts>
17     </task>
18   </body>
19   <outputPorts>
20     <outputPort name="res" type="collection/file" />
21   </outputPorts>
22   <links>
23     <link from="forEach1/collA" to="A/elementA" />
24     <link from="forEach1/collB" to="A/elementB" />
25     <link from="A/res" to="forEach1/res" />
26   </links>
27 </parallelForEach>
```

A *cross product* (all-to-all) data iteration strategy of two collections flowing into task `A` can be implemented in IWIR as displayed in Listing 5. In this example, we have two data collections `collA` and `collB` as input (lines 3, 5) to a `parallelForEach` task called `forEach1`. This compound task contains another `parallelForEach` task (lines $9-35$) which contains an atomic task `A` (lines $17-25$) invoked $l(\texttt{collA}) \cdot l(\texttt{collB})$ times, where $l(\texttt{X})$ is the number of elements in the collection `X`. These two nested loops that iterate over the collection elements compute the cross product that is represented as a collection of nested level 2 as an outport of `forEach1` (line 38).

Listing 5 Cross product iteration example.

```
1  <parallelForEach name="forEach1">
2    <inputPorts>
3      <inputPort name="collB" type="collection/file"/>
4      <loopElements>
5        <loopElement name="collA" type="collection/file"/>
6      </loopElements>
7    </inputPorts>
8    <body>
9      <parallelForEach name="forEach2">
10       <inputPorts>
11         <inputPort name="elementA" type="file"/>
12         <loopElements>
13           <loopElement name="collB" type="collection/file"/>
14         <loopElements>
15       </inputPorts>
16       <body>
17         <task name="A" tasktype="consumer">
18           <inputPorts>
19             <inputPort name="elementA" type="file"/>
20             <inputPort name="elementB" type="file"/>
21           </inputPorts>
22           <outputPorts>
23             <outputPort name="res" type="file"/>
24           </outputPorts>
25         </task>
26       </body>
27       <outputPorts>
28         <outputPort name="res" type="collection/file"/>
29       </outputPorts>
30       <links>
31         <link from="forEach2/elementA" to="A/elementA"/>
32         <link from="forEach2/collB" to="A/elementB"/>
33         <link from="A/res" to="forEach2/res"/>
34       </links>
35     </parallelForEach>
36   </body>
37   <outputPorts>
38     <outputPort name="res" type="collection/collection/file"/>
39   </outputPorts>
40   <links>
41     <link from="forEach1/collA" to="forEach2/elementA"/>
42     <link from="forEach1/collB" to="forEach2/collB"/>
43     <link from="forEach2/res" to="forEach1/res"/>
44   </links>
45 </parallelForEach>
```

9.2 Image registration workflow

To illustrate show the workflow translation process using IWIR, we present a case study that uses an *image registration* workflow performing a common medical image spatial alignment

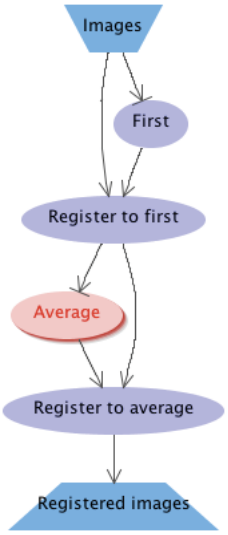

Fig. 15 Image registration workflow in MOTEUR. (color online)

Listing 6 Image registration workflow excerpt in GWENDIA.

```
1  ...
2  <processor name="First" >
3   <in name="in" type="string" depth="1"/>
4   <out name="out" type="string" depth="0"/>
5   <beanshell>
6   </beanshell>
7  </processor>
8  <processor name="Register to first">
9   <in name="ref" type="string" depth="0"/>
10  <in name="float" type="string" depth="0"/>
11  <out name="out" type="string" depth="0"/>
12  <iterationstrategy>
13   <cross>
14    <port name="ref"/>
15    <port name="float"/>
16   </cross>
17  </iterationstrategy>
18  <beanshell>
19  </beanshell>
20 </processor>
21 ...
```

procedure. The workflow has been originally programmed in MOTEUR as displayed in Figure 15. 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 misaligned in space. The workflow automatically realigns the images in two alignment steps:

- `Register to first` aligns all images ($\{I_0, I_1, I_2, \ldots\}$) to the first one ($I_0$);
- `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 that extracts the first, respectively compute the mean image from the list.

Listing 6 shows an excerpt from the workflow in GWENDIA [25], the native workflow description language used by the MOTEUR system. The excerpt contains the activities (*processors* in GWENDIA terminology) `First` (lines 1 – 6) and `Register to first` (lines 17 – 19). We can observe that `First` has one input (`in` – line 2) and one output port (`out` – line 3) of type string representing data file URLs. Activities in GWENDIA may receive inputs with different nesting levels 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 triggering the activity. An input port depth of 0 denotes that the activity will trigger for each individual scalar value received. An input port depth of n means that the activity will trigger once for every nested structure of depth n received on the port. In Listing 6 (line 2), the activity `First` has a port depth one and will therefore consume and trigger once for each element of a one-dimensional array of strings (references to image files). The output port of `First` (line 3) has a depth of zero, resulting in one string (file reference) per execution of `First`. The second activity in Listing 6 shows that all the input and output ports of `Register to first` have a depth of zero (lines 8 – 10). Furthermore, we can observe that the input ports `ref` and `float` are specified in a `cross iterationstrategy`

(lines 11 – 16) and will trigger the activity for all possible combinations. In this particular workflow, `Register to first` receives all the images $\{\mathtt{I}_0, \mathtt{I}_1, \mathtt{I}_2, \ldots\}$ on port `float` and the first image $\mathtt{I}_0$ on port `ref`. This results in an execution of `Register to first` for the cross product combination of the two inputs: $\{(\mathtt{I}_0, \mathtt{I}_0), (\mathtt{I}_1, \mathtt{I}_0), (\mathtt{I}_2, \mathtt{I}_0), \ldots\}$, leading to the creation of a set of images (given as references to their locations). The rest of the workflow follows the same structure in executing the second alignment step.

Listing 7 shows the same portion of the workflow translated to IWIR. We can see that the input port of the task `First` has been translated to type `collection/file` (line 3). Since the GWENDIA workflow defined the port depth as one, we had to explicitly specify in IWIR that this input port expects a collection of files to start the task execution. We can also see that the GWENDIA task `Register to first` has been translated into two IWIR tasks `Register-to-first:cross` and `Register-to-first`. As described in Section 9.1, the cross product iteration strategy used in `Register to first` activity can be expressed in IWIR using two `parallelForEach` tasks that 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 receives a collection of files and the other input port a single file. Each collection element is then used together with the file on the second port for every execution of the `Register-to-first` task.

Listing 7 Image registration workflow excerpt in IWIR.

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

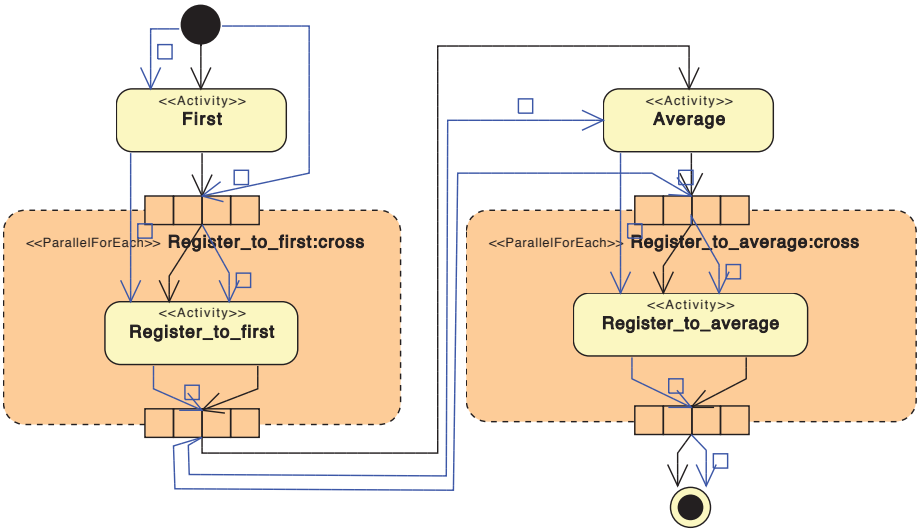

Fig. 16 Image registration workflow in ASKALON. (color online)

Finally in the last step, we loaded the resulting IWIR workflow into the graphical user interface of the ASKALON workflow environment. This automatically triggers a translation to its native AGWL language and renders a graphical view in its UML-based interface, as displayed in Figure 16.

9.3 Image manipulation workflow

In this section, we demonstrate a common use case of the proposed fine-grain workflow interoperability framework. We devised a scenario in which a workflow developer wishes to collaborate in the production of a workflow with other developers. To achieve this, the developer designs a first preliminary version of a workflow in the favourite workflow system and language, and uses the FGI architecture to disseminate it amongst other developers in

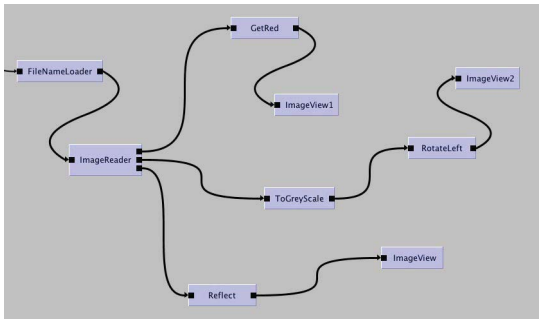

Fig. 17 Image manipulation workflow in Triana. (color online)

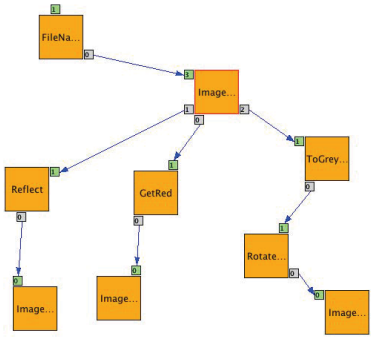

Fig. 18 Image manipulation workflow in WS-PGRADE. (color online)

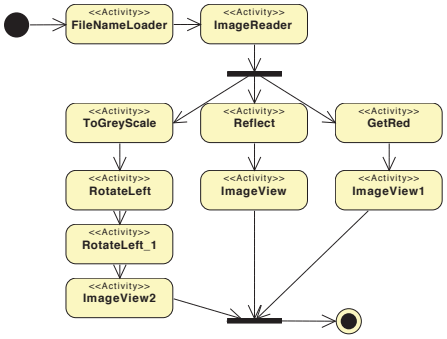

Fig. 19 Modified image manipulation workflow in ASKALON. (color online)

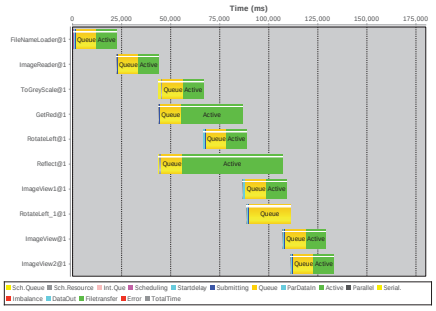

Fig. 20 Modified image manipulation workflow execution in ASKALON. (color online)

the form of an IWIR bundle. The other developers are then able to further edit and modify the workflow using their own language and execute it on their own enactment engines too. The new modified workflow versions may be given back, again in the form of an IWIR workflow bundle, to the original developer for evaluation, execution, and possibly further development. To demonstrate this collaborative interoperability scenario, we employ the current development versions of the Triana, WS-PGRADE and ASKALON systems.

In the first step, the workflow developer `A` working with the Triana system develops an initial image manipulation workflow, as shown in Figure 17. The workflow loads an image file (in the tasks `FileNameLoader` and `ImageReader`) and uses it as input to three parallel workflow paths. The first of the parallel paths filters out everything but the red channel (`GetRed`) and opens an image viewer (`ImageView1`) that displays the resulting image. The second parallel path converts the image to a pure greyscale image (`ToGreyScale`), rotates the resulting image counter-clockwise by 90° (`RotateLeft`), and finally opens another image viewer (`ImageView2`) displaying the resulting image. The third parallel path transforms the image into its mirror image through reflection (`Reflect`) and opens an image viewer (`ImageView`) too.

To collaborate with developers `B` and `C`, developer `A` now uses the IWIR export function in Triana to export the workflow to the IWIR bundle format. This functionality follows the three steps proposed in our FGI architecture (see Section 3):

1. The abstract part of the native Triana taskgraph workflow is converted to an abstract IWIR workflow graph containing seven distinct IWIR task types (`FileNameLoader`, `ImageReader`, `GetRed`, `ToGreyScale`, `Reflect` and `ImageView`);
2. For each of the seven task types, the required binaries are identified and a JSDL template specifying its invocation is created (i.e. the CTR);
3. An IWIR bundle containing the IWIR workflow graph and the CTRs for all seven task types is generated.

The resulting IWIR bundle is sent to developers `B` and `C` working with the workflow systems WS-PGRADE and ASKALON. When the developer `B` imports the bundle into WS-PGRADE, it appears as shown in Figure 18. Even though the graphical representation changed, it is still easy to see that the structure of the workflow has not changed. Developer `C` working with ASKALON imports the IWIR bundle and decides to modify it to better suit his requirements which demand the greyscaled image be rotated by 180° counter-clockwise (as opposed to 90° in the original version). To achieve this, he duplicates the `RotateLeft`

task and produces a new modified version of the workflow, as shown in Figure 19. After this successful modification, he executes the workflow using the native ASKALON workflow system running on top of the the Austrian Grid DCI. Figure 20 shows the execution trace of the workflow in the ASKALON performance monitoring interface. The horizontal bars represent the workflow tasks having two visible states, queued and active, and the horizontal axis the execution time.

As a final step, the developer `C` decides to give the new modified workflow back to the original workflow developer `A`. To achieve this, he exports the ASKALON workflow to an IWIR bundle and forwards it to his colleague who opens and visualises it as a Triana taskgraph, as illustrated in Figure 21. This scenario executed using the current development versions of Triana, WS-PGRADE and ASKALON shows therefore how the FGI architecture is able to support the collaborative editing and development of workflows by scientists working in different environments with different user interfaces and workflow languages.

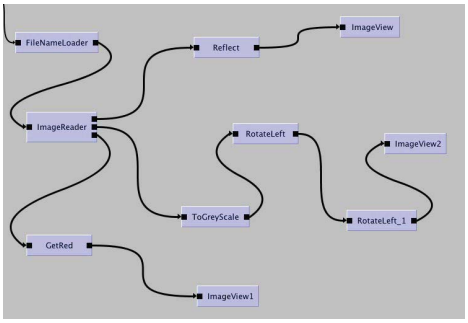

Fig. 21 Modified image manipulation workflow in Triana. (color online)

10 Conclusions and Future Work

In this paper we presented the architecture designed in the EU FP7 SHIWA project [4] to support fine-grained interoperability (FGI) of scientific workflows. Our solution separates the interoperability concerns in two layers: abstract and concrete. At the abstract layer, we proposed a novel Interoperable Workflow Intermediate Representation (IWIR) designed to enable portability of workflows originally written in different languages across numerous scientific workflow systems. 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. At the concrete layer, the interoperability is based on JSDL templates and the IWIR bundle concept designed for packaging concrete task representations (CTRs) and their invocation. To support integration of new workflow systems into the interoperability framework, we developed a number of tools including an IWIR tool for scanning, parsing and manipulating IWIR documents, an IWIR XML schema, an API to interact with IWIR bundles, and a JSDL template creator. We showed why the BPEL standard is not a good candidate in being used as an intermediate language to enable the interoperability of scientific workflow systems. We presented examples of using IWIR to model common data iteration strategies such as dot and cross product. We showed two real-world case studies using two workflow applications designed in a native environment, translated to the IWIR bundle format, and executed by a foreign workflow system in a foreign DCI. Additionally, we showed how workflow developers can employ

the FGI architecture to cooperate, reuse, and share workflows across different, previously incompatible workflow systems and DCIs. FGI is currently successfully supported by the four pilot workflow systems that participate in the SHIWA project: ASKALON, MOTEUR, WS-PGRADE and Triana.

Planned future work includes support for a wider range of technologies for CTR representation, particularly virtual machine images for Cloud infrastructures. We also intend to create a repository for CTRs. This repository can be used to create a one-to-many mapping between abstract task types and concrete CTRs. An intended API that can be integrated into workflow systems or converters allows for more flexibility in scheduling and optimization at conversion time or even dynamically at runtime. Another area we want to target is research on data annotation, starting with a simple semantic-enabled system that helps to better describe data consumed and produced by the concrete task representations. Additionally, we work together with workflow communities and workflow system developers to identify open issues and evolve IWIR to provide support for future requirements.

Acknowledgements The Austrian Science Fund project TRP 237-N23 and the European Union project 261585/SHIWA and funded this research.

References

1. Open Archives Initiative Object Reuse and Exchange (OAI-ORE). `http://www.openarchives.org/ore/`.
2. Resource Description Framework (RDF) . `http://www.w3.org/RDF/`.
3. Simple Knowledge Organization System (SKOS), 2009. `http://www.w3.org/TR/skos-reference/`.
4. SHIWA: SHaring Interoperable Workflows for large-scale scientific simulation on Available DCIs. `http://www.shiwa-workflow.eu`, 2011.
5. Ali Anjomshoaa, Fred Brisard, Michel Drescher, Donal Fellows, An Ly, Stephen McGough, Darren Pulsipher, and Andreas Savva. Job Submission Description Language (JSDL) Specification, Version 1.0. Technical report, Global Grid Forum, November 2005.
6. Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0 (W3C Recommendation). Technical report, World Wide Web Consortium, January 2007.
7. K.M. Chao, M. Younas, N. Griffiths, I. Awan, R. Anane, and CF Tsai. Analysis of Grid Service Composition with BPEL4WS. In *Advanced Information Networking and Applications, 2004. AINA 2004. 18th International Conference on*, volume 1, pages 284–289. IEEE, 2004.
8. R. Chinnici, J.J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. *W3C Recommendation*, 26, 2007. `http://www.w3.org/TR/wsdl20/`.
9. James Clark and Steve DeRose. XML Path Language (XPath) 1.0 (W3C Recommendation). Technical report, World Wide Web Consortium, 1999.
10. E. Deelman, G. Singh, M.H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, et al. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming*, 13(3):219–237, 2005.
11. F. DeRemer and H.H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. In *Software Engineering, IEEE Transactions on*, number 2, pages 80–86. IEEE, 1976.
12. W. B. Dobrusky and T. B. Steel. Universal computer-oriented language. *Commun. ACM*, 4:138–, March 1961.
13. E. Elmroth, F. Hernández, and J. Tordsson. Three Fundamental Dimensions of Scientific Workflow Interoperability: Model of Computation, Language, and Execution Environment. volume 26, pages 245–256. Elsevier, 2010.
14. O. Ezenwoye, S. Sadjadi, A. Cary, and M. Robinson. Grid Service Composition in BPEL for Scientific Applications. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, pages 1304–1312. Springer, 2007. ISBN: 978-3-540-76889-0.

15. Thomas Fahringer, Radu Prodan, and et al. *ASKALON: A Development and Grid Computing Environment for Scientific Workflows*, chapter Frameworks and Tools: Workflow Generation, Refinement and Execution. Workflows for e-Science. Springer Verlag, 2007.
16. Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployment of data-intensive applications on grids with moteur. *International Journal of High Performance Computing Applications*, 22(3):347–360, 2008.
17. Andrew Harrison, Dave Rogers, and Ian Taylor. SHIWA Desktop. SHIWA Deliverable D5.2, December 2010.
18. D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, et al. Web Services Business Process Execution Language Version 2.0. *OASIS Standard*, 11, 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.
19. Peter Kacsuk. P-GRADE portal family for grid infrastructures. *Concurrency and Computation: Practice and Experience*, 23(3):235–245, March 2011.
20. F. Leymann. Choreography for the Grid: towards fitting BPEL to the resource framework. In *Concurrency and Computation: Practice and Experience*, volume 18, pages 1201–1217. Wiley Online Library, 2005. ISSN: 1532-0634.
21. Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java^{TM}Virtual Machine Specification, Java SE 7 Edition. Technical report, Oracle America, Inc., July 2011.
22. J. Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179, 2003.
23. Paolo Missier, Daniele Turi, Carole Goble, and et al. Taverna workflows: Syntax and semantics. In *IEEE International Conference on e-Science and Grid Computing*, Dec 2007.
24. B.P. Model. Business Process Modeling Notation (BPMN) Version 2.0. *OMG Specification, Object Management Group*, 2011. `http://www.omg.org/spec/BPMN/2.0/`.
25. Johan Montagnat, Benjamin Isnard, Tristan Glatard, Ketan Maheshwari, and Mireille Blay 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.
26. J. Nitzsche, T. Van Lessen, D. Karastoyanova, and F. Leymann. BPEL light. In *Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007, Proceedings*, Lecture Notes in Computer Science, pages 214–229. Springer, 2007. ISBN: 978-3-540-75183-0.
27. Kassian Plankensteiner, Radu Prodan, Thomas Fahringer, Johan Montagnat, N. Cerezo, Dave Rogers, Ian Harvey, Akos Balasko, and et al. Fine-grained interoperability architecture and case studies. SHIWA Deliverable D6.2, March 2012.
28. Kassian Plankensteiner, Radu Prodan, Thomas Fahringer, Johan Montagnat, and et al. Interoperable workflow intermediate representation. SHIWA Deliverable D6.1, December 2010.
29. A. Slominski. Adapting BPEL to Scientific Workflows. In *Workflows for e-Science*, pages 208–226. Springer, 2007. ISBN: 978-1-84628-757-2.
30. W. Tan, P. Missier, I. Foster, R. Madduri, D. De Roure, and C. Goble. A Comparison of Using Taverna and BPEL in Building Scientific Workflows: the case of caGrid. In *Concurrency and Computation: Practice and Experience*, volume 22, pages 1098–1117. Wiley Online Library, 2009. ISSN: 1532-0634.
31. Ian Taylor, Matthew Shields, Ian Wang, and Rana Rana. Triana applications within Grid computing and peer to peer environments. *Journal of Grid Computing*, 1(2), 2003.
32. B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, and J. Patel. Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling. In *Workflows for e-Science*, pages 428–449. Springer, 2007. ISBN: 978-1-84628-757-2.
33. Paul Damian Wells. A universal intermediate representation for massively parallel software development. *SIGPLAN Not.*, 39(5):48–57, May 2004.
34. W.P.D.I.X.M.L. WfMC. Process Definition Language (XPDL), WfMC Standards. Technical report, WFMC-TC-1025, http://www. wfmc. org, 2001. `http://www.wfmc.org/xpdl.html`.
35. H. Zhang, X. Fan, R. Zhang, J. Lin, Z. Zhao, and L. Li. Extending BPEL 2.0 for Grid-Based Scientific Workflow Systems. In *Asia-Pacific Services Computing Conference, 2008. APSCC'08. IEEE*, pages 757–762. IEEE, 2008.