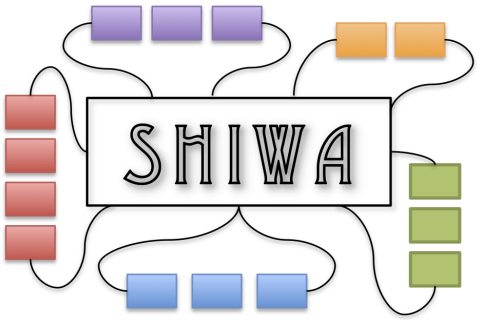

Project number: RI 261585

Project acronym: SHIWA

Project full title: **SH**aring **I**nteroperable **W**orkflows for large-scale scientific simulations on **A**vailable DCIs

Research Infrastructures
INFRA-2010-1.2.2 Simulation Software and Services

D6.2: Fine-grained Workflow Interoperability: Framework and Case Studies

Due date of deliverable: 31/03/2012 Start date of project: 01/07/2010	Actual submission date: 31/03/2012 Duration: 24 months
Lead beneficiary: UIBK	WP6
Dissemination Level: public	Version 1.0

SHIWA is supported by the FP7 Capacities Programme under contract number RI-261585

Contents

List of Figures

List of Tables

Status and Change History

Status	Name	Date	Signature
Draft	K. Plankensteiner, R. Prodan, J. Montagnat, N. Cerezo, A. Balasko, T. Kukla, D. Rogers, I. Harvey	19/03/2012	n.n electronically
Reviewed	Vladimir Korkhov	27/03/2012	n.n electronically
Approved	Péter Kacsuk	31/03/2012	n.n electronically

Table 1: Status of deliverable

Version	Date	Pages	Author	Modification
0.1	8/3/2012	all	J. Montagnat , N. Cerezo	GWENDIA converter and use case
0.2	8/3/2012	all	K. Plankensteiner	TOC outline and cleanup
0.3	13/3/2012	all	A. Balasko	Sections JSDL, gUSE, DCI Bridge
0.4	13/3/2012	all	K. Plankensteiner	Sections Architecture, IWIRtool
0.5	17/3/2012	all	T. Kukla	Section SHIWA Repository
0.6	17/3/2012	all	R. Prodan	Sections: Introduction, conclusions
0.7	19/3/2012	all	D. Rogers, I. Harvey	Sections: FGI bundles, Triana Taskgraph
0.8	19/3/2012	all	K. Plankensteiner	Cleanup, Integration
1.0	28/3/2012	all	K. Plankensteiner, D. Rogers	Changes according to internal reviewer comments

Table 2: Change History

Glossary

BLAST	Basic Local Alignment Search Tool
CGI	Coarse-Grained Interoperability
CO	Carbon monoxide
DCI	Distributed Computing Infrastructure
EGI	European Grid Initiative
FGI	Fine-Grained Interoperability
CTR	Concrete Task Representation
JRA	Joint Research Activity
MPI	Message Passing Interface
NA	Networking Activity
SA	Service Activity
WP	Work Package

Table 3: Glossary

1 Introduction

The objective of the SHIWA JRA2 workpackage is to develop solutions for fine-grained interoperability. In contrast to coarse-grained interoperability developed in JRA1 which regards nested non-native workflows as a black box executed by different workflow systems, fine-grained interoperability focuses on the transformation of workflow representations in order to achieve workflows migration from one system to another. The power of the fine-grained workflow interoperability stands in exploiting the most appropriate enactor for a certain workflow application, independently from the language in which it was created.

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 [2], GWENDIA [5], P-GRADE [3], SCUFL [4], Triana [6]). 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 all workflows 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. An important design property of IWIR shall be 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 the minimum amount 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) [1] 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 experiment and the scientist's objectives (e.g. performance, reliability, cost);

3. It enables runtime interoperability between different workflow systems (as opposed to static meta-workflow design-time supported by the coarse-grained interoperability). 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 ($O(m)$ complexity). Similarly, porting m interoperable workflows to a new DCI platform requires the development of one single IWIR-compliant back-end (again $O(1)$ complexity), while language-to-language translations would require again 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.

Figure 1 displays the schematic architecture of the fine-grained interoperability framework targeted in the SHIWA JRA2 workpackage. 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. A special case to this scenario is the Pegasus workflow, for which a GUI-based workflow composition tool that directly interfaces to IWIR is planned. 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

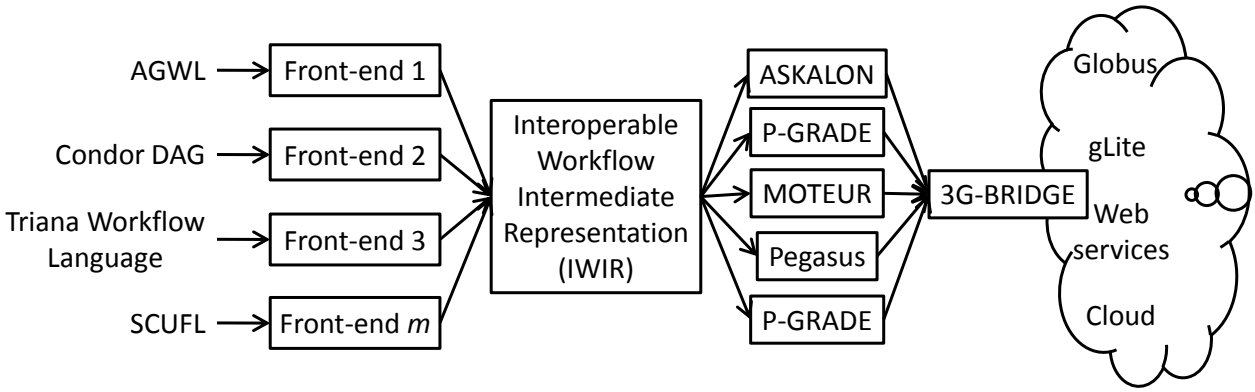

Figure 1: Schematic fine-grained interoperability framework architecture.

After the IWIR specification has been defined in the deliverable D6.1, the purpose of this deliverable is to present the fine-grained interoperability architecture and its implementation support in the four workflow systems that are part of the SHIWA project: ASKALON, Moteur, P-Grade, and Triana. The deliverable is organised as follows. The next section presents the general fine-grained interoperability architecture, followed in Section 3 by a description of the FGI bundle technology underneath. Section 4 presents the design and implementation of the IWIRtool as a Java package that provides general support for parsing, manipulating, and validating IWIR data structures. Section 5 reports on the IWIR workflow converters implemented by the four workflow systems for their native languages: AGWL, GWENDIA, gUSE, and Triana taskgraph. Additionally, conversion examples are shown. While the IWIR workflow converters deal with the abstract part of a workflow only, concrete computational tasks are described using JSDL Templates, as shown in Section 6. Section 7 reports on the integration of fine-grained interoperability-based workflows in the SHIWA repository and Section 8 concludes the deliverable.

2 Architecture

When talking about the architecture of the fine-grained-interoperability solution and FGI-based workflow applications, we distinguish two parts of a workflow, corresponding to two levels of abstraction: the *concrete* part and the *abstract* part of the workflow.

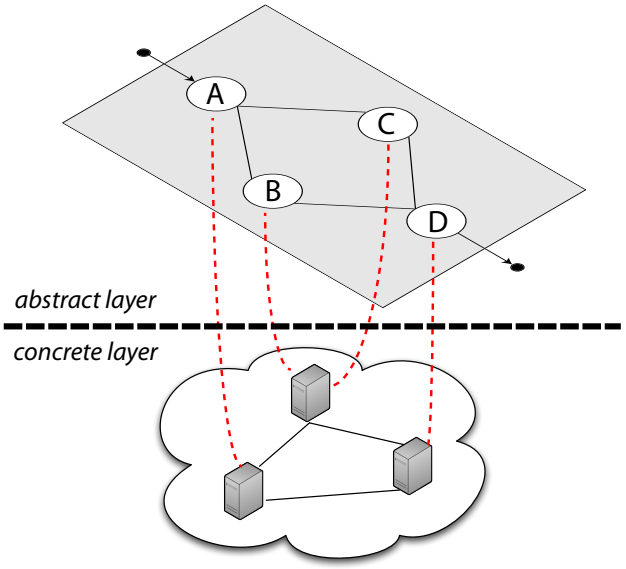

Figure 2: Abstract Layer and Concrete Layer in a Workflow

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.

Both parts combined, abstract and concrete, make the workflow a fully specified, executable application. Figure 2 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 SHIWA FGI environment, the abstract and concrete part of an FGI-compatible workflow are packaged in a single archive file, the FGI Bundle (see Section 3).

2.1 IWIR

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*

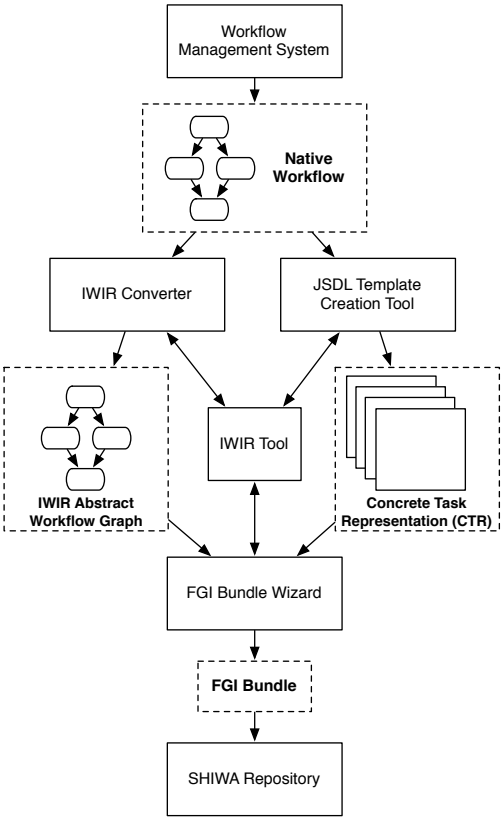

Figure 3: Transforming a native workflow to an FGI-compatible workflow package

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 data-flows 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. IWIR was specified in deliverable D6.1.

2.2 Creation of an FGI-compatible workflow

The act of transforming a native workflow application to an FGI-compatible workflow bundle can be broken down into the following steps:

1. Convert the *abstract* part of the workflow to an IWIR-based graph representation
2. For each Task Type referenced in the IWIR-based graph representation, convert the *concrete* task implementation to a DCI-independent concrete task representation (CTR)
3. Create an FGI Bundle containing the IWIR-based graph representation and all CTRs

All of these steps are explained in more detail in the following sections. Figure 3 illustrates the components and data involved in the process.

2.2.1 Conversion of the abstract part of the workflow to IWIR

In order to convert the abstract part of a workflow (i.e. the workflow logic as expressed by the original workflow language) to IWIR, Work Package 6 developed a set of IWIR converters. Currently

we provide IWIR converters for the following native workflow languages:

- AGWL (ASKALON) - see Section 5.1
- Triana Taskgraph (Triana) - see Section 5.2
- gUSE (WS-PGRADE) - see Section 5.4
- GWENDIA (MOTEUR) - see Section 5.5

As shown in Figure 3, the conversion of the abstract part of the workflow will lead to an *IWIR Abstract Workflow Graph*, representing the workflow logic as an IWIR workflow document. To create IWIR workflow documents, the IWIR converters use the functionality of our Java-based library called IWIRtool (see Section 4).

2.2.2 Creation of DCI-independent concrete task representations (CTR)

An IWIR abstract workflow graph as created by the IWIR Converter components contains information about precedence relations between computational tasks, their input/output signature as well as 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 both, a description of the abstract part as well as information specifying the concrete part of the workflow.

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 as simple as providing a statically linked Linux x86 executable file for execution of the computational task on x86-based Linux systems without any additional library requirements. It can also mean providing several executables together with library dependencies for different platforms or even providing a virtual machine image pre-equipped with all tools and libraries necessary to execute the task. It is our intention to keep the possibilities for concrete representations of computational tasks wide open to extension for the future, and we provide the possibility of a 1-to-N mapping between abstract IWIR Task Types and CTRs. For the initial implementation of FGI-compatibility of the SHIWA workflow engines we will start off by representing computational tasks using executable Linux x86 binaries.

The second part of a CTR is a JSDL Template file that describes how to use the files contained in part one to invoke the computational task. Figure 3 shows that JSDL Templates are created using a component called *JSDL Template Creation Tool*, which is described in Section 6.1. The specification of JSDL Templates for use in CTRs is described in Section 6.

2.2.3 Creation of an FGI Bundle

An FGI bundle (see Section 3) 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 IWIR Task Types to CTRs. In other words, an FGI bundle can be understood as a self-contained interoperable workflow, described in a common representation, that is usable in every FGI-compatible workflow system.

As can be seen in Figure 3, the creation of FGI bundles is enabled by a component called the *FGI Bundle Wizard*. The FGI Bundle Wizard is described in detail in Section 3.4.

2.3 Native execution of an FGI-compatible workflow

A SHIWA FGI Bundle (see Section 3) contains all of the information and data required to execute the contained workflow on any SHIWA FGI-compatible workflow execution system. The SHIWA FGI architecture does not mandate that a workflow system executing an FGI Bundle follows a certain fixed execution procedure.

In this section, we present a proposed execution scheme that is well suited for integration into many workflow systems and will be used as shown for the integration of several of the projects workflow systems into the FGI environment. Figure 4 gives an overview of the components and data involved in this process.

2.3.1 Proposed execution Scheme for FGI workflows

Upon import of an FGI Bundle, the target workflow management system (WFMS) uses the FGI Bundle Library (see Section 3) to extract the contained IWIR Abstract Workflow Graph. This workflow graph is converted to the native language of the system using an IWIR converter (see Section 5).

Depending on the type of workflow representation in the target WFMS it may be required to read the CTR information at import time or at runtime. For simplicity reasons, we only describe the latter situation in this document. Now the execution of the workflow graph can start. When the workflow engine reaches an atomic task T, it uses the FGI Bundle Library to request the CTR that matches the Task Type of T. This CTR consists of a JSDL Template file as well as several other files (executables, libraries, input files) that are required. The execution of task T is then done as follows:

1. Extract CTR from the FGI Bundle using the FGI Bundle Library by providing the Task Type of task T
2. Upload the files contained in the CTR to a location reachable by the DCI Bridge (see Section 6.4)
3. Instantiate the JSDL Template to a concrete JSDL document using the JSDL Template Instantiation Tool by providing URL references to the data required for each input port of task T as well as URL references to the uploaded files contained in the CTR
4. Send the instantiated JSDL document to the DCI Bridge for execution (see Section 6.2)
5. Upon notification of task completion by the DCI Bridge, retrieve the output data and continue with the workflow execution.

3 FGI Bundles

Section 3.1 below describes and assesses the general hierarchy already present in SHIWA bundles, as defined in deliverable D5.2, in order to determine how FGI can be facilitated through the bundling API; looking at how the current relationships present in a bundle could be interoperated to support an FGI Bundle. When originally defining the metadata of a CGI bundle (see D5.2) much consideration was taken to model the signature of a workflow implementation on the signature of an IWIR task, as theoretically a CGI workflow can be considered a node in an IWIR workflow. This means that much of the bundle metadata should be easily modified to represent a single IWIR task.

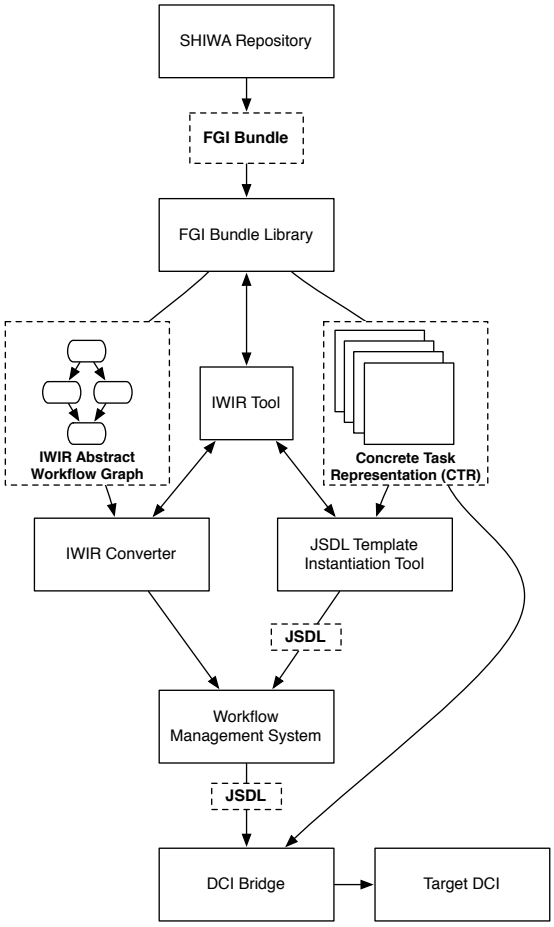

Figure 4: Proposed native execution of an FGI-compatible workflow package

3.1 General Bundle Hierarchy

Over the development of the SHIWA CGI bundle schema, a three tier structure has emerged as a means of describing the elements of a workflow. Figure 5 shows the three levels to the bundle hierarchy. The leftmost set identifies the current nomenclature used by CGI to describe the elements within the three tiers. This name set has also been inherited by the current bundle schema to identify these elements, as up to this point bundling was only truly considered a CGI solution. The right hand set identifies the equivalent IWIR terms as identified in D6.1. In the centre of the diagram, the new terminology that is proposed for the bundle schema is displayed.

3.1.1 Abstract Level Concepts

The Abstract Task is used to aggregate together tasks of the same function and signature in order to facilitate interoperability. One element of this consists of a set of both input and output ports and a unique name which is known as the abstract task's signature. Each port in a signature will declare a data type; the format the port requires the data to be in. Datatypes are bound to the XML Schema set of datatypes (http://www.w3.org/2001/XMLSchema#), as specified in devlierable D6.1. An abstract task is environment independent; allowing workflow developers to design a workflow that is not constrained to a specific operating system or workflow environment.

An IWIR Task Type is translated directly to an abstract task when serialised to bundle format. A direct translation is available due to the fact that the CGI counterpart and precursor to the Abstract task, the Abstract Workflow, was originally modelled on the Task Type concept proposed

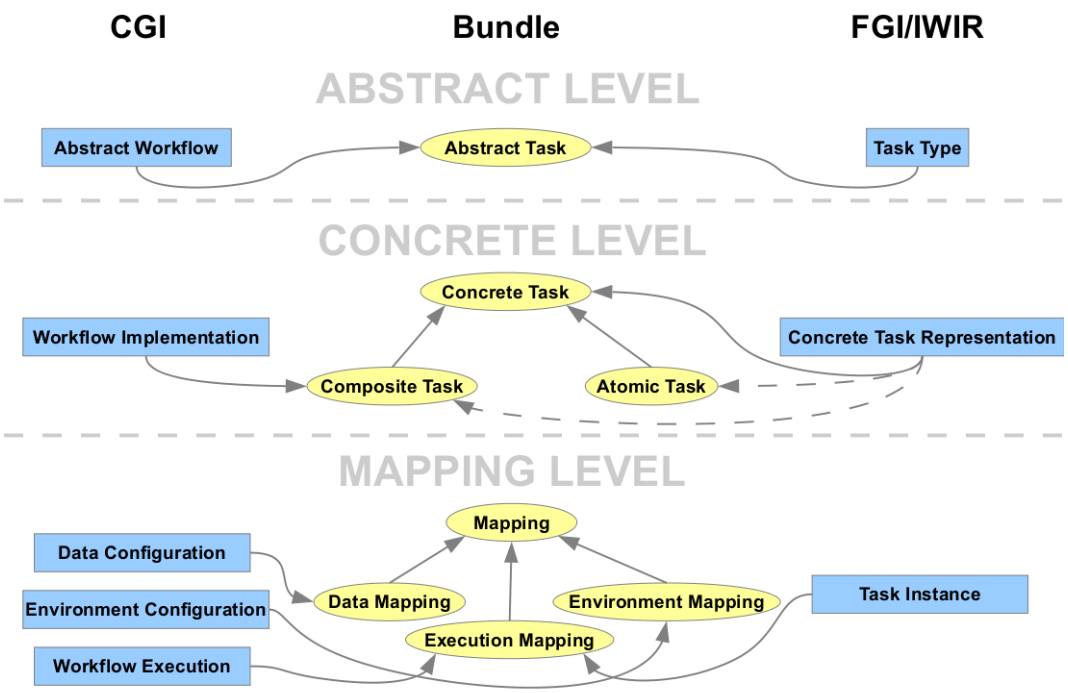

Figure 5: Bundle Hierarchy with CGI and FGI mappings

in D6.1.

The abstract task is primarily used for the cataloging of concrete tasks described below, and as such has little use in the current implementation of FGI bundles.

3.1.2 Concrete Level Concepts

A specific implementation of an abstract task is known as a concrete task. This may inherit its signature from its parent abstract workflow, or it may declare its own finer grained version of the parent signature. Along with this, it will identify its main executable code, known as the definition files, and any dependencies that the implementation will rely on to be run.

Section 2.2.2 describes the Concrete Task Representation; a task type bound to a specific implementation. Each CTR will be represented inside of an FGI bundle with an abstract task, with it's definition files being the executable file and JSDL of the CTR. Whilst the CTR is to be DCI independent it may still be reliant upon other DCI independent executable code and so these reliances are described using the dependency set.

The CGI element at this level is known as a Workflow Implementation, which is an engine specific implementation of a workflow. To distinguish a workflow from a CTR when in bundle form, there will be two sub-concepts of abstract task; an atomic task that represents a single node, and a composite task which represents a group of atomic tasks marshalled into a workflow.

3.1.3 Mapping Level Concepts

Mappings associate datasets to elements exposed at the Abstract and Concrete level. There are currently three types of mappings;

Data Mappings provide input data and output locations for a task which may be associated to a port present in an abstract or concrete task's signature either through a string, file or remote URI. A bundle may contain a data mapping, allowing it to be run immediately, otherwise

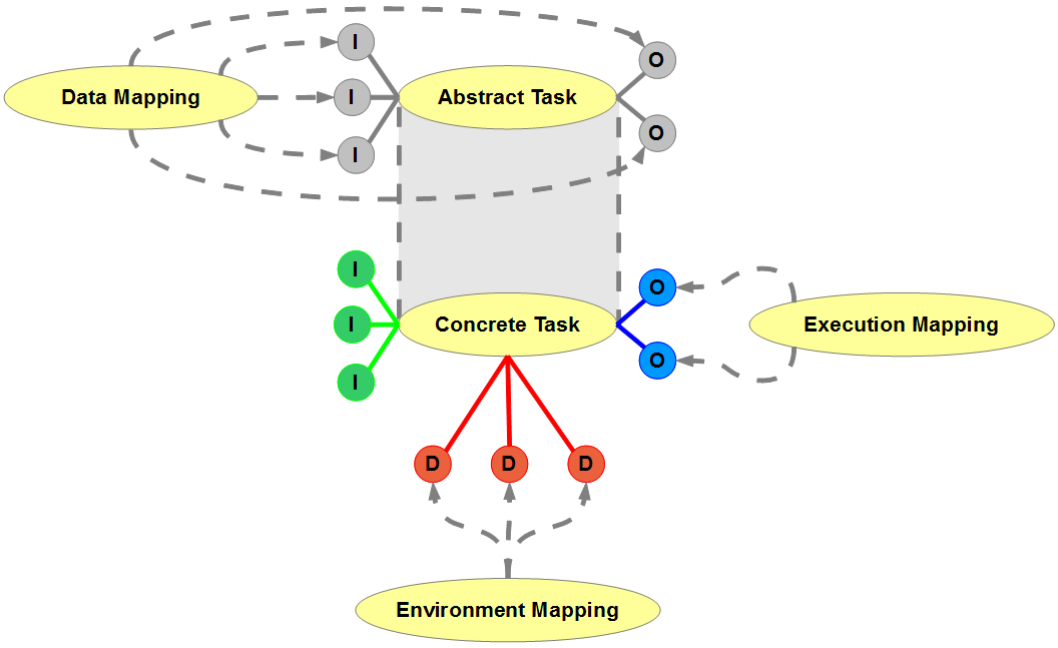

Figure 6: Mapping level concepts relation to concrete concepts

a data mapping must be created. This can be done using the SHIWA Desktop bundling API. In an FGI bundle, a data mapping may be present which maps to the composite task representing the IWIR workflow.

Environment Mappings map to a concrete tasks dependency set. There is a tight relationship between environment mappings and concrete tasks; a concrete task will generally not run unless all it's dependencies are realised via an environment mapping. Again, an environment mapping can be created using the SHIWA Desktop bundling API.

Task Executions hold the output data produced by a concrete task after it has been run. The output data of an FGI workflow should be serialised to a Task Execution upon completion.

Figure 6 shows how each concept can be mapped onto the concrete and abstract concepts.

3.1.4 OAI-ORE Resource Maps

A SHIWA bundle contains a cocktail of the concept elements described above to describe a workflow or a composite part of a workflow. Each element is described using an OAI-ORE Resource Map which aggregates together a collection of files, ranging from descriptive metadata files to raw data to supporting literature. In the scope of ORE these files are known as Aggregated Resources and collectively they form an Aggregation of the concept element. The bundle terminology goes further by assigning these aggregated resources into three categories;

The Primary Resource is the metadata file that describes the main properties of the aggregation. When the aggregation is representing an element from the concrete level task it will have a pointer to a definition file, this being the workflow graph or executable code of the task.

Secondary Resources are the primary resources of child aggregations, and so will declare that they are described by another Resource Map.

Tertiary Resources are the raw data files and supporting information files.

An example of a Resource map (specifically a CGI workflow) is provided in Listing 1.

Listing 1: Example Resource Map

```
<rdf:RDF
  xml:base="http://shiwa-workflow/bundle/" >

  <-- The root element of the Resource Map; identifies the aggregation that the RM
        describes -->

  <rdf:Description rdf:about="rem/root">
    <dc:creator rdf:nodeID="A0"/>
    <dcterms:created>2012-03-28T00:00:38+01:00</dcterms:created>
    <dc:title>Resource Map for Triana Image Merger</dc:title>
    <dc:identifier>83a0fe54-4fd6-49e0-b59a-bb72e3bc87f8</dc:identifier>
    <ore:describes rdf:resource="aggr/e79b8126-e18f-4c86-90a0-5a01fe24aa14"/>
    <rdf:type rdf:resource="http://www.openarchives.org/ore/terms/ResourceMap"/>
  </rdf:Description>

  <!-- The Resource Map's aggregation; e79b...aa14 is the UUID of the Primary Resource,
         declared inside of metadata.rdf -->

  <rdf:Description rdf:about="aggr/e79b8126-e18f-4c86-90a0-5a01fe24aa14">
    <ore:aggregates rdf:resource="metadata.rdf"/>
    <ore:aggregates rdf:resource="c48c8227-1e29-431f-a795-289fe9ef5dcc/metadata.rdf"/>
    <ore:aggregates rdf:resource="bd6ff821-1b6d-4743-8492-2dfe385c849b/metadata.rdf"/>
    <ore:aggregates rdf:resource="bundle2.odg"/>
    <ore:aggregates rdf:resource="triana_wf2.xml"/>
    <ore:aggregates rdf:resource="Triana_wf2.png"/>
    <ore:aggregates rdf:resource="Triana_wf2.png2567684204760176565.tmp"/>
    <rdf:type rdf:resource="http://www.openarchives.org/ore/terms/Aggregation"/>
  </rdf:Description>

  <!-- The Primary Resource; identified as it is of type shiwa:workflow one of the
         bundle hierarchy element -->

  <rdf:Description rdf:about="metadata.rdf">
    <dcterms:created>2011-05-31T01:00:00+01:00</dcterms:created>
    <rdf:type rdf:resource="http://shiwa-workflow.eu/concepts#workflow"/>
  </rdf:Description>

  <!-- Secondary Resources; Identified as they point to their own Resource Map -->

  <rdf:Description rdf:about="c48c8227-1e29-431f-a795-289fe9ef5dcc/metadata.rdf">
    <ore:isDescribedBy rdf:resource="c48c8227-1e29-431f-a795-289fe9ef5dcc/resourceMap.rdf"/>
  </rdf:Description>
  <rdf:Description rdf:about="bd6ff821-1b6d-4743-8492-2dfe385c849b/metadata.rdf">
    <ore:isDescribedBy rdf:resource="bd6ff821-1b6d-4743-8492-2dfe385c849b/resourceMap.rdf"/>
  </rdf:Description>

  <!-- All remaining resources can be considered Tertiary Resources -->

  <rdf:Description rdf:about="Triana_wf2.png2567684204760176565.tmp">
    <rdf:type rdf:resource="http://shiwa-workflow.eu/concepts#bundlefile"/>
    <dc:description></dc:description>
  </rdf:Description>
  <rdf:Description rdf:about="triana_wf2.xml">
    <rdf:type rdf:resource="http://shiwa-workflow.eu/concepts#definition"/>
  </rdf:Description>
  <rdf:Description rdf:about="Triana_wf2.png">
    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/img"/>
  </rdf:Description>
  <rdf:Description rdf:about="bundle2.odg">
    <rdf:type rdf:resource="http://shiwa-workflow.eu/concepts#bundlefile"/>
    <dc:description></dc:description>
  </rdf:Description>
</rdf:RDF>
```

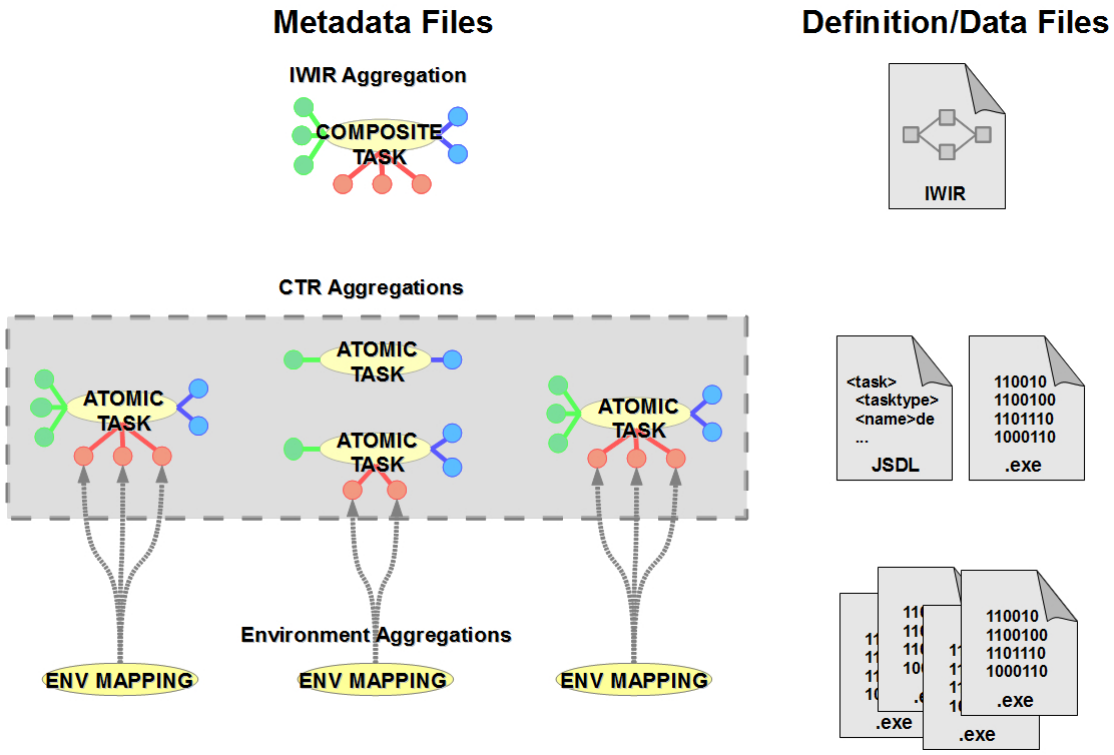

Figure 7: FGI Bundle Structure

3.2 FGI Bundle Structure

It has been proposed that an FGI bundle will contain a composite task as its top level aggregation, representing the IWIR workflow as a whole. The primary resource will declare the definition file as being the IWIR graph, and the aggregation's signature representing the overall inputs and outputs of the workflow, any dependencies that apply to the workflow in its entirety will be defined at this level. Figure 7 describes all the elements present in an FGI Bundle.

Each task inside of the workflow with have their own atomic task aggregation, listed as secondary aggregations to the IWIR workflow aggregation. The primary resource of the task aggregation will contain metadata describing the task, the executable code, any dependencies that are specific to the task and a JSDL Template file (described in Section 6).

3.3 Bundle API alterations

In order to achieve the bundle structure described above, some small alterations are required of the overall bundle framework. The first change relates to the definition file present in a concrete element. In a CGI workflow there was only ever one definition file, this being the workflow graph, but a CTR requires at least two definition files; the JSDL Template file and one or more executables. This means the bundling API present within the SHIWA Desktop Data package will have to be altered to provide a set of files upon request of a composite task's definition.

The second alteration of the API comes in the form of extending the descriptive classes to align the terminology with the bundle concepts described above in Section 3.1. The original CGI specific classes will remain but will now extend classes named from the bundle terminology.

The final alteration will be the creation of an *FGIController* class which will be used to both build and read FGI Bundles, and a *CTRHandler* which will be an interface that is to be implemented

in order to present the information belonging to a CTR and used much in the same way as the WorkflowEngineHandler provides information about a Workflow Implementation. Initially the implementation of the TaskDeploymentHandler will only happen within SHIWA Desktop, and be used in conjunction with the FGI Bundle Creation Wizard described in Section 3.4, but it is believed that in future Workflow engines will be able to work directly with the FGIController to bundle FGI workflow automatically.

3.3.1 FGIController Bundle creation methods

function:	initialiseController(WorkflowEngineHandler workflowEngineHandler)
returns:	void
description:	Initialise the FGIController using a WorkflowEngineHandler.
function:	addTask(CTRHandler ctrHandler)
returns:	void
description:	Add a CTR to the FGIController using a CTRHandler.
function:	toBundle()
returns:	SHIWABundle
description:	Generate a SHIWABundle from the Resources present in the FGIController.

3.3.2 FGIController Bundle retrieval methods

function:	initialiseController(SHIWABundle bundle)
returns:	void
description:	Initialise the FGIController with a new SHIWABundle.
function:	getIWIR()
returns:	File
description:	Get the IWIR file the SHIWABundle holds.
function:	getTaskJDSL(String taskName)
returns:	File
description:	Get the JSDL Template file the SHIWABundle holds for the named task.
function:	getTaskDefinition(String taskName)
returns:	Set<File>
description:	Get the executable files the SHIWABundle holds for the named task.
function:	getTaskValue(String taskName, String portName)
returns:	String
description:	Get the input value for a particular port/dependency for the named task. This can be a String value, a URI to a remote location or a local file location.

3.3.3 CTRHandler

function:	getTaskDefinition()
returns:	Set<File>
description:	An set of definition files should be returned from this method.
function:	getTaskJDSL()
returns:	InputStream
description:	An input stream from the JDSL Template file should be returned from this method.
function:	getTaskName()
returns:	String
description:	The name that the task deployment file should have inside the bundle.
function:	getSignature()
returns:	TransferSignature
description:	The TransferSignature of the Workflow.

3.4 SHIWA Desktop FGI Bundle Creation Wizard

Figure 8 describes the process by which an FGI bundle can be created using the SHIWA Desktop FGI Bundle Wizard which is to be developed and packaged with SHIWA Desktop.

The FGI Bundle Wizard will be a tool within SHIWA Desktop that allows a workflow developer to create an FGI Bundle from an IWIR workflow definition file. A workflow will be passed to SHIWA Desktop by the workflow engine. If the workflow identifies as an IWIR workflow the definition file is passed on to the IWIR tool for inspection and the WorkflowEngineHandler is used to initialise an FGIController.

The IWIR tool will provide SHIWA Desktop with a list of unique CTRs. Upon receiving the list of tasks, the FGI Bundle Wizard interface will be launched which prompts the user to provide data files corresponding to each CTR. A CTRHandler will be generated for each CTR, which is passed to the FGIController to add to the FGI Bundle.

Once the wizard is finished, SHIWA Desktop GUI will be loaded, and the user can add any additional metadata information to the IWIR workflow implementation, as they would to any CGI style workflow implementation. Saving and submission of the SHIWABundle is then handled by SHIWA Desktop in the usual manner.

4 IWIRtool

IWIRtool is 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 tool 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 both when parsing existing IWIR XML documents as well as during the creation of new workflows using the API of IWIRtool. Detailed information about these validation processes can be found in Section 4.2.

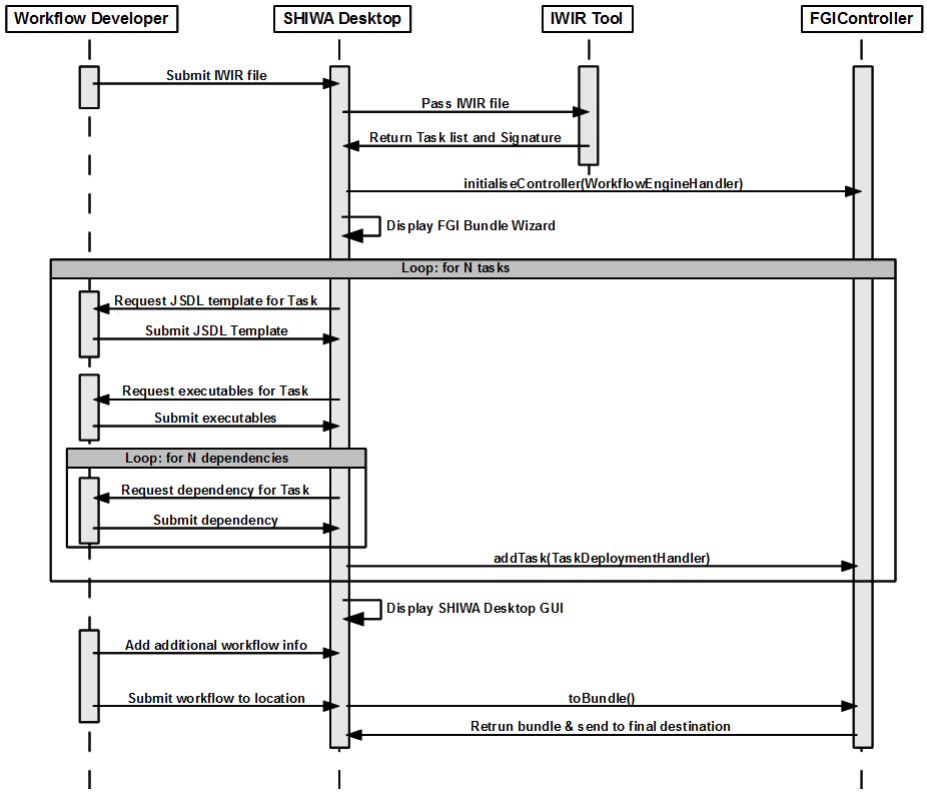

Figure 8: Sequence diagram for FGI Bundle Wizard

4.1 Architecture of IWIRtool

IWIRtool offers Java classes for every task type, port type and link type, for properties, constraints and conditions, several utility classes and a package for validating IWIR condition expressions. Important classes are described in more detail below.

Tasks The classes for the different task types form a hierarchy as follows:

- AbstractTask (abstract class)
  - Task
  - AbstractCompoundTask (abstract class)
    - IfTask
    - AbstractSimpleCompoundTask (abstract class)
      - Blockscope
      - ForTask
      - ForEachTask
      - ParallelForTask
      - ParallelForEachTask
      - WhileTask

Each class represents one of the task types declared in the IWIR specification. Tasks in IWIR form a hierarchical structure as illustrated in Figure 9.

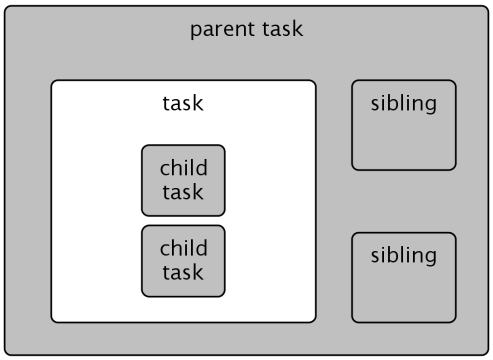

Figure 9: The relation between tasks.

Ports The classes for the different port types form a hierarchy as follows:

- AbstractPort (abstract class)
  - LoopCounter
  - AbstractDataPort (abstract class)
    - InputPort
    - OutputPort
    - LoopPort
    - LoopElement
    - UnionPort

Ports are subdivided into two groups: Input Ports and Output Ports. Input Ports are objects of type InputPort, LoopCounter, LoopPort and LoopElement. Output Ports are objects of type OutputPort and UnionPort.

Each IWIR port has a name, a data type, as well as IWIR properties and constraints. The data type is one of the following:

- DataType (abstract class)
  - SimpleType: one of file, string, integer, double or boolean
  - CollectionType: contains another DataType object

Links The classes corresponding to IWIRs different link types form a hierarchy as follows:

- AbstractLink (abstract class)
  - DataLink: data flow link between two ports
  - ControlLink: control link between two tasks

Conditions A condition is represented by the ConditionExpression class. It contains the condition both as string and as an abstract syntax tree built from ConditionNode objects used for validation of the condition, as described in Section 4.2.

4.2 Workflow validation features

IWIRtool provides on-the-fly validation of IWIR workflows when parsing as well as during construction of workflows using IWIRtool itself. To achieve full validation, IWIRtool uses several different approaches:

- Validation against the IWIR XML Schema
- Validation of IWIR Condition expressions
- Data-flow and Control-flow cycle detection
- Data-flow validation
- Type checking
- Name validation
- Scope validation

We provide an XML Schema document specifying the syntax of IWIR

Validation against the IWIR XML Schema Each time when reading an IWIR workflow from a file or writing an IWIR workflow into a file the workflow is validated against the IWIR XML schema. This is done with the help of javax.xml.validation, a framework for the validation of XML documents.

Validation of IWIR Condition expressions Upon creation of Condition objects, IWIRtool checks it against the following EBNF grammar:

```
CONDITION := CONJUNCTION {or CONJUNCTION}
CONJUNCTION := EQUATION {and EQUATION}
EQUATION := RELATION {EQOP RELATION}
RELATION := FACTPR {COMPOP FACTOR}
FACTOR := not FACTOR | (CONDITION) | STRING | PORT | integer | double | bool
COMPOP := &gt; | &gt;= | &lt; | &lt;=
EQUOP := != | =
STRING := "string"
PORT := string
```

The operator precedence is given as follows:

```
not >> &gt; | &gt;= | &lt; | &lt;= >> != | = >> and >> or
```

Tokens are delimited using spaces or parentheses. The validation is done by creating an abstract syntax tree (AST) of the condition. If the validation fails, a ConditionParseException is thrown.

Data-flow and Control-flow cycle detection Whenever a link is added to a task, IWIRtool searches for cyclic dependencies. This is done by traversing all paths these new links create. If a task occurs more than once in a path a cycle exists, as illustrated in Figure 10. The tool throws a NotWellFormedException and the creation of the object is aborted.

Data-flow validation, Type checking IWIR defines several restrictions on data flow, e.g. data flow may not directly cross scopes. IWIRtool validates the data-flow against the restrictions and checks for type compatibility according to the IWIR specification whenever a link is added to a task. If a link violates one of the restrictions the iwirTool throws a NotWellFormedException and the creation of the object is aborted.

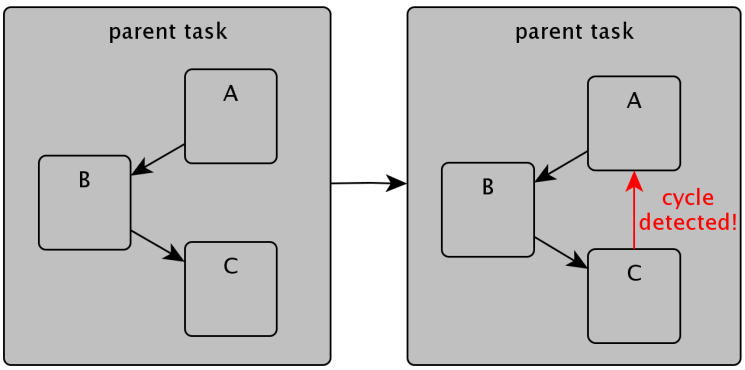

Figure 10: IWIRtool detected a cycle during the creation of a link.

Name and Scope validation When creating a new task or port object IWIRtool verifies its name against the IWIR specification and additionally checks if names are unique within the current scope. If the name is not valid or unique, IWIRtool throws a NotWellFormedException and the creation of the object is aborted.

5 IWIR workflow converters

In order to convert the abstract part of a workflow (i.e. the workflow logic as expressed by the original workflow language) to IWIR, Work Package 6 developed a set of IWIR converters. Currently we provide IWIR converters for the following native workflow languages:

- AGWL (ASKALON) - see Section 5.1
- Triana Taskgraph (Triana) - see Section 5.2
- gUSE (WS-PGRADE) - see Section 5.4
- GWENDIA (MOTEUR) - see Section 5.5

The following sections describe the details of the IWIR converters.

5.1 AGWL

The Abstract Grid Workflow Language AGWL is the native workflow language of the ASKALON workflow system. We have created two software components, agwl2iwir and iwir2agwl, responsible for the translation from AGWL to IWIR abstract workflow graphs and vice versa.

5.1.1 Conversion from AGWL to IWIR

The agwl2iwir tool is able to translate the abstract part of a given AGWL workflow into an equivalent IWIR workflow. The tool traverses the AGWL workflow in a top-down approach starting with the top-level activities and translates each visited activity into a corresponding task. The translation process includes the translation of AGWL ports into IWIR ports, as well as AGWL properties and AGWL constraints into IWIR properties and IWIR constraints.

The following paragraphs describe the AGWL to IWIR translation process for AGWL activites, ports, properties and constraints as well as the creation of links.

Tasks The translation of AGWL activities into IWIR tasks depends on the class of the activity. In general the process works as follows: first, a task with the corresponding class is created. The IWIR task name is either the same as the AGWL activity name, or, if a property with key *agwl-name* exists, the value of this property is used instead. Then, the ports (see paragraph *Ports* below), the properties and constraints and the contained child activites are translated. If the activity contains a condition, which is the case for the IfActivity, SwitchActivity, DoWhileActivity and the WhileActivity, the condition has to be translated (see paragraph *Conditions* below). The translation process differs in the case of a DoWhileActivity, SwitchActivity, SequenceActivity, ParallelActivity and DAGActivity, which is detailed below.

DoWhileActivity As the DoWhileActivity does not have a direct IWIR counterpart, we have to convert the DoWhileActivity into a construct involving an IWIR WhileTask. To convert the DoWhile into a While, we extract the first loop iteration and add it before the While. The translation process of a DoWhileActivity is illustrated in Figure 11 and works as follows: the agwl2iwir tool creates a new Blockscope and adds the translated body tasks to it. Then it creates a WhileActivity with the same body tasks, ports, constraints, properties and condition. This WhileActivity is then translated with the standard procedure and the resulting WhileTask is added to the body of the Blockscope.

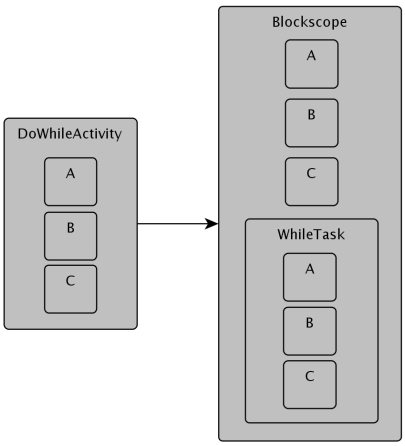

Figure 11: The translation of a DoWhileActivity into a WhileTask.

A concrete example for the translation of a DoWhileActivity into a WhileTask as seen in Figure 11 is shown below. Listing 2 represents the XML representation of an AGWL workflow containing a DoWhile loop. The translation to IWIR results in the XML representation shown in Listing 3.

Listing 2: DoWhile example in AGWL

```
<cgwd author="margit" domain="Generic" name="workflow" version="1.0.0">
  <cgwdInput>
    <dataIn name="input" source="" type="xs:integer"/>
  </cgwdInput>
  <cgwdBody>
    <dowhile name="doWhile">
      <condition conn="and">
        <acondition data1="doWhile/loop" data2="1" operator="&lt;="/>
      </condition>
      <dataLoops>
        <dataLoop initSource="workflow/input" loopSource="C/output"
          name="loop" type="xs:integer"/>
```

```
      </dataLoops>
      <loopBody>
        <activity name="A" type="aType">
          <dataIns>
            <dataIn name="input" source="doWhile/loop" type="xs:integer"/>
          </dataIns>
          <dataOuts>
            <dataOut name="output" type="xs:integer"/>
          </dataOuts>
        </activity>
        <activity name="B" type="aType">
          <dataIns>
            <dataIn name="input" source="A/output" type="xs:integer"/>
          </dataIns>
          <dataOuts>
            <dataOut name="output" type="xs:integer"/>
          </dataOuts>
        </activity>
        <activity name="C" type="aType">
          <dataIns>
            <dataIn name="input" source="B/output" type="xs:integer"/>
          </dataIns>
          <dataOuts>
            <dataOut name="output" type="xs:integer"/>
          </dataOuts>
        </activity>
      </loopBody>
      <dataOuts/>
      <dataUnions>
        <dataUnion name="result" source="doWhile/loop" type="agwl:collection"/>
      </dataUnions>
    </dowhile>
  </cgwdBody>
  <cgwdOutput>
    <dataOut name="output" source="doWhile/result" type="agwl:collection"/>
  </cgwdOutput>
</cgwd>
```

Listing 3: DoWhile example translated to IWIR

```
<IWIR xmlns="http://shiwa-workflow.eu/IWIR" version="1.1" wfname="workflow">
  <blockScope name="doWhile_block">
    <inputPorts>
      <inputPort name="loop" type="integer"/>
    </inputPorts>
    <body>
      <task name="A" tasktype="aType">
        <inputPorts>
          <inputPort name="input" type="integer"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="output" type="integer"/>
        </outputPorts>
      </task>
      <task name="B" tasktype="aType">
        <inputPorts>
          <inputPort name="input" type="integer"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="output" type="integer"/>
        </outputPorts>
      </task>
      <task name="C" tasktype="aType">
        <inputPorts>
          <inputPort name="input" type="integer"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="output" type="integer"/>
        </outputPorts>
      </task>
      <while name="doWhile">
```

```
        <inputPorts>
          <loopPorts>
            <loopPort name="loop" type="integer"/>
          </loopPorts>
        </inputPorts>
        <condition>loop &amp;lt;= 1</condition>
        <body>
          <task name="A" tasktype="aType">
            <inputPorts>
              <inputPort name="input" type="integer"/>
            </inputPorts>
            <outputPorts>
              <outputPort name="output" type="integer"/>
            </outputPorts>
          </task>
          <task name="B" tasktype="aType">
            <inputPorts>
              <inputPort name="input" type="integer"/>
            </inputPorts>
            <outputPorts>
              <outputPort name="output" type="integer"/>
            </outputPorts>
          </task>
          <task name="C" tasktype="aType">
            <inputPorts>
              <inputPort name="input" type="integer"/>
            </inputPorts>
            <outputPorts>
              <outputPort name="output" type="integer"/>
            </outputPorts>
          </task>
        </body>
        <outputPorts>
          <unionPorts>
            <unionPort name="result" type="collection/integer"/>
          </unionPorts>
        </outputPorts>
        <links>
          <link from="doWhile/loop" to="A/input"/>
          <link from="A/output" to="B/input"/>
          <link from="B/output" to="C/input"/>
          <link from="C/output" to="doWhile/loop"/>
          <link from="doWhile/loop" to="doWhile/result"/>
        </links>
      </while>
    </body>
    <outputPorts>
      <outputPort name="result" type="collection/integer"/>
    </outputPorts>
    <links>
      <link from="doWhile_block/loop" to="A/input"/>
      <link from="A/output" to="B/input"/>
      <link from="B/output" to="C/input"/>
      <link from="C/output" to="doWhile/loop"/>
      <link from="doWhile/result" to="doWhile_block/result"/>
    </links>
  </blockScope>
</IWIR>
```

SwitchActivity Because IWIR does not contain a switch statement, the translation leads to nested If tasks. An example is illustrated in Figure 12. The contents of the first remaining *case* branch is translated to IWIR and placed in the and *then* branch of a newly created IfTask. Then, we move to the *else* branch and recursively repeat the procedure for every remaining*case* branch of the switch activity. If the AGWL switch activity contains a default branch, its translated tasks are added to the innermost else branch. Otherwise the else branch of the innermost IfTask is omitted.

A concrete example for the translation of a SwitchActivity into nested IWIR If tasks as seen

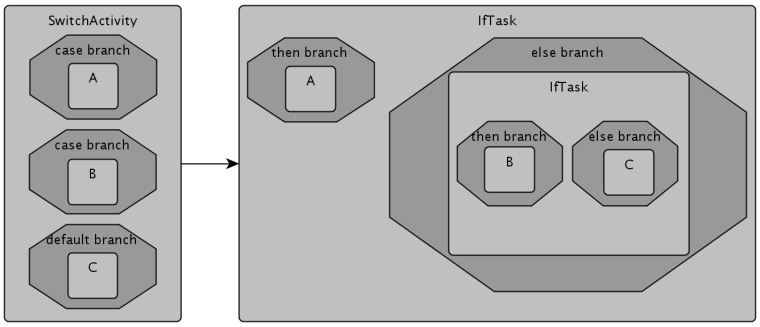

Figure 12: The translation of a SwitchActivity into nested IfTasks.

in Figure 12 is shown below. Listing 4 represents the XML representation of an AGWL workflow containing a DoWhile loop. The translation to IWIR results in the XML representation shown in Listing 5.

Listing 4: Switch example in AGWL

```
<cgwd author="margit" domain="Generic" name="workflow" version="1.0.0">
  <cgwdInput>
    <dataIn name="eval" source="" type="xs:integer"/>
    <dataIn name="data" source="" type="agwl:file"/>
  </cgwdInput>
  <cgwdBody>
    <switch name="switch">
      <dataEval name="eval" source="workflow/eval" type="xs:integer"/>
      <case break="true" value="1">
        <activity name="A" type="aType">
          <dataIns>
            <dataIn name="input" source="workflow/data" type="agwl:file"/>
          </dataIns>
          <dataOuts>
            <dataOut name="output" type="agwl:file"/>
          </dataOuts>
        </activity>
      </case>
      <case break="true" value="2">
        <activity name="B" type="aType">
          <dataIns>
            <dataIn name="input" source="workflow/data" type="agwl:file"/>
          </dataIns>
          <dataOuts>
            <dataOut name="output" type="agwl:file"/>
          </dataOuts>
        </activity>
      </case>
      <default>
        <activity name="C" type="aType">
          <dataIns>
            <dataIn name="input" source="workflow/data" type="agwl:file"/>
          </dataIns>
          <dataOuts>
            <dataOut name="output" type="agwl:file"/>
          </dataOuts>
        </activity>
      </default>
      <dataOuts>
        <dataOut name="result" source="A/output,B/output,C/output" type="agwl:file"/>
      </dataOuts>
    </switch>
  </cgwdBody>
  <cgwdOutput>
```

```
    <dataOut name="output" source="switch/result" type="agwl:file"/>
  </cgwdOutput>
</cgwd>
```

Listing 5: Switch example translated to IWIR

```
<IWIR xmlns="http://shiwa-workflow.eu/IWIR" version="1.1" wfname="workflow">
  <if name="switch">
    <inputPorts>
      <inputPort name="eval" type="integer"/>
      <inputPort name="input" type="file"/>
    </inputPorts>
    <condition>eval = 1</condition>
    <then>
      <task name="A" tasktype="aType">
        <inputPorts>
          <inputPort name="input" type="file"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="output" type="file"/>
        </outputPorts>
      </task>
    </then>
    <else>
      <if name="switch_1">
        <inputPorts>
          <inputPort name="eval" type="integer"/>
          <inputPort name="input" type="file"/>
        </inputPorts>
        <condition>eval = 2</condition>
        <then>
          <task name="B" tasktype="aType">
            <inputPorts>
              <inputPort name="input" type="file"/>
            </inputPorts>
            <outputPorts>
              <outputPort name="output" type="file"/>
            </outputPorts>
          </task>
        </then>
        <else>
          <task name="C" tasktype="aType">
            <inputPorts>
              <inputPort name="input" type="file"/>
            </inputPorts>
            <outputPorts>
              <outputPort name="output" type="file"/>
            </outputPorts>
          </task>
        </else>
        <outputPorts>
          <outputPort name="result" type="file"/>
        </outputPorts>
        <links>
          <link from="switch_1/input" to="B/input"/>
          <link from="switch_1/input" to="C/input"/>
          <link from="B/output" to="switch_1/result"/>
          <link from="C/output" to="switch_1/result"/>
        </links>
      </if>
    </else>
    <outputPorts>
      <outputPort name="result" type="file"/>
    </outputPorts>
    <links>
      <link from="switch/input" to="A/input"/>
      <link from="switch/input" to="switch_1/input"/>
      <link from="switch/eval" to="switch_1/eval"/>
      <link from="A/output" to="switch/result"/>
      <link from="switch_1/result" to="switch/result"/>
    </links>
```

```
66    </if>
67  </IWIR>
```

SequenceActivity, ParallelActivity, DAGActivity Tasks of an IWIR workflow that do not depend on each other, i.e. there exists no explicit data or control-flow path from one task to the other, can be executed in parallel, independent of the order in which they are given in the XML representation. In an AGWL workflow document, the order of occurrence of activities already implies a sequential control flow dependency. In AGWL, parallel statements have to be made explicit. Therefore, the translation of a SequenceActivity, a ParallelActivity and a DAGActivity differs from the standard translation process. As the dependencies between IWIR tasks are expressed with the link construct there is no need to translate SequenceActivities, ParallelActivities and DAGActivities into corresponding IWIR tasks. These tasks are omitted and only their child tasks are translated, the implicit sequential control dependencies amongst them are expressed with the help of IWIR control links. The agwl2iwir tool adds links between tasks if one of the following cases applies for their corresponding AGWL activities:

- Succeeding activities in an explicit or implicit sequence are linked:
  - activities inside a SequenceActivity (explicit sequence)
  - activities inside a DAGNode (implicit sequence)
  - activities inside the body of a ForActivity, ForEachActivity, ParallelForActivity, ParallelForEachActivity, WhileActivity or DoWhileActivity (implicit sequence)
  - activities inside the same branch of an IfActivity or SwitchActivity (implicit sequence)
- The first activity of a DAGNodes is linked to the last activity of a preceding DAGNodes.

As crucial dependencies between tasks are given only by the data links it is often desirable to omit the control links in the translation process. Therefore, the user can choose whether control links should be created during the translation process or not.

Ports AGWL ports have a one-to-one correspondance to IWIR ports, therefore each port is translated in the same way: an IWIR port of the corresponding type is created with the same name and the translated value type. The source is not translated, as dependecies in IWIR are represented by the link construct. The task that is responsible for a specific link uses the source information in order to create and add the link, as explained in Section 5.1.1.

Properties and constraints When translating an AGWL property or an AGWL constraint, an IWIR property, respectively IWIR constraints, with the same name and value is created.

Conditions The translation of conditions depends on the type of the AGWL condition, which is either a SimpleCondition object or an AtomicConditions object. In the former case the condition is already given as a string, therefore only operators and ports have to be adapted. There is a one-to-one correspondance between AGWL operators and IWIR operators, therefore they can be directly translated. AGWL conditions can refer to ports in any part of the workflow, whereas IWIR conditions are restricted to the current scope. This can make the creation and linking of intermediate ports necessary. This process is described in Section 5.1.1. In the case of the AtomicConditions object, which is an and-combined or or-combined list of $< data >< operator >< data >$, the given condition has to be decomposed to get a string representation. Again, ports and operators have to be adapted. Finally, a concrete example for the translation of conditions is given. The AGWL condition given as AtomicConditions in an XML representation is the following:

```
<condition conn="and">
  <acondition data1="activity/agwlport" data2="0" operator="&gt;="/>
  <acondition data1="activity/agwlport" data2="10" operator="&lt;="/>
</condition>
```

Given as SimpleCondition the AGWL condition is as follows:

```
<condition> activity/agwlport &gt;= 1 and activity/agwlport &lt;= 10 </condition>
```

The translated IWIR condition looks similar to the SimpleCondition, except the AGWL port is exchanged for the IWIR port:

```
<condition> iwirport &gt;= 1 and iwirport &lt;= 10  </condition>
```

Links The process of creating a link based on the source attribute of an AGWL port is different for the two groups of input ports and output ports. The link for an output port is added to the task itself, whereas the link for an input port is added to the parent task. This is because an AtomicTask contains no links, but the input ports of an AtomicActivity can have a source attribute, while the output ports can not. Therefore, during the translation of an AGWL activity into an IWIR task, the agwl2iwir tool creates links for its own output ports and the input ports of its child tasks. The source of the link is the source of the port, and the target of the link is the port itself.

AGWL conditions can refer to ports in any part of the workflow, whereas IWIR conditions are restricted to the current scope. This can make the creation and linking of intermediate ports necessary to reach the correct source port. This process is illustrated in Figure 13.

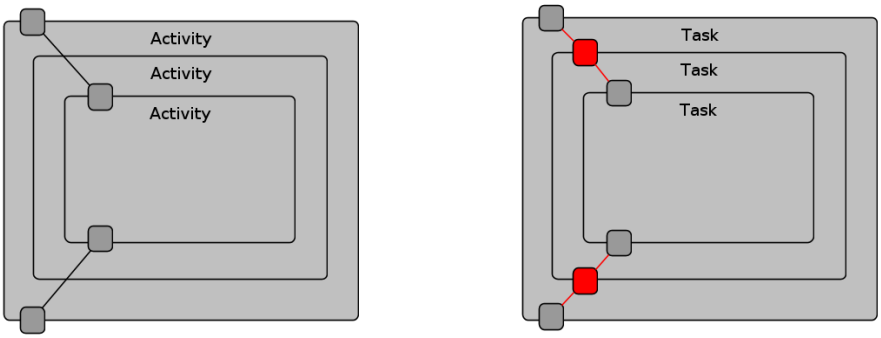

Figure 13: During the translation of links additional ports have to be created.

When searching for the source of an output port of the task itself the search procedure is as follows:

1. Search in the input ports of the task itself
2. Search in the output ports of the child tasks recursively. If the source port is not located at the task's direct child, output ports of all intermediate tasks have to be created and linked accordingly
3. Check if an input port of the task itself links to the same source
4. Create a new InputPort with the same source information: the search for the source port continues recursively at the parent task, until it can be found or the top-level task is reached

The search procedure for the source of an input port of a child task is similar, except step one is omitted and in step two the child task itself is not searched. Finally, we show a concrete example to illustrate the creation of links out of AGWL source port information. An AGWL port in XML representation is the following:

```
<activity name="targetActivity" type="aType">
  <dataIns>
    <dataIn name="targetAgwlPort" source="sourceActivity/sourceAgwlPort" type="agwl:file">
  </dataIns>
  ...
```

The created IWIR link in an XML representation is the following, where the activities *targetActivity* and *sourceActivity* translated to the task *targetTask*, respectively *sourceTask* and the AGWL ports *targetAgwlPort* and *sourceAgwlPort* translated to the IWIR port *targetIwirPort* and *sourceIwirPort*:

```
...
<links>
  <link from="sourceTask/sourceIwirPort" to="targetTask/targetIwirPort"/>
</links>
```

5.1.2 Conversion from IWIR to AGWL

The agwl2iwir tool is able to translate a given IWIR workflow into an equivalent AGWL workflow. The tool traverses the IWIR workflow in a top-down approach starting with the top-level task and translates each visited IWIR task into a corresponding AGWL activities. This process includes the translation of IWIR ports into AGWL ports, as well as IWIR properties and IWIR constraints into AGWL properties, respectively AGWL constraints. For most IWIR tasks there is exactly one corresponding AGWL activity. Exceptions are the WhileTask, the IfTask and the Blockscope. A WhileTask can be translated into a WhileActivity, or, if certain conditions are fullfilled, into a DoWhileActivity. In the case of an IfTask either an IfActivity is constructed or, if certain conditions are fullfilled, a SwitchActivity. A Blockscope can be translated either into a SequenceActivity, a ParallelActivity or a DAGActivity, depending on the data and control-flow links connecting the contained tasks.

The translation from an IWIR workflow into an AGWL workflow starts by translating the ports of the top-level task into global ports of the AGWL workflow. Then the top-level task is translated together with its ports, properties, constraints and child tasks. Recursively, all child tasks are translated into activites using this process. The resulting activity is defined as the top-level activity of the new AGWL workflow.

Tasks The translation of tasks depends on the task class. In general the process works as follows: first, an activity with the corresponding type is created. The activity uses either the same name or, if it is not unique throughout the whole workflow, a unique identifier is created. This unique name is created based on the task name and the name of all of its parent tasks, i.e. $< toplevel task >$. $< parenttask >$. $< childtask >$. The original name is stored as a property of the activity using the key *agwl-name*. Then, the ports (see paragraph Ports), the properties and constraints and the child tasks are translated. If the task contains a condition, which is the case for the IfTask and the WhileTask, also the condition is translated (see paragraph Conditions).

This standard process does not apply for the IfTask, the WhileTask and the Blockscope. The translation processes for these task classes are detailed below.

IfTask An IfTask can be translated into either an IfActivity or a SwitchActivity, depending on the structure of the IfTask. If the IfTask is not suitable to be translated into a SwitchActivity it is translated into an IfActivity according to the standard process. In order to result in a SwitchActivity, we have to find nested IfTasks, i.e. the *else* branch of the IfTask has to contain exactly one single IfTask, recursively. Furthermore, the IfTasks have to have the same ports, the conditions have to match a certain pattern and the tasks and its ports must not have any constraints. If these requirements are fulfilled, the nested IfTasks are translated into one single AGWL SwitchActivity

by translating each single one of the nested IfTasks into a separate case branch. If the else-branch of the innermost IfTask consists of at least one IWIR task, this translates to the default branch of the SwitchActivity. Figure 14 illustrates an IfTask suitable for the translation into a SwitchActivity.

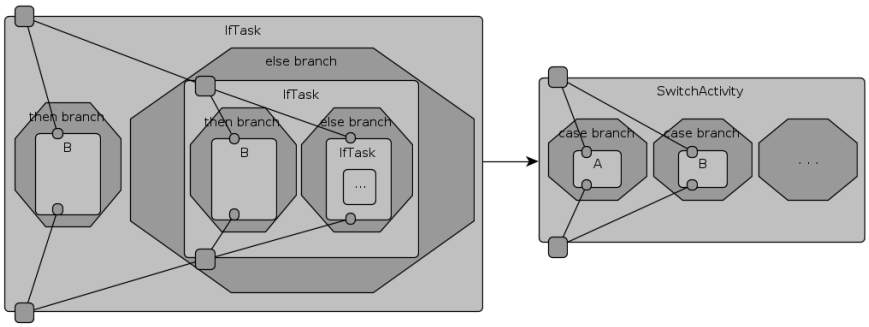

Figure 14: The translation of an IfTask into a SwitchActivity.

WhileTask A WhileTask can be translated either into a WhileActivity or into a DoWhileActivity, depending on the structure of the WhileTask and on its sibling tasks. If the WhileTask is not suitable to be translated into a DoWhileActivity, it is translated into a WhileActivity according to the standard process. A DoWhileActivity is only created if the complete body of the WhileTask is also contained in its parent task as direct predecessors to the WhileTask.

Blockscope On the one hand, Blockscopes can be explicit tasks, and on the other hand, every ForTask, ForEachTask, ParallelForTask, ParallelForEachTask, IfTask and WhileTask contains an implicit Blockscope: its body. In the following explanation, the term Blockscope covers both of these concepts.

In an AGWL workflow, dependencies are given by explicitly representing SequenceActivities, ParallelActivities and DAGActivities. The dependencies between the tasks in an IWIR workflow are given implicitly by means of the link construct. Therefore, to translate a Blockscope to AGWL, the dependencies of its contents have to be matched against sequential and parallel patterns.

First, the links contained in the Blockscope are parsed and a directed dependency graph is created. Every node of the graph represents a task. A connection from node A to node B represents a link from task A to task B. With the help of this graph, the tasks are matched against the parallel pattern and matching tasks are translated to an AGWL Parallel activity structure. Then, the remaining tasks are matched against the sequential pattern. Matching tasks are translated to an AGWL Sequence structure. Finally, the remaining tasks are translated into a DAGActivity.

In order to detect which tasks can be translated to an AGWL Parallel structure, the following procedure is executed for every node N in the graph: we construct all possible paths starting in N, to find the first node where all of these paths merge. All nodes prior this common successor are checked for the following requirements:

- There is no link between two nodes of different paths.
- All direct successors of N on all the paths have the same set of predecessors, the set P. For every path A, the predecessors of all nodes on path A are either contained in A or in the set P.
- For all paths A, No node on a path A links to a node outside of A.

If all of these requirements hold, a ParallelActivity is created out of all nodes on the paths from the starting node to the common successor. Examples of tasks executable in parallel are shown in Figure 15, whereas patterns of tasks not executable in parallel are illustrated in Figure 16. In the latter Figure the red links indicate dependencies that prevent the translation into an AGWL ParallelActivity.

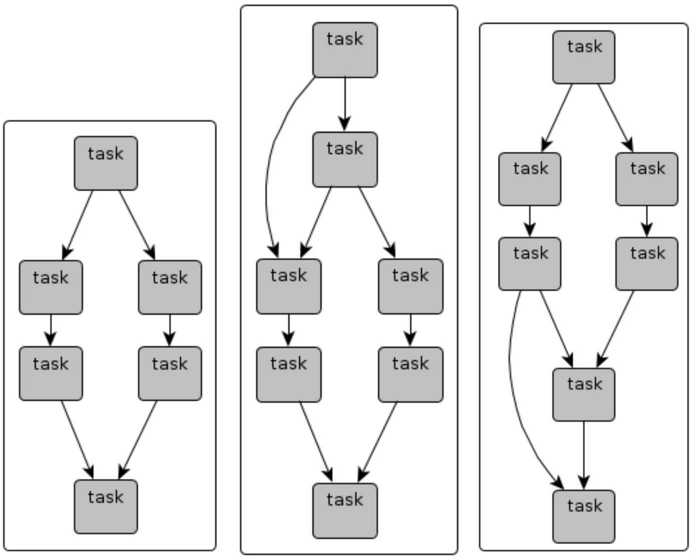

Figure 15: Task dependencies suitable for translation into an AGWL Parallel structure.

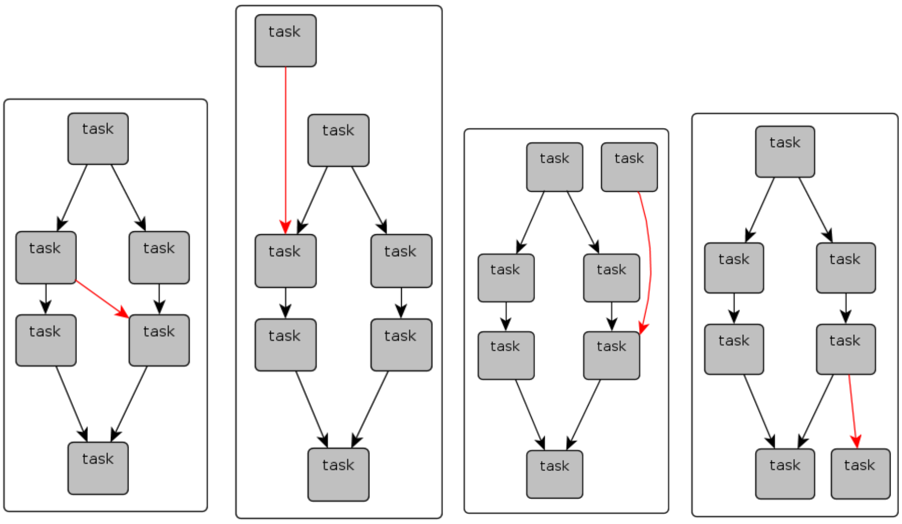

Figure 16: Task dependencies not suitable for translation into an AGWL Parallel structure.

The detection of a SequenceActivity follows next. The remaining nodes are checked if all, or some of them, are linked sequentially. This is the case if A links only to B and B has no other predecessor. Tasks that do not match either of these two cases are put into a DAGActivity structure upon translation.

Ports IWIR ports have a one-to-one correspondance to AGWL ports, therefore each port gets translated in the same way: an AGWL port of the corresponding type is created with the same name and data type. The source attribute of the AGWL port is created based on the link specified in the IWIR task containing the port, or its parent. Finally, its properties and constraints are translated.

Properties and constraints When translating an IWIR property or an IWIR constraint, an AGWL property, respectively AGWL constraints, with the same name and value is created.

Condition The translation of an IWIR condition results in one of the two differnt types of AGWL conditions, depending on the structure of the condition. If it is possible, then an AtomicConditions is created, which is either an and-combined or an or-combined list of $< data >< operator >< data >$. Otherwise a SimpleCondition is created, which is a simple string. Furthermore, the operators have to be adapted. If the condition contains ports the translation also needs to search for the corresponding AGWL port.

Finally, a concrete example for the translation of conditions is given. The IWIR condition as XML representation is the following:

```
<condition> iwirport &gt;= 1 or iwirport &lt;= 10  </condition>
```

The translated AGWL condition represented as a SimpleCondition looks similar to the IWIR condition, except the IWIR port is exchanged for the AGWL port:

```
<condition> activity/agwlport &gt;= 1 or activity/agwlport &lt;= 10 </condition>
```

Given as AtomicConditions the corresponding AGWL condition is as follows:

```
<condition conn="or">
  <acondition data1="activity/agwlport" data2="1" operator="&lt;="/>
  <acondition data1="activity/agwlport" data2="10" operator="&lt;="/>
</condition>
```

5.1.3 Integration in ASKALON

We have integrated the iwir2agwl tool as well as the agwl2iwir tool into the graphical user interface of ASKALON. This allows workflows to be imported from as well as exported to an IWIR representation directly in the native ASKALON user interface.

5.2 Triana Taskgraph

Triana has been extended with importers and exporters for the IWIR workflow format. The GUI has an option in the extensions menu to import an IWIR file, and a command-line option allows the importing of IWIR either in file format, or as part of a SHIWA bundle.

Exporting from Triana taskgraph format to IWIR can also be achieved via both GUI and command-line options. Also, within the GUI, when submitting a SHIWA bundle to a workflow repository, the option is given to first translate the designed Triana workflow into the IWIR format before creating the bundle. This allows direct publication of an IWIR described workflow to the SHIWA repository.

The convertor has been written as an addon for Triana, so the installation and execution of a conversion is through the use of Triana. The triana-shiwa module is placed in either the toolbox or modules folder before startup, and is discovered when either the GUI loads, or the ”-c” option is given at the command line. e.g.

```
./triana.sh -w taskgraph.xml -c iwir
```

In the case of conversion from IWIR, the opposite invocation is required.

```
./triana.sh -w iwir.xml -c taskgraph
```

If an impossible translation or incorrect language name is given, an appropriate error message is returned.

5.2.1 Triana Taskgraph to IWIR

Conversion from Triana Taskgraph to IWIR is performed iteratively, traversing through each "layer" of the taskgraph, producing a series of Blockscopes containing either AtomicTasks or further Blockscopes. The connections between nodes in the taskgraphs are retained during this traversal, and a mapping of Triana Nodes to IWIR Ports is created. Once the IWIR Tasks and their ports are created, the Triana connections are traversed, and the Node-¿Port mappings are used to create the connections between the scopes of the Blockscopes.

Unit parameters are used within Triana to define the variables used when the workflow executes. These are stored within the IWIR in the properties tag as name/value pairs.

The AtomicTasks "TaskType" is set to the canonical classname of the Triana units class, i.e.

```
common.string.Concat
```

This will likely be extended in future to:

```
Triana:defaultToolboxes:common.string.Concat
```

in order to distinguish between different implementations of tasktypes for other workflow engines.

Various audio and graphic manipulation workflows created in Triana have been successfully converted through IWIR and subsequently been imported into the ASKALON workflow engine, proving the functionality of the conversions. The validity of the ASKALON conversion back to IWIR and the importation back into Triana was displayed, and was the first demonstration of a workflow being edited on two workflow engines within the SHIWA project.

Below is an audio manipulation workflow created in Triana, followed by its conversion into IWIR.

Listing 6: Audio Manipulation workflow in Triana Taskgraph format

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<tool>
    <toolname>HauntedHouse</toolname>
    <package>audio.processing.userpresets</package>
    <version>0.1-SNAPSHOT</version>
    <inportnum>1</inportnum>
    <outportnum>1</outportnum>
    <inparam/>
    <outparam/>
    <input>
        <node index="0">
            <type>triana.types.audio.MultipleAudio</type>
        </node>
    </input>
    <output>
        <node index="0">
            <type>triana.types.audio.MultipleAudio</type>
        </node>
    </output>
    <parameters>
        <param name="guiY" type="gui">
            <value>4.0</value>
        </param>
        <param name="guiX" type="gui">
            <value>1.969298245614035</value>
```

```
        </param>
        <param name="popUpDescription" type="unknown">
            <value>No description for tool</value>
        </param>
    </parameters>
    <tasks>
        <taskgraph>
            <toolname>Large Room</toolname>
            <package>audio.processing.reverb</package>
            <version>0.1</version>
            <inportnum>1</inportnum>
            <outportnum>1</outportnum>
            <inparam/>
            <outparam/>
            <input>
                <node index="0">
                    <type>triana.types.audio.MultipleAudio</type>
                </node>
            </input>
            <output>
                <node index="0">
                    <type>triana.types.audio.MultipleAudio</type>
                </node>
            </output>
            <parameters>
                <param name="guiY" type="gui">
                    <value>0.02631578947368407</value>
                </param>
                <param name="guiX" type="gui">
                    <value>1.2017543859649122</value>
                </param>
                <param name="popUpDescription" type="unknown">
                    <value>No description for tool</value>
                </param>
            </parameters>
            <tasks>
                ....

                ....

                <connections>
                    <connection type="NonRunnable">
                        <source node="3" taskname="Fader"/>
                        <target node="0" taskname="CombDelay3"/>
                    </connection>
                    ....

                  ....
                    <connection type="NonRunnable">
                        <source node="0" taskname="NodeMixer"/>
                        <target node="0" taskname="AllPass"/>
                    </connection>
                </connections>
                <groupnodemapping>
                    <input>
                        <node externalnode="0" node="0" taskname="Fader"/>
                    </input>
                    <output>
                        <node externalnode="0" node="0" taskname="AllPass1"/>
                    </output>
                </groupnodemapping>
            </tasks>
        </taskgraph>
        <task>
            <toolname>Reverse</toolname>
            <package>audio.processing.tools</package>
            <version>0.1</version>
            <proxy type="Java">
                <param paramname="unitName">
                    <value>Reverse</value>
                </param>
```

```
                <param paramname="unitPackage">
                    <value>audio.processing.tools</value>
                </param>
            </proxy>
            <inportnum>1</inportnum>
            <outportnum>1</outportnum>
            <inparam/>
            <outparam/>
            <input>
                <type>triana.types.audio.MultipleAudio</type>
            </input>
            <output>
                <type>triana.types.audio.MultipleAudio</type>
            </output>
            <parameters>
                <param name="guiY" type="gui">
                    <value>0.05263157894736814</value>
                </param>
                <param name="guiX" type="gui">
                    <value>0.0</value>
                </param>
            </parameters>
        </task>
        <task>
            <toolname>Reverse1</toolname>
            <package>audio.processing.tools</package>
            <version>0.1</version>
            <proxy type="Java">
                <param paramname="unitName">
                    <value>Reverse</value>
                </param>
                <param paramname="unitPackage">
                    <value>audio.processing.tools</value>
                </param>
            </proxy>
            <inportnum>1</inportnum>
            <outportnum>1</outportnum>
            <inparam/>
            <outparam/>
            <input>
                <type>triana.types.audio.MultipleAudio</type>
            </input>
            <output>
                <type>triana.types.audio.MultipleAudio</type>
            </output>
            <parameters>
                <param name="guiY" type="gui">
                    <value>0.0</value>
                </param>
                <param name="guiX" type="gui">
                    <value>2.4649122807017543</value>
                </param>
            </parameters>
        </task>
        <connections>
            <connection type="NonRunnable">
                <source node="0" taskname="Reverse"/>
                <target node="0" taskname="Large Room"/>
            </connection>
            <connection type="NonRunnable">
                <source node="0" taskname="Large Room"/>
                <target node="0" taskname="Reverse1"/>
            </connection>
        </connections>
        <groupnodemapping>
            <input>
                <node externalnode="0" node="0" taskname="Reverse"/>
            </input>
            <output>
                <node externalnode="0" node="0" taskname="Reverse1"/>
            </output>
```

```
        </groupnodemapping>
    </tasks>
</tool>
```

Listing 7: Audio Manipulation converted to IWIR

```
<IWIR version="1.1" wfname="HauntedHouse" xmlns="http://shiwa-workflow.eu/IWIR">
  <blockScope name="HauntedHouse">
    <inputPorts>
      <inputPort name="in1" type="string"/>
    </inputPorts>
    <body>
      <task name="NodeMixer" tasktype="audio.processing.tools.NodeMixer">
        <inputPorts>
          <inputPort name="in2" type="string"/>
          <inputPort name="in4" type="string"/>
          <inputPort name="in1" type="string"/>
          <inputPort name="in3" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="out1" type="string"/>
        </outputPorts>
        <properties>
          <property name="normal" value="100"/>
        </properties>
      </task>
      <task name="Fader" tasktype="audio.processing.tools.Fader">
        <inputPorts>
          <inputPort name="in1" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="out1" type="string"/>
          <outputPort name="out2" type="string"/>
          <outputPort name="out3" type="string"/>
          <outputPort name="out4" type="string"/>
        </outputPorts>
        <properties>
          <property name="volumeInDB" value="0"/>
        </properties>
      </task>
        ....

        ....
      <task name="AllPass1" tasktype="audio.processing.delay.AllPass">
        <inputPorts>
          <inputPort name="in1" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="out1" type="string"/>
        </outputPorts>
        <properties>
          <property name="delayInMs" value="5.000"/>
          <property name="feedback" value="70"/>
          <property name="chunked" value="false"/>
        </properties>
      </task>
      <task name="Reverse1" tasktype="audio.processing.tools.Reverse">
        <inputPorts>
          <inputPort name="in1" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="out1" type="string"/>
        </outputPorts>
      </task>
      <task name="Reverse" tasktype="audio.processing.tools.Reverse">
        <inputPorts>
          <inputPort name="in1" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="out1" type="string"/>
        </outputPorts>
```

```
      </task>
    </body>
    <outputPorts>
      <outputPort name="out1" type="string"/>
    </outputPorts>
    <links>
      <link from="CombDelay1/out1" to="NodeMixer/in2"/>
      <link from="NodeMixer/out1" to="AllPass/in1"/>
      <link from="Fader/out2" to="CombDelay1/in1"/>
      <link from="AllPass1/out1" to="Reverse1/in1"/>
      <link from="Reverse/out1" to="Fader/in1"/>
      <link from="AllPass/out1" to="AllPass1/in1"/>
      <link from="HauntedHouse/in1" to="Reverse/in1"/>
      <link from="Reverse1/out1" to="HauntedHouse/out1"/>
    </links>
  </blockScope>
</IWIR>
```

5.2.2 IWIR to Triana Taskgraph

Each Blockscope within IWIR becomes a taskgraph within Triana, and each AtomicTask becomes a task. This basic rule follows throughout all forms of conversion, with the addition of control tasks when required. The Blockscope task within IWIR allows the collation of multiple AtomicTasks together, including other compound tasks (including other Blockscopes). This pattern matches the usage of taskgraphs within Triana exactly. In this way, conversion to and from pipeline (DAG) workflows without control tasks or loops is possible with a high degree of accuracy.

Other compound tasks within IWIR required the production of new Unit task objects within Triana, in order to achieve the same functionality defined by the language, and used by other engines. The production of these control Units is under constant adaption to allow the construction of as many of the compound tasks as possible.

The control Units created for Triana map directly to the control tasks listed in IWIR, namely ForEach, For, If, ParallelForEach, ParallelFor and While. When a compound tak of this type is found when traversing the IWIR workflow, a Triana Taskgraph is created containing a control unit of the relevant type. The inputs and outputs of the compound task are then connected to the control task, and the task performs the appropriate action on execution of the graph. For example, a handle to the IWIR ConditionExpression object is held in the If and While Units, in order to be evaluated during execution of the workflow.

Using the package and class name of tools as TaskType means importing an IWIR task can easily be converted back to a Triana unit. As all toolboxes are loaded into Triana in initialisation, it is easy to check if the relevant Triana unit is present, and if it can be loaded. If the TaskType is not recognised, a mechanism is in place to retrieve an executable from an external "tasktype repository"; this will be extended once the codebase for hosting and retrieval of TaskTypes has been implemented. For the time being, the importer uses a placeholder "passThrough" unit instead of the expected atomic task, which simply outputs whatever it receives as input. Clearly this means the workflow will not perform as expected, but the abstract form of the workflow is shown visually within the Triana GUI, so it is potentially possible to replace the missing unit with another equivalent Unit if one is available.

5.2.3 Triana Modifications

It was necessary during the construction of the importer to modify Triana in a number of ways. Triana's inports and outports, known as nodes, do not have names. Instead Triana allocates them numbers when they are created. In some cases these numbers can be substituted for names, and vice versa, but it is important to keep track of which portname maps to which node number for a task.

Another issue is the case where the output node of one atomic task within the IWIR is connected to another port outside the scope of the first. In a Triana taskgraph this would be a connection between two nodes, one on the sending node, and the receiving node would be attached to the parent taskgraph so that the scopes of the two nodes become the same. This is a different idea to the IWIR description, where the output port is joined to an output port of the parent compound task, and then from that node onwards to the atomic tasks receiving port. Traversing scopes of graphs to connect nodes proved complex, especially in the case where two leaves of a workflow graphs tree were connected multiple parents up.

5.3 Pegasus and IWIR

Work has been made to incorporate the Pegasus workflow system within the SHIWA project, in that IWIR workflows can be imported into Triana, and exported as Pegasus DAX files, as well as importing DAX and exporting as IWIR.

Current issues are that the input files are listed within the workflow definition file, and so the workflow appears to have no inputs. Future work will be done to ascertain which files described within the DAX xml format are not outputs from the Job items listed within the DAX definition. There will be an assumption that if a file is not an output, and so is not created during the running of the workflow, then it is an input required on initialisation of the graph, and so can be listed as input ports for use when creating the workflow's signature. This signature will be used to define the input ports of the IWIR description when produced by Triana.

There exists a limitation within Pegasus in that only DAG (directed acyclic graphs) may be run. The majority of Triana workflows are DAG graphs so the translation here is relativly easy. However, the production of any loops or control task driven workflows will require intervention by a workflow designer for either the original engine or Triana to recreate the required data flow in DAG format before conversion to Pegasus DAX is possible.

5.4 gUSE

To provide interoperability between gUSE and other workflow management systems, two standalone JAVA applications are developed to perform conversion between native gUSE, and IWIR languages. One for handling conversion in gUSE-IWIR direction,and one for backward conversions. Each converter are standalone applications, integration into WS-Pgrade web-portal is planned.

5.4.1 Usage

Converters of both directions can be used via command line. GuseIwirConverter.jar performs the conversion from gUSE to IWIR language. Usage of the converters are shown in Listing 8 and Listing 9.

Listing 8: guse-iwir converter

```
1 java   jar  GuseIwirConverter.jar filename.data
```

In this command filename.data is the gUSE graph descriptor file to be converted. The output will be named as filename_out.iwir, and will be placed in the same directory as where the input file is. For the opposite direction IwirGuseConverter.jar can be used:

Listing 9: iwir-guse converter

```
1 java   jar  IwirGuseConverter.jar filename
```

In this command IWIR graph descriptor file named to filename will be converted to gUSE workflow descriptor file, and the output will be placed in the same directory as where the input file

it. The output file will be named workflow.data. The converter will make the compressed version of the gUSE graph named workflow_name.zip. This type of output is supported by WS-PGRADE portal to be uploaded successfully.

5.4.2 Restrictions

1. In case of gUSE-> IWIR conversion with the recent IWIR specification we could not convert those jobs which have dot- and cross product relation among its inputs at the same time. Therefore the converter does not support conversion of this kind of jobs, and exception will be thrown.

2. The if-tasks are not convertible, because in gUSE conditions can be defined only for the input ports.

3. In case of IWIR-> gUSE conversion only the parallelForEach loops are convertible.

5.4.3 Conversion example

Let us consider 2 simple workflows shown in Figure 17 and 18. The structure of the workflows are the same, only 2 jobs are defined and connected. The first one is to generate a set of inputs, that will be transferred to the second one as an input. In the two figures the name second jobs are different according to the following input generalization strategy. In the first figure DotJob job will get dot product of all inputs, in the second figure CrossJob will get Chartesian-product of sets of inputs. As many job instances will be submitted as many members the generated set has. Native gUSE description of dot producted workflow is shown in Listing 10, the result of its conversion in Listing 11. Cross-producted workflow is described in Listing 12 and the converted IWIR structure does in Listing 13.

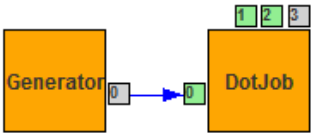

Figure 17: Workflow with dot product

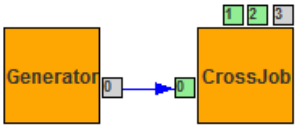

Figure 18: Workflow with cross product

Listing 10: dot Product gUSE-IWIR direction

```
1
2 <real abst="" graf="Dot_wf_Graf" name="Dot_wf" text="2012-1-10">
3 <job name="Generator" text="Description of Job" x="8" y="8">
4    <output name="OUTPUT" seq="0" text="Description of Port" x="38" y="68">
5      <port_prop desc="null" inh="null" key="intname" label="null" value="OUTPUT"/>
6      <port_prop desc="null" inh="null" key="maincount" label="null" value="2"/>
7    </output>
```

```
</job>
<job name="DotJob" text="Description of Job" x="8" y="8">
<input name="INPUT1" prejob="Generator" preoutput="0" seq="0" text="Description of Port" x
    ="38" y="-7">
    <port_prop desc="null" inh="null" key="intname" label="null" value="INPUT1"/>
    <port_prop desc="null" inh="null" key="value" label="null" value="3"/>
    <port_prop desc="null" inh="null" key="dpid" label="null" value="0"/>
    <port_prop desc="null" inh="null" key="waiting" label="null" value="one"/>
    <port_prop desc="null" inh="null" key="pequaltype" label="null" value="0"/>
</input>
  <input name="INPUT2" prejob="" preoutput="" seq="1" text="Description of Port" x="23" y
      ="-7">
    <port_prop desc="null" inh="null" key="intname" label="null" value="INPUT1"/>
    <port_prop desc="null" inh="null" key="value" label="null" value="3"/>
    <port_prop desc="null" inh="null" key="dpid" label="null" value="1"/>
    <port_prop desc="null" inh="null" key="waiting" label="null" value="one"/>
    <port_prop desc="null" inh="null" key="pequaltype" label="null" value="0"/>
    <port_prop desc="null" inh="null" key="max" label="null" value="10"/>
 </input>
 <input name="INPUT3" prejob="" preoutput="" seq="2" text="Description of Port" x="8" y
     ="-7">
    <port_prop desc="null" inh="null" key="intname" label="null" value="INPUT1"/>
    <port_prop desc="null" inh="null" key="value" label="null" value="3"/>
    <port_prop desc="null" inh="null" key="dpid" label="null" value="2"/>
    <port_prop desc="null" inh="null" key="waiting" label="null" value="one"/>
    <port_prop desc="null" inh="null" key="pequaltype" label="null" value="0"/>
    <port_prop desc="null" inh="null" key="max" label="null" value="10"/>
 </input>
     <output name="OUTPUT" seq="3" text="Description of Port" x="38" y="68">
        <port_prop desc="null" inh="null" key="intname" label="null" value="OUTPUT"/>
        <port_prop desc="null" inh="null" key="maincount" label="null" value="1"/>
      </output>
</job>
</real>

```

Listing 11: Dot product gUSE-IWIR direction

```
<IWIR version="1.1" wfname="Dot_wf_example" xmlns="http://shiwa-workflow.eu/IWIR">
  <blockScope name="Dot_wf">
    <inputPorts>
      <inputPort name="bs_in0" type="collection/file"/>
      <inputPort name="bs_in1" type="collection/file"/>
    </inputPorts>
    <body>
      <task name="Generator" tasktype="type">
        <inputPorts/>
        <outputPorts>
          <outputPort name="OUTPUT" type="collection/file"/>
        </outputPorts>
      </task>
      <parallelForEach name="dotforeach0">
        <inputPorts>
          <loopElements>
            <loopElement name="INPUT1" type="collection/file"/>
            <loopElement name="INPUT2" type="collection/file"/>
            <loopElement name="INPUT3" type="collection/file"/>
          </loopElements>
        </inputPorts>
        <body>
          <task name="DotJob" tasktype="type">
            <inputPorts>
              <inputPort name="INPUT1" type="file"/>
              <inputPort name="INPUT2" type="file"/>
              <inputPort name="INPUT3" type="file"/>
            </inputPorts>
            <outputPorts>
              <outputPort name="OUTPUT" type="file"/>
            </outputPorts>
          </task>
        </body>
```

```
        <outputPorts>
          <outputPort name="OUTPUT" type="collection/file"/>
        </outputPorts>
        <links>
          <link from="dotforeach0/INPUT1" to="DotJob/INPUT1"/>
          <link from="dotforeach0/INPUT2" to="DotJob/INPUT2"/>
          <link from="dotforeach0/INPUT3" to="DotJob/INPUT3"/>
          <link from="DotJob/OUTPUT" to="dotforeach0/OUTPUT"/>
        </links>
      </parallelForEach>
    </body>
    <outputPorts>
      <outputPort name="bs_out_0" type="collection/file"/>
    </outputPorts>
    <links>
      <link from="Generator/OUTPUT" to="dotforeach0/INPUT1"/>
      <link from="topLevel/bs_in0" to="dotforeach0/INPUT2"/>
      <link from="topLevel/bs_in1" to="dotforeach0/INPUT3"/>
      <link from="dotforeach0/OUTPUT" to="topLevel/bs_out_0"/>
    </links>
  </blockScope>
</IWIR>
```

Cross product The composition is the same as above, except that the inputs of the second job are in cross product relation and one input is a simple file input.

Listing 12: Cross product gUSE-IWIR direction

```
<real abst="" graf="Cross_wf_Graf" name="Cross_wf" text="2012-1-10">
<job name="Generator" text="Description of Job" x="8" y="8">
   <output name="OUTPUT" seq="0" text="Description of Port" x="38" y="68">
        <port_prop desc="null" inh="null" key="intname" label="null" value="OUTPUT"/>
        <port_prop desc="null" inh="null" key="maincount" label="null" value="2"/>
   </output>
</job>
<job name="DotJob" text="Description of Job" x="8" y="8">
  <input name="INPUT1" prejob="Generator" preoutput="0" seq="0" text="Description of Port" x
      ="38" y="-7">
    <port_prop desc="null" inh="null" key="intname" label="null" value="INPUT1"/>
    <port_prop desc="null" inh="null" key="value" label="null" value="3"/>
    <port_prop desc="null" inh="null" key="dpid" label="null" value="0"/>
    <port_prop desc="null" inh="null" key="waiting" label="null" value="one"/>
    <port_prop desc="null" inh="null" key="pequaltype" label="null" value="0"/>
  </input>
  <input name="INPUT2" prejob="" preoutput="" seq="1" text="Description of Port" x="23" y
      ="-7">
    <port_prop desc="null" inh="null" key="intname" label="null" value="INPUT1"/>
    <port_prop desc="null" inh="null" key="value" label="null" value="3"/>
    <port_prop desc="null" inh="null" key="dpid" label="null" value="0"/>
    <port_prop desc="null" inh="null" key="waiting" label="null" value="one"/>
    <port_prop desc="null" inh="null" key="pequaltype" label="null" value="0"/>
    <port_prop desc="null" inh="null" key="max" label="null" value="10"/>
 </input>
 <input name="INPUT3" prejob="" preoutput="" seq="2" text="Description of Port" x="8" y
     ="-7">
    <port_prop desc="null" inh="null" key="intname" label="null" value="INPUT1"/>
    <port_prop desc="null" inh="null" key="value" label="null" value="3"/>
    <port_prop desc="null" inh="null" key="dpid" label="null" value="0"/>
    <port_prop desc="null" inh="null" key="waiting" label="null" value="one"/>
    <port_prop desc="null" inh="null" key="pequaltype" label="null" value="0"/>
    <port_prop desc="null" inh="null" key="max" label="null" value="10"/>
 </input>
     <output name="OUTPUT" seq="3" text="Description of Port" x="38" y="68">
        <port_prop desc="null" inh="null" key="intname" label="null" value="OUTPUT"/>
        <port_prop desc="null" inh="null" key="maincount" label="null" value="2"/>
      </output>
</job>
</real>

```

Listing 13: Cross product gUSE-IWIR direction

```
1
2  <IWIR version="1.1" wfname="Cross_wf_example" xmlns="http://shiwa-workflow.eu/IWIR">
3    <blockScope name="Cross_wf">
4      <inputPorts>
5        <inputPort name="bs_in0" type="file"/>
6        <inputPort name="bs_in1" type="collection/file"/>
7      </inputPorts>
8      <body>
9        <task name="Generator" tasktype="type">
10         <inputPorts/>
11         <outputPorts>
12           <outputPort name="OUTPUT" type="collection/file"/>
13         </outputPorts>
14       </task>
15       <parallelForEach name="foreach0_0">
16         <inputPorts>
17           <inputPort name="INPUT2" type="file"/>
18           <inputPort name="INPUT3" type="collection/file"/>
19           <loopElements>
20             <loopElement name="INPUT1" type="collection/file"/>
21           </loopElements>
22         </inputPorts>
23         <body>
24           <parallelForEach name="foreach0_1">
25             <inputPorts>
26               <inputPort name="INPUT1" type="file"/>
27               <inputPort name="INPUT2" type="file"/>
28               <loopElements>
29                 <loopElement name="INPUT3" type="collection/file"/>
30               </loopElements>
31             </inputPorts>
32             <body>
33               <task name="DotJob" tasktype="type">
34                 <inputPorts>
35                   <inputPort name="INPUT2" type="file"/>
36                   <inputPort name="INPUT1" type="file"/>
37                   <inputPort name="INPUT3" type="file"/>
38                 </inputPorts>
39                 <outputPorts>
40                   <outputPort name="OUTPUT" type="file"/>
41                 </outputPorts>
42               </task>
43             </body>
44             <outputPorts>
45               <outputPort name="OUTPUT" type="collection/file"/>
46             </outputPorts>
47             <links>
48               <link from="DotJob/OUTPUT" to="foreach0_1/OUTPUT"/>
49               <link from="foreach0_1/INPUT1" to="DotJob/INPUT1"/>
50               <link from="foreach0_1/INPUT3" to="DotJob/INPUT3"/>
51               <link from="foreach0_1/INPUT2" to="DotJob/INPUT2"/>
52             </links>
53           </parallelForEach>
54         </body>
55         <outputPorts>
56           <outputPort name="OUTPUT" type="collection/file">
57             <constraints>
58               <constraint name="flatten-collection" value="1"/>
59             </constraints>
60           </outputPort>
61         </outputPorts>
62         <links>
63           <link from="foreach0_1/OUTPUT" to="foreach0_0/OUTPUT"/>
64           <link from="foreach0_0/INPUT1" to="foreach0_1/INPUT1"/>
65           <link from="foreach0_0/INPUT3" to="foreach0_1/INPUT3"/>
66           <link from="foreach0_0/INPUT2" to="foreach0_1/INPUT2"/>
67         </links>
68       </parallelForEach>
69     </body>
70     <outputPorts>
```

```
71        <outputPort name="bs_out_0" type="collection/file"/>
72      </outputPorts>
73      <links>
74        <link from="Generator/OUTPUT" to="foreach0_0/INPUT1"/>
75        <link from="topLevel/bs_in0" to="foreach0_0/INPUT2"/>
76        <link from="topLevel/bs_in1" to="foreach0_0/INPUT3"/>
77        <link from="foreach0_0/OUTPUT" to="topLevel/bs_out_0"/>
78      </links>
79    </blockScope>
80  </IWIR>
```

5.5 Conversion from GWENDIA to IWIR

GWENDIA [5] defines four data types (`integer`, `double`, `string` and `URI`) that are also available in IWIR (the `URI` type corresponding to IWIR `file` type as it is simply a reference to a file). GWENDIA also manipulate homogeneous nested arrays of data items of any type, that are represented as homogeneous collections of data of any types in IWIR.

The GWENDIA language has the concepts of workflow activities, data links and coordination links which map to IWIR tasks and links (between data ports or between activities) respectively. It also includes conditionals that can be represented by IWIR `if` construct. The biggest difference between GWENDIA and IWIR is the implicit data parallelism described in GWENDIA workflow that can be represented in IWIR through loop constructs but that need to be made explicit. In this section, we explain how GWENDIA implicit iterations are converted to explicit ones in IWIR.

5.5.1 Iterations in GWENDIA

The dimension of a GWENDIA array is referred to as its nesting level. GWENDIA input and output processor ports have depths that corresponds to the nesting level of the data structure that they consume and produce respectively. For instance, a processor computing the mean of an array of integers might take a 1D array (nesting level = 1) as input and produce a scalar (nesting level = 0) as output.

GWENDIA iterates over data structures received when processing the workflow. There are two kinds of iterations that can occur:

- Implicit iterations happen when a processor is provided with "more" data than it expects, as in data with a higher nesting level. The processor is fired once for each element at the appropriate nesting level in the input data.
- Explicit iterations are caused by explicit data combination operators (known as iteration strategies) that are needed for each activity with more than one input data port (thus mixing more than one data flow).

The two most common iteration strategies are the *dot product* and the *cross product* (with its variant *flat cross product*). (GWENDIA also features a *match product* that implies some knowledge on the data semantics and is not supported in IWIR).

Dot product matches the elements by pairs, based on their indexes. For this strategy to make sense, both arguments must be of the same nesting level.

Cross product builds all possible combinations of all elements. It iterates on its first argument and, for each element, creates a list of all combinations with its second argument. The cross product increases the nesting level of data.

Flat cross product is a variation of the cross product that does not increase the nesting level of data: it produces the same combinations in the same order as the regular cross product, but the resulting list is flattened, hence the name.

5.5.2 Conversion example

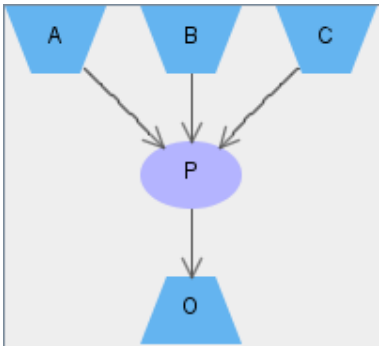

Figure 19: Simple GWENDIA workflow

Let us consider the simple workflow shown in Figure 19 with only one activity `P` that has three inputs `A`, `B` and `C` and one output `O`. The GWENDIA XML representation of this workflow is shown in Listing 14. It shows in line 23 that a dot product iteration strategy is considered between all 3 workflow inputs.

Listing 14: Simple workflow (GWENDIA)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <workflow name="Name" version="0.1" author="Author">
3    <description>Description</description>
4    <interface>
5      <source name="A" type="string">
6        <source-comment>Comment A</source-comment>
7      </source>
8      <source name="B" type="string">
9        <source-comment>Comment B</source-comment>
10     </source>
11     <source name="C" type="string">
12       <source-comment>Comment C</source-comment>
13     </source>
14     <sink name="O" type="string" />
15   </interface>
16   <processors>
17     <processor name="P" >
18       <in name="in1" type="string" depth="0" />
19       <in name="in2" type="string" depth="0" />
20       <in name="in3" type="string" depth="0" />
21       <out name="out" type="string" depth="0" />
22       <iterationstrategy>
23         <dot>
24           <port name="in1" />
25           <port name="in2" />
26           <port name="in3" />
27         </dot>
28       </iterationstrategy>
29     </processor>
30   </processors>
31   <links>
32       <link from="A" to="P:in1" />
33       <link from="B" to="P:in2" />
34       <link from="C" to="P:in3" />
35       <link from="P:out" to="O" />
36   </links>
37 </workflow>
```

Listing 15 is the result of converting this GWENDIA workflow with scalar input data (no nesting). There is no iteration, in this case, since nesting levels and port depths match, thus the iteration strategy does not matter, in the sense that it won't impact the resulting IWIR workflow.

Listing 15: Base workflow (IWIR)

```
<IWIR version="1.1" wfname="Name" xmlns="http://shiwa-workflow.eu/IWIR">
  <blockScope name="Name">
    <inputPorts>
      <inputPort name="A" type="string"/>
      <inputPort name="B" type="string"/>
      <inputPort name="C" type="string"/>
    </inputPorts>
    <body>
      <task name="P">
        <inputPorts>
          <inputPort name="in1" type="string"/>
          <inputPort name="in2" type="string"/>
          <inputPort name="in3" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="out" type="string"/>
        </outputPorts>
      </task>
    </body>
    <outputPorts>
      <outputPort name="O" type="string"/>
    </outputPorts>
    <links>
      <link from="Name_main_block/A" to="P/in1"/>
      <link from="Name_main_block/B" to="P/in2"/>
      <link from="Name_main_block/C" to="P/in3"/>
      <link from="P/out" to="Name_main_block/O"/>
    </links>
  </blockScope>
</IWIR>
```

5.5.3 Dot product

Just by increasing the nesting levels of the input data, we can generate any number of different IWIR workflows from the same GWENDIA workflow. For instance, if we make all inputs lists (nesting level = 1), we obtain Listing 16. Note that the nesting levels of all tree inputs must match, since we're using a dot product.

Listing 16: Dot product (IWIR)

```
<IWIR version="1.1" wfname="Name" xmlns="http://shiwa-workflow.eu/IWIR">
  <blockScope name="Name_main_block">
    <inputPorts>
      <inputPort name="A" type="collection/string"/>
      <inputPort name="B" type="collection/string"/>
      <inputPort name="C" type="collection/string"/>
    </inputPorts>
    <body>
      <parallelForEach name="P:dot:0">
        <inputPorts>
          <loopElements>
            <loopElement name="in1" type="collection/string"/>
            <loopElement name="in2" type="collection/string"/>
            <loopElement name="in3" type="collection/string"/>
          </loopElements>
        </inputPorts>
        <body>
          <task name="P">
            <inputPorts>
              <inputPort name="in1" type="string"/>
              <inputPort name="in2" type="string"/>
              <inputPort name="in3" type="string"/>
```

```
            </inputPorts>
            <outputPorts>
              <outputPort name="out" type="string"/>
            </outputPorts>
          </task>
        </body>
        <outputPorts>
          <outputPort name="out" type="collection/string"/>
        </outputPorts>
        <links>
          <link from="P:dot:0/in1" to="P/in1"/>
          <link from="P:dot:0/in2" to="P/in2"/>
          <link from="P:dot:0/in3" to="P/in3"/>
          <link from="P/out" to="P:dot:0/out"/>
        </links>
      </parallelForEach>
    </body>
    <outputPorts>
      <outputPort name="O" type="collection/string"/>
    </outputPorts>
    <links>
      <link from="Name_main_block/A" to="P:dot:0/in1"/>
      <link from="Name_main_block/B" to="P:dot:0/in2"/>
      <link from="Name_main_block/C" to="P:dot:0/in3"/>
      <link from="P:dot:0/out" to="Name_main_block/O"/>
    </links>
  </blockScope>
</IWIR>
```

Note that `inputPort` `A`, `B` and `C` as well as `outputPort` `O` are now of type `collection/string` corresponding to a 1D array (nesting level = 1). There is a new block `parallelForEach` created specifically to perform the dot-product iteration, by iterating simultaneously over the three input collections and handing three scalars to the task `P` each time. The `forEach` loop is explicitly `parallel` because all iterations are implicitly parallel in GWENDIA. Each additional nesting level would require an additional `ParallelForEach` loop. For instance, if the inputs were of type `collection/collection/string` (nesting level = 2), the first `ParallelForEach` loop would unwrap them into `collection/string` and the second one into `string` that can be fed to the task `P`.

5.5.4 Cross product

The cross product produces more complex results than the dot product, because each input must be iterated separately, for all combinations to be computed. Listings 16 and 17 are generated from the same workflow and the same input data (nesting level = 1), but the former applies a dot product iteration strategy and the latter a cross product.

In this case, there is exactly one `ParallelForEach` loop for each input. In general, since every input data must be iterated over until its type matches that of the input port and only one input is iterated at a time, the number N of loops generated for a cross product is:

$$N = \sum_{i \in inputs} level(data_i) - level(port_i)$$

Listing 17: Cross product (IWIR)

```
    ...<body>
      <parallelForEach name="P:cross:0">
        <inputPorts>
          <inputPort name="in2" type="collection/string"/>
          <inputPort name="in3" type="collection/string"/>
          <loopElements>
            <loopElement name="in1" type="collection/string"/>
          </loopElements>
```

```
        </inputPorts>
        <body>
          <parallelForEach name="P:cross:1">
            <inputPorts>
              <inputPort name="in1" type="string"/>
              <inputPort name="in3" type="collection/string"/>
              <loopElements>
                <loopElement name="in2" type="collection/string"/>
              </loopElements>
            </inputPorts>
            <body>
              <parallelForEach name="P:cross:2">
                <inputPorts>
                  <inputPort name="in1" type="string"/>
                  <inputPort name="in2" type="string"/>
                  <loopElements>
                    <loopElement name="in3" type="collection/string"/>
                  </loopElements>
                </inputPorts>
                <body>
                  <task name="P">
                    <inputPorts>
                      <inputPort name="in1" type="string"/>
                      <inputPort name="in2" type="string"/>
                      <inputPort name="in3" type="string"/>
                    </inputPorts>
                    <outputPorts>
                      <outputPort name="out" type="string"/>
                    </outputPorts>
                  </task>
                </body>
                <outputPorts>
                  <outputPort name="out" type="collection/string"/>
                </outputPorts>
                <links>
                  <link from="P:cross:2/in1" to="P/in1"/>
                  <link from="P:cross:2/in2" to="P/in2"/>
                  <link from="P:cross:2/in3" to="P/in3"/>
                  <link from="P/out" to="P:cross:2/out"/>
                </links>
              </parallelForEach>
            </body>
            <outputPorts>
              <outputPort name="out" type="collection/collection/string"/>
            </outputPorts>
            <links>
              <link from="P:cross:1/in1" to="P:cross:2/in1"/>
              <link from="P:cross:1/in2" to="P:cross:2/in2"/>
              <link from="P:cross:1/in3" to="P:cross:2/in3"/>
              <link from="P:cross:2/out" to="P:cross:1/out"/>
            </links>
          </parallelForEach>
        </body>
        <outputPorts>
          <outputPort name="out" type="collection/collection/collection/string"/>
        </outputPorts>
        <links>
          <link from="P:cross:0/in1" to="P:cross:1/in1"/>
          <link from="P:cross:0/in2" to="P:cross:1/in2"/>
          <link from="P:cross:0/in3" to="P:cross:1/in3"/>
          <link from="P:cross:1/out" to="P:cross:0/out"/>
        </links>
      </parallelForEach>
    </body>...
```

5.5.5 Mixed strategy

The application of the mixed strategy $(a \otimes b) \odot c$ in GWENDIA. That precise case produces the IWIR workflow of Listing 18.

Listing 18: Mixed strategy (IWIR)

```
...<body>
  <parallelForEach name="P:cross:0:0">
    <inputPorts>
      <inputPort name="in2" type="collection/string"/>
      <loopElements>
        <loopElement name="in1" type="collection/string"/>
      </loopElements>
    </inputPorts>
    <body>
      <parallelForEach name="P:cross:0:1">
        <inputPorts>
          <inputPort name="in1" type="string"/>
          <loopElements>
            <loopElement name="in2" type="collection/string"/>
          </loopElements>
        </inputPorts>
        <body>
          <task name="Identity" tasktype="Iteration-Identity">
            <inputPorts>
              <inputPort name="in1" type="string"/>
              <inputPort name="in2" type="string"/>
            </inputPorts>
            <outputPorts>
              <outputPort name="out_in1" type="string"/>
              <outputPort name="out_in2" type="string"/>
            </outputPorts>
          </task>
        </body>
        <outputPorts>
          <outputPort name="out_in1" type="collection/string"/>
          <outputPort name="out_in2" type="collection/string"/>
        </outputPorts>
        <links>...</links>
      </parallelForEach>
    </body>
    <outputPorts>
      <outputPort name="out_in1" type="collection/collection/string"/>
      <outputPort name="out_in2" type="collection/collection/string"/>
    </outputPorts>
    <links>...</links>
  </parallelForEach>
  <parallelForEach name="P:dot:1">
    <inputPorts>
      <loopElements>
        <loopElement name="in1" type="collection/collection/string"/>
        <loopElement name="in2" type="collection/collection/string"/>
        <loopElement name="in3" type="collection/collection/string"/>
      </loopElements>
    </inputPorts>
    <body>
      <parallelForEach name="P:dot:0">
        <inputPorts>
          <loopElements>
            <loopElement name="in1" type="collection/string"/>
            <loopElement name="in2" type="collection/string"/>
            <loopElement name="in3" type="collection/string"/>
          </loopElements>
        </inputPorts>
        <body>
          <task name="P">
            <inputPorts>
              <inputPort name="in1" type="string"/>
              <inputPort name="in2" type="string"/>
              <inputPort name="in3" type="string"/>
            </inputPorts>
            <outputPorts>
              <outputPort name="out" type="string"/>
            </outputPorts>
          </task>
        </body>
```

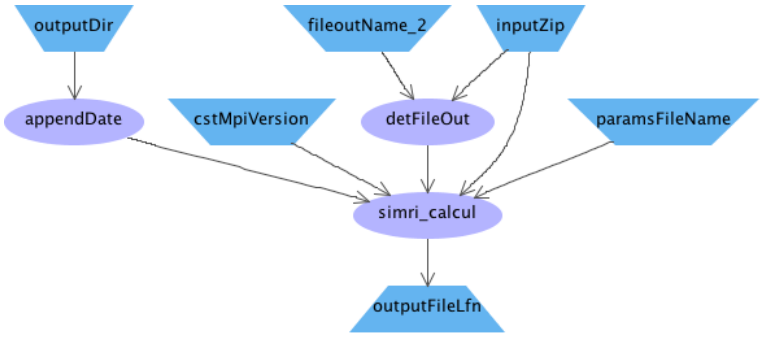

Figure 20: SIMRI workflow graphical representation.

```
            <outputPorts>
              <outputPort name="out" type="collection/string"/>
            </outputPorts>
            <links>...</links>
          </parallelForEach>
        </body>
        <outputPorts>
          <outputPort name="out" type="collection/collection/string"/>
        </outputPorts>
        <links>...</links>
      </parallelForEach>
    </body>...
```

The GWENDIA converter produces a literal translation of the iteration strategy rather than an optimal one: the cross product is performed first, which amounts to iterating on each input (`A` and `B`) and combining them as-is (hence the use of the identity task). The combinations produced by the cross product (literally all possible ones) are then combined in a dot product, much like in Section 5.5.3.

5.5.6 Example: SIMRI Magentic Resonance Images Simulation

SIMRI is a 3D MRI simulator workflow which original GWENDIA representation is graphically depicted in Figure 20. The GWENDIA workflow was transformed into the IWIR code shown in Listing 19. For the sake of simplicity, the workflow inputs, outputs and all data links internal to the IWIR nested structures have been omitted from this example.

Listing 19: SIMRI workflow represented in IWIR

```
<IWIR version="1.1" wfname="SIMRI" xmlns="http://shiwa-workflow.eu/IWIR">
  <blockScope name="simri_mpich_calcul_16cores_main_block">
    <inputPorts>...</inputPorts>
    <body>

      <task name="appendDate" tasktype="appendDate">
        <inputPorts>
          <inputPort name="dir" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="result" type="string"/>
        </outputPorts>
      </task>

      <parallelForEach name="detFileOut:cross:0">
        <inputPorts>
          <inputPort name="fileoutName" type="string"/>
          <loopElements>
            <loopElement name="fileObjLfn" type="collection/string"/>
          </loopElements>
        </inputPorts>
```

```
22          <body>
23            <task name="detFileOut" tasktype="detFileOut">
24              <inputPorts>
25                <inputPort name="fileObjLfn" type="string"/>
26                <inputPort name="fileoutName" type="string"/>
27              </inputPorts>
28              <outputPorts>
29                <outputPort name="fileOut" type="string"/>
30              </outputPorts>
31            </task>
32          </body>
33          <outputPorts>
34            <outputPort name="fileOut" type="collection/string"/>
35          </outputPorts>
36          <links>...</links>
37        </parallelForEach>
38
39        <parallelForEach name="simri_calcul:dot:0:0">
40          <inputPorts>
41            <inputPort name="input3" type="string"/>
42            <inputPort name="input1" type="string"/>
43            <inputPort name="input0" type="string"/>
44            <loopElements>
45              <loopElement name="input4" type="collection/string"/>
46              <loopElement name="input2" type="collection/string"/>
47            </loopElements>
48          </inputPorts>
49          <body>
50            <task name="simri_calcul" tasktype="simri_calcul">
51              <inputPorts>
52                <inputPort name="input3" type="string"/>
53                <inputPort name="input2" type="string"/>
54                <inputPort name="input4" type="string"/>
55                <inputPort name="input1" type="string"/>
56                <inputPort name="input0" type="string"/>
57              </inputPorts>
58              <outputPorts>
59                <outputPort name="result0" type="string"/>
60              </outputPorts>
61            </task>
62          </body>
63          <outputPorts>
64            <outputPort name="result0" type="collection/string"/>
65          </outputPorts>
66          <links>...</links>
67        </parallelForEach>
68      </body>
69
70      <outputPorts>...</outputPorts>
71      <links>...</links>
72    </blockScope>
73  </IWIR>
```

The 3 SIMRI workflow activities `appendDate`, `defFileOut`, and `simri_calcul` appear as IWIR tasks in Listing 19 at lines 6, 23, and 50 respectively. All feature different iteration strategies. The task `appendDate` is just taking a single string value as input (a directory name) and produce a single string value as output (the directory name append with the execution date). It is therefore not iterated and appears as a single task in the IWIR `blockScope`.

The `defFileOut` activity receives a single string value on one of its input ports (the output base file name) and an array of strings on its other input port (the LFNs of the input object to process). For each input LFN it produces an output string, leading to an array of strings as a result (a collection of strings in IWIR terminology). This activity is therefore iterated over all LFN items in the input array through the wrapping `parallelForEach` IWIR construct at line 15.

Finally, the `simri_calcul` activity is exposing a complex iteration strategy represented in Figure 21. This strategy combines a dot product (pair-wise combination of two input parameters) and cross products (combinatorial combination of all input parameters) [5]. This case also results in

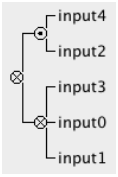

Figure 21: GWENDIA iteration strategy of `simri_calcul` activity. The ⊙ symbol represents a dot product and the ⊗ symbol represents a cross product.

a wrapping textttparallelForEach IWIR construct for the activity (line 39). The two input ports in the `loopElements` section (line 44) are the two inputs of the dot product shown in Figure 21. The other inputs appear as simple parameters (non-loop elements in the `inputPorts` section at line 40). The activity is iterated over all pairs of `input2`/`input4` array values and consequently produces an array of results.

It should be noted that in the original the GWENDIA workflow, the `appendDate` task was implemented as a *beanshell* java script. As it is a non-portable activity type, it was needed to create a new activity for the target Askalon workflow manager implementing the same function (simple appending of date in this case).

6 JSDL Templates

Conversion of workflow structure from source to target languages using middle IWIR Language solves the issue of representation the same graph strucure in both workflow systems. But as IWIR describes structure of the workflows only, the converters do not transfer information of task's source language implementation e.q. settings of input or output parameterization, path of the input files and executables, requirements of the resource where the task should be submitted to. As DCI Bridge accepts standardized JSDL job description, an obvious idea to resolve this problem is to describe each task's implementation in JSDL format.

6.1 JSDL Template Creation Tool

As this job description stores input and output paths that have references to files accessible from the source workflow manager only, they must be updated by actual values before sending the task to the DCI Bridge for submission. Figure 22 shows the process of template creation. To recognize tags that are modifiable by the users of the target workflow system, a skeleton JSDL description will be created by replacing values of the modifiable attributes to placeholders. Placeholders are constructed from some pre-defined strings according to the properties of the task and the attribute to be replaced. Table 4 shows the possible placeholders depending on the properties and the attribute of the task. Some of them are used many times on the same level of the xml structure, e.g. a new ns2:Argument tag is created for each command line argument separated by whitespace. To be able to recognize the order of the these kind of arguments, placeholders that are created for them must contain a unique identifier. In case of PLACEHOLDER_COMMANDLINE this identifier is index. In other cases the values of URI placeholders contain the name of the file as unique identifiers.

Listing 21 shows a general JSDL description that contains only one task with 2 input and 2 output ports (see Listing 20). As DCI Bridge returns standard output, standard error, the whole jsdl and log files for further investigation in different and predefined channels, they must be defined in JSDL description as well. Concrete values that must be replaced by placeholders are highlighted by red. Listing 22 shows the resulting JSDL Template, where all possible values are replaced by placeholders (highlighted by green).

The JSDL Template Creation Tool is planned to be implemented as a Liferay-based JSR-268 portlet.

Table 4: JSDL attributes and Placeholders association

JSDL Tag	purport	Placeholder
ns2:Argument	Command line argument	<PLACEHOLDER_COMMANDLINE_$index>
ns2:UserName	Name of the user,who submitted the job	<PLACEHOLDER_USERNAME >
ns2:GroupName	Group, commonly identifies the portal where the submission is done from	<PLACEHOLDER_GROUP>
<Source>URI </Source>	path of the real file associated as source datastage	<PLACEHOLDER_SOURCEFILESERVER_$filename>
<Target>URI </Target>	path of the real file associated as target datastage	<PLACEHOLDER_TARGETFILESERVER_$filename>
ns3:DCIName	Name of the DCI	<PLACEHOLDER_DCINAME>
ns3:MyProxy	Hostname of MyProxy server, if exists	<PLACEHOLDER_MYPROXY>
ns3:ManagedResource	Name of the Resource where the task will be submitted to	<PLACEHOLDER_RESOURCE>
ns4:wfiservice	Web service reference that handles statuses that will be sent by the task	<PLACEHOLDER_STATUSSERVICE>
ns4:proxyservice	Web service reference that provides required credential	<PLACEHOLDER_CREDENTIALPROVIDER>

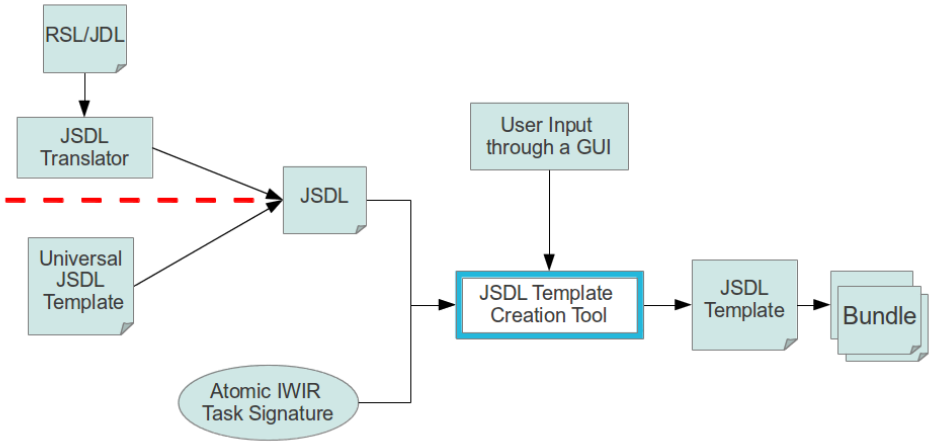

Figure 22: Process of JSDL template creation

6.1.1 Example

Listing 20: Sample template creation

```
<task name="Identity" tasktype="Iteration-Identity">
                <inputPorts>
                  <inputPort name="in1" type="string"/>
                  <inputPort name="in2" type="string"/>
                </inputPorts>
                <outputPorts>
                  <outputPort name="out_in1" type="string"/>
                  <outputPort name="out_in2" type="string"/>
                </outputPorts>
              </task>
```

Listing 21: Concrete JSDL in Source workflow system

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<JobDefinition xmlns="http://schemas.ggf.org/jsdl/2005/11/jsdl"
xmlns:ns2="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix"
xmlns:ns3="uri:MBSchedulingDescriptionLanguage" xmlns:ns4="extension.dci"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://schemas.ggf.org/jsdl/2005/11/jsdl">
    <JobDescription>
        <JobIdentification>
            <JobName>sample.sh</JobName>
            <JobAnnotation>cae3415f-0bc2-469b-b092-7d58b818875a</JobAnnotation>
        </JobIdentification>
        <Application>
            <ApplicationName>guse://535676014437440zentest/0/Identity/0</ApplicationName>
            <ns2:POSIXApplication_Type>
                <ns2:Executable>execute.bin</ns2:Executable>
                <ns2:Argument>cmd1</ns2:Argument>
                <ns2:Argument>cmd2</ns2:Argument>
                <ns2:Output>stdout.log</ns2:Output>
                <ns2:Error>stderr.log</ns2:Error>
                <ns2:UserName>10239</ns2:UserName>
                <ns2:GroupName>http://192.168.143.166:8080/wspgrade</ns2:GroupName>
            </ns2:POSIXApplication_Type>
        </Application>
        <Resources>
            <CandidateHosts/>
            <OperatingSystem>
                <OperatingSystemType>
                    <OperatingSystemName>LINUX</OperatingSystemName>
                </OperatingSystemType>
            </OperatingSystem>
        </Resources>
        <DataStaging>
            <FileName>execute.bin</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Source>
                <URI>http://192.168.143.168:8080/storage/getFile?
                path=http:__192.168.143.166:8080_wspgrade/10239/
                IWIR_SAMPLE_01/Identity/execute.bin</URI>
            </Source>
        </DataStaging>
        <DataStaging>
            <FileName>in2</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Source>
                <URI>http://192.168.143.168:8080/storage/getFile?
                path=http:__192.168.143.166:8080_wspgrade/
                10239/IWIR_SAMPLE_01/Identity/inputs/1/0</URI>
```

```
            </Source>
        </DataStaging>
        <DataStaging>
            <FileName>in1</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Source>
                <URI>http://192.168.143.168:8080/storage/getFile?
                path=http:__192.168.143.166:8080_wspgrade/
                10239/IWIR_SAMPLE_01/Identity/inputs/0/0</URI>
            </Source>
        </DataStaging>
        <DataStaging name="out.in2">
            <FileName>out.in2</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI>http://192.168.143.168:8080/storage/FileUploadServlet?
                path=http:__192.168.143.166:8080_wspgrade/
                10239/IWIR_SAMPLE_01/Identity/outputs/535676014437440zentest/0/&amp;link=out.in2</URI>
            </Target>
        </DataStaging>
        <DataStaging name="out.in1">
            <FileName>out.in1</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI>http://192.168.143.168:8080/storage/FileUploadServlet?
                path=http:__192.168.143.166:8080_wspgrade/
                10239/IWIR_SAMPLE_01/Identity/outputs/535676014437440zentest/0/&amp;link=out.in1</URI>
            </Target>
        </DataStaging>
        <DataStaging>
            <FileName>stderr.log</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI>http://192.168.143.168:8080/storage/FileUploadServlet?
                path=http:__192.168.143.166:8080_wspgrade/
                10239/IWIR_SAMPLE_01/Identity/outputs/535676014437440zentest/0</URI>
            </Target>
        </DataStaging>
        <DataStaging>
            <FileName>stdout.log</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI>http://192.168.143.168:8080/storage/FileUploadServlet?
                path=http:__192.168.143.166:8080_wspgrade/
                10239/IWIR_SAMPLE_01/Identity/outputs/535676014437440zentest/0</URI>
            </Target>
        </DataStaging>
        <DataStaging>
            <FileName>gridnfo.log</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI>http://192.168.143.168:8080/storage/FileUploadServlet?
                path=http:__192.168.143.166:8080_wspgrade/
                10239/IWIR_SAMPLE_01/Identity/outputs/535676014437440zentest/0</URI>
            </Target>
        </DataStaging>
        <DataStaging>
            <FileName>guse.jsdl</FileName>
```

```
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI>http://192.168.143.168:8080/storage/FileUploadServlet?
                path=http:__192.168.143.166:8080_wspgrade/
                10239/IWIR_SAMPLE_01/Identity/outputs/535676014437440zentest/0</URI>
            </Target>
        </DataStaging>
        <DataStaging>
            <FileName>guse.logg</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI>http://192.168.143.168:8080/storage/FileUploadServlet?
                path=http:__192.168.143.166:8080_wspgrade/
                10239/IWIR_SAMPLE_01/Identity/outputs/535676014437440zentest/0</URI>
            </Target>
        </DataStaging>
    </JobDescription>
    <ns3:SDL_Type>
        <ns3:Constraints>
            <ns3:Middleware>
                <ns3:DCIName>local</ns3:DCIName>
                <ns3:MyProxy/>
                <ns3:ManagedResource>dci-bridge host(64bit)</ns3:ManagedResource>
            </ns3:Middleware>
            <ns3:Budget>0</ns3:Budget>
        </ns3:Constraints>
    </ns3:SDL_Type>
    <ns4:extension_type>
        <ns4:wfiservice>http://192.168.143.168:8080/wfi/JobStatusService?wsdl</ns4:wfiservice>
        <ns4:proxyservice>http://192.168.143.166:8080/wspgrade/CredentialProvider?wsdl</ns4:proxyservice>
    </ns4:extension_type>
</JobDefinition>
```

Listing 22: Concrete JSDL in Source workflow system

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<JobDefinition xmlns="http://schemas.ggf.org/jsdl/2005/11/jsdl"
xmlns:ns2="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix"
xmlns:ns3="uri:MBSchedulingDescriptionLanguage" xmlns:ns4="extension.dci"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://schemas.ggf.org/jsdl/2005/11/jsdl">
    <JobDescription>
        <JobIdentification>
            <JobName>sample.sh</JobName>
            <JobAnnotation>cae3415f-0bc2-469b-b092-7d58b818875a</JobAnnotation>
        </JobIdentification>
        <Application>
            <ApplicationName>guse://535676014437440zentest/0/Identity/0</ApplicationName>
            <ns2:POSIXApplication_Type>
                <ns2:Executable>execute.bin</ns2:Executable>
                <ns2:Argument><PLACEHOLDER_COMMANDLINE_INDEX></ns2:Argument>
                <ns2:Argument><PLACEHOLDER_COMMANDLINE_INDEX></ns2:Argument>
                <ns2:Output>stdout.log</ns2:Output>
                <ns2:Error>stderr.log</ns2:Error>
                <ns2:UserName><PLACEHOLDER_USERNAME></ns2:UserName>
                <ns2:GroupName><PLACEHOLDER_GROUP></ns2:GroupName>
            </ns2:POSIXApplication_Type>
        </Application>
        <Resources>
            <CandidateHosts/>
            <OperatingSystem>
                <OperatingSystemType>
                    <OperatingSystemName>LINUX</OperatingSystemName>
```

```
                </OperatingSystemType>
            </OperatingSystem>
        </Resources>
        <DataStaging>
            <FileName>execute.bin</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Source>
                <URI><PLACEHOLDER_FILESERVER_execute.bin></URI>
            </Source>
        </DataStaging>
        <DataStaging>
            <FileName>in2</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Source>
                <URI><PLACEHOLDER_FILESERVER_in2></URI>
            </Source>
        </DataStaging>
        <DataStaging>
            <FileName>in1</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Source>
                <URI><PLACEHOLDER_FILESERVER_in1></URI>
            </Source>
        </DataStaging>
        <DataStaging name="out.in2">
            <FileName>out.in2</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI><PLACEHOLDER_FILESERVER_out.in2></URI>
            </Target>
        </DataStaging>
        <DataStaging name="out.in1">
            <FileName>out.in1</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI><PLACEHOLDER_FILESERVER_out.in1></URI>
            </Target>
        </DataStaging>
        <DataStaging>
            <FileName>stderr.log</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI><PLACEHOLDER_FILESERVER_stderr.log></URI>
            </Target>
        </DataStaging>
        <DataStaging>
            <FileName>stdout.log</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI><PLACEHOLDER_FILESERVER_stdout.log></URI>
            </Target>
        </DataStaging>
        <DataStaging>
            <FileName>gridnfo.log</FileName>
            <CreationFlag>overwrite</CreationFlag>
            <DeleteOnTermination>true</DeleteOnTermination>
            <Target>
                <URI><PLACEHOLDER_FILESERVER_gridnfo.log></URI>
            </Target>
        </DataStaging>
        <DataStaging>
            <FileName>guse.jsdl</FileName>
```

```
98              <CreationFlag>overwrite</CreationFlag>
99              <DeleteOnTermination>true</DeleteOnTermination>
100             <Target>
101                 <URI><PLACEHOLDER_FILESERVER_guse.jsdl></URI>
102             </Target>
103         </DataStaging>
104         <DataStaging>
105             <FileName>guse.logg</FileName>
106             <CreationFlag>overwrite</CreationFlag>
107             <DeleteOnTermination>true</DeleteOnTermination>
108             <Target>
109                 <URI><PLACEHOLDER_FILESERVER_guse.logg></URI>
110             </Target>
111         </DataStaging>
112     </JobDescription>
113     <ns3:SDL_Type>
114         <ns3:Constraints>
115             <ns3:Middleware>
116                 <ns3:DCIName><PLACEHOLDER_DCINAME></ns3:DCIName>
117                 <ns3:MyProxy><PLACEHOLDER_MYPROXY></ns3:MyProxy>
118                 <ns3:ManagedResource><PLACEHOLDER_RESOURCE></ns3:ManagedResource>
119             </ns3:Middleware>
120             <ns3:Budget>0</ns3:Budget>
121         </ns3:Constraints>
122     </ns3:SDL_Type>
123     <ns4:extension_type>
124         <ns4:wfiservice><PLACEHOLDER_STATUSSERVICE></ns4:wfiservice>
125         <ns4:proxyservice><PLACEHOLDER_CREDENTIALPROVIDER></ns4:proxyservice>
126     </ns4:extension_type>
127 </JobDefinition>
```

6.2 JSDL Template Instantiation

Once the Bundle contains skeletons of JSDL descriptions, they must be instantiated by current values of real paths of input files; command line arguments set by users; output filepaths, where the generated outputs must be transferred to; place of the service that can handle task statuses and a web-service reference that provides credential for the submission. It must be done automatically and on-the-fly during the submission progress. Therefore a simple java library must be defined that can be used to replace placeholders to current values. This java library provides a simple API with the functions and signatures shown in Table 5. Every function has a getter method as well to be able to get the current value of the attribute.

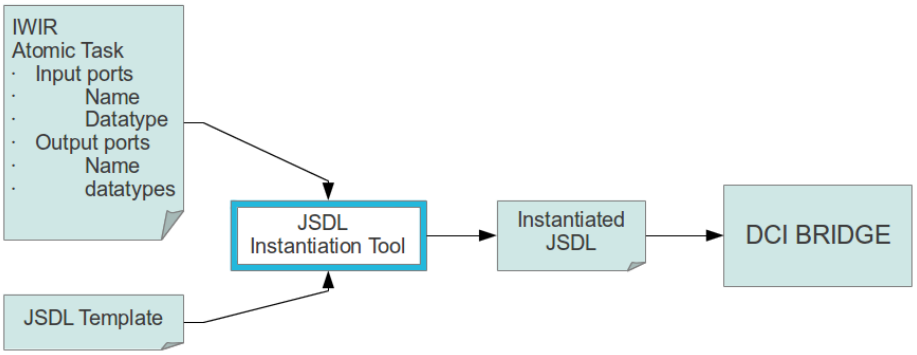

Figure 23: Process of JSDL instantiation

Table 5: Instantiation Tool API

Function name	Arguments	Description
setInput	String name, String URI	Replaces <PLACEHOLDER_FILESERVER_$name> to URI
setOutput	String name, String URI	Replaces <PLACEHOLDER_FILESERVER_$name> to URI
setExecutable	String URI	Replaces <PLACEHOLDER_FILESERVER_execute.bin> to URI
setCredentialProvider	String URI	Replaces <PLACEHOLDER_CREDENTIALPROVIDER> to URI
setStatusHandler	String URI	Replaces <PLACEHOLDER_STATUSSERVICE> to URI
setUser	String username	Replaces <PLACEHOLDER_USERNAME> to username
setDCI	String dci	Replaces <PLACEHOLDER_DCI> to dci
setMyProxy	String myproxy	Replaces <PLACEHOLDER_MYPROXY> to myproxy
setResource	String resource	Replaces <PLACEHOLDER_RESOUCE> to resource
setGroup	String groupname	Replaces <PLACEHOLDER_GROUP> to URI

6.3 JSDL Translator

The JSDL Translator is developed as a component of the Meta-Broker to be responsible for translating the resource specification language defined by the user to the language of the appropriate resource broker that the Meta-Broker selects to use for a given job. But as it is developed as a web-service, the translator component can be used as a standalone job description converter service. From all possible job specification language a subset of basic job attributes can be chosen, which can be denoted relatively in the same way in each document. The translation of these parts is almost trivial. Supported attributes are listed in Table 7 and Table 8. The rest of these attributes describe special job handling, various scheduling features and remote storage access. Generally these cases can hardly be matched among the different systems, because only few of them support the same solution. We gathered these special attributes of the different job description languages. If an attribute of a job description language cannot be expressed in JSDL, we specify it as an extension. These attributes are collected and specified in a proposed JSDL extension called jsdl-metabroker and are defined in a schema called mbsdl.xsd. Regarding other languages we express the missing attribute in comments in order to keep the translations consistent. Some examples on the mapping among these attributes are shown in Table 6.

Class of translator web-service object called TRService contains one constructor method.

Listing 23: Constructor method of JSDL Translator

```
1 public TRService()
```

Table 6: A subset of special job description language attributes.

RSL (GTbroker)	*xRSL (NorduGrid)*	*JDL (EGEE)*	*JSDL*
(*sched=random*)	(*sched=random*)	FuzzyRank=true;	extension
(*sched=CPU/Memory/Disk*)	(*sched=CPU/Memory/Disk*)	rank=other.GlueHostProcessorClockSpeed/GlueHostMainMemoryRAMSize/GlueSAStateAvailableSpace;	extension
(*minMemory=int*), (*mindisk=int*)	(memory=int), (disk=int)	Requirements: (GlueHostMainMemoryRAMSize int); anyMatch(other.storage.CloseSEs, target.GlueSAStateAvailableSpace int);	resources jsdl:IndividualDiskSpace jsdl:IndividualPhysicalMemory ... /resources
(*skiptime=int*)	(*skiptime=int*)	/*skiptime=int*/	extension
rescheduling by default	(rerun=max.5)	RetryCount=max.10;	extension

JSDL Translator provides a method called translateXDL to transform Base64 encoded job descriptions written in JDL or RSL to the extended Base64 encoded JSDL format.

Listing 24: Method for translating in JSDL Translator

```
1 public java.lang.String translateXDL(java.lang.String xDLtype,java.lang.String xDL)
2 throws java.io.IOException
```

The method requires 2 parameters, the first one must be the type of the native description language document (guse/rsl/jdl), the second is the Base64 encoded native description language document, and returns a Base64 encoded JDSL or throws an IOException in case of read or write errors.

The following schemas are used: jsdl.xsd, jsdl-posix.xsd, mbsdl.xsd and jsdl-arc.xsd.

6.4 DCI Bridge

The DCI Bridge component was developed to support only SHIWA's FGI solution, however later on it turned out that it is useful for any OGSA Basic Execution Service 1.0 (BES) enabled workflow management system to solve its DCI interoperability issues. The DCI Bridge is a web service based application, which provides standard access to various distributed computing infrastructure (DCIs) such as: grids, desktop grids, clusters, clouds and service based computational resources (it connects through its DCI plug-ins to the external DCI resources). The main advantage of using the DCI Bridge as web application component of workflow management systems is, that it enables workflow management systems to access various DCIs using the same well defined communication interface (shown in Fig. 24.). When a user submits a workflow, its job components can be submitted transparently into the various DCI systems using the OGSA Basic Execution Service 1.0 (BES) interface. As a result, the access protocol and all the technical details of the various DCI systems are totally hidden behind the BES interface. The standardized job description language of BES is JSDL.

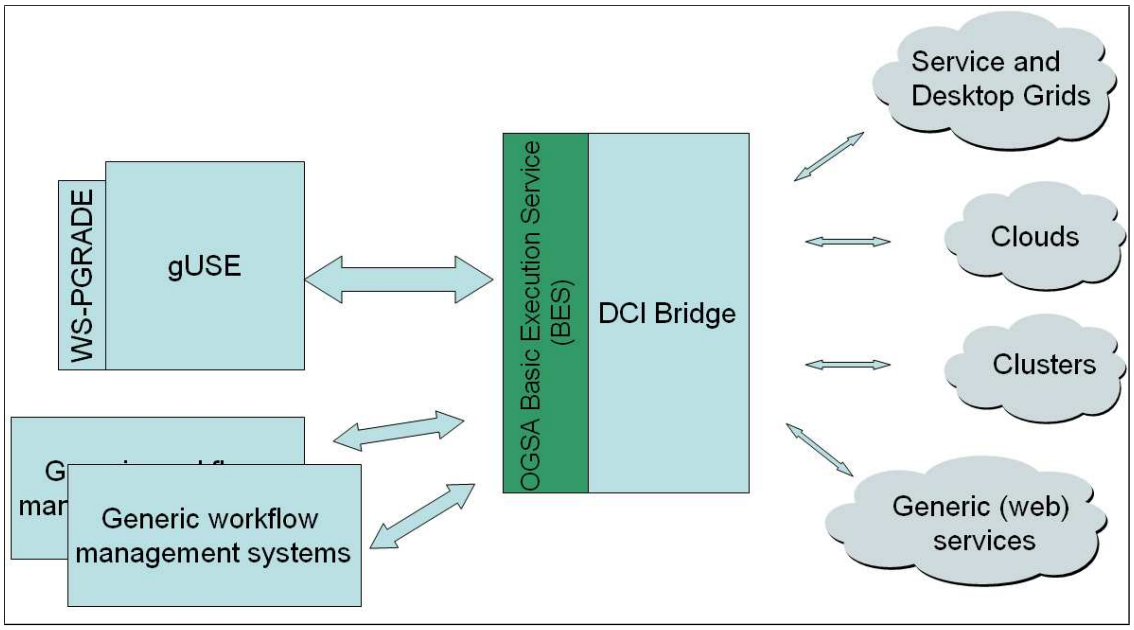

Figure 24: Schematic overview of the DCI Bridge and its external communication channels

The Runtime System accepts standardized JSDL job description documents which are based on well-defined XML schema, and contains information about the job inputs, binaries, runtime settings and output locations. The core JSDL itself is not powerful enough to fit all needs, but fortunately it has a number of extensions. For example, DCI Bridge makes use of the JSDL-POSIX extension. Beside the JSDL-POSIX extension, DCI Bridge makes use of two legacy extensions: one for defining execution resources, and one for proxy service and callback service access. The execution resource extension is needed both for the core DCI Bridge in order to define specific execution resource needs and for the Metabroker service. The proxy service extension is needed for jobs targeted to DCIs, which rely on X.509 proxy certificates for job submission. The callback service extension is needed if status change callback functionality is needed: the DCI Bridge will initiate a call to the service specified in the extension upon every job status change. User credential handling of DCI Bridge is based on content-based approach instead of a channel-based ones. This means that user credentials (proxies or SAML assertions) are not handled by the communication channel, but are rather specified in the JSDL extension mentioned earlier. This approach allows DCI Bridge to implement varied DCI-dependent credential handling options within the different DCI plug-ins instead of relying on the capabilities of the servlet container running the DCI Bridge service. Of course it is still possible to run DCI Bridge as a secured service (for example as a service accessible through authenticated HTTP, HTTPS or even HTTPG), but credentials used for establishing connection to the service are not passed to the destination plug-in, it solely makes use of the credentials described in the JSDL extension.

7 FGI-based Workflows in the SHIWA Repository

The JRA1 work package developed and deployed the SHIWA Repository that stores descriptions of workflows to enable browsing, searching, uploading, publishing and downloading workflows and related metadata. The Shiwa Repository enables workflow developers to define multiple implementations for a single workflow. Each implementation is specific to a given workflow engine, that is able to interpret and execute it. In order to support sharing of FGI workflows, an abstract “IWIR” workflow engine has been added to the list of engines supported by the repository.

7.1 Publishing and Retrieving FGI Bundles

If a given workflow has an FGI bundle, the workflow developer can upload it to the repository by creating a new IWIR implementation of the workflow and uploading the bundle file as the definition of the new workflow implementation. Once an FGI bundle is published in the repository, it can be retrieved as a single file. FGI bundles can be published and retrieved by:

- using the web based GUI provided by the SHIWA Repository;
- using the GUI of the SHIWA Desktop that can interface with the SHIWA Repository; or
- using the servlet interface of the SHIWA Repository.

The former two interfaces allow manual interaction with the repository for users and workflow developers, while the servlet interface allows automated bundle exchange for third party software components.

8 Conclusions

We presented in this deliverable the fine-grained interoperability architecture and its implementation support in the four workflow systems that are part of the SHIWA project: ASKALON, Moteur, P-Grade, and Triana. The realisation of the fine-grained interoperability architecture is based on two components: FGI bundle technology and IWIRtool as a Java package that provides general support for parsing, manipulating, and validating IWIR data structures. We reported details on the IWIR workflow converters implemented by the four workflow systems for their native languages: AGWL, GWENDIA, gUSE, and Triana task graph and showed conversion examples. Finally, we covered the description and execution of concrete computational tasks using JSDL Templates and presented the integration of fine-grained interoperability-based workflows in the SHIWA repository.

References

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

[2] Thomas Fahringer, Jun Qin, and Stefan 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.

[3] Péter Kacsuk and Gergely 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.

[4] 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.

[5] 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 *Workshop on Workflows in Support of Large-Scale Science(WORKS'09)*, , pages 1–10, Portland, USA, November 2009. ACM.

[6] 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.

Table 7: Supported JSDL attributes

JSDL	JSDL-POSIX
jsdl:ApplicationName	jsdl-posix:Executable
jsdl:ApplicationVersion	jsdl-posix:Argument
jsdl:JobAnnotation	jsdl-posix:Input
jsdl:JobProject	jsdl-posix:Output
jsdl:JobName	jsdl-posix:WallTimeLimit
jsdl:ExclusiveExecution	jsdl-posix:FileSizeLimit
jsdl:OperatingSystem	jsdl-posix:CoreDumpLimit
jsdl:CPUArchitectureName	jsdl-posix:DataSegmentLimit
jsdl:HostName	jsdl-posix:LockedMemoryLimit
jsdl:FileName	jsdl-posix:MemoryLimit
jsdl:CreationFlag	jsdl-posix:OpenDescriptorsLimit
jsdl:DeleteOnTermination	jsdl-posix:PipeSizeLimit
jsdl:URI	jsdl-posix:StackSizeLimit
jsdl:CPUArchitecture	jsdl-posix:CPUTimeLimit
jsdl:IndividualNetworkBandwidth	jsdl-posix:ProcessCountLimit
jsdl:IndividualPhysicalMemory	jsdl-posix:VirtualMemoryLimit
jsdl:IndividualVirtualMemory	jsdl-posix:ThreadCountLimit
jsdl:IndividualCPUSpeed	jsdl-posix:UserName
(jsdl:IndividualCPUTime)	jsdl-posix:GroupName
(jsdl:IndividualCPUCount)	
(jsdl:IndividualDiskSpace)	
jsdl:TotalCPUTime	
jsdl:TotalCPUCount	
jsdl:TotalPhysicalMemory	
jsdl:TotalVirtualMemory	
jsdl:TotalDiskSpace	
jsdl:TotalResourceCount	
jsdl:Range	
jsdl:LowerBoundedRange	
jsdl:UpperBoundedRange	
jsdl:Exact	
jsdl:LowerBound	
jsdl:UpperBound	
jsdl:DataStaging	
jsdl:Source	
jsdl:Target	

Table 8: Attribute extensions

mbsdl	jsdl-arc
mbsdl:Middleware	jsdl-arc:IsExecutable
mbsdl:VirtualOrganisation	jsdl-arc:GridTimeLimit
mbsdl:InformationSystem	jsdl-arc:Directory
mbsdl:MyProxy	jsdl-arc:Reruns
mbsdl:ProxyName	jsdl-arc:RemoteLogging
mbsdl:ServerName	jsdl-arc:URL
mbsdl:PortNumber	
mbsdl:Policy	
mbsdl:OtherConstraint	
mbsdl:OtherPolicy	
mbsdl:LRMSPolicy	
mbsdl:Agreement	
mbsdl:AdvanceReservation	
mbsdl:MDS	
mbsdl:BDII	
mbsdl:ARC	
mbsdl:WebMDS	
mbsdl:JobType	
mbsdl:Budget	
mbsdl:RemoteFileAccess	
mbsdl:FaultToleranceMechanisms	
mbsdl:GridAccessControl	
mbsdl:Priority	
mbsdl:EmailNotification	
mbsdl:PolicyName	
mbsdl:StartTime	
mbsdl:Duration	
mbsdl:TimeOut	
mbsdl:DCIName	
mbsdl:ProxyName	
mbsdl:Name	
mbsdl:ServerName	
mbsdl:PortNumber	
mbsdl:Target	
mbsdl:ConfidenceLevel	
mbsdl:ResourceName	
mbsdl:Date	
mbsdl:Value	