Bundle and Pool Architecture for Multi-Language, Robust, Scalable Workflow Executions

David Rogers, Ian Harvey, Tram Truong Huu, Kieran Evans, Tristan Glatard, Ibrahim Kallel, Ian Taylor, Johan Montagnat, Andrew Jones, Andrew Harrison

Noname manuscript No.
(will be inserted by the editor)

Bundle and Pool Architecture for Multi-Language, Robust, Scalable Workflow Executions

David Rogers · Ian Harvey · Tram Truong Huu · Kieran Evans · Tristan Glatard · Ibrahim Kallel · Ian Taylor · Johan Montagnat · Andrew Jones · Andrew Harrison ·

Received: date / Accepted: date

Abstract In this paper, we leverage the previous work on the SHIWA bundling format and expand on this specification in order to facilitate workflow execution within a multi-workflow environment. We introduce a scalable and robust execution pool environment that supports workflows consisting of sub-workflows built upon a multitude of different workflow engines and environments, and also provide a common workflow representation for seamless connectivity through serialization to workflow bundles. We also present a meta-workflow scenario based upon this system.

Workflow bundles employ the lightweight Open Archives Initiative Object Reuse and Exchange (ORE) Web-based standard, to provide a common format for representing and sharing workflows and the associated metadata required for their execution. This generalized bundling approach is already available within five workflow engines and has proven a useful environment for inter-workflow experimentation.

The execution pool facilitates federated access to multiple distributed computing infrastructures supported by the underlying workflow engines subscribed to the pool. Workflow bundles are exposed using the eXtensible Messaging and Presence Protocol (XMPP), which provides the necessary commu-

K. Evans, I. Harvey, A. Jones, D. Rogers and I. Taylor
School of Computer Science, Cardiff UK
E-mail: {k.evans, Ian.Harvey, Andrew.C.Jones, d.m.rogers, Ian.J.Taylor}@cs.cardiff.ac.uk

A. Harrison
Carefx Corporation
E-mail: aharrison@carefx.com

T. Glatard, I. Kallel
CREATIS - CNRS UMR 5220 - INSERM U1044 - University Lyon 1 - INSA Lyon
E-mail: glatard@creatis.insa-lyon.fr

J. Montagnat and T. Truong Huu
CNRS / UNS, I3S lab, MODALIS team
E-mail: johan@i3s.unice.fr, tram@polytech.unice.fr

nication backbone to enable multiple workflow engine agents to asynchronously publish and subscribe to bundles in meta-workflow pipelines.

We present experiments showing the scalability and robustness of the pool execution approach with results showing that overheads remain controlled for up to 150 workflow agents, and that agent failures have very limited impact. We then demonstrate the applicability of our architecture by describing how a Java-based music analysis workflow can be distributed within such a multi-workflow environment consisting of the Triana and MOTEUR workflow engines.

1 Introduction

Workflows provide a structured means of describing complex functional execution and data pipelines for a scientific experiment and hence expose the underlying scientific processes for enabling the reproducibility of results. They allow the specification of the scientific process as sub-elements of a task, each of which can be independently developed, validated, refined and composed for different configurations [1]. By allowing a user to formalize the data processing for execution and collection/visualization of results in an automated fashion, workflows provide the infrastructure for modeling the scientific process as a whole.

There are a wide array of popular workflow systems available for researchers to design, test and run large-scale scientific workflows [2,3,4,5,6,7,8,9,10]. These workflow systems are often tailored to a specific set of scientific domains such as astrophysics or bioinformatics, or they may be bound to specific regional infrastructures, and so a workflow researcher may only ever be exposed to a small range of workflow systems. In the case of large-scale or multi-disciplinary research it may be desirable to exploit multiple workflow environments. This will only be practical when the benefits of running tasks on more specialized workflow systems outweigh the overhead of developing specialized sub-workflows, and when data transfer between multiple workflow environments can be minimized. It may also be the case that workflows developed in previous research may be able to form part of newly developed workflow experiments. Reusability and modularity therefore become important concepts when developing large-scale workflows.

This work is motivated by the coarse-grained requirements of the SHIWA project which aims to leverage existing workflow solutions and enable cross-workflow and inter-workflow federative exploitation of Distributed Computing Infrastructure (DCI) resources by applying both a coarse- and fine-grained strategy [11]. In our previous work within the SHIWA project, we described the method of modeling a scientific workflow experiment by referencing its constituent components using the Object Reuse and Exchange (ORE) standard [12], developed by the Open Archives Initiative (OAI), to expose them using a single aggregated Web resource known as a SHIWA Bundle [13] – which facilitates Course Grained Interoperability (CGI) between workflow en-

gines by allowing workflows to be treated as black boxes. We proposed that a formalization of encapsulating a workflow that also allows the capturing of the scientific research techniques, tools documentation and methods, could help the reproducibility and validation of research methods in the eScience community. Such a model would not only increase efficiency in supporting larger scale research via the development of meta-workflows, but also encourage reuse and the sharing of tools, methods and processes thus lowering the learning barrier for scientists who would like to take advantage of a DCI environment. However, for such sharing to take place, users would benefit from using the familiar Web environment for sharing and using concepts that they are acquainted with, without having to understand the complexities of the multiple different workflow systems currently in use on today's DCIs.

The main contribution of this current paper is to present and demonstrate the use of a *CGI pool*, which enables the automatic execution of the SHIWA bundle technology so that pipelines of workflows connecting several different workflow engines can be achieved within one environment. We describe the updates we made to the SHIWA Bundle concept that were required in order to facilitate workflow execution within the multi-workflow environment of the pool and demonstrate this approach by running an experiment that connected a Triana workflow engine running a workflow on TrianaCloud with multiple MOTEUR sub-workflows running on the European Grid Infrastructure (EGI).

A multi-workflow environment is a collection of execution services that provides support for workflows potentially implemented in a variety of different languages that may need to run on multiple different engines or DCIs. Through the use of SHIWA Bundles, engine interfaces can be made uniform, providing a single input/output format for the execution system, but the environment still has to be able to select a proper engine to execute a workflow expressed in a given language. Adding a new language implies reconfiguring the environment, which may be time consuming. In addition, linking a list of endpoints to a list of languages is failure-prone due to the dependence of the service to end points located within different institutions, and with different capabilities. Without any further precaution, the environment may well overwhelm a particular engine due to excessive submission, or lose time in trying to contact engines that no longer exist.

The CGI pool is a coarse-grained execution pool that is capable of asynchronously communicating bundles using XMPP [14,15] between different workflow execution engines. We report on the implementation and integration in MOTEUR and Triana of CGI pool support. Through the pool workflow engines can conveniently discover a workflow, execute it, and then publish it back into the execution pool for execution by another workflow engine. In this fashion, meta-workflow pipelines can be built across different workflow engines. An advantage of this approach is that the workflow execution service is distributed to a number of agents that can be dynamically started and tuned to the workload, making the service scalable and robust to agent failures. In addition, this meta-workflow approach provides a single workflow bundling format for representing workflows and their components via a single SHIWA bun-

dle. This effectively reduces the meta-workflow solution from a many-to-many problem to a many-to-one and a one-to-many problem, where each workflow engine is only required to understand bundles in order to interoperate in this coarse-grained model and plug into the execution pool.

After showing the pool's reliability and scalability, we demonstrate the usefulness of the bundle and execution pool approach by distributing the execution of this application in multiple workflow environments: the Triana workflow engine running on TrianaCloud and MOTEUR running on the European Grid Infrastructure (EGI). We use Triana as a master workflow engine which is capable of publishing bundles to the CGI pool backbone. The Triana and MOTEUR pool agents get the bundle from the execution pool to execute on their native environments. The final objective is to reduce the total execution time of the complete dataset.

The paper is organized as follows. The next section describes related work on this topic. Section 3 introduces the first contribution to the workflow community, SHIWA Bundles; the overall design of the SHIWA bundle ORE schema and the associated Resource Description Framework (RDF)[1] vocabulary used to describe workflow artefacts and their associated data and metadata. It also discusses how the bundling mechanism fits into the overall vision for workflow interoperability and reuse. In Section 4 we introduce the execution pool which provides a flexible, robust and scalable means of distributing bundles for execution. Section 5 presents our integration of SHIWA bundles and pools into the MOTEUR and Triana workflow systems. Section 6 presents experiments demonstrating the scalability and robustness of the pool. Finally we have Section 7, providing a use-case experiment that describes and tests two means of running bundles through the European Grid Infrastructure and through a local cloud.

2 Background and Related Work

Recently, workflow interoperability has gained popularity within the distributed computing community, and so the SHIWA project was undertaken with the aim of achieving interoperability between workflow systems at various levels. SHIWA acknowledges that different levels of interoperability suit different systems, and that imposing one set of standards and structures is not the most appropriate approach of leveraging a system's capabilities. SHIWA defines two modes of interoperability: coarse-grained (CGI) and fine-grained (FGI). Coarse-grained interoperability makes use of third-party workflow engines as "black boxes" by embedding specific functionality supplied by a workflow into another [16]. The fine-grained approach allows the same workflow to be moved between different distributed computing infrastructures by translating workflow languages from one workflow engine to another through the use of the common Intermediate Workflow Representation Language (IWIR) language

[1] `http://www.w3.org/RDF/`

which is implemented by ASKALON [3], MOTEUR, P-Grade [5], Pegasus [6] and Triana, as part of the SHIWA project [17].

The Workflow Management Coalition (WfMC) [18] was the first to comprehensively address workflow interoperability within the business community, defining various standards including the Workflow Standard-Interoperability Abstract Specification. In this specification, different strategies can be used to achieve workflow interoperability: (1) Direct Interaction, through the use of a common API; (2) Message Passing, by exchanging information and sending packets of data messages through a communication network; (3) Bridging Strategy, by applying a bridging mechanism using a gateway technique to move data and tasks between systems via protocol converters; and (4) Shared Data Store, by transferring data and tasks between workflow Systems using a shared database.

A related approach, which was a precursor to our ORE work, was developed within the OMII-UK WHIP project [19]. The WHIP project focused on creating a desktop launcher application for different workflow engines by using an OS mapping of the WHIP file extension and MIME type for launching within a Portal, which is in contrast to the SHIWA Desktop plug-in model that provides a uniform interface to the Portal. However, the means by which data was shared between the desktop application and the Web server was through a WHIP bundle. WHIP bundles, like SHIWA bundles, were modeled in conformance with the ORE, but they differed in approach by binding to the Atom feed format for dissemination of such aggregations.

Another currently running project WF4Ever [20] is also focused at achieving interoperability. The authors in [21] also favour the ORE approach, arguing that publishing linked data does not meet the requirements of reuse because validation and reproducibility of scientific results requires multiple sources of information, such as provenance, quality, credit, attribution and methods. Although the authors call such ORE aggregations "Research Objects" for sharing and publishing workflows, the structure is compatible in essence with our bundle concept and on-going discussions are taking place in order to align this effort made through the myExperiment project and our work. myExperiment [22] is a Web 2.0-oriented interface for sharing scientific workflows, inspired by social networking sites. Users can upload arbitrary files or logically group resources into "packs". It has also been used to expose WHIP bundles. The myExperiment team is currently working on an ORE implementation for sharing of research objects.

Pegasus [6] supports large-scale workflows on Grid resources and is usually integrated into a portal environment using Web forms, e.g. in the Telescience project [23]. Pegasus has used WHIP bundles for bundling workflow descriptions, inputs, outputs and DCI characteristics to implement a pipeline-centric provenance model applied to use cases from the astronomy community [24]. This supports the use of an ORE-bundling approach.

P-GRADE [25] is designed to work using Webstart and has a custom integration within its portal, which can enable the creation, execution and monitoring of workflows. However, it provides no means to expose and

share workflows with users of other workflow engines. Service-based workflows, such as Triana, Kepler and Taverna, have sophisticated front end user interfaces for interaction with users for the design and/or execution of workflows. To date, Taverna and Triana have supported the use of ORE through WHIP bundles. For Kepler, the Hydrant project[2] provides a web-based portal for uploading and sharing workflows, but this is specific to Kepler.

Other related work in this context includes the VLE-WFBus project [26], where a number of different workflow systems are made interoperable through a run-time infrastructure. Each of the workflow systems connected by a workflow bus is wrapped and treated as a sub-workflow. The role of the workflow bus is to propagate information about the data objects to the correct sub-workflows, schedule the sub-workflows, and interface to the execution environment. VLE therefore adopts a message passing or bridging strategy in order to achieve workflow interoperability. Another related effort with respect to message passing is the SWIF system [27], which employs the use of WS-Notification to provide asynchronous communication channels between distributed workflow systems.

3 Workflow Bundles: Design and Architecture

Since the coarse-grained approach is concerned with atomic execution of workflows running in their own individual environments, the fundamental research issue is how to address the communication and sharing of data between different workflow engines. To this end, the concept of a SHIWA bundle emerged, which forms the basis of the work reported here. The SHIWA workflow bundle is defined as a compound object that contains all information pertaining to a scientist's experiment — the engine-specific workflow definition, at a minimum, along with potentially the input data, output data from previous runs, executable dependencies of the workflow, provenance data, documentation, research output and references to other web artifacts, such as related work.

To address language independence, aggregation and standardization criteria, it is modeled using the Object Reuse and Exchange (ORE) Internet standard ORE has an RDF vocabulary for describing aggregations of Web resources. ORE uses a *Resource Map* resource to model collections of related resources; this collection is known as an *Aggregation* within ORE. In terms of SHIWA bundles, these aggregations are the maps of resources that define the compound objects that represent a workflow experiment (see Section 3.2).

The choice of a bundle format using ORE for modeling compound objects is not arbitrary. A number of design considerations and requirements are met through the use of ORE:

– Integration with the Web Architecture. Using ORE makes all referenced resources available at URL endpoints, thus creating a transparent resource

[2] http://code.google.com/p/hydrant-kepler/

map. This means aggregations can be accessed via a wide variety of agents. Typically HTTP URLs are used for resources, meaning that resources can be accessed from any device that supports HTTP. This makes it very easy for users to retrieve, view, share and edit their aggregations using web-based tools that they are used to using. Large files or data files may be available at GSI FTP endpoints which has less ubiquitous support. However, these are not typically user-facing resources; rather, these would be execution files needed by eScience middleware.
- Aggregating resources in a single file is useful for publishing, archiving and handling situations where URLs cannot be assumed to be persistent. A self-contained bundle provides local locations for resources that do not have a public URL, that is, the URL is local to the bundle itself. Once a SHIWA bundle is published to a server environment, it is typically 'unpacked', making resources available that were previously not addressable by generating public URLs for those resources in the bundle that were referenced relative to the bundle itself. The bundle concept also allows resources to be 'repacked' for archiving, or deployment into a firewall-restricted execution environment.
- While SHIWA currently uses the XML serialization of RDF, various serializations exist including N3, Turtle and RDFa. Furthermore, ORE has a binding to the Atom Syndication Format[3]. This flexibility of exchange format means compound objects can be integrated with a wide variety of systems, for example as web pages through the combination of HTML 5 and RDFa.
- ORE supports evolution of existing aggregations by simply adding a new URL to a resource map. This provides a very simple way for users to work with their existing aggregations and develop them over time.

3.1 SHIWA Properties

The use-cases addressed by the SHIWA bundle include human publication, search and sharing of workflow artifacts, as well as execution of workflows in a variety of environments. While the aim is to introduce as few as possible new RDF terms, some of the requirements of the use-cases are not covered by existing vocabularies. Bundles employ widely used RDF vocabularies, such as the Dublin Core (DC)[4] metadata elements that broadly describe resources and Friend Of A Friend (FOAF)[5] elements that describe human entities. Beyond this, we employ the Simple Knowledge Organization System (SKOS)[6], which provides a means of creating thesaurus-like collections of *SKOS Concepts* without resorting to defining new, and hence less interoperable, vocabularies. SKOS

[3] `http://www.toolsietf.org/html/rfc4287`
[4] `http://www.dublincore.org/`
[5] `http://www.foaf-project.org/`
[6] `http://www.w3.org/TR/skos-reference/`

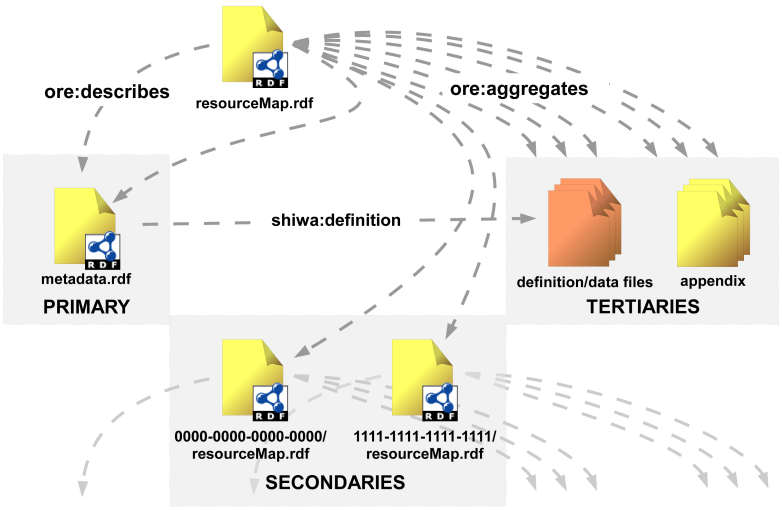


Fig. 1: Anatomy of an Aggregation in a SHIWA Bundle

is used to model those elements that are specific to workflows and the requirements of the bundles' metadata. Some of the SHIWA properties are addressed within this paper; the remaining SHIWA properties were defined in SHIWA Deliverable D5.2.[7]

3.2 SHIWA Aggregations

The SHIWA bundle has a directory based structure, which is managed by one or more Resource Maps. While the structure of a SHIWA bundle is only constrained by the presence of a *resourceMap.rdf* file in the root directory of the bundle, the SHIWA software implementation makes use of certain conventions in organising the files within a bundle. These help in the interpretation of the structure of the artefact contained in the bundle. The aggregation's RDF description is defined in a *metatdata.rdf* file at the root of the bundle. Aggregations of resources, each described by its own Resource Map, are organised within their own directories. There will always be one root Resource Map within the bundle that acts as the entry point of the bundle. The Resource Map metadata file and supporting file structure will be found in the root directory of the bundle. All files of sub-resource maps are stored in sub-directories, with the UUID value of the sub-aggregation as the directory title.

Figure 1 illustrates the physical composition of an aggregation, along with the relationships between the files which is maintained by the resource map. Each file referenced by an Aggregation's Resource Map is described using an ORE *Aggregated Resource*, which allows the purpose and status of each file to

[7] `http://www.shiwa-workflow.eu/documents/10753/626f809c-7853-40ce-a3b2-eb41a29a9ecd`

be described in further detail by the metadata in the resource map metadata. The aggregated resources in the resource map are broken into three sub-types:

- The Primary Resource is the metadata file that describes the main properties of the aggregation. The resource map uses the RDF tag *ore:describes* to identify this resource.
- Secondary Resources are the primary resources of child aggregations, which therefore are described by their own resource map. These are identifiable as they declare they are described by a different resource map using *ore:describedBy*.
- The Tertiary files present within an aggregation contain the concrete data related to the aggregation. Within the resource map, the tertiary resources will declare the type of file they are using *rdf:type*:
  shiwa:definition The definition file is the main implementation file of a concrete task.
  shiwa:datafile Data files are the input and output data associated with mapping concepts. These files are referenced within the primary resource of a data, environment or execution mapping.
  shiwa:bundlefile Any other tertiary files are defaulted to the bundlefile tag. These files may be supporting documents relating to a task or mapping such as readme files, related publications or screen shots.

In terms of the physical structure within an aggregation, the primary resource file and the tertiary file structure that belong to a specific aggregation will be located in the same directory as the resource map whilst any secondary resources (and their subsequent file structure) are located in individual subdirectories.

3.3 Aggregation Types

There are five types of aggregation used to represent common metadata structures required to model workflows and their composite tasks. These can be organized in a multitude of ways that represent workflows and workflow components in different states within the life-cycle of workflow development and execution. Each one of these elements is represented within a bundle via its own resource map and supporting file structure as described above. In a bundle, each aggregation will have a UUID associated with it to allow cross referencing between aggregations within the metadata. Figure 2 illustrates the relationships between aggregations, with I representing input ports, O output ports and D dependencies. We describe each of these aggregation types in the following paragraphs.

Concrete Tasks represent computational tasks. These can be individual tasks within a workflow, or a workflow in its entirety. They will contain all the executable data required to run the task, as well as highlighting any dependencies on external systems and environments the task has. The input and output of the task will be described using a *task signature* (Section 3.4).

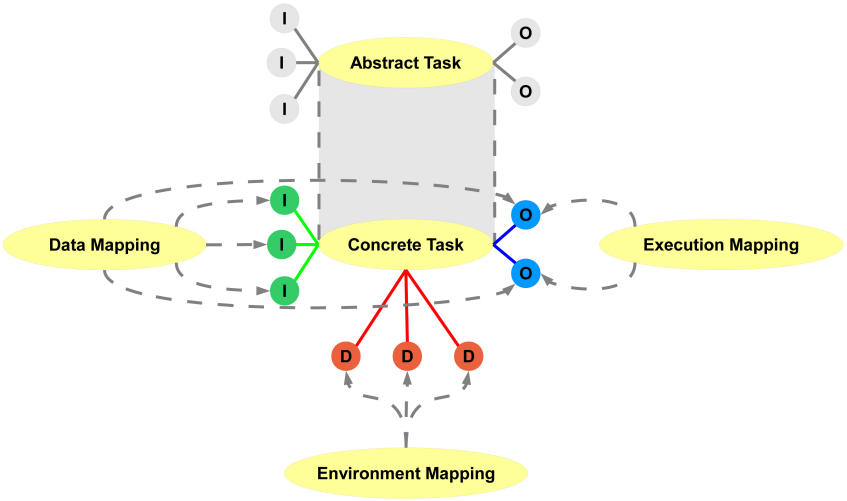


Fig. 2: Bundle Metadata Structures and their interactions

There is a CGI specific extension of the Concrete Task aggregation found inside the bundling API (Section 5.1), the Workflow Implementation, which allows the Concrete Task to be enhanced with more CGI specific properties. The Workflow Implementation is used for bundles found within the CGI pool (see Section 3.5).

Abstract Tasks are used to aggregate together tasks of the same function and signature in order to facilitate interoperability. An abstract task is environment independent, allowing developers to design workflows that are not constrained to a specific operating system or workflow environment.

Data Mappings are sets of data that can be applied to a particular task. Data is mapped to the input ports and output ports identified in the workflow's signature using the *shiwa:reference* concept. This data can make the task immediately executable, or may only fill in some of the data required to execute the task. Data mappings support validation of workflow bundles via test input data and expected results data. They also support workflow reuse – for example, through defining workflow parameters suitable for particular types of experiments.

Environment Mappings describe things such as virtual organizations or middleware required by the workflow to execute properly. This is important, not just for programmatic selection of workflows, but also for users to understand whether they will be able to execute the workflow themselves, given their own profile and VO memberships. These are mapped to the dependencies exposed by a Concrete Task.

Execution Mappings hold the output data produced by a concrete task after it has been run. These map only to the output ports of a task's signature (unlike a Data Mapping).

3.4 Task Signature

A *task signature* is generated from the metadata and provides an interface that distinguishes a concrete task instance in terms of its task type, the data it receives, and the data it outputs. These are derived from the *tasktype* concept as well as the definition of the *inport* and *outport* elements of the task. The aim of the signature is to enable programmatic grouping and discrimination between tasks to allow runtime selection and embedding of task and workflow bundles into other workflows.

Along with the name of the task artifact, the elements that make up the task signature can be directly mapped to the IWIR atomic task definition enabling a smooth adoption and integration between FGI and CGI tasks represented in both bundles and IWIR graphs.

A signature does not model any internal 'wiring' of the task or workflow — it merely describes the task/workflow as a black box. While this gives a high-level view of the task, it may not be enough for either a human or software agent to make decisions about its applicability to a particular function or environment.

3.5 Bundle Configurations used in the execution pool

SHIWA Bundles may be organized in a variety of ways as a means of achieving different tasks. Two types of bundle configuration are involved in workflow execution with the pool: Workflow Execution Bundles and Workflow Result Bundles, which are illustrated by Figure 3 and Figure 4 and described below.

Workflow Execution Bundle. These bundles are used to initialize workflow executions through the pool and consist of a Workflow Implementation and both a Data and an Environment Mapping. This should provide all the information required to start a new workflow execution, with the onus being on the Workflow Engine to marshal the data provided in order to execute.

Workflow Result Bundle. Once a Workflow has completed running, the results set will be placed in a new bundle via an Execution Mapping and sent back to the pool for the original submitter to retrieve. The original Workflow Implementation aggregation will also be placed in the bundle, so that the result data can be associated with an initial submission.

4 Execution Pool

Workflow execution services usually consist of a collection of engine endpoints that a meta-engine may use to submit and monitor workflow executions. When multiple workflow languages are considered, the meta-engine has to be extended to associate endpoints with workflow languages. Engine interfaces can be made uniform, for instance using the bundle format described in Section 3.

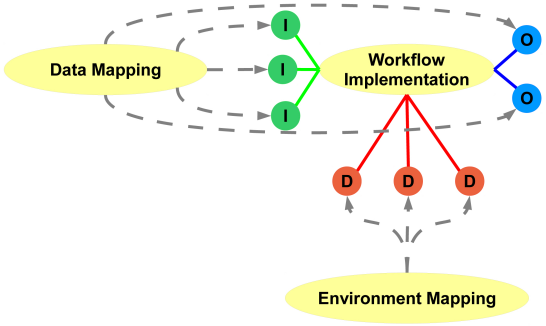


Fig. 3: Workflow Configuration Bundle

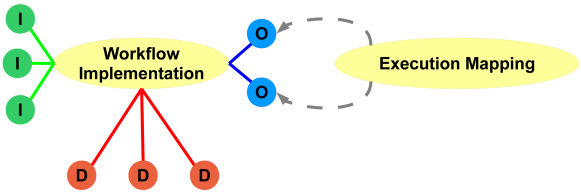


Fig. 4: Worklfow Result Bundle

The meta-engine, however, still has to select an appropriate engine to execute a workflow expressed in a given language, and adding a new language implies reconfiguring the meta-engine. In addition, linking a list of endpoints to a list of languages is failure-prone due to the dependence of the meta-engine upon services hosted in different institutions, and with different capabilities. Without any further precaution, the meta-engine may well overwhelm a particular workflow engine due to excessive submission, or lose time in trying to contact engines that are out of service. To tackle these issues, we present here a pool architecture to execute workflow bundles. The pool is implemented using SHIWA bundles, and it is interfaced with MOTEUR and Triana.

4.1 Architecture Description

The pool architecture is shown on Figure 5, with the numbered arrows corresponding to the life-cycle of a successful workflow execution (along with their message types and properties), and circle-terminated arrows denoting broadcasts. The system has a central pool, to which clients can submit workflow execution bundles, monitor them and get results as workflow result bundles. Agents, also called workflow executors, are distributed and they can connect to the pool, get execution bundles, spawn workflow engines through engine plugins, send status updates to the pool and clients, and transfer result bundles back to the pool.

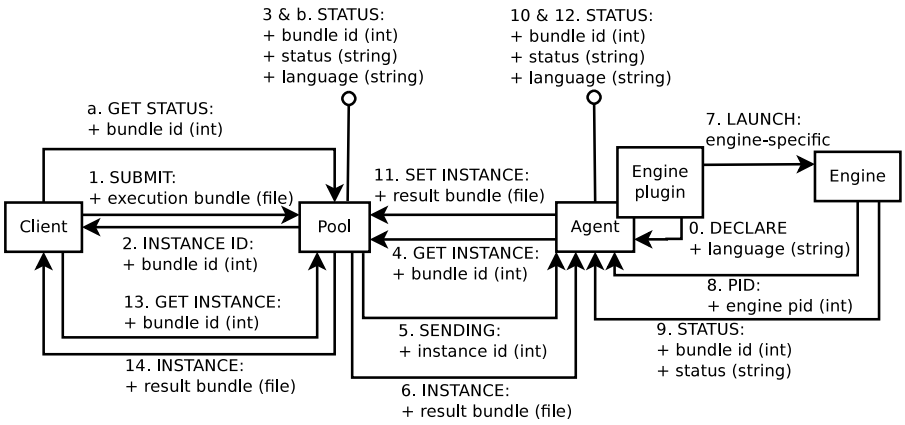


Fig. 5: Interactions between client, pool, agent, and engine.

When an agent starts, its embedded engine plugins declare their supported workflow language so that agents know which bundles they are able to execute. Engines are (lightly) instrumented to report bundle statuses to their agent. Workflow bundle statuses are kept both by the pool and by the agent, as shown in Figure 6. When a bundle is submitted, the pool puts it in status `PENDING`. The pool periodically broadcasts a status message for pending bundles so that new agents are informed. To improve scalability, a single message containing the statuses of all pending bundles is sent. A timeout ($T_{pending}$) can be set on `PENDING` bundles. It expires when no agent is able to take the bundle – for example, because the workflow language or any other bundle dependency is not supported, or because all agents are busy. The bundle is then put in status `KILLED`. If bundle requests are made by agents, then the pool selects an agent, sends the bundle to it, and puts the bundle in status `SENDING`. The bundle is then transferred to the select agent, and put in status `SENT`, or `FAILED` if the transfer fails. A timeout (T_{sent}) is started to detect bundles blocked at this stage. If it expires before the agent sends a `RUNNING` status message, then the bundle is put in status `KILLED`. Once the bundle is running, the pool waits for status updates from the agent until the bundle is `FINISHED` or `FAILED`. The connection with the agent is also periodically checked. In case it is lost, then the bundle is put in status `KILLED` after a timeout. When the agent receives a `PENDING` status message, it checks the workflow language of the bundles concerned, and also the number of locally active bundles.

If conditions are met for execution, the agent selects a random bundle, sets the status of the bundle to `WAITING`, requests the bundle from the pool, and starts a timeout ($T_{waiting}$) count. If the timeout expires (e.g. the pool has selected another agent to run the bundle or the pool did not receive the request message) then the bundle is deleted. Otherwise, the bundle is put in status `SENDING` until the pool sends a `FAILED` status message, or the agent receives the bundle. In the latter case, the bundle is put in status `LAUNCHING`

and a workflow engine is spawned. A timeout $T_{launching}$ is spawned to kill the bundle in case the engine fails to start. The bundle status is then updated by the engine until completion or failure. The agent also kills running bundles when their engine crashed after timeout $T_{engine\_crashed}$.

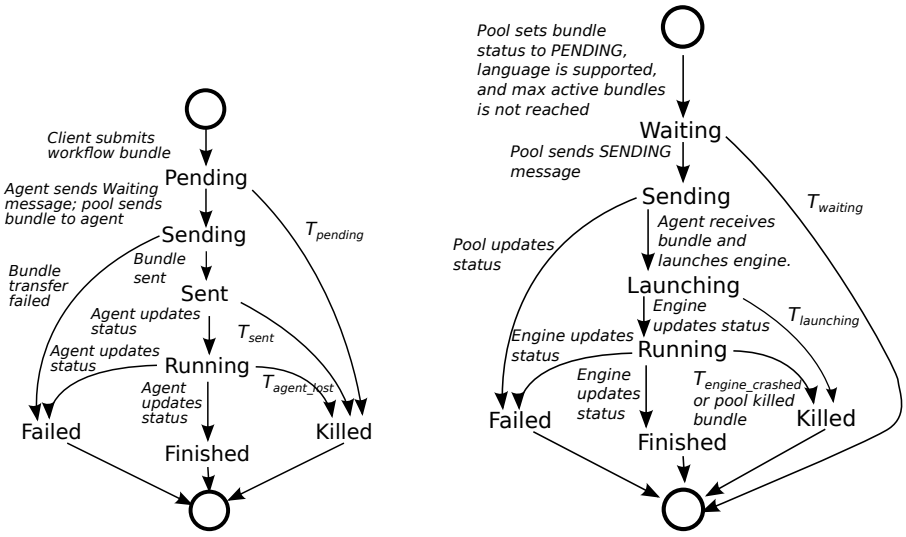
4.2 Pool Properties

This architecture is scalable because several agents supporting the same workflow engine can coexist without any special configuration of the pool. The centralized pool handles concurrency to ensure that at most one agent will execute any given bundle. Timeout $T_{waiting}$ on Figure 6b guarantees that agents will not wait forever in case of concurrent execution requests. In addition, the maximal number of active bundles in an agent is configured in the agent (see first transition in Figure 6b), so that some agents may accept a few executions only while others, e.g. agents deployed on a cluster frontend, could support more. Therefore the architecture can be customized to heterogeneous execution infrastructures without reconfiguring the pool. This architecture is robust to agent crashes because (i) a crash will impact only the workflows being run by the agent and the time of the crash, and (ii) failover agents can be dynamically started without reconfiguring the pool.

The pool model is also robust to agent overload. In the case where an agent reaches its maximal capacity, it will stop requesting bundles from the pool to avoid being overwhelmed. And in the case of agent downtime, no additional latency is introduced because of submission failures. Workflow bundles are handled independently from their language. Language-specificity required for the execution lies in the agent plugins which perform the required conversions between the bundle and the native workflow format. Similarly, clients are responsible for language-specific bundling of the input data before executions are submitted to the pool. Supported workflow languages are declared by engine plugins to their agent at start up, so that new workflow languages can be dynamically supported without any modification in the pool, client, and agent. Note that meta-workflows are intrinsically supported because engines can submit workflows of different languages to the pool.

5 Bundling and Pool Integration into MOTEUR and Triana

In this section, we explain how bundling support has been integrated into Triana and MOTEUR, which makes them capable of operating within the execution pool environment. MOTEUR and Triana are the first two workflow engines to be integrated with the CGI pool as they have been developed and maintained by CNRS and Cardiff University respectively. There are a number of tools provided within the bundling toolkit that facilitate this process, which are discussed first.



(a) In pool (b) In agent

Fig. 6: State machines of workflow bundles. Initial and terminal states are figured with circles. T indicates timeouts.

5.1 SHIWA Desktop

The *SHIWA Desktop* software has been developed in order to create an interface that can be integrated into workflow systems desktop applications in order to provide a single common process for creating and manipulating and publishing bundles to the Web. The underlying element of this software is a Bundling API called SHIWA Desktop Data, which enables fine-grained creation of SHIWA Bundles programmatically. This is complemented by a *WorkflowEngineHandler* interface which individual workflow systems need to implement to support bundling. Above the SHIWA Desktop Data API are two user interfaces which are intended to be the common means of bundle creation: a GUI which can integrate with GUI-based Workflow engines and a Command Line Interface for workflow engines without a GUI.

There are five major parts of the Desktop Data API which are used in CGI and the pool:

RDF Element Objects. A collection of objects representing the elements found in the RDF files; these provide a data structure to the rest of the API and allow the metadata information to be serialized to and serialized from their RDF representation.

SHIWABundle Object. Reads in SHIWA bundles, converting the metadata into the data structure mentioned above.

WorkflowController Class. Provides methods for retrieving aggregations from a SHIWABundle object; exposing the main aggregations concerned with CGI for simpler retrieval.

DataUtils. Provides functionality for downloading bundles from, and uploading bundles to, remote locations, allowing users to store workflows on external repositories or deploy them to external execution pools.

WorkflowEngineHandler Interface. The primary mechanism for integrating with a workflow engine, allowing ease of publishing and deployment of workflows.

A full overview of the SHIWA Desktop Data API can be found in SHIWA Deliverable D5.3.[8]

A workflow engine handler is an engine-specific component that understands the internal operations of an engine and its object models. The interface is designed to be simple to implement, and it provides a means of pre-populating technical metadata, that should not be specified by a user, directly from the engine environment. Apart from providing SHIWA Desktop with simple information such as the workflow engine name and version, and the workflow language of the workflow, the handler must also be able to create a *TransferSignature* object and return an *InputStream* to a serialization of the workflow definition.

A TransferSignature is a simple container object that has a name, and a list of inputs and outputs. It is also possible to associate data (either inline or by reference, e.g. URL) with the inputs. Each input and output has a unique name, a data type, and optional input data associated with it, as well as an optional human-readable description.

All these are provided through the interface and can be adjusted using the SHIWA Desktop GUI at a later stage. This means generic descriptions could be given by the handler, to be improved upon by the user if required.

The GUI can be run either within a Workflow Engine's own GUI or as a stand-alone application, and is intended as the main interface of SHIWA Desktop. The GUI has been integrated into both Triana and MOTEUR, providing users with identical interfaces for generating bundles from workflows and publishing and retrieving bundles to and from remote locations.

5.2 Pool Implementation

Prototype pool, agent, and client implementations were created in Java. The Extensible Messaging and Presence Protocol (XMPP[9]) was chosen for the communication layer due to its ability to enable communication among distributed peers with no inbound connectivity. Only the XMPP server has to have a port open. The smack Java API[10] v3.2.2 was used to handle XMPP

[8] `http://www.shiwa-workflow.eu/documents/10753/8bc729cf-34ac-4bfe-bb96-9ce8ebf9f8ca`

[9] `http://xmpp.org`

[10] `http://www.igniterealtime.org/projects/smack`

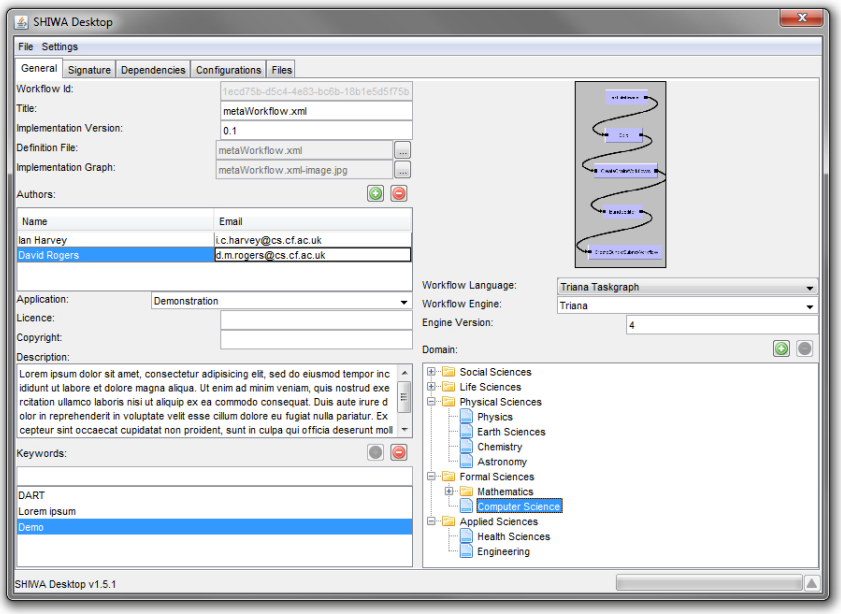


Fig. 7: SHIWA Desktop GUI

operations. XMPP messages exchanged during the life-cycle of a workflow are the ones shown in Figure 5. The pool has 3 threads used to receive/process messages, transfer files, and monitor timeouts ($T_{pending}$, T_{sent}, and T_{agent_lost} on Figure 6a). A local database is used to store workflow bundle statuses and bundle file paths (input and output) so that restarting the pool does not impact active bundles. The agent also has 3 threads to receive/process messages, transfer files, and monitor timeouts ($T_{waiting}$, $T_{launching}$, and $T_{engine\_crashed}$). Again, a local database is used to store bundle statuses and engine UNIX process identifiers so that restarting the agent does not impact active bundles.

A Java interface based on the Java Simple Plugin Framework (JSPF[11]) is provided to write engine plugins. It has two methods: bundle execution launch and result bundle creation.

5.3 Triana Integration

Once a workflow has been designed in Triana it can be wrapped in a bundle by sending the workflow to SHIWA Desktop using the *TrianaEngineHandler*, which implements the WorkflowEngineHandler interface described in the SHIWA Desktop package. The workflow file, the workflow's name, a screenshot produced by Triana, and a description of the inputs to, and outputs from, the workflow are sent via the handler. These descriptions include the port number

[11] `http://code.google.com/p/jspf`

and the type of data expected, and this is used to automatically fill in data within the SHIWA Desktop panel.

The handler is instantiated with a Triana *Task*, which is the *TaskGraph* object retrieved from the Triana *ApplicationFrame*. This *Task* object contains accessors to all the inports, outports, parameters and connectivity options for the workflow, and, if run from the Triana GUI, will have all the recent information created by the user.

A SHIWA Desktop *task signature* is produced by creating a new *SHIWADesktopPanel* object with the *TrianaEngineHandler* as its argument. The *SHIWADesktopPanel* returns a *JPanel*, which Triana places in a customized *JDialog*, and displays within the SHIWA desktop environment (Figure 7). While Triana has the ability to display numerous TaskGraphs at the same time, it is the currently selected object which is used. If for any reason a taskgraph is not available, an appropriate error is shown.

A pool agent plugin was also developed to execute Triana workflows submitted to the pool. The TrianaCloud Broker (Section 5.4.1) is registered to the pool via a small intermediary application; the Triana Filter, which implements the engine plugin interface and passes bundles directly to the filter.

5.4 Executing Triana in a cloud environment

The Triana workflow engine was designed to load extensions during its initialization. A new extension has been added that unpacks a bundle and sets up the runtime environment prior to Triana being invoked. It attempts to retrieve all files required for the workflow; either via a download from a remote URI, or via a reference to files within the bundle. These are copied into a runtime folder before Triana itself is invoked, ensuring runtime efficiency. An attempt to run the workflow will be made regardless of whether or not the dependencies are met. In future this state will be caught, and will fail early, to allow runs to attempt runs in other environments, potentially running in parallel to this execution.

This early invocation of bundles means Triana can now unbundle, execute, and consequently bundle an entire workflow using command line parameters. This addition was important to ensure the remote execution of the workflow engine within a cloud environment. The broker, defined and explained below, is deliberately ignorant of how Triana executes a workflow, so a worker running on a cloud node runs the command line invocation relevant for bundle execution. These workers, on registration with the broker, have specified the type of data they are able to accept, so any information packet forwarded via the broker to the worker can be executed using the same launcher.

5.4.1 The Broker

The broker is in charge of distributing the task to the environment it is working with, and is built upon the RabbitMQ[12] message broker platform, which is used for passing tasks around the system. RabbitMQ is an open source message brokering system based upon the AMQP[13] messaging protocol. It provides a mechanism for systems to pass messages with a high degree of reliability and scalability. As RabbitMQ does not place any limitations on the content of these messages, it is ideal for the Broker to pass any form of data between listening clients - in this case bundles between workflow systems.

Due to the design of the broker it can run tasks on any number of different systems, and can do so simultaneously. The broker is ignorant to the nature of the task, acting as an intermediary between whatever system has submitted the task and the underlying workers. This allows the broker and workers to be used to distribute any task, not just Triana based ones (assuming an appropriate executor is written). To facilitate the distribution of these tasks, they are accompanied by some metadata. The metadata contains a routing key in the format *a.b.c...* to ensure that the correct Executor is started by the Worker, and only if the Worker has that executor enabled. An example would be *addition.triana*: this could signify that this task can be run by a Triana executor, which is capable of running the addition task. If the Triana executor was able to run all Triana based tasks, it would listen for **.triana* and would pick up the addition task. Currently, the task data (a bundle in the case of Triana and this experiment) is passed around as a byte array within the task object. When being sent over RabbitMQ, it is converted into BSON (Binary JSON). Future versions of the broker will operate differently; with the data being put into a shared storage area and a URI referring to the data put into the task. This will help reduce bandwidth requirements somewhat. The shared storage will be independent of the broker, allowing it to be tailored to the deployment being used.

The broker also contains a receiver. This keeps track of all the tasks that are sent out, along with their ID. All returned (completed) tasks go to the receiver, which then looks up their ID to find the original task so that it can update it to reflect it is complete.

5.4.2 The Worker

Like the broker, the worker does not understand tasks at all. When run, it looks for all the executor plugins in a plugin folder, and queries them for appropriate routing keys. Once a list of these keys is built up, it listens to RabbitMQ using these keys. When a task arrives, it passes it on to the Executor responsible for that routing key for the task. When the Executor finishes, the worker sends the task and results back to the broker.

[12] http://www.rabbitmq.com/
[13] http://www.amqp.org/

5.4.3 The Executor

Executors are plugins for the Worker. In the Java implementation, they only need to implement 3 methods: *getRoutingKey*, *setData*, and *executeTask*. The first, getRoutingKey, simply returns the routing key so that the worker can receive tasks from RabbitMQ. The second allows the Worker to put the task data into the Executor so the executor can access it. The last, executeTask, does the execution, and returns the results. The implementation of executeTask depends completely on what it needs to do. It may simply take an MD5 sum of the task data, perform some complex analysis, or start another process to, for example, run a non-Java application with the data as an input file of some sort. The Triana Executor just passes the data into Triana (the data in this case is a bundle file), Triana runs the workflow in the bundle, then returns a bundle to the executor.

5.5 MOTEUR Integration

The binding to the MOTEUR workflow engine follows a similar pattern to the Triana implementation. MOTEUR has a very simple API for handling workflow descriptions. A *Workflow* object can be read from and written to files or streams with ease. Therefore the *MoteurWorkflowEngineHandler* has constructors that take either an in-memory representation of a *Workflow* object, or a file object from which a workflow can be read. The *Workflow* object also has methods to retrieve *Input* and *Output* objects. These represent the top level data interfaces to the workflow that we are interested in.

The *MoteurWorkflowEngineHandler* also supports methods for pre-mapping data to certain inputs. If any data has been pre-staged through the handler's API, then this data is added to the workflow's signature and converted to a configuration by the Desktop component when parsing the Signature returned to it. Additionally, all information available in the workflow is captured into the *MoteurWorkflowEngineHandler* object. Examples of such information include workflow title, description and version, workflow authors information and workflow image.

The two most pertinent methods are the *getSignature* and *getWorkflowDefinition* methods. These use the MOTEUR workflow API to extract the relevant information for creating a coarse-grained description of the workflow, and allow the SCUFL or Gwendia XML definition to be read into a bundle file accordingly.

Similarly to Triana, MOTEUR GUI also supports menus for retrieving and publishing workflow bundles. When a workflow is designed or loaded in the MOTEUR GUI, the publish menu is enabled to allow user to create the bundle or upload it to remote repository. All information available in the *MoteurWorkflowEngineHandler* object is then automatically filled into the SHIWA Desktop GUI. If there exists an input file, a data configuration is created. All data items for each input port of the workflow are mapped to

each input port in the data mapping of the bundle. When a bundle is retrieved, MOTEUR uses the SHIWA Desktop API to interpret the bundle and extract the workflow definition and the data mapping and load them to the MOTEUR GUI.

To submit and monitor workflows using the execution pool, MOTEUR provides *SHIWAPoolInvoker* interface that is responsible for invoking the execution of a bundle. A thread is spawned to wait for arriving workflow bundle and submit to the execution pool, using the API provided by Pool Client described in section 4. A second thread is used for monitoring the execution of submitted workflow bundles. It periodically contacts the execution pool to get the status of submitted bundles. When a bundle finishes its execution, the thread retrieves the output bundle, interprets it and extracts the relevant information, using the SHIWA Desktop API.

An engine plugin was also written to execute MOTEUR workflows submitted to the pool. This plugin spawns the engine in a new JVM forked through a system call (step 7 in Figure 5). A listener plugin in MOTEUR was developed to send bundle statuses to the agent (steps 8 and 9 in Figure 5).

6 Experiments and Results on Pool scalability and Robustness

Two experiments were conducted to demonstrate the scalability and robustness of the pool architecture. Version 0.7 of the pool, agent and client was used to conduct these experiments. Sources are publicly available online[14].

6.1 Deployed Infrastructure

For both experiments, the pool and client were deployed on different machines, on the same network as the XMPP server. Agents and engines were deployed on the academic cloud infrastructure offered by StratusLab[15]. We used a FedoraCore 16 x86_64 virtual machine (VM) image with Java, MySQL, and our agent installed. XMPP accounts were manually created for agents, and login/passwords were configured in the deployed VM instances before the experiments started. VMs were deployed on the StratusLab site at Laboratoire de l'Accélérateur Linéaire prior to the experiments. Timeout values were as follows: T_{sent}=30s, T_{agent_lost}=10s, $T_{engine\_crashed}$ = 3s, and $T_{waiting}$ = $T_{launching}$ = 5s. The maximal number of active workflows per agent was 3. The pool broadcasted status messages for pending bundles at a frequency increasing linearly with the number n of pending bundles, with a maximal value of 5 min.

[14] `http://vip.creatis.insa-lyon.fr:9002/projects/cgi-executor`
[15] `http://stratuslab.org`

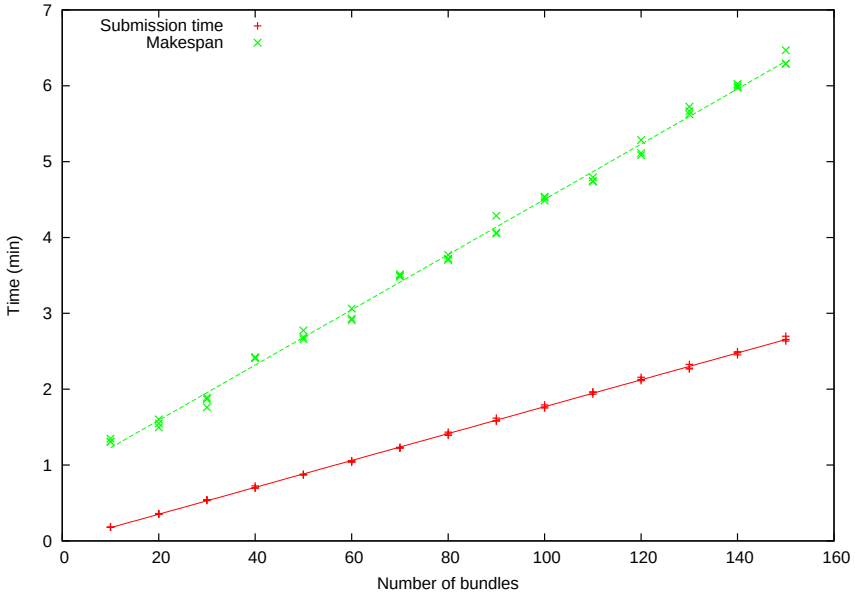


Fig. 8: Evaluation of the pool scalability, w.r.t the number of concurrent active bundles

6.2 Scalability

Scalability was tested both in the number of concurrent workflow bundles (`Exp1-a`), and in the number of available agents (`Exp1-b`). In both cases, a simple MOTEUR workflow consisting of a single activity sleeping for 1 minute was used. Bundles were submitted sequentially to the pool. Three repetitions were done for each number of bundles. For each repetition we measured the total submission time and makespan (duration between the beginning of the submission of the first bundle and the completion of the last one).

For `Exp1-a`, the number of deployed agents was 10, and therefore the maximal achievable throughput was 30 bundle/min (10 agents are deployed and each one can have 3 active workflows). The number of concurrent workflow bundles varied from 10 to 150 by steps of 10.

Figure 8 shows the evolution of the makespan and submission time with respect to the number of concurrent active bundles. Least-square regression lines are also plotted. Both submission time and makespan are close to their regression line, demonstrating the good scalability of the system. Variability among repetitions is low. The submission time is mostly bound by the transfer time of bundles (3.6KB), which is hampered by XMPP's base64 encoding of the transferred files. The makespan linear regression has an inverse slope of 27.5 bundle/min, which is close to the maximum achievable throughput on the deployed infrastructure. The median makespan for 10 bundles is 1min 18s,

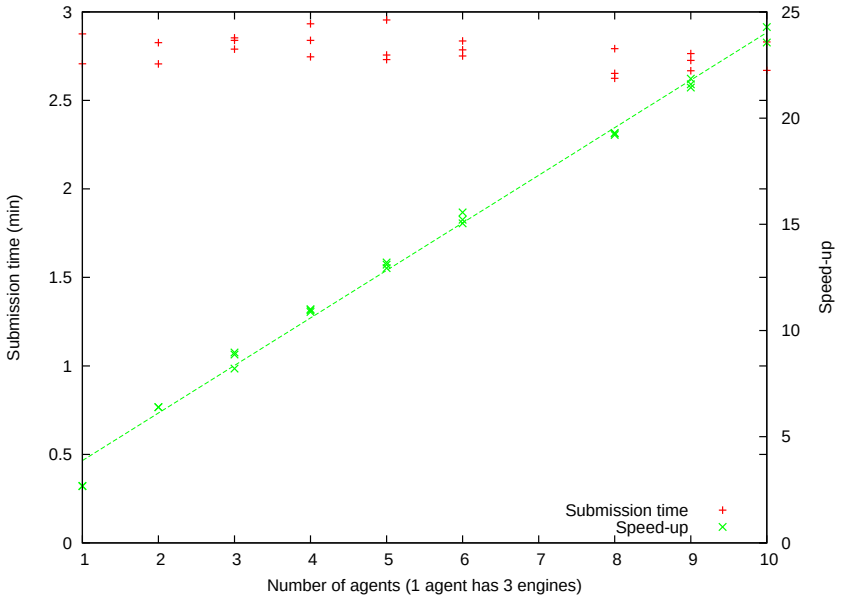


Fig. 9: Evaluation of the pool scalability, w.r.t the number of available agents

indicating that the average latency is 18s. Among these, 11s are due to submission time. The remaining 7s are due to data transfers, expiration of $T_{waiting}$ (5s) and the broadcast frequency of pending bundles by the pool (between 5s and 6s in this case).

For `Exp1-b`, the number of bundles was 150, and the number of agents varied from 1 to 10 by steps of 1. Figure 9 shows the evolution of the submission time and speed-up w.r.t to the number of deployed agents. The speed-up is computed as the ratio between the cumulative workflow execution time (150 minutes) and the makespan. As expected, submission time is quite stable, with a subtle decrease for 8, 9 and 10 agents. Measured speed-up values are well approximated by their regression line, which indicates that overheads remain controlled, leading to scalable performance of the system. The slope of the regression line is 2.21, and the median speed-up for 1 agent is 2.6: these remain under their optimal values (3) due to the overheads mentioned before.

6.3 Reliability

Reliability of the system against agent faults was studied in two configurations: `flapping` and `crash`. In both cases, two agents were deployed. In configuration `flapping`, robustness against temporary connection losses of the agent was tested. One of the agents behaved normally, while the other disconnected from the pool for 5s every 10s. These values were chosen coherently with the value of T_{agent_lost}. In configuration `crash`, both agents behaved correctly during the

Test			Flapping			Crash		
	#Killed	MkSpn		#Killed	MkSpn		#Killed	MkSpn
#1	0	321	#1	0	319	#1	3	454
#2	0	318	#2	0	326	#2	3	385
#3	0	317	#3	0	319	#3	3	454

Table 1: Robustness of the execution pool to flapping and crashed agents.

first 90s, and then one agent was shutdown until the end of the experiment. The makespan (MkSpn) and number of killed bundles was measured in both cases, and in a test configuration where both agents behaved correctly.

Results are reported in Table 1. As expected, the pool architecture is totally robust to flapping agents. In this configuration, no execution was killed, and the makespan compares to the one obtained in the test configuration. Agent crash only has limited impact on the system. It only impacts the executions that were running when the crash occurred (3 in our case), without any consequence on the subsequent executions. The makespan increases compared to the test configuration due to the availability of only 1 agent after the crash.

7 Experiments and Results on Meta-Workflow Execution

7.1 Experiment Use Case

In this section, we describe the meta-workflow scenario, which uses distributed DART [28] Music Information Retrieval (MIR) workflows to perform a parameter sweep experiment in order to discover the optimal parameter settings for the sub-harmonic summation pitch detection algorithm. The meta-workflow is a Triana workfllow, which generates Triana and MOTEUR bundles containing DART workflows that are then put in the pool. As depicted in Figure 10, this application has three parameters:

- *freqpoints_max* - Number Of Top Frequency Points (NTFP) : Vary 1 to 501 in 10 point intervals (51 in total)
- *harmonics_max* - Number of Harmonics: Vary 1-32 (5 Octaves)
- *audio_file* - Audio Input Files: 6 audio files

This parameter sweep experiment creates 9,792 concurrent jobs in total, with 1,632 jobs per audio input file. Each run results in an output file which will be downloaded to a local folder. When all runs finished their execution, the results folder is zipped.

This application was originally executed on a Cloud installation at Cardiff University using the BOINC[16] software to distribute it. The execution time of the total data set is just over 3 days (run on 120 machines with 1 core 1GB of memory). In this experiment, we demonstrate the usefulness of the bundle and execution pool approach by distributing the execution of this application to

[16] `http://boinc.berkely.edu`

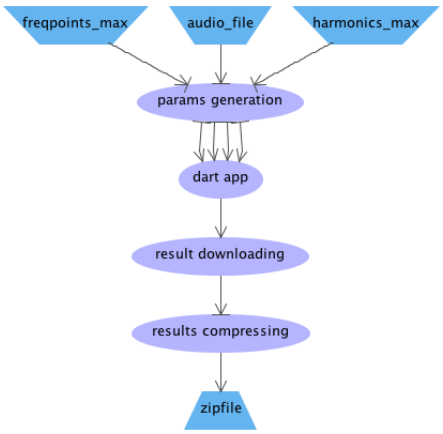


Fig. 10: DART Music Information Retrieval (MIR) workflow

two workflow environments: Triana workflow engine running on TrianaCloud and MOTEUR running on the European Grid Infrastructure (EGI). We use Triana to run the meta-workflow which is set up to publish the MOTEUR bundles to the CGI pool backbone and send the Triana bundles (as the end location was known) directly to TrianaCloud. MOTEUR pool agents retrieved the bundles from the execution pool to execute on the EGI. The final objective is to reduce the total execution time of the complete dataset. Each environment will therefore run over only 3 of the audio files. We have kept the value of *freqpoints_max* constant at 501, resulting in 51 intervals. this means we have 153 jobs to run per *harmonics_max* value and so we vary the value of *harmonics_max* from 1 to 10 in order to test the performance of the execution environment with increasing loads. When the *harmonics_max* value increases, the number of concurrent jobs submitted to each computing infrastructure increases accordingly as presented in Table 2.

harmonics_max		2	5	10
Number of concurrent jobs	153	306	765	1530

Table 2: Number of concurrent jobs submitted to the infrastructure in function of the value of the *harmonics_max* parameter

In the following sections, we analyze the application performance on both infrastructures. We are interested in the total execution time and we give a discussion to compare these two infrastructures.

Type	Succeeded	Failed	Incomplete	Total	Retries	Total Run
Tasks	862	0	0	862	0	862
Jobs	765	0	0	765	0	765
Sub-Workflows	32	0	0	32	0	32

Table 3: Output from statistics script, based on Triana (Stampede) logging data

7.2 Distribution using MOTEUR

MOTEUR is an intrinsically data-parallel workflow engine. It enables simultaneous, asynchronous execution of multiple data fragments. It adopts a nested data array-centric model, where arrays of data fragments are pushed through the workflow inputs, flow through the workflow data links, and cause the workflow activities to be executed in parallel, potentially as many times as data fragments received. To determine the number of parallel iterations applying to each activity, the GWENDIA language defined the notion of activity input port depth. At depth 0 (default) each data fragment causes an iteration of the target activity to process it. A higher port depth means that a complete array needs to be received before the activity can be executed. As many array nesting levels as the port depth are then collected. This implements a data synchronization barrier at the level of activity invocation. Similarly, an executed activity can either return a single data fragment, or a structured collection of such fragments in the form of a nested array, for further parallel processing by subsequent activities. Figure 10 shows a MOTEUR workflow exemplifying both behaviours. The "params generation" is an array-producing activity, which outputs deliver complete arrays of data fragments to be processed independently by subsequent "dart app" and "result downloading" activities. Conversely, the "result compressing" activity synchronizes a complete array of data fragment results before invocation. It therefore processes all downloaded results simultaneously. The jobs generated by the activities invoked are distributed on the target Distributed Computing Infrastructure for concurrent execution by the core MOTEUR engine using its asynchronous invocation capability.

7.3 Performance timings for Triana and MOTEUR

7.3.1 Triana with TrianaCloud

The Triana environment attached to the cloud is fronted by the Triana broker. When the broker receives a bundle a new Triana instance is spun up on the cloud and the bundle is passed to it and run. Each node generated will require startup and wind down time, but these startups are performed in parallel, and so waiting time will be near constant regardless of the number of jobs required.

Workflow Wall Time	22 mins, 55 secs	1375 seconds
Cumulative Job Wall Time (CJWT)	1 day, 2 hrs	96944 seconds
CJWT as Seen From Submit Side	21 hrs, 16 mins	76616 seconds

Table 4: Triana (Stampede) cumulative wall time report

It would only be through splitting the workload up into more bundles and subsequent workers, that waiting time would differ.

Execution logging in TrianaCloud is performed by the Stampede logging system [29,30], providing Triana with detailed logging information as seen in Table 3, where information on the final state of jobs within the workflow is given. Stampede also provides us with cumulative wall times to give an indication of the effectiveness of distributing the workload across multiple nodes as shown in Table 4.

The jobs were split between 32 sub-workflows when submitting the workflow to the cloud, each of these sub-workflows require three additional jobs to be created for retrieving the execution data, starting up the DART workflow and re-bundling and submitting the results. The submission workflow is also modeled as a job within the Stampede logs, and so along with the extra worker jobs account for the extra jobs found in Table 3.

Table 5 shows the execution times and cumulative runtimes of several workflow runs. A single DART job takes roughly a minute to run, and so it is not surprising to see that the final runtime of the workflows increase in line with the number of jobs in a bundle, as the bundles are executed simultaneously. It must be noted that the jobs are not evenly distributed between the bundles, jobs are assigned to a bundle until the bundle is "full" potentially leading to a single bundle containing a smaller number of jobs to execute, accounting for the drop in execution time of the third and fourth runs.

harmonics_max	**# Concurrent jobs [per bundle]**	**Execution time (min)**	**Waiting time (min)**	**Runtime (min)**	**Cumulative Runtime (min)**
1	153 [5]	5.92	0.55	6.47	311.72
2	30 [10]	10.02	0.55	10.57	586.17
5	765 [24]	22.36	0.56	22.92	1615.73
10	1530 [49]	41.43	0.55	41.98	3542.30

Table 5: The execution times of the application on TrianaCloud in function of *harmonics_max*

7.3.2 MOTEUR with EGI

In the MOTEUR environment, all workflow jobs are submitted to the European Grid Infrastructure (EGI) through the DIRAC pilot system [31] thanks

to the submission back–end GASW [32]. The execution time of the application is presented in table 6. We see that when the number of jobs submitted to the infrastructure increases, the execution time also increases.

Indeed, EGI is a production infrastructure with many users running their applications. Users jobs therefore are put into a batch system queue to wait for a computing resource. As shown in Figure 11, a timeline diagram for the execution of 153 jobs submitted to the infrastructure, each job is represented by a line starting with red color for the waiting time. The time for input and binary downloading is in yellow; the running time is represented in green, and result uploading is in blue. The waiting time increases when the number of jobs submitted to the infrastructure increases as presented in Table 6, where the value represents the average waiting time for each job to obtain an available computing resource. Such long waiting times seriously hamper short tasks such as DART's. Furthermore, DART is a Java-based application and the Java runtime environment is not available on EGI. Users' jobs therefore have to install JRE on the fly to be able to execute. This leads to the fact that jobs have to take time to download and install JRE.

harmonics_max	**# Concurrent jobs**	**Execution time (min)**	**Waiting time (min)**	**Runtime (min)**
1	153	32.72	10.78	43.5
2	306	83.17	24.99	108.16
5	765	185.27	57.52	242.79
10	1530	301.68	109.98	411.66

Table 6: The total execution time of the application on EGI in function of *harmonics_max*

Comparing to TrianaCloud, the total execution time of the application on EGI might be longer. Indeed, on TrianaCloud, computing resources are dedicated to the user. Users jobs are executed with very little waiting time. Furthermore, users have overall control of their computing resources and can install all necessary software before executing the application, which was not the case on EGI. It can also be seen that the clustering of jobs in the TrianaCloud execution greatly speeds up execution time (by a factor of ten), as the data need only be downloaded once per bundle as opposed to being downloaded once per job when the workflow is run on EGI.

8 Conclusions and Future Work

In this paper, we have updated and expanded on the SHIWA Bundle format, providing a formalization of the SHIWA aggregation types and the bundle configurations that can be used in the CGI pool. This format has been adopted by the EU-funded SHIWA project as a means of providing a common input data

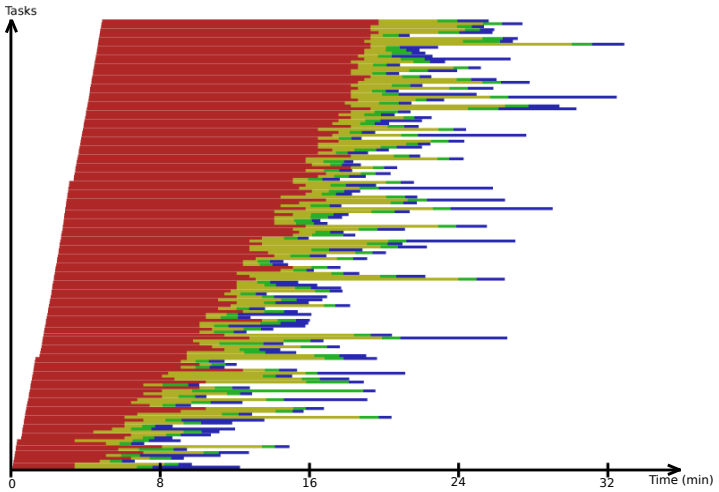


Fig. 11: Jobs timeline of DART application on the EGI

format interface, both for sharing workflows and for enabling the automatic execution of workflows native to that platform in a coarse-grained fashion. We have explained that a SHIWA bundle comprises one or more aggregations of resources, each aggregation having its own Resource Map. A SHIWA-specific core schema is specified, to support the SHIWA concepts that need to be represented, but these have been released to the public and we are involved in on-going discussions with the intention of stabilizing these definitions across different projects. For example, we are discussing this format with the Pegasus, myExperiment and Taverna teams, and plan to reach out to other communities also, over time. Bundles make it possible for workflows to be embedded and for the creation of meta-workflow pipelines connecting one workflow system to another. At the highest level, bundles therefore enable heterogeneous workflow execution through the creation of meta-workflow pipelines.

We have described a language-independent pool model that exploits the portable nature of bundles to provide a dynamic and flexible workflow execution environment. Our execution pool enables decentralised, fully configurable workflow execution services and through a set of interfaces for creating and manipulating bundles, we have integrated bundles with the Triana and MOTEUR workflow engines in order to operate in the pool. The scalability experiments described in this paper have shown that the execution pool was scalable with respect to the number of workflow bundles and agents, and that it was robust to flapping agents and agent crashes. We have used this pool as part of a multi-DCI, multi-language, meta-workflow execution. We have also begun to explore the benefits provided by the CGI pool's approach to workflow execution both as an execution environment for workflow experiments. As a means of comparing the performance of different workflow environments

attached to the pool against one another in order to determine the optimal execution environments for a specific workflow.

We have successfully presented a proof of concept system where each stage of a meta-workflow's execution can be orchestrated by the CGI pool, but some stages of the experiment would benefit from refinement through future work. The Triana meta-workflow was responsible for passing the meta-worklfow bundles into the CGI pool. Moteur required a specific input file to be passed with the Moteur bundle, which Triana was able to provide. The output files produced by Moteur were not connected to ports, instead a single output zip file was added to the bundle, which Triana was able to retrieve and unzip, but this meant that the full meta-workflow could not be completed as the retrieval required extra inspection. For a full experiment, a more structured returned bundle from Moteur would allow passing on of outputs directly within the meta-workflow, instead of having to retrieve the output zip from the bundle through deep bundle inspection. Additionally an interface within Moteur which creates the input xml file itself would remove the need for Triana to send Moteur specific inputs as well as the general workflow data. This would allow metrics to be achieved describing the overhead associated with Triana locating and retrieving output bundles from the pool.

Further experiments could attach other workflow engines to the pool, for example ASKALON or WS-PGRADE, which also are able to read and understand the bundle format. On top of this, future work should look at providing researchers with metrics and tools that aid in the developments of large multi workflow language workflows, highlighting the best environments registered to the pool for executing particular subtasks of the multi workflow. For bundles, we are investigating the possibility of making SHIWA bundles interoperable with the myExperiment ORE format so that SHIWA bundles can be exported to myExperiment for dissemination and searching within a social networked environment. Furthermore, we would look to integrating the two ORE formats to provide a true heterogeneous means of representing workflows and workflow related research within the DCI community.

9 Acknowledgements

The authors would like to thank the SHIWA project for its financial support. The SHIWA (SHaring Interoperable Workflows for large-scale scientific simulations on Available DCIs), is an Integrated Infrastructure Initiative (I3) project co-funded by the European Commission (under contract number 261585) through the Seventh Framework Programme. We would like to thank our collaborators in SHIWA, in particular Tamas Kukla, Kassian Plankensteiner and Vladimir Korkhov for their consultation during the development of the bundling schema and SHIWA Desktop. We thank StratusLab (`http://stratuslab.org`) for providing infrastructure and support for the experiments.

For Triana, we would like to thank our sponsors, PPARC (GridOneD and Geo 600) for the development of Triana, UK STFC TRIACS project ST/F002033/1 for the Triacs work, Wellcome Trust for the Sintero work and the EU for the Gridlab project to help the development of the distributed computing capabilities and SHIWA for the development of the SHIWA bundles that provide the cloud-based distributed mechanisms, described in the Triana sections of this paper.

References

1. Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, 2009.
2. Marek Wieczorek, Radu Prodan, and Thomas Fahringer. Scheduling of scientific workflows in the askalon grid environment. *SIGMOD Record*, 34(3):56–62, 2005.
3. T. Fahringer, R. Prodan, R.Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. *Workflows for e-Science*, chapter ASKALON: A Development and Grid Computing Environment for Scientific Workflows, pages 143–166. Springer, New York, 2007.
4. I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *16th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 423–424. IEEE Computer Society, New York, 2004.
5. Peter Kacsuk. P-grade portal family for grid infrastructures. *Concurr. Comput. : Pract. Exper.*, 23:235–245, March 2011.
6. E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D.S. Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.
7. Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.
8. Andrew Harrison, Ian Taylor, Ian Wang, and Matthew Shields. WS-RF Workflow in Triana. *International Journal of High Performance Computing Applications*, 22(3):268–283, August 2008.
9. Roger Barga, Jared Jackson, Nelson Araujo, Dean Guo, Nitin Gautam, and Yogesh Simmhan. The trident scientific workflow workbench. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 317–318, Washington, DC, USA, 2008. IEEE Computer Society.
10. Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployment of data-intensive applications on grids with MOTEUR. *Int. J. High Perform. Comput. Appl.*, 22:347–360, August 2008.
11. The SHaring Interoperable Workflows for large-scale scientific simulations on Available DCIs Project . `http://www.shiwa-workflow.eu/`.
12. Open Archives Initiative. Object, Reuse and Exchange (ORE). `http://www.openarchives.org/ore/`, 2009.
13. Andrew Harrison, Ian Harvey, Andrew Jones, David Rogers, and Ian Taylor. Object Reuse and Exchange for Publishing and Sharing Workflows. In *Proceedings of the 6th Workshop on Workflows in Support of Large-Scale Science at Supercomputing*, Seattle, 2011.
14. Extensible Messaging and Presence Protocol (XMPP): Core.
15. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence.

16. Vladimir Korkhov, Dagmar Krefting, Tamas Kukla, Gabor Terstyanszky, Matthan Caan, and Silvia Olabarriaga. Exploring workflow interoperability tools for neuroimaging data analysis. In *Proceedings of the 6th Workshop on Workflows in Support of Large-Scale Science at Supercomputing*, Seattle, 2011.
17. Kassian Plankensteiner, Johan Montagnat, and Radu Prodan. IWIR: a language enabling portability across grid workflow systems. In *Proceedings of the 6th Workshop on Workflows in Support of Large-Scale Science at Supercomputing*, Seattle, 2011.
18. The Workflow Management Coalition. `http://www.wfmc.org/`.
19. Andrew Harrison and Ian Taylor. Web enabling desktop workflow applications. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, 2009.
20. The EU Wf4Ever Project. `http://www.wf4ever-project.org/`.
21. Sean Bechhofer, John Ainsworth, Jitenkumar Bhagat, Iain Buchan, Phillip Couch, Don Cruickshank, Mark Delderfield, Ian Dunlop, Matthew Gamble, Carole Goble, Danius Michaelides, Paolo Missier, Stuart Owen, David Newman, David De Roure, and Shoaib Sufi. Why linked data is not enough for scientists. In *Sixth IEEE e-Science conference (e-Science 2010)*, August 2010.
22. David De Roure, Carole Goble, and Robert Stevens. Designing the myexperiment virtual research environment for the social sharing of workflows. In *Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pages 603–610, Washington, DC, USA, 2007. IEEE Computer Society.
23. ST. Peltier, AW. Lin, D. Lee, S. Mock, S. Lamont, T. Molina, M. Wong, ME. Martone, and MH. Ellisman. The Telescience Portal for Advanced Tomography Applications. *Journal of Parallel and Distributed Applications*, 63(5):539–550, 2003.
24. Paul Groth, Ewa Deelman, Gideon Juve, Gaurang Mehta, and Bruce Berriman. Pipeline-centric provenance model. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, pages 4:1–4:8, New York, NY, USA, 2009. ACM.
25. Attila Kertész, Gergely Sipos, and Péter Kacsuk. Brokering Multi-grid Workflows in the P-GRADE Portal. In *Euro-Par 2006: Parallel Processing*, volume 4375, pages 138–149. Springer, Berlin, 2007.
26. Zhiming Zhao, Suresh Booms, Adam Belloum, Cees de Laat, and Bob Hertzberger. Vle-wfbus: a scientific workflow bus for multi e-science domains. In *Proceedings of the 2nd IEEE International conference on e-Science and Grid computing*, pages 11–19, Amsterdam, the Netherlands, December 4- December 6 2006. IEEE Computer Society Press.
27. Alqaoud Ahmed, Taylor Ian, and Jones Andrew. Scientific workflow interoperability framework. *International Journal of Business Process Integration and Management*, Volume 5(Number 1):93 – 105, 2010.
28. Ian Taylor, Eddie Al-Shakarchi, and Stephen David Beck. Distributed Audio Retrieval using Triana (DART). In *International Computer Music Conference (ICMC) 2006, November 6-11, at Tulane University, USA.*, pages 716–722, 2006.
29. Dan Gunter, Ewa Deelman, Taghrid Samak, Christopher X. Brooks, Monte Goode, Gideon Juve, Gaurang Mehta, Priscilla Moraes, Fabio Silva, D. Martin Swany, and Karan Vahi. Online workflow management and performance analysis with stampede. In *CNSM*, pages 1–10. IEEE, 2011.
30. Taghrid Samak, Dan Gunter, Monte Goode, Ewa Deelman, Gideon Juve, Gaurang Mehta, Fabio Silva, and Karan Vahi. Online fault and anomaly detection for large-scale scientific workflows. In Parimala Thulasiraman, Laurence Tianruo Yang, Qiwen Pan, Xingang Liu, Yaw-Chung Chen, Yo-Ping Huang, Lin huang Chang, Che-Lun Hung, Che-Rung Lee, Justin Y. Shi, and Ying Zhang, editors, *HPCC*, pages 373–381. IEEE, 2011.
31. Adrian Casajus, Ricardo Graciani, Stuart Paterson, Andrei Tsaregorodtsev, and the Lhcb Dirac Team. DIRAC Pilot Framework and the DIRAC Workload Management System. *J. Phys. Conf. Ser.*, 219(1–6), 2010.
32. Rafael Ferreira da Silva, Sorina Camarasu-Pop, Baptiste Grenier, Vanessa Hamar, David Manset, Johan Montagnat, Jérôme Revillard, Javier Rojas Balderrama, Andrei Tsaregorodtsev, and Tristan Glatard. Multi–infrastructure Workflow Execution for Medical Simulation in the Virtual Imaging Platform. In *HealthGrid 2011*, Bristol, UK, June 2011.