Consistency and Collaboration for Fine-Grained Scientific Workflow Development: The dispel4py Information Registry[*]

Iraklis A. Klampanos[†], Paul Martin and Malcolm P. Atkinson

August 6, 2019

Abstract

This paper reports experience designing technology to support large-scale distributed computations that arise in scientific research as well as in other modern contexts. The challenge is to support data-intensive work across multiple autonomous sites, where experimentation and collaborative development are simultaneously encouraged across the same computing infrastructure. Focusing on fine-grained streaming workflows for data-intensive tasks, and in particular on requirements arising through the use of dispel4py within the eScience context, we specify appropriate registry modules and their interactions with other core components, designed to achieve the aforementioned goals. We then discuss the design and usage of a prototype information registry designed to support Dispel and dispel4py workflows. Finally, we demonstrate our method's suitability through a seismic ambient-noise cross-correlation example, drawn from the field of seismology.

[*]This report was completed as part of the FP7 project *VERCE* (`http://www.verce.eu`), project no.: 283543, when all authors were working at the School of Informatics of the University of Edinburgh, in 2015.

[†]Corresponding author. Email: iaklampanos@gmail.com

Contents

1 Introduction **3**

2 Rationale and Related Work:
Distributed, Data-Intensive Workflows **4**
2.1 Workflows 4
2.2 Scientific Experimental Procedure 6

3 Registry Requirements **7**
3.1 Distribution 8
3.2 Registrable Programming and Scientific Components 8
3.2.1 Workflow Components 9
3.2.2 Scientific and Domain-Specific Objects 10
3.3 Identification and Versioning 11
3.4 Dynamic, Flexible Enactment and Optimisation 11

4 Registry Design and Structure **12**
4.1 Packages 13
4.2 Workspaces 13
4.3 Logical Schema 16
4.4 Current Implementation and Usage 18

5 Application Scenario: Seismic Ambient Noise Cross-Correlation **18**

6 Conclusions and Future Work **20**

1 Introduction

Data-intensive computation in increasingly useful in a number of contexts, such as in science, engineering, medicine, government, etc. [1]. We use the term "data-intensive" to characterise computation that either requires or generates large volumes of data, or has complex data access patterns due to algorithmic or infrastructural reasons. In most of these types of computation, research proceeds by bringing all of the required data into one administrative context before they can be further analysed and explored by teams that included domain scientists, data-analysis specialists as well as computer scientists. Similar arrangements are typical in many other contexts, such as those reported in [2, 3, 4].

We argue that there are cases where this collection of data and software, under a single administrative regime, is not the optimum solution. First, the owners of the data are spread across organisations and they may wish to restrict access to certain data sets while also collaborating on methods with colleagues elsewhere. Second, the data or software is rapidly changing at some sites, so that co-location reduces timeliness. Last, the cost or time involved in extracting, transforming, moving and organising the data into a single location can be too great.

We posit that scientific workflow platforms provide the necessary functionality for orchestrating the actions pertaining to the above data-intensive activities, while providing researchers with an abstraction that makes the inherent growth of, especially distributed, software libraries and data more manageable. Such platforms should enable researchers to browse, extend and create new computational components, organise remote data resources and stores in a way relevant to the work at hand and allow them to compose or edit workflows deployable to the distributed infrastructure. Given sufficient support mechanisms, the platform should be able to facilitate different degrees of supervision during the execution of a workflow, as well as test and production enactment.

In this paper we study the workflow information registry, as a logical component of such a platform, and in particular dispel4py[1] [5], on the basis of a functional reference implementation[2]. We focus on eScience, due to its inherent resource and administration heterogeneity, as well as due to its, often, collaborative nature. The registry stores and is able to produce information regarding workflow and other research components. It is typically interrogated by user interfaces (UIs) to aid the researcher, and it may provide computational component specifications and metadata to the execution or enactment engine for optimisation. Furthermore, such registry should be aware of execution modes (test, development, etc.) and states (incomplete, failed, complete, etc.), being in communication with the provenance system and logs. It follows that consistency is achieved by the registry behaving as a source of reference when code is deployed. Efficiency is achieved by fragmenting the workflow to run elements at the locations near their data, by mapping to specialised implementations, and by inserting transformations when necessary into the data streams coupling the

[1]`http://dispel4py.org`
[2]`https://github.com/iaklampanos/dj-vercereg`

distributed elements, all of which require consultation with the registry. An earlier implementation of this model can be found in [6] and the use of the registry for optimisation is reported in [7].

This paper is organised as follows. The next section introduces related work and scientific research procedures. Section 3 presents requirements that would need to be met by the information registry for use within a scientific setting. Section 4 introduces the design and logical structure of the information registry component as well as its interactions with related components of the eScience platform. Section 5 provides a use-case drawn from seismology and how the registry can be used to aid optimisation and orchestration. Last, in Section 6 we provide concluding remarks and pointers for future work.

2 Rationale and Related Work: Distributed, Data-Intensive Workflows

Experiments conducted by researchers can no longer be considered as isolated operations performed at one locale within a modest time frame. Instead, experiments may require the orchestration of resources over a prolonged period, with a gradual accumulation of results data throughout. In addition, researchers work in collaborative environments, where they are increasingly expected to exchange methods (formalised as software), results and analyses, often implicitly or explicitly requiring the movement of large volumes of data.

This scenario calls for the flexible deployment of persistent computational services across many contexts, and the establishment of high-throughput data channels between them. These services must command substantial local resources and be able to re-configure themselves on demand. Where necessary, services have to be wound down to release scarce resources and re-deployed closer to active data sources. This is not a scenario which permits manual coordination on large scales; certainly not on the part of a user constituency which is ostensibly concerned with science over software engineering and resource management.

2.1 Workflows

In the context of persistent services, a workflow can be seen as a, possibly indefinite, configuration of data-intensive machinery towards a specific purpose. Connections are established between service instances which re-gear themselves towards providing the processing elements prescribed by the workflow. Data flows into processing elements via such channels and flows out to elements further down the workflow. Flow control is handled by the enactment services themselves, in accordance with agreed protocols.

Architectural overview There must exist a common interface and behavioural specification for data-intensive machinery (persistent services) which can be deployed on distributed resources, forming a distributed enactment platform. There must also exist a choreography service to broker the choreography of persistent services; control over the workflow must be distributed between the choreography service and the enactment services in such a manner as to allow easy monitoring of progress while conferring onto the enactment services the autonomy they need to conduct their part of the workflow in the most independent and flexible way possible. It is also clear that there must be a standard way to specify workflows to be imposed on the enactment platform. This suggests the need for a language in which to describe the processing elements used in a workflow and the connections between them, specifying the data-types involved and any desired characteristics of the overall workflow, which might affect how it should be deployed and choreographed onto available resources.

The Dispel Workflow Specification Language The role of a workflow language is to provide a standard *lingua franca* for describing the choreography of logical components collectively composing a workflow.

Since we view the enactment system as a collection of concurrent persistent computational services acting in concert, our view of workflows is data-oriented rather than control-oriented, as defined in [8]. The deployment of processing elements onto enactment resources is taken as given, the task being delegated to the choreography service. Data elements are streamed as and when ready from one processing element to another, allowing components of a given workflow to operate in parallel as long as all inputs can be provided. The buffering and replication of data is delegated to the individual enactment services, allowing connections to be established directly between co-dependent elements without the need for buffers and filters to be defined explicitly.

Dispel is a workflow language introduced by the ADMIRE project[3]. A Dispel script essentially describes a high-level process to construct a workflow rather than a specific workflow instance. This allows the use of imperative constructs, such as iteration, selection and sub-procedures to be employed in order to describe arbitrarily complex, parametrisable workflows. Dispel's imperative nature also allows the construction of composite processing elements from the composition of existing workflow elements without resort to external configuration or registration.

Upon interpretation (generally by submission to a gateway) a workflow is produced. A Dispel workflow is an abstract network of processing elements between which data can be streamed. More specifically, a *processing element (PE)* is an operator with a number of connection interfaces through which data is either consumed or output, while a *connection* streams data from one output interface to at least one input interface.

Dispel relies on the concept that a standard library of abstract processing elements can be mapped into a variety of different execution contexts (*e.g.* OGSA-DAI[9]), provided that there exists a single logical specification of the

[3]Advanced Data Mining and Integration Research for Europe: `http://www.admire-project.eu`.

behaviour of each element for which there may exist many compliant implementations. This separation of workflow language from specific execution contexts distinguishes Dispel from many similar languages such as SCUFL[10], MoML[11] or ZigZag[12].

A Dispel script is typically used to either describe a logical workflow for submission, or to define and make available new workflow components. It can, therefore, be seen as providing a means of interaction between users and the platform, through the platform's registry, the required properties and design of which are presented in this paper. Its purpose is therefore diverse, as it can be used for deployment/enactment of a workflow or for, implicit or explicit, registration of workflow entities, with both actions requiring appropriate entity resolution at the registry level.

The dispel4py Python Library and Enactment Engine There have been application contexts where the Dispel language and its associated components, described above, such as required gateways and its dependency on OGSA-DAI, could not be used due to practical reasons. Some of these reasons included the difficulty to provide and maintain additional hardware resources to host the gateways, security considerations due to the operation of gateways and OGSA-DAI in scientific supercomputing centres, as well as a steeper learning curve than scientists and end-users would often be willing to overcome in order to put it to production use. In order to address these points, and to provide a more accessible ramp for scientists and researchers to adopt streaming data-intensive workflows, we designed dispel4py[4][5].

dispel4py is a new library and enactment engine which allows users, primarily scientists, to specify their data-intensive workflows in Python, therefore allowing for the use of existing and efficient scientific libraries, such as *scipy*, while their workflow specifications adhere to most Dispel rules. dispel4py allows for local execution of workflows for testing and debugging purposes, before users can execute them on large-scale distributed resources. The dispel4py engine also provides a number of *mappings* to different kinds of resources, currently: Apache Storm[5] clusters, shared-memory multi-CPU machines as well as to MPI clusters, therefore enabling the use of fine-grained, data-intensive, streaming workflows on production clusters effortlessly and with the possibility to execute the same workflow on a different target platforms with minimum, if at all, change.

This paper focuses on the design of a workflow component information registry designed for dispel4py.

2.2 Scientific Experimental Procedure

Replicability is essential for verifying and validating experiments, as well as to ensure trust in shared processes. Given the potentially disparate nature of

[4]`http://dispel4py.org`
[5]`https://storm.apache.org`

resources conscripted into the enactment platform, it may be that processes executable on different resources may require different underlying implementations. Even if this is the case however, there should be a standard logical description of any given processing element which accurately describes its behaviour in all valid contexts. This entails an external registry service dedicated to the maintenance and publication of those descriptions which can then be referred to in workflow specifications dispatched to the coordination service. A consequence of this is that if the logical operation of a component *does* change over time, then the conduct of experiments using that component will change; it must be possible therefore to both conduct a previous experiment under the new specifications (benefiting from any improvements made) and conduct the experiment as it was before (so as to verify prior results).

Approaches such as *myExperiment*[6][13] and *wf4ever*[7][14] are instrumental in catering for workflow preservation and experiment replicability, providing users with rich UIs and social networking functionality to connect and collaborate. However, to our experience, the level of abstraction of such solutions is higher than many scientists would like to use on a day-to-day basis. While these solutions store metadata about the properties and lifecycle of workflows and associated research objects, they often also treat them as black boxes, leaving the execution details to the users.

Another approach targeted at coarse-grained scientific workflows is taken by *gUSE*[8][15]. gUse provides tools for managing generic workflows and associated metadata, as well as executing them on user-specified target contexts. Similar to myExperiment and wf4ever, gUSE views workflow components as blackboxes, with each component being a complete program to execute on a specific resource. Clearly, approaches such as the above aim to provide a different service to the scientific community than Dispel and dispel4py, and they therefore have different registration needs. dispel4py and its associated registry, discussed below, allow for the specification and sharing of workflows and workflow components on a finer-grain level and are complementary to approaches such as myExperiment and gUSE.

3 Registry Requirements

In order to address the requirements of global consistency, local agility and simplicity, posed by the distributed and large-scale nature of modern science, we build our architecture around three logical elements: a data-intensive workflow language – in the case of dispel4py, a Python library adhering to the semantics of the Dispel language, an enactment/execution layer and a registry of computational components. Together these elements form the core of our distributed architecture, with their combined semantics dictating patterns of control and data passing, orchestration as well as user-interaction. In this section we dis-

[6]http://www.myexperiment.org
[7]http://www.wf4ever-project.org
[8]http://guse.hu

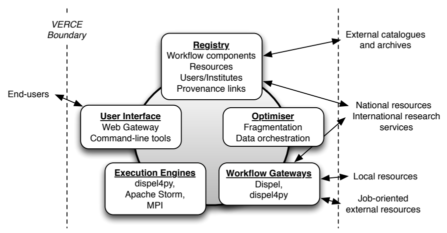


Figure 1: Components interacting with the workflow registry.

cuss the main requirements of the information registry, mainly in relation to the other two components, as a service that enables collaboration while ensuring consistency across the distributed infrastructure.

3.1 Distribution

The environment we study is inherently distributed, both in terms of computing as well as of human resources. The modern scientist will typically make use of a number of disparate computing resources, such as grids, high-performance (HPC) facilities, private institutional resources, Web or desktop applications, depending on the task at hand. Inevitably, the same applies to the location of relevant data, be it initial, intermediate or resulting data. The scientist is typically required to explicitly arrange data movement and processing as well as archiving and analysis of results. Furthermore, an increasingly important requirement in modern science is to support scientific collaboration. Through common projects or other means, scientists need to collaborate with colleagues working in different countries as productively as with their officemates. To collaborate means being able to share methods, datasets and results, replicate and extend other experiments, provoke discussions and generally be able to feedback newly acquired knowledge into the experimental process.

In order to automate part of this process, we need a solution, incorporating a registration framework, able to take into account the inherent heterogeneity and be consistently accessible from remote locations. Further, to support scientific collaboration, it would need to store some information about research organisations, people and scientific networks. Based on this information, such an eScience framework would be able to automate certain sections of the experimental process and facilitate collaboration.

3.2 Registrable Programming and Scientific Components

In a distributed scientific setting, the information registry will need to hold information useful and relevant to a number of entities, internal or external to the platform (Figure 1). As a facilitator for information and meta-data, the registry is interrogated by services, such as the enactment service and the user interface, in oder to optimise workflow creation, execution effectiveness, parallelisation, data movement and presentation based on contextual and domain-specific information. Furthermore, it is conceivable that designers and engineers may not have complete knowledge of the extent or the form/specification of the information that would eventually be required to be maintained within the registry. Therefore, the integration with and reuse of external scientific catalogues and related resources is required. In this work, apart from the core registrable entities, we assume that the registry allows for integration with arbitrary external data sources through some extensible mechanism (*e.g.* through the use of PIDs[9]), so as to provide continuity of objects of interest over time and location. However, such mechanisms, their provision and integration is not the focus of this study. Instead we concentrate on the core entities, which would be relevant to any eScience platform that aims to deliver a distributed and collaborative programming environment: (a) processing elements and other language (for the purposes of this work, Dispel or dispel4py[10]) components and (b) scientific (or research) objects:

3.2.1 Workflow Components

The language components that should be registered are those that would be required by the platform's local or remote enactment engines to realise and execute a workflow graph. The registry can also be seen as hosting and managing the consistent, distributed programming library, it should contain definitions and specifications of components that the domain experts and the data-intensive engineers should have at their disposal when creating a new workflow or when they are customising or modifying an existing one. In the case of dispel4py, such components include all nameable entities that may appear in a dispel4py workflow. These components are the following:

Type the core Dispel modelling unit. Types are most frequently used to define PEs and can be broadly divided into abstract and concrete. Abstract PE types only define available connection interfaces, while concrete types specify an internal topology of PEs and other components, realising the intended functionality. Even though PE types do not typically coincide with their enactable or executable counterparts, this difference is not crucial to the discussion of the information registry, hence we refer to both entities interchangeably.

Function (or constructor) a Dispel structure used for defining arbitrary Dispel topologies, wrapping them into PE types. Functions are evaluated and expanded into enactable workflow segments during run-time.

Literal is a named Dispel literal. Such literals may be scientific constants, data-store references, etc.

[9] *e.g.* via EPIC: `http://www.pidconsortium.eu`

[10] As the semantics of Dispel and dispel4py are by design as close as possible, the two terms will be used interchangeably for the remainder of this study.

Implementation an arbitrary block of programming code, usually implementing the functionality of a PE. Such implementations could range from scientific code written in Python to code performing job submission to HPCs, etc. In the case of dispel4py, implementations specify workflows, PEs, or reusable functions in Python.

Connection a typed data inlet or outlet, part of the definition of PEs.

Data Type a language-specific entity used for specifying the data types of connections of PEs. Connections may be designated as being input or output and are typically named per PE definition. Each connection may carry two types: a *structural* type, or *s-type* and a *domain* type, or *d-type*. S-types are language-dependent and in the case of dispel4py they can be 'str', 'int', or a user-defined class, etc. D-types depend on the application domain and can be of an arbitrary structure (*e.g.* semantic, graph-based descriptions, etc.). Potentially, the use of d-types can be used to extend the functionality of the registry by enriching the descriptions of the stored elements by, for instance, linking to externally defined metadata schemas.

Workflow a partial or complete workflow, expressed using combinations of the elements above, able to carry out a well-specified task when deployed. Aside from other language components, workflows may also be associated with arbitrary domain-specific objects (see Section 3.2.2 below).

3.2.2 Scientific and Domain-Specific Objects

The workflow-based platform should be able to register and keep track of relevant scientific objects in a form usable by components as diverse as the user interface and the enactment layer. Domain-specific objects of interest may significantly vary in form, function and size. However, with respect to their relationship to the scientific workflow, such objects can be broadly categorised as being:

External to the workflow are domain-specific objects which play no part in the enactment or execution of a workflow, but which have some significance to the experiment, user or other entity supported by the platform. Examples of such objects may include a relevant results data-set to be used for comparison or replication purposes, a copy of or reference to a published article to complement an experiment, etc.

Internal to the workflow are domain-specific objects and references to datasets, which are results of specific stages of a workflow. Such objects can be expressed in terms of language components, versions of which would also be present in the registry (Section 3.2.1), in combination with user-specified parameters and previous outcomes. The registry should keep track of scientific objects of interest by associating language components with provenance trace information, obtained through the provenance component of the platform.

3.3 Identification and Versioning

Essential to the operation of any registry or object store is a robust identification and versioning model. A registry component suitable for such distributed, collaborative workflow platform should provide for dynamic assignment of identifiers to stored digital entities. It should also provide for automatic versioning, where appropriate – i.e. when a user adds a new processing element to the distributed Dispel library, the registry should assign an ID to it, while it should also be able to keep track of future versions automatically. The automated approach to object identification and versioning is essential due to the complexity and potential scale of such a system.

As the registry of such system provides information to a number of other components, to keep IDs and versions consistent leads to two further requirements, namely that object IDs need to be consistent across remote and heterogeneous sites and computing resources as well as between past and future versions of the same container object. To illustrate this last point, consider the situation where a workflow has been registered, making use of some PE a, and that a is then modified by some researcher into a'. Whether $a \equiv a'$ should depend on the context of use of the workflow, and by implication, on the context of reference of the object. For provenance purposes, a should always signify a different object than a', however from an enactment perspective, if the same workflow were to be deployed again after the refinement of a into a', the latter version should probably be used by default.

To add collaboration into versioning would require that the overall identification and versioning model is able to cater for individual users and groups and that its default behaviour during object resolution takes into account current common practices. At the same time, traceability of the resolution process of individual objects and divergence from the default behaviour would also be desirable.

3.4 Dynamic, Flexible Enactment and Optimisation

Optimising the enactment and execution of scientific workflows is a non-trivial research task, which requires information and meta-data regarding the available execution contexts. The role of the registry in this regard can be of central importance. The workflow platform, through an appropriate registry sub-component, should be able to query metadata regarding associated computing resources. Such metadata may include descriptions of resources in terms of processing power, disk capacity, location, average workload, front-ends or access points, supported authentication mechanisms, subscription information of relevant research groups, etc. This metadata would be of a mostly static (or infrequently-changing) nature, with dynamic, real-time data being handled by a suitable provenance component. Some of this metadata could be stored and managed internally, while there should be interfacing to external metadata resources where appropriate. Irrespective of the location of these metadata or of the internal management mechanisms, the registry should provide the workflow platform with a comprehensive and consistent API, mapping the underlying information to project-specific requirements.

4 Registry Design and Structure

In order to address the diverse set of requirements of our system, we build on and extend the package metaphor, which is already present in the Dispel workflow specification language [6] as well as it is present in the Python language (and therefore in dispel4py). This metaphor defines a hierarchical system for organising and naming digital objects of interest. In traditional programming, packages are used as containers of source code units, which are somehow related. From the perspective of the registry, packages may contain either programs (written in Dispel or other languages) or other kinds of digital objects. The registry, therefore, associates each registrable entity with a package, denoted by a *de facto* dot-separated path.

Using packages to complement the Registry's identification system has a number of advantages: it is a human-readable system, which users are able to reference directly from either source code or GUIs, its hierarchical structure leads to intuitively unique fully-qualified names for the contained objects, and it is a system many users are familiar with, as it exists in other programming environments too (e.g. in Python, Java, C#, etc.).

Introducing Workspaces While the use of packages is a feature we wish to retain, for the reasons outlined above, it also has disadvantages when used for object identification in a large, heterogeneous programming environment. As briefly discussed in Section 3.3, the main drawback arises when users are allowed to modify registered entities, as such modifications would potentially lead to ambiguity when enacting a previously registered workflow (or an enactable subgraph of one).

Consider a concrete version of the example of Section 3.3: A workflow F makes use of the seismology-related PE `eu.verce.seismo.InstrumentCorrection`, which modifies a waveform to compensate for characteristics of the observing station. Suppose then that a scientist who makes use of F wishes to modify the implementation of this processing element so that uses instrument correction information available to her, locally. As we are dealing with a distributed programming platform, introducing this change under the specific PE, would cause the execution of F to behave differently before and after the modification, universally within the workflow platform. While this would be acceptable to the user who introduced the modification, it would be surprising to other users. The modified version of the PE, say `eu.verce.seismo.InstrumentCorrection`$'$, would now contain the locally desired functionality, while corresponding to the same PE signature as before.

From an execution perspective, when the platform is requested to enact and deploy F, during PE resolution[11], and unless a priority mechanisms has been introduced, the registry would not be able to resolve the PE in question deterministically, as there would be two versions under the same fully-qualified name (id) for same entity.

Based only on a traditional packaging system, this problem could be addressed by a combination of explicit version requests, alongside suitable default behaviours (*e.g.* "if a version has not been explicitly provided, resolve to the most recent version"). While this approach would take care of this problem from an enactment perspective, it would nonetheless make interacting with the platform a tedious task at the same time it would impede repeatability. Assuming that such platform would be used by a potentially large number of scientists with individual and research-group agendas, default behaviours within a global scope would be of little value. On the other hand, if we were to implement local-scope default behaviours, exchanging information and collaborating with other scientists, outside of one's immediate research environment, would not be encouraged. To disallow *ad hoc* modifications completely would also be an option – it would be a direct equivalent to the standard library of a traditional programming language, *i.e.* the open public cannot modify the library, unless the modifications are fed back to the developing community or company through official channels. This is an approach, however, that would be of little value to collaborating researchers with different, and frequently changing, requirements and it would incur a higher cost of maintenance to the platform. In order to avoid such problems, we choose to design for collaboration inside the registry itself, by providing native support for *workspaces*.

In the following two sections, we provide further discussion of the platform's building blocks which appear both in the registry as well as the workflow specification language, namely of packages, workspaces and their relationship.

4.1 Packages

Dispel and dispel4py use package mechanisms similar to those in the Java programming language. As Dispel provides a means to specify types, processing elements and other workflow components, i.e. the core elements of any workflow-based system, it is clear that the registry should also be able to reflect the language's structures, including packages. As the use of packages in programming is well understood, we don't expand further on Dispel packages in this paper.

4.2 Workspaces

A workspace represents a snapshot of the whole ecosystem of library and other (e.g. external, domain-specific, etc.) components, as these are created, refined or being put to use within a specific, user-defined context. Workspaces are

[11] In this work, we consider the resolution mechanism as being part of the registry component due to its central role within the architecture and the coupling of its function with entities described in the registry.

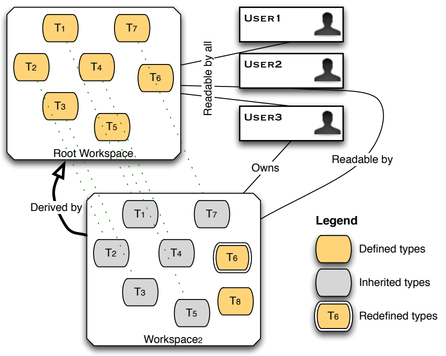


Figure 2: An example workspace hierarchy and its relationship with users. Instead of types we could have any named, and therefore registrable, Dispel element.

organised as a hierarchy, with each workspace being potentially an extension of another one. Each workspace contains new, modified or implicit pointers to packages of parent workspaces and, consequently, digital objects of interest defined within its parent workspace.

A workspace is typically associated with a user or a user group or with applications or experiments sharing similar goals. The exact use of a package is left to the user and to the characteristics and constraints of the application at hand. Each workspace represents a certain Dispel-based environment, which can be invoked by users and utilised for look up, resolution and execution of workflows by the enactment component.

Workspaces, as an abstraction, are orthogonal to packages, *i.e.* a workspace spans the packages of the types and other objects it contains.

On a conceptual level, each workflow component can be seen as bearing some semantic significance to the purpose or intention of the library. Often such semantics are encoded in package- and class-names, for instance, while in other cases additional information may also be employed, such as semantics-specific types (*e.g.* Dispel's d-types), relationships between components, text or other descriptions, etc. Let us denote a conceptual workflow component as $T \leftarrow \mathbb{T}$, where $\mathbb{T}$ is the set of all conceptual components defined in the distributed workflow environment.

Users of the registry should be able to reference and introduce conceptual components, and also modify and specify/implement them. Let us denote a specification of a component as t, and its association to a conceptual/abstract component T, as $t : T$. In our model, each workflow component specification, t, can only be associated with a single abstract component, T, within each workspace. Components associated with the same conceptual/abstract entity are said to be *siblings*, denoted by $\widetilde{t_i, t_j}^{T}$, where T is, optionally[12], the "parent"

[12]To specify the conceptual type of the siblings is sometimes optional, as each component can

abstract type. For the purposes of this work, no assumption is asserted on the structure siblings may be organised in – *i.e.* all siblings may be on the same level, or arranged hierarchically, or otherwise, under a parent type.

Workspace Definition A workspace then can be defined as a set of logical workflow components, such that no two components can share the same conceptual type in the same workspace:

$$W = \{t | \nexists t' \in W \ . \ \widetilde{t, t'}\}$$

In other words, conceptual types can be used as an in-workspace identifiers for logical workflow components. This modelling decision was taken to enforce a semantic relationship between components and IDs, therefore making the use of types easier. This decision does not incur any loss of generality, since new types and workspaces can be created as needed. Given that the scope of conceptual types is the whole of the distributed workflow system, it follows (and is indeed desirable) that there will exist siblings of the same conceptual types in different workspaces, and therefore, that $|\mathbb{T}_W \cap \mathbb{T}_{W'}| \geq 0$, where $\mathbb{T}_i$ denotes the set of conceptual types for which there exist specifications in workspace i, for any two arbitrary workspaces W and W'.

Cascading Workspaces Each workspace is designed to be an independent, yet parent-complete representation of the run-time ecosystem available to the scientist or the developer at any given time, derived either from a "standard library" of workflow components on offer, or by another user-defined workspace. This model of workspace definition leads to hierarchies of workspaces, much like package hierarchies, only orthogonal to them.

In terms of component inheritance, each new workspace will inherit all the components of its parent workspace *by-reference* (changes to the components in the parent workspace will be reflected on the children), while modifications to components will take place in a *by-value* fashion (*i.e.* modifications to components in the child will not be propagated up to the parent), as depicted in Figure 2. This, effectively, creates cascading hierarchies of workspaces, where member components get inherited, until modified.

Single Vs. Multiple Inheritance The cascading workspace model presented above can be materialised either by single or my multiple inheritance, with the latter being a more general case. In the case of multiple inheritance, as in traditional programming languages, there is the need for a precedence operator, denoted as $t_1 \succ t_2$, for two workflow component specifications, t_1 and t_2. Choosing between single and multiple inheritance would depend on the application at hand and on the way workspaces would be employed during the lifetime of a system. If multiple inheritance were to be adopted, the precedence operator,

have exactly one "parent", and so there can only be one parent under which two components can be siblings.

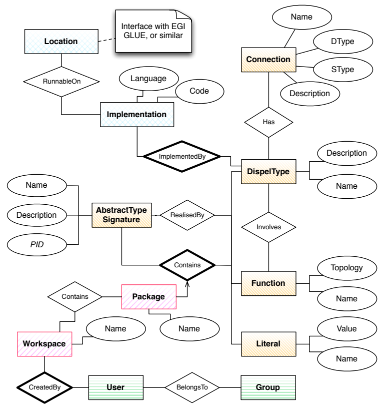


Figure 3: An Entities-Relationships (ER) conceptual model for the eScience registry, depicting the most important entities and attributes. Entities are colour- and pattern-coded according to the logical group relevant to their function, *e.g. Connection* and *DispelType* are part of the core Dispel registry, while *Workspace* is related to functionality pertaining to the organisation chosen within a specific application. The attributes presented are a subset of the actual attributes implemented, and many of them are instead realised as additional entities. Relationships depicted in bold are interfacing relationships between different logical parts of the registry.

and therefore type resolution, where not defined explicitly in Dispel, could take place over (1) the type placement in the Dispel script, (2) provenance information, such as recency of the specification of the clashing workflow component, (3) user-specified preference, etc. For the purposes of this study, we assume a single-inheritance model, leaving the exploration of alternative strategies for type resolution under multiple inheritance as future work.

Relationship to Packages Workflow components are rarely conceived in isolation, instead being grouped with similar components or with dependencies. Related components can therefore be expected to be packaged together when accessed or modified inside or moved between workspaces. In the case of multiple inheritance, briefly discussed above, precedence could be expressed at package rather than at component level.

4.3 Logical Schema

In order for an information registry component to be usable within a complex distributed system, it needs to be able to store data and interpret queries outside its core, workflow specification function. Figure 3 shows the conceptual model for such registry, focusing on Dispel-specific components. Here, four distinct registry areas are shown and colour-coded: the core language-related represented

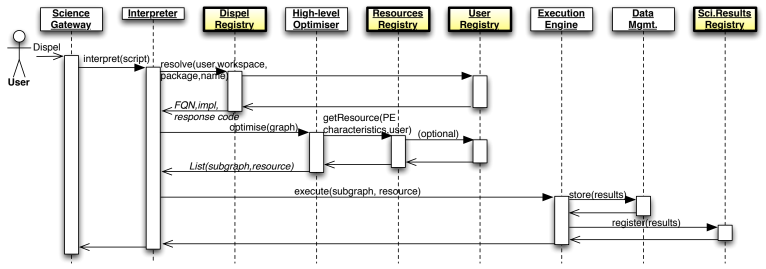


Figure 4: A sequence diagram depicting a hypothetical interaction between a number of components of a Dispel-powered eScience solution. Components which logically belong to the information registry appear with bold outline. Calls and responses are indicative of the intended functionality, and they should be implementable according to the registry schema of Figure 3.

by the entities *AbstractTypeSignature*, *DispelType*, *Function*, *Connection* and *Literal*; the implementation area, which pertains to specific implementations which are enactable at computing facilities with certain characteristics, and with example entities being *Location* and *Implementation*; the organisational area, which might implement different policies depending on the application at hand, and with representative entities *Package* and *Workspace*; and the user-management area, with representative entities *User* and *Group*.

For reasons of interoperability and sustainability, certain logical areas of the registry need to be made compatible with known and well-accepted standards, so as to take advantage of useful third-party catalogues and registries. For instance, the implementation logical area of the registry, which is designed to interact with the enactment and optimisation services of the workflow platform, could retain compatibility with schemas such as EGI GLUE[13], primarily in the European context, or with constituent Dublin Core[14] schemas for resource description. Similarly, the user-management area could be made compatible with relevant Dublin Core schemas, such as FOAF[15]. In maintaining interoperability and consistency between such workflow information registry and external catalogues, the registration of appropriate prefixes through initiatives such as the European Persistent Identifier Consortium – EPIC[16] would be very important. In terms of implementation, presented in the following section, the dispel4py information registry could be extended so that it facilitates the integration of dispel4py workflows in larger heterogeneous infrastructures.

[13]`http://go.egi.eu/glue2-standard`
[14]`http://dublincore.org`
[15]`http://xmlns.com/foaf/spec/`
[16]`http://www.pidconsortium.eu`

4.4 Current Implementation and Usage

A prototype version of the Information Registry has been developed for use with dispel4py, and has been released as an open-source project at `https://github.com/iaklampanos/dj-vercereg`. The current prototype is implemented in Django[17], a popular Python-based Web framework, and takes the form of a RESTful API. The registry backend is provided by a relational MySQL database server. The adoption of Django and MySQL, allows for fast prototyping – due to the level of integration of MySQL in Django, while achieving a performant solution, deployable on thoroughly tested and reliable software, such as the Apache Web Server. While the use of a relational database fulfils the basic registration requirements of dispel4py, we envisage extensions that make better use of the intrinsic semantic relations between workflow components, types, provenance information, etc., either as part of the core implementations or as extensions that interrogate external resources.

The current implementation allows for basic user and group management, which in turn allows for the creation and modification of workspaces. It also covers all core Dispel-specific entities, such as PEs, functions (in the case of dispel4py functions refer to python functions that may define a new workflow), literals and implementations. It does not yet contain resource descriptions for middleware and hardware, nor does it contain descriptions of data resources and products. Based on the requirements of our current work, we intend to look into these two extensions in the future, also taking into account provenance and optimisation considerations. Making a case for the potential usefulness of registries in the context of fine-grained, streaming workflows, the example of the next section assumes these two components exist.

Once the registry has been deployed, users can navigate dynamic, browsable documentation, under the path `/docs`, built on Django Swagger[18]. After deployment, and once the location of the registry and the default workspace have been specified, dispel4py users can invoke PEs and other Dispel entities inside their workflows. This is achieved by overriding the Python `import` keyword so that it queries the registry unless a local package with the same name has been found. We believe this last detail is of great importance, as it makes the use of the registry transparent to users when coding, allowing them to focus on the task at hand. More information on how to make use of the registry in dispel4py workflows can be found at `http://dispel4py.org`.

5 Application Scenario: Seismic Ambient Noise Cross-Correlation

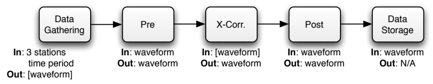
The Green's function of the medium between two seismographic stations can be deduced from the inter-correlation of the seismic noise recorded at these stations

[17]`https://www.djangoproject.com`

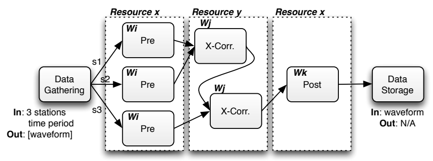
[18]`https://github.com/marcgibbons/django-rest-swagger`

[16] During recent years, rapid practical implementation of these innovative statistical methods, both in seismology and acoustics, have lead to breakthroughs in high-resolution imaging – in seismology and exploration geophysics – and acoustic communications [17].

The calculation of cross-correlation is performed on waveforms retrieved from arbitrary numbers of stations, on regional or global levels and over varying time periods. This entails gathering, managing and processing large volumes of data from a number of arbitrary sources, potentially annotating and storing byproducts, before cross-correlating the traces; therefore making this a data-intensive scientific task. Cross-correlation is a parallelisable procedure, which can be executed either on a local cluster or at an external facility. Post-processing, visualisation and interactive steering typically follow the cross-correlation calculation.



(a) Abstract workflow for ambient noise cross-correlation.



(b) Workflow for ambient noise cross-correlation after high-level optimisation.

Figure 5: Workflow stages for ambient noise cross-correlation before and after initial, high-level optimisation.

Focusing on the interactions of the Information Registry and the user as well as the other components, and without loss of generality, a cross-correlation experiment can be seen as having three broad phases: (1) pre-processing of multiple waveforms; (2) cross-correlation and (3) post-processing (which may include visualisation components). Before processing can commence, data needs to be gathered from relevant sources, while at the end of the workflows result data will typically need to be stored by some appropriate data-management service. In typical cases, intermediate results could correspond to hundreds of thousands of files. Let us assume that the abstract workflow consists of one pre-processing, one cross-correlation and one post-processing component, as in Figure 5(a).

The dispel4py registry contains information to enable the optimiser of the enacting component to extract appropriate PE implementations. Such information includes user-specified workspaces (explicit or derived), *e.g.* W_i, W_j, W_k of Figure 5(b). The parallelisation strategy may be derived by information such as the types of the input and output streams of PEs (*e.g.* whether they expect single or multiple input streams), by observable indicators such as output/input ratio as well as by other properties. In the future, by consulting the registry about resources, the optimiser would also be able to make informed decisions about which resource (*e.g.* x, y of Figure 5(b)) should be responsible for each workflow segment. Furthermore, the presence of such a registry component ensures that references to PEs and workflows, such as the ambient noise cross-correlation presented here, can be exchanged between users and still be resolvable and executable consistently throughout the eScience platform. By storing provenance and other metadata, such as workspace names of PEs involved, the choice of resources and the workflow segmentation, the system through the shared information registry, can support the controlled re-enactment of scientific workflows and the consistent replication of past experiments.

At this stage, the dispel4py library and engine does not include an optimiser to make such decisions; rather the users are required to specify which part of the workflow gets executed on which resource. As such, the information registry is currently used primarily for preserving consistency and encouraging sharing and collaboration between researchers. Furthermore, accessing intermediate results is semi-automatic, since, if the above is executed as a single workflow, PEs are aware of files previously created and can stream them into subsequent parts of the workflow with minimal user intervention. More details of the implementation of the above workflow in dispel4py can be found in [5] as well as in published material of the EU VERCE project[19].

The example above is provided in order to highlight the importance of fine-grained registries within the eScience context as well as their potential for intelligent workflow management and enactment. Furthermore, we believe that such registries are suited to aiding researchers not only in sharing their research methods, but also in collaborating on a more manageable and detailed level, which is closer to their everyday working needs.

6 Conclusions and Future Work

In this paper we presented the role of an information registry within the Dispel-powered eScience framework. Further, we provided a suitable design for such a registry component taking into account the distributed nature of the target system and of the way scientists conduct research as well as their collaboration needs. We demonstrated how a mechanism of workspaces can help retain consistency across what is essentially a distributed programming platform, while being meaningful to users. Our choice of using Python-based technologies both for the information registry and for dispel4py increases their sustainability as well as their prospects of adoption by relevant research communities. Furthermore, the tight and mostly transparent integration between the two technologies allow users to control additions and changes to their workflows with minimal deviation from their usual way of working.

[19]`http://verce.eu`

At the same time, the design of such a wide-ranging eScience workflow platform is far from complete. Pointers for future work include researching and specifying appropriate APIs which would allow interoperability with other eScience platforms and registries, the integration of external stores and catalogues in an extensible and domain-unaware fashion as well as, crucially, to kick-off an iterative design process with increasing numbers of members of multiple interested scientific communities.

References

[1] A. J. G. Hey, S. Tansley, and K. Tolle, *The Fourth Paradigm: Data-Intensive Scientific Discovery.* Microsoft Research, 2009.

[2] T. Segaran and J. Hammerbacher, *Beautiful Data: The Stories Behind Elegant Data Solutions.* O'Reilly, 2009.

[3] A. Shoshani and D. Rotem, *Scientific Data Management: Challenges, Technology and Deployment*, ser. Computational Science Series. Chapman and Hall/CRC, 2010.

[4] W. H. Dutton and P. W. Jeffreys, *World Wide Research: Reshaping the Sciences and Humanities.* MIT Press, 2010.

[5] R. Filguiera, I. Klampanos, A. Krause, M. David, A. Moreno, and M. Atkinson, "Dispel4py: A python framework for data-intensive scientific computing," in *Proceedings of the 2014 International Workshop on Data Intensive Scalable Computing Systems*, ser. DISCS '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 9–16. [Online]. Available: http://dx.doi.org/10.1109/DISCS.2014.12

[6] M. Atkinson, R. Baxter, P. Brezany, O. Corcho, M. Galea, J. van Hemert, M. Parsons, and D. Snelling, *THE DATA BONANZA: Improving Knowledge Discovery for Science, Engineering and Business*, A. Y. Z. (series), Ed. John Wiley & Sons Ltd., April 2013.

[7] C. S. Liew, "Optimisation of the enactment of fine-grained distributed data-intensive workflows," Ph.D. dissertation, School of Informatics, University of Edinburgh, 2012.

[8] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-Science: An overview of workflow system features and capabilities," *Future Generation Computer Systems*, vol. 25, no. 5, pp. 528–540, 2009.

[9] B. Dobrzelecki, A. Krause, A. Hume, A. Grant, M. Antonioletti, T. Alemu, M. P. Atkinson, M. Jackson, and E. Theocharopoulos, "Integrating Distributed Data Sources with OGSA-DAI DQP and Views," *Philisophical Transactions of the Royal Society A*, vol. 368, no. 1926, pp. 4133–4145, 2010.

[10] D. Hull, K. Wolstencroft, R. Stevens, C. A. Goble, M. R. Pocock, P. Li, and T. Oinn, "Taverna: a tool for building and running workflows of services." *Nucleic Acids Research*, vol. 34, pp. 729–732, 2006.

[11] E. A. Lee and S. Neuendorffer, "MoML – A Modeling Markup Language in XML – Version 0.4," University of California at Berkeley, Tech. Rep., March 2000.

[12] X. Llorá, B. Ács, L. S. Auvil, B. Capitanu, M. E. Welge, and D. E. Goldberg, "Meandre: Semantic-Driven Data-Intensive Flows in the Clouds," in *IEEE Fourth International Conference on eScience*. IEEE Press, 2008, pp. 238–245.

[13] D. De Roure, C. Goble, and R. Stevens, "The Design and Realisation of the myExperiment Virtual Research Environment for Social Sharing of Workflows," *Future Generation Computer Systems*, vol. 25, pp. 561–567, 2009. [Online]. Available: doi:10.1016/j.future.2008.06.010

[14] K. Belhajjame, J. Zhao, D. Garijo, K. M. Hettne, R. Palma, Ó. Corcho, J. M. Gómez-Pérez, S. Bechhofer, G. Klyne, and C. A. Goble, "The research object suite of ontologies: Sharing and exchanging research data and methods on the open web," *CoRR*, vol. abs/1401.4307, 2014. [Online]. Available: http://arxiv.org/abs/1401.4307

[15] Á. Balasko, Z. Farkas, and P. Kacsuk, "Building science gateway by utilizing the generic ws-pgrade/guse workflow system," *Computer Science*, vol. 14, p. 307, Jan-01-2013 2013. [Online]. Available: http://journals.agh.edu.pl/csci/article/view/284

[16] M. Campillo and A. Paul, "Long-range correlations in the diffuse seismic coda," *Science*, vol. 299, no. 5606, 2003.

[17] E. Galetti and A. Curtis, "Generalised receiver functions and seismic interferometry," *Tectonophysics*, vol. 532–535, no. 0, pp. 1 – 26, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0040195111005026