Imperial College London

Standard Description of Big Data Neuroimaging Experiments

Author:
Claudia Mihai

Supervisor:
Dr. Jonathan Passerat-Palmbach

Second Marker:
Dr. Bernhard Kainz

June 26, 2016

Submitted in part fulfillment of the requirements for the degree of
Bachelor of Engineering in Computing of Imperial College London

Abstract

Since its beginnings, science has been defined by experiments and observations. As a direct consequence of the increase in data, scientists can no longer rely on batch processing and scripting systems alone. A solution to this problem has emerged in the form of workflow systems, which facilitate reproducibility and verifiability of experiments.

We first review some of the outstanding workflow platforms out there, highlighting their advantages and disadvantages. We continue with a critical side-by-side feature analysis based on the OpenMOLE DSL and the Common Workflow Language developing standard. Upon this investigation, a modularised system is designed to bridge the gap between the two technologies. The proposed system consists of two key components: a parser for SALAD, a YAML-based language with dual grammar; and a code generator for the OpenMOLE DSL. A basic implementation is offered, acting as a proof of concept for our research.

Offering a qualitative evaluation of the project's outcome, we also identify the main sources of incompatibility between the OpenMOLE DSL and the Common Workflow Language standard, presenting a detailed roadmap of future work.

Acknowledgements

First of all, I would like to thank my project supervisor, Dr. Jonathan Passerat-Palmbach, for contagious enthusiasm, unwavering support, and continued guidance throughout the duration of the project.

I am also beyond thankful to my friends and family for listening to all of the impromptu presentations, even while I was still figuring out things myself.

To Prof. Emanuela Cerchez, for single-handedly inspiring generations.

Contents

Chapter 1

Introduction

1.1 Motivation

Since its very beginnings, science has been defined by experiments and observations. With the advancement of technology, a plethora of data has become readily available for scientists to extract and analyse. This phenomenon has given rise to new issues in the various fields of science. More specifically, the novel problems are largely caused by the absolute necessity of reproducible experiments and verifiable results, and the difficulty of guaranteeing these with the increase of data.

Technological progress has helped automate tasks and has provided certainty as well as predictability throughout history. It is then no wonder that when faced with the abovementioned issues, humankind has once again turned to technology. While batch processing and scripting systems have provided great help to scientists, more recent solutions involve workflows and workflow systems [1].

The necessity of workflows has been brought on by the growing level of complexity in scientific experiments and applications. In order to provide a long-term solution to this problem, workflows present a high-level view of the experiment, allowing users to focus on the scientific protocol behind it. Thus, workflows offer details on what analysis is to be completed, but not on exactly how this will happen. Besides a description of the analysis, they also serve as permanent records of the experiments; thus, the experiments, or even only intermediate steps, can easily be re-run and verified within the community.

Workflows further allow a closer collaboration between scientists worldwide through the introduction of workflow-sharing environments, such as myExperiment [2]. This potentially increases both the quality and the rate of data analysis [3], and facilitates the creation of a pool of scientific methods, permitting the avoidance of unnecessary reinvention.

There has been a particularly interesting development of workflow systems targeting the field of bioinformatics and neuroimaging. Many such platforms have been built in the past few years, such as Taverna [4], Galaxy [5–7], Biopipe [8], BioWBI [9], Wildfire [10], KDE Biosciences [11], Biowep [12], BioWMS [13], and OpenMOLE [14].

While each of the existing platforms has various capabilities along with advantages and disadvantages, an analysis of their influence in the research environment reveals progress acceleration within the targeted field. Furthermore, these systems have even facilitated new levels of research in neuroimaging analysis [15], introducing methods such as data mining and meta-analysis.

1.2 Contributions

Given that the introduction of workflow systems into the field of bioinformatics is relatively recent, it is natural that there is still a lot of research and development to be pursued in this area. Therefore, the first goal of the present paper is a brief analysis of some of the existing scientific workflow platforms, with a particular focus on their languages and engines.

Upon the completion of this initial research, an in-depth analysis and concept matching based on the chosen workflow engine (OpenMOLE) and workflow language specification (Common Workflow Language) are presented in Chapter 3.

This is followed by the design, implementation and evaluation of two modules, detailed in Chapters 4, 5 and 6. The first module is a parser to be used with any given workflow following the Common Workflow Language specification. The parser creates an abstract syntax tree (AST) from the workflow, which is then passed to the translation module. The translator transforms the AST into Scala code that the OpenMOLE platform understands and executes.

From a higher level perspective, the aim is to provide an abstract intuitive interface for scientists. This would allow them to fully focus on their analyses rather than tangled code. Partial functionality has already been implemented within OpenMOLE for this purpose. Thus, the platform abstracts away details of running tasks on various High Performance Computing (HPC) platforms.

Therefore, the goal of this paper is to further extend the functionality and capabilities of the OpenMOLE workflow engine. Integrating it with a developing industry standard aims to contribute to the facilitation of the reproducibility of experiments, the verifiability of results, and the extensive analysis of data.

In doing so, the present paper sets out to contribute to a standard description of neuroimaging experiments, and aid in opening new paths in the analysis of data within neuroinformatics.

Chapter 2

Background Reading

2.1 Workflows

According to the Oxford English Dictionary[1], a workflow is “the sequence of industrial, administrative, or other processes through which a piece of work passes from initiation to completion; the passage of a piece of work through this sequence.”

Here we focus on scientific workflows. Following the definition above, this suggests that for our purposes a workflow is a sequence of scientific processes through which a piece of work goes from initiation to completion. In medical imaging, for example, workflows are also denoted as *pipelines*, so the two terms may be used interchangeably.

For example, the presence of a non-linear threshold or critical parameter is common in complex systems. To illustrate this, we take a NetLogo [16] model simulating the spread of a fire through a forest, which we summarise in Figure 2.1 [17]. In order to explore this model, we would need to run it on a wide range of possible density values. Encapsulating the model into a workflow allows us to more easily explore the parameter space, repeatedly run the experiment and log results, all with minimal effort[2].

To further expand on scientific workflows, we take a generic scientist named Alice as an example. She may have to run an experiment on various data sets. First, Alice has to ensure that she is running the same experiment on all of the different data. Then, she would also like to keep track of the data provenance - where the data has been collected from. For example, this may help her identify additional correlations between the origin of the data and the results. Alice would also need a clear view of the steps of the experiment, and the ability to pass some steps’ output as input to other steps. It is likely that she would also need high computing power; therefore, she would like to delegate the computing workload to the European Grid Infrastructure[3] (EGI).

We immediately notice the sheer number of factors Alice needs to keep track of throughout her work. This is in addition to actually analysing the data and results she will have found. Besides her wide range of skills in her scientific field of choice, she also requires technical knowledge of task delegation on remote environments. It goes without saying that her work may be made a lot easier if there was some way to automatically keep track of all the necessary information.

This is why we introduce workflows. Within a workflow document, Alice can define her

[1]http://www.oed.com/

[2]Running the model on the OpenMOLE platform takes 23 SLOC.

[3]http://www.egi.eu/

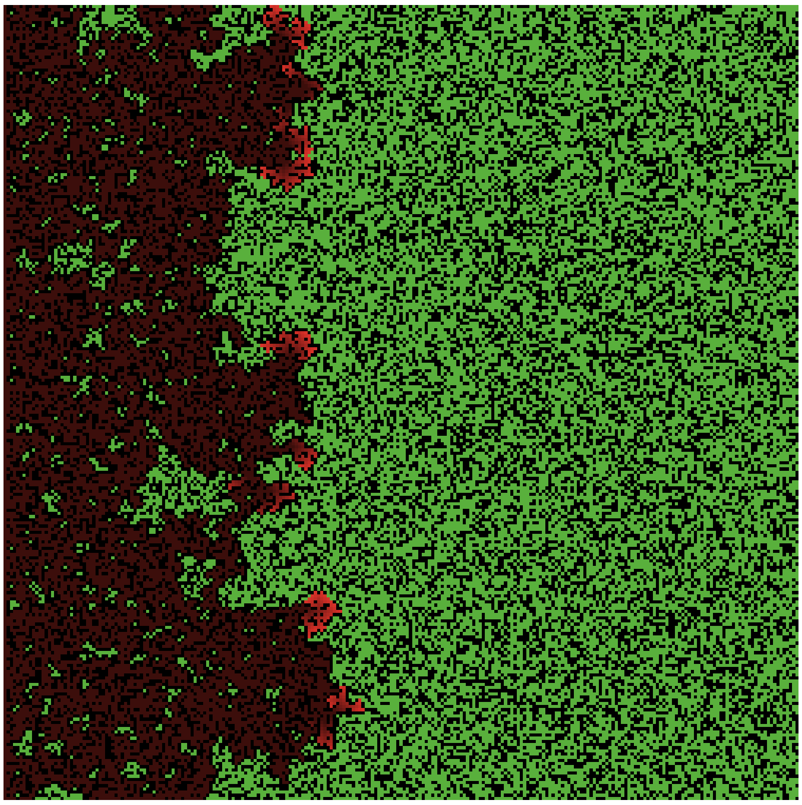


Figure 2.1: Sample Fire model, simulating the spread of a fire through a forest. Shows that the fire's chance of reaching the right edge of the forest depends critically on the density of trees.

input and output files. Then, she can define each task separately, and easily connect them to each other. Sending a task's output to multiple instances of the following task would be functionality implemented within the workflow platform. Furthermore, depending upon the workflow system she chooses to use, it may keep track of data provenance for her. It may also provide task delegation to remote environments as a service. Therefore, she would be able to distribute her experiment to the EGI with 1-3 lines and minimal effort, only providing the credentials necessary to access it.

We now introduce some of the workflow systems Alice may choose from.

2.2 Workflow Systems

Since the mid-nineties, many workflow systems have appeared in the scientific research environment. Some are developed for a specific research field, while others attempt to be generic solutions. However, they all have their own achievements and limitations. This section offers a brief overview of some of the main workflow systems, presenting both their capabilities and limitations.

2.2.1 General Structure

A workflow platform, or a workflow management system, is a piece of software that coordinates the operation of individual workflow components. Thus, it provides an infrastructure responsible for the setup, execution, and monitoring of a scientific workflow.

It generally consists of several main components, such as a workflow engine, a workflow specification language, a workflow editor, and a client program for job submission [18]. Additional modules can be integrated into the system, depending upon the developers' goals.

Several main workflow platforms are presented in the sections below.

2.2.2 Taverna

Taverna is a scientific workflow system consisting of several components: the Taverna Engine, powering Taverna Workbench as well as Taverna Server for remote execution; and the SCUFL workflow representation language [19].

It is built primarily for the field of bioinformatics, organising a wide range of remote web services into a collection usable by scientists worldwide. However, Taverna is domain-independent and has been used in a variety of fields, including Arts, Astronomy, Geoinformatics, Multimedia, and Social Sciences.

SCUFL (Simple Conceptual Unified Flow Language) allows workflow representation as Directed Acyclic Graphs. Workflow inputs are called sources, while workflow outputs are sinks. The basic execution units in SCUFL are processors. It presents two types of information flow: control flow (i.e., coordination links), which consists of tasks that may have an effect on the execution environment beyond I/O; and data flow (i.e., data links), for function composition. The user does not have to learn any new syntax; all of the workflow editing is graphical.

Finally, we note that Taverna explicitly allows for partial execution of a given workflow. The Taverna Workbench interface is shown in Figure 2.2, and constitutes a clear advantage of the platform. While it is a complex and interesting scientific workflow system, further analysis of the Taverna's components is beyond the scope of this project.

2.2.3 Galaxy Project

Galaxy is a web-based platform for data-intensive bioinformatics analysis. It aims to make computational biomedical research accessible, reproducible, and transparent. The Galaxy Project does so by providing a platform for running tools and workflows for computational analysis. Furthermore, it provides a tool for publishing and sharing web-based analyses and descriptions of experiments.

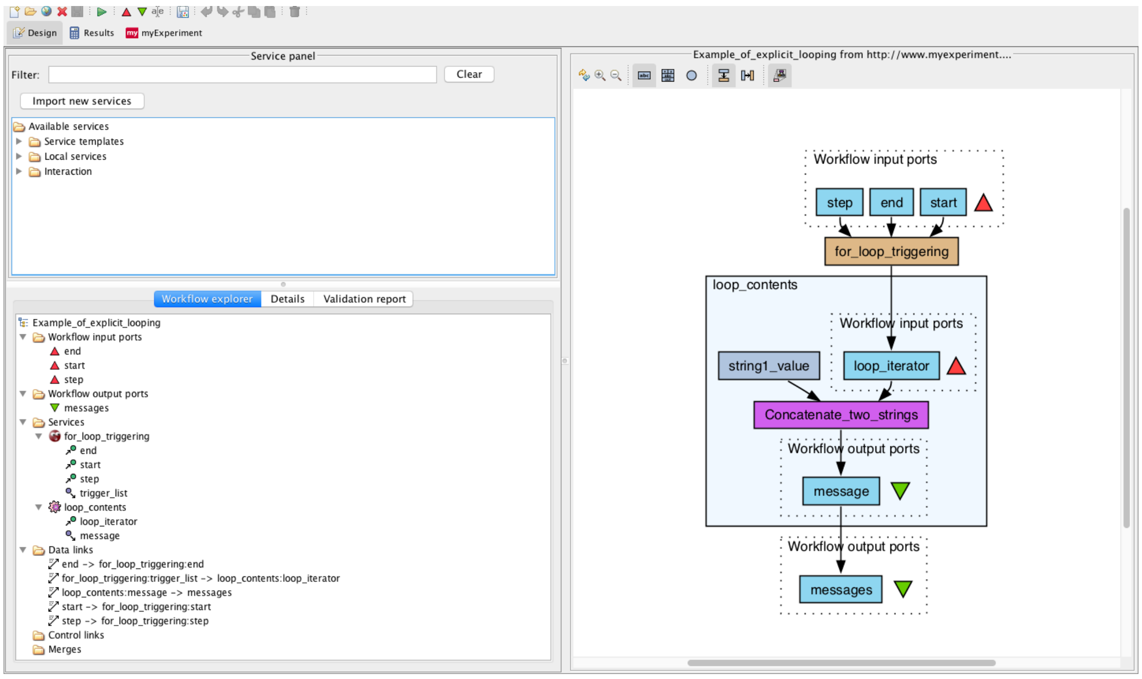


Figure 2.2: Taverna Workbench

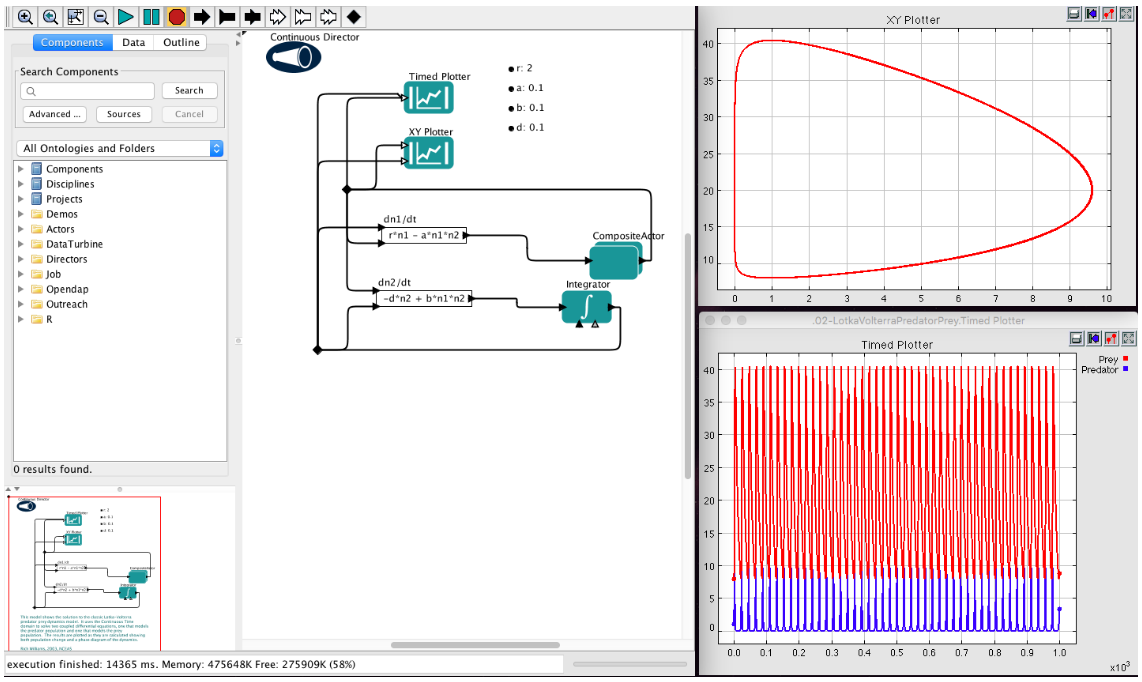
Rather than a workflow system, Galaxy can be described as a Reproducible Research System (RRS) [5]. Its strengths lie within the genomic analysis subfield of bioinformatics, providing a web-based interface that facilitates access to genomic data and various computational tools. Beyond access to data and tools, Galaxy also aims to provide solutions to the problems of reproducibility and transparency. Through its automatic generation of metadata for analysis steps, it ensures partial documentation, leaving a smaller gap to be filled in by the user's annotations. The problem of transparency is tackled by means of two different methods: the ability to share items such as datasets, workflows, and histories through Galaxy's sharing model and web-based framework; and Galaxy Pages, allowing users to extensively document every detail of their experiment. Moreover, Galaxy Pages also embeds items from the experiment, allowing easier reproduction of experiments by readers.

Overall, it is clear how the Galaxy Project extensively tackles the issues in bioinformatics. However, it is also largely limited to the subfield of genomic analysis. Finally, there are efforts to integrate the Galaxy Project with the Taverna workflow system, given the minimal overlap of their capabilities [21] and their complementary functionalities.

2.2.4 Kepler

Kepler is a scientific workflow system built upon the dataflow-oriented system, Ptolemy II [22]. Following its predecessor, Kepler presents the user with an intuitive graphical user interface (GUI) for workflow editing and execution [23], similarly to Taverna.

The Kepler Project is distinguishable through its actor-oriented approach. Actors represent operations, are connected through relations, and together constitute the workflow model. They are only aware of their own actions, inputs, and outputs. Meanwhile, the directors form the execution model or the Model of Computation, and are able to schedule actors. Depending upon the user's needs, they can choose from a set of directors, whether plugged in by themselves or pre-defined.



execution finished: 14365 ms. Memory: 475648K Free: 275909K (58%)

Figure 2.3: Kepler Project

The dataflow is represented through tokens that actors send to each other. Through the use of directors and limited abilities of actors, Kepler lacks a purely functional composition model. Therefore, it may be less applicable to analytical tasks involving dynamic scheduling decisions [24].

Although Kepler's limitations concerning a more dynamic scheduling process do restrain its use, the choice is justifiable by its attempt to focus on both data analysis and process simulations.

Figure 2.3 shows a workflow solving the classic Lotka-Volterra predator prey dynamics model. It describes the relative populations of a predator and its prey over time (top: in relation to each other; bottom: in time). All credits for the workflow go to Rich Williams of the National Center for Ecological Analysis and Synthesis [4].

2.2.5 BioWBI

This system actually consists of two tools: the Bioinformatic Workflow Builder Interface (BioWBI), and the Workflow Execution Engine (WEE). It is developed specifically for the field of bioinformatics, and constitutes a tool for building and executing related workflows [9]. Furthermore, it presents data integration as well as integration of specialised tools and algorithms within bioinformatics.

BioWBI is a web application providing an interface for creating workflows and sets of data. Meanwhile, WEE is the back-end application that receives requests from BioWBI, executes the analyses, and returns the results. The two components communicate with each other using Web Services Technologies.

Moreover, BioWBI is also a collaborative environments for bioinformaticians. This is supported by the platform users' abilities to share tools and workflows with each other, thus contributing to a growing workflow library.

[4]NCEAS: https://www.nceas.ucsb.edu/

BioWBI is the most specialised workflow system we found. It is also the only open-source system we found to have been developed within the industry[5] rather than solely in academia.

Finally, it should be noted that while the published research and ideas behind the development of BioWBI and WEE stand, the system itself seems to have been removed.

2.2.6 Conclusions

Following the brief presentation of several workflow management systems, we try to identify similarities and differences, advantages and disadvantages.

First of all, we notice that each system's development started from a single field, such as bioinformatics. However, while BioWBI and Galaxy remained limited to their initial scope, Kepler and Taverna show increased scalability, having been used in a wide range of domains.

Secondly, each system presents its own strengths and weaknesses. For example, Kepler appears successful in its aim to tackle both data analysis and process simulations, but its use in analytical tasks is limited by the lack of functional composition. Meanwhile, Taverna and Galaxy differ so greatly in their capabilities that they complement each other with minimal overlap.

Finally, Taverna, Galaxy, and Kepler present the user with a GUI for workflow editing. However, they lack the ability to edit text-based workflows. On the other hand, BioWBI does not mention support for graphical edits on a workflow or visualisation, but specifies its own syntax for workflow design.

[5]Older citations of the paper mention that BioWBI and WEE were developed within the Life Sciences Practice Team for IBM Business Consulting Services.

2.3 Project Components

2.3.1 Common Workflow Language

The CWL (Common Workflow Language) is a vendor-neutral and portable standard for workflow representation [25]. It is a collaborative open source specification, and the Common Workflow Language group mainly consists of researchers in biology and medicine. Designed for data-intensive science, the CWL specification defines a data and execution model permitting implementation over a wide range of computing platforms. Motivated by the necessity of portable data analysis workflows, specifications such as the CWL aim to enable data scientists to describe powerful, portable, and reproducible workflows and analysis tools.

We present some of the main concepts used in the CWL specification.

Data Concepts

First, we define some data notions used throughout the CWL specification.

1. Object: equivalent to the "object" type in JSON. Consists of an unordered set of *fields*; a field is a `(name, value)` pair.

   Note: `name` must be a string; `value` may be a string, number, boolean, array, or object.

2. Document: a file that contains either an *object*, or an *array of objects.*
3. Process: the basic unit of computation. It produces output data by performing computation on some accepted input data.
4. I/O objects: an input object describes the inputs to a process invocation; similarly, an output object describes the output of a process invocation.
5. Metadata: information not used directly in the computation, such as any additional information about workflows, tools, or inputs. Resembles comments in most programming languages.

Execution Concepts

Following the data model definitions, we need to go through some execution notions next.

1. Parameter: a named symbolic input or output of a process. When assigned values, parameters *construct the input or output object* used for concrete process invocation.
2. Process: in an execution perspective, a process is an abstraction acting as the foundation concept for *Command Line Tools* (CLT) and *Workflows.*

   2.1. CLT: a process consisting of the execution of a *standalone, non-interactive program.*

   2.2. Workflow: a multi-step process. Its steps are connected, forming a *directed acyclic graph.* The step outputs are connected to downstream step inputs in a cascading fashion. Independent steps within a workflow may run concurrently.

3. Runtime environment: the *hardware and software environment* executing a Command Line Tool. It includes the hardware architecture and resources, the operating system, software runtime as well as various libraries, modules, packages, utilities, or data files required to run the CLT.

4. Workflow platform: a *specific hardware and software implementation* of the CWL specification. It is able to interpret a CWL document, and run the processes that this specifies. In doing so, it should be capable of completing tasks such as scheduling and performing process invocation, setting up the runtime environment, and dealing with input availability as well as output collection.

Hints and requirements

Any CWL document may present certain hints and/or requirements.

A *process requirement* is a *compulsory* condition imposed upon the semantics or runtime environment of a process. An example is `CreateFileRequirement`, which requests the creation of a file in the output directory before executing a CLT. Failure to meet a process requirement results into a *fatal error*, and process execution should not be attempted.

Similarly, a *process hint* is an *optional* condition. Failure to meet a hint results into an *appropriate warning*, but process execution may carry on.

Workflow details

We now provide some additional information related to the Workflow class of CWL documents. As stated earlier, a workflow describes a set of steps and dependencies between these processes. Independent processes may execute in any order, or even concurrently. When all of its steps have finished execution, a workflow is *complete*.

The CWL specification supports *nested workflows*; i.e., a workflow may be specified as another workflow's step.

In order to make workflow use more efficient, the CWL specification further introduces two complementary concepts: *scatter* and *gather* (see Figures 2.4 and 2.5). A scatter operation indicates that the associated step should be executed separately over a list of input elements. Jobs within a scatter operation are independent. Its complementary feature is gather, which allows collection of output from multiple tasks (e.g., following a scatter operation) to a single task.

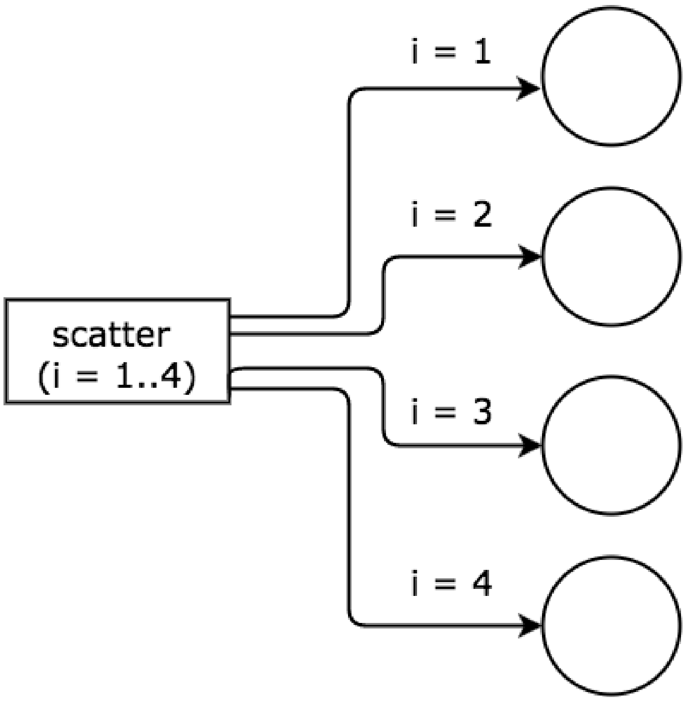


Figure 2.4: Scatter

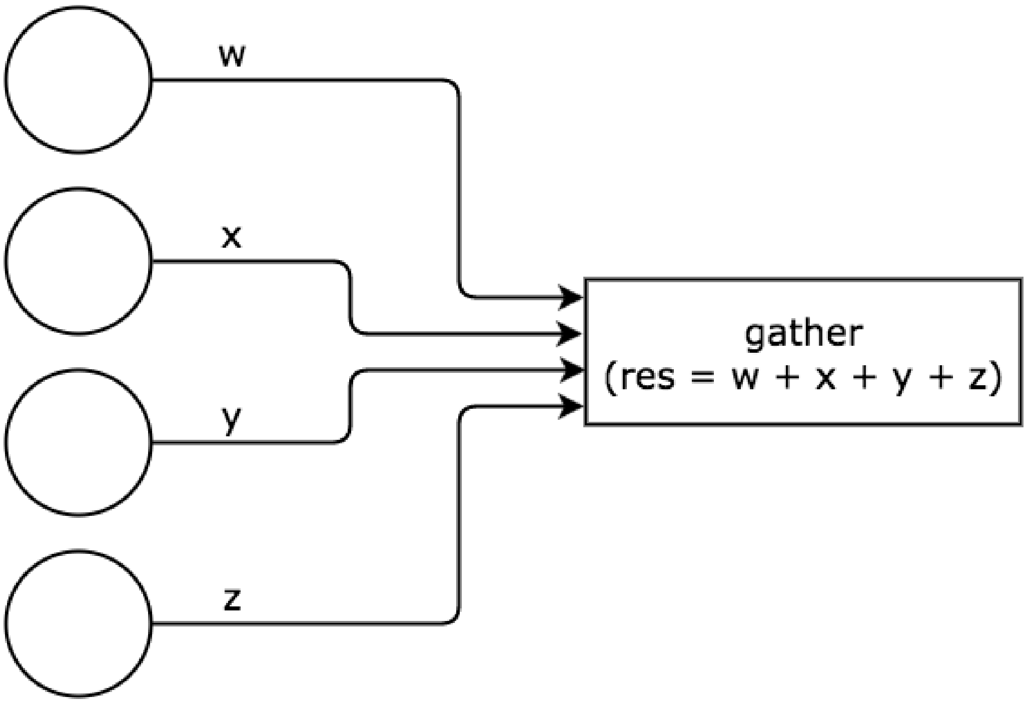


Figure 2.5: Gather

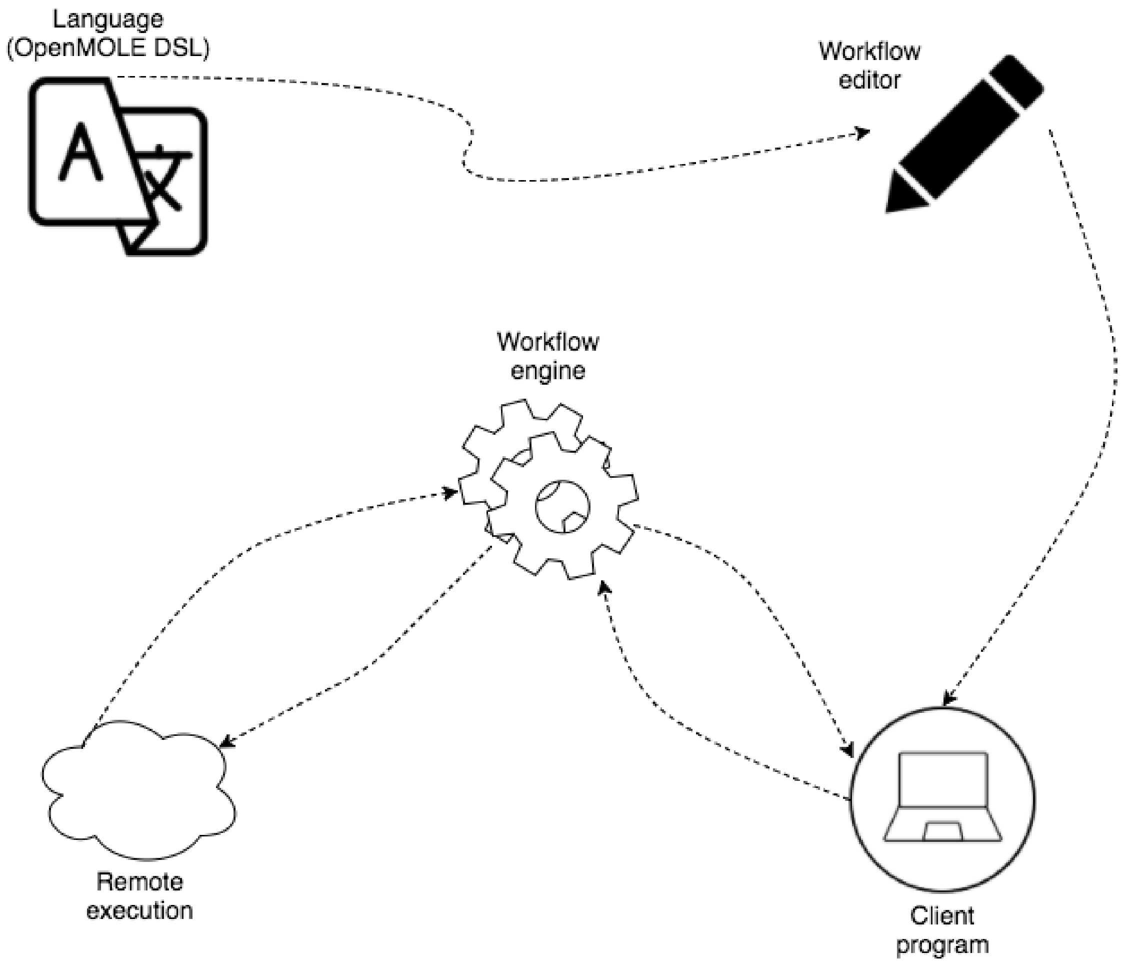


Figure 2.6: The OpenMOLE system

2.3.2 OpenMOLE

OpenMOLE is a scientific workflow engine employing a cloud-based approach for model experimentation. It is designed to exploit the power of distributed computing systems for naturally parallel processes.

By abstracting away the technical details of execution on remote environments, it exposes these as services instead. Moreover, OpenMOLE makes efficient use of distributed execution environments by revealing and utilising the natural parallelism of model experiments.

This natural parallelism includes, but is certainly not limited to, Design of Experiments [26], automated calibration processes [27], particle swarm optimisation [28], genetic algorithms [29], and applications of viability theory [30]. Generally, the concept applies to processes composed of subtasks that do not need to communicate with each other, such as parameter optimisation in machine learning.

OpenMOLE's Domain Specific Language (DSL) is an extension of the Scala language, designed for distributed computing. The fundamental concepts involved in OpenMOLE's functionality are presented below. All examples are credited to the OpenMOLE team, and consist of excerpts extracted from their Documentation[6].

[6]http://www.openmole.org/current/Documentation.html

Tasks

Tasks are the atomic execution elements of OpenMOLE, allowing for encapsulation of programs. As portable, reentrant, and immutable pieces of software, tasks have no interfering side effects and can be run concurrently. They deal with I/O via input (arguments) and output (return values) variables, and are linked to each other by transitions.

We declare a simple task below. Declarations for `i` are assumed to have already been stated prior to this declaration.

```
val model =
  ScalaTask("val res = i * 2") set (
    inputs += i,
    outputs += (i, res)
  )
```

Transitions

Transitions can be of several types: flat, divergent, or convergent.

1. Flat transition: also known as a simple transition, it specifies a precedence relationship between two tasks and is marked by the operator `--`.

2. Divergent transition: a special type of transition, it is also known as an exploration transition as it links some task to an Exploration task. This highlights the parallelisation aspect of OpenMOLE, which is permitted through the exploration of given parameters' spaces. It is marked by the operator `-<`.

3. Convergent transition: as its alternative name of aggregation transition suggests, it aggregates the results of different tasks into a single stream. Its purpose is the resynchronisation of the execution streams. It is marked by the operator `>-`.

Hooks

Given that the tasks do not write files, display values, or generally present side effects, they can only export output via *hooks*. They make up the mechanism allowing tasks to save or display results provided by their execution on remote environments. Hooks carry out their actions upon task completion, as specified in their workflows.

Below we illustrate transitions and hooks, composing some tasks into a workflow (i.e., pipeline). We assume that tasks `t1` and `t2` have been previously declared.

```
exploration -< t1 -- (t2 hook ToStringHook())
```

Samplings

OpenMOLE opts for a broad meaning of parameters, including numbers, files, random streams, or images. Parameter exploration is facilitated by *samplings*, which are of multiple types.

1. Complete sampling[7]: the most common way to explore a model.

   ```
   val explo =
     ExplorationTask (
       (i in (0 to 10))
     )
   ```

2. Zip samplings: can be of three types, namely ZipSampling (`zip`), ZipWithIndexSampling (`withIndex`), and ZipWithNameSampling (`withName`). It corresponds to the traditional zip operations from functional programming.
   File variables represent the content of a file, so the ZipWithNameSampling allows preservation of the file's name. If this information is of interest to the user, they can thus zip the file variable with a `String` variable (e.g., the file's path) in the dataflow.

   ```
   val s1 = p1 in (0 to 100)
   val s2 = p2 in (0 to 100)

   val s3 = s1 zip s2
   ```

3. Random samplings: can be generated from an initial sampling using `shuffle`, which results into a randomly shuffled version of the sampling. A fresh random sampling can be generated using `UniformDistribution[T]`, where `T` represents the type of random numbers to be generated.

   ```
   val s4 = s2.shuffle
   ```

4. Higher level samplings: OpenMOLE also handles higher level samplings such as `repeat` and `bootstrap`.

5. Literature samplings: some widely spread samplings from the literature are implemented within the platform. Examples are Latin hypercube sampling (`LHS`), and low discrepancy sequences (e.g., `SobolSampling`).

6. File samplings: file exploration in OpenMOLE can be done by using Domains. This allows the user to range along a set of files, whether in one or multiple directories. A more powerful method to inject files into the dataflow consists of using sources, which are described later on.

7. CSV sampling: allows the user to inject their own sampling directly into a task through CSV files. This feature is invoked using `CSVSampling`.

Various operators can be used on the samplings. These include `take` for selection of the first N values of a sampling, `filter` for selection of values from a sampling according to a specific condition, and `sample` for selection of N random values from a sampling. There is also the `x` operator, which allows for combination of different samplings upon first unrolling all domains. Finally, the keyword `is` allows a full domain to be stored in a variable of type array, which can then be passed along to other tasks.

```
val s5 = s1 take 10
val s6 = s1 filter ("p1 > 10")
val s7 = s1 sample 5
```

[7]A complete sample is a set of objects from a parent population that includes all objects satisfying a given set of selection criteria

Remote execution environments

A key feature of OpenMOLE is the ability to delegate the workload to a remote execution environment. This can be done in a declarative manner, using the keyword `on`. No additional installation or configuration on the remote machine is necessary, but authentication information must be provided by the user. Multiple tasks can also be grouped into a single job using the keyword `by`. In the event that no environment is specified, the default behaviour is local machine execution. Otherwise, there are several environments available to choose from.

1. Multi-thread: executes tasks concurrently on the local machine.
2. SSH: allows execution on a remote server.
3. Clusters: permits execution on clusters managed by PBS/Torque, SGE, Slurm, Condor, or OAR.

   ```
   val env =
     CondorEnvironment(
       "login",
       "machine.domain"
     )
   ```

4. EGI: facilitates execution on the European Grid Infrastructure[8] using DIRAC[9] as a submission engine by default.

Sources

Sources are an OpenMOLE concept designed to inject data into the dataflow from CSV files, databases, sensors, etc. They are complementary to hooks as well as an advanced addition to file sampling, both of which have been described above. A source is executed prior to each execution of its corresponding tasks, and only use of file-based sources is currently available on the platform. Example uses of sources include listing files and/or directories within a directory as well as exploring a directory.

Other

Advanced exploration methods are implemented within OpenMOLE in order to automatically generate workflows toward solving exploration problems. However, these are beyond the scope of the present paper and will not be discussed further.

Putting everything together

We present a full workflow example from the OpenMOLE documentation in order to connect the essential information presented above.

[8]http://www.egi.eu/
[9]http://diracgrid.org/

```
// Define the variables that are transmitted between the tasks
val i = Val[Double]
val res = Val[Double]

// Define the design of experiment
val exploration = ExplorationTask(i in (0.0 to 100.0 by 1.0))

// Define the model
val model =
  ScalaTask("val res = i * 2") set (
    inputs += i,
    outputs += (i, res)
  )

// Define the execution environment: here, a local execution
environment with 4 threads
val env = LocalEnvironment(4)

// Define the workflow
exploration -< (model on env hook ToStringHook())
```

Comparison to other workflow systems

Referring to the workflow management and execution systems presented in 2.2, we identify OpenMOLE's defining characteristics. Firstly, we notice that it lacks a GUI for workflow editing; however, it permits code-based workflow editing, employing an extended version of the Scala language. Therefore, despite the current lack of an intuitive GUI, OpenMOLE does not impose a completely new syntax with which the user must familiarise.

Secondly, we note that OpenMOLE can support users within a diversity of fields, as long as they possess some degree of programming knowledge.

Finally, the most defining feature of the engine is its explicit support for model experimentation and naturally parallel processes. To this, we add the exposure of high computing power platforms as services, performed through an efficient abstraction of technical details. Therefore, while it can attend to a range of computational power needs, OpenMOLE is tailored to users who frequently perform parameter exploration, optimisation, and other similar tasks.

2.4 Related Technologies

2.4.1 YAML

Overview

YAML (YAML Ain't Markup Language) is a data serialisation language. It is designed to be human-friendly, and it works with modern programming languages, facilitating storage and transfer of data.

YAML uses three basic primitives: *mappings*, for hashes or dictionaries; *sequences*, for arrays or lists; and *scalars*, for strings or numbers. Together with a simple typing system and an aliasing mechanism, this forms a complete language for serialising all native data structures. Additionally, YAML uses both a flow grammar and an indentation one. This particular feature aims to support both JSON syntax and enhanced human readability, allowing the user to employ a preferred syntax.

Despite its versatility, YAML is largely used for more common tasks, such as configuration and log files, cross-language data sharing, and debugging complex data structures. We now present a more detailed view of the language's features.

Basic concepts

A YAML node consists of a single native data structure. Its content may be one of the three primitives previously defined: a mapping, a sequence, or a scalar.

Each node is associated with a tag, used to attach meta information to the node. Tags specify the expected node type, and may provide additional information such as a set of allowed content values for validation of the node. During the processing of a YAML document, tags that are not explicitly defined by the user must be resolved in order to allow content validation (see 2.4.1 Schemas).

Flow style

The flow style presented by the YAML language permits folding of long content lines in order to enhance readability; allows for additional control over the construction of native data structures; and employs an aliasing mechanism that allows for reuse of previously constructed object instances. We show how each primitive of YAML is represented in the flow grammar, and offer details on its aliasing mechanism.

1. Aliasing: YAML nodes may be *anchored* using the `&` character; *alias* nodes are denoted by the `*` character. The alias refers to the most recent preceding anchored node. An unanchored node may not be invoked as an alias.

2. Scalars: the flow scalars can be presented in three different styles: double-quoted (for escaped sequences), single-quoted (when escaping is not necessary), and plain style (human-readable, but context-sensitive).

3. Collections: entries are separated by the `,` character.

   3.1. Sequences: denoted by `[` and `]` characters.

   ```
   [one, two, three, four]
   ```

3.2. Mappings: denoted by { and } characters.

```
{one: 1, two: 2, three: 3, four: 4}
```

Block style

YAML's block style stands for its indentation grammar. While it is a more human-readable notation, it is also less compact than the flow style.

The indentation count is started by the content's first non-empty line. All children nodes must be indented at least one additional space relative to their parent nodes.

1. Scalars: can be presented in literal (noted `|`) or folded (noted `>`) style. The literal style considers whitespace as content; meanwhile, the scalar style subjects scalars to line folding, allowing long lines to be broken.

2. Collections:

   2.1. Sequences: a series of nodes; each node is denoted by a leading `-` character, which is separated from the node by a white space.

   ```
    - one
   - two
   - three
   - four
   ```

   2.2. Mappings: a series of pairs of nodes; each pair is an *entry*, consisting of a key node and an optional value node (i.e., `key: value`). A node may also be explicitly marked as a key (`?key`), in which case the value node (`: value`) must be written on the next line.

   While the value itself may be missing, its marker (`:`) must follow a key.

   ```
   one: 1
   two: 2

   ? three
   : 3
   ? four
   : 4
   ```

Character stream

YAML defines two complementary views of data: YAML representations, the data objects we have presented so far; and the YAML character stream, which presents these data objects in a human-readable text format.

A YAML stream consists of zero or more documents, independent from each other. Within a document, the end of directives is signalled by the `---` marker, and the end of the document itself by the `...` marker. Multiple streams may be concatenated, or additional documents may be appended to them.

Schemas

The YAML specification proposes a standard set of schemas to be used for tag resolution. While these are beyond the scope of our project, they represent an important feature of YAML. The schemas are listed below in ascending order of complexity.

1. Failsafe schema: only covers the generic YAML types, i.e., mapping, sequence, string.
2. JSON schema: in addition to the types supported by the failsafe schema, it also supports the null, boolean, integer, and floating point types; it is a result of YAML 1.3's increased compatibility with JSON.
3. Core schema: while it supports the same tags as the JSON schema, it allows for a more human-readable presentation.
4. Extended: encompasses a variety of schemas that may be further implemented by the user in order to support a given language's native data structures (e.g., sets); ideally, the additional types are to be chosen from a list of tags[10] proposed by the YAML developers.

2.4.2 SALAD

SALAD (Semantic Annotations for Linked Avro Data) is a schema language for describing structured linked data in JSON and YAML documents. It builds upon JSON-LD[11] and the Apache Avro[12] data serialisation system, adding features for rich data modelling such as inheritance, template specialisation, object identifiers, object references, documentation generation, and transformation to RDF.

SALAD is being developed concurrently with the Common Workflow Language specification, and was first introduced with version 3 of CWL[13]. It permits URI resolution and strict document validation while also enabling generation of JSON-LD contexts, RDF schema, and RDF triples.

Syntax

SALAD documents are written using a JSON-compatible subset of YAML. This excludes YAML features that are not supported by JSON, such as tags.

Data concepts

First, we explain some notions relating to the document model of SALAD, which we will later use to define additional concepts. Note that the definitions for *object* and *document* correspond to the ones mentioned earlier in 2.3.1 Common Workflow Language.

1. Object: equivalent to the "object" type in JSON. Consists of an unordered set of *fields*; a field is a `(name, value)` pair.

   Note: `name` must be a string; `value` may be a string, number, boolean, array, or object.

[10]http://yaml.org/type/
[11]JSON-LD specification: https://www.w3.org/TR/json-ld/
[12]Apache Avro documentation: https://avro.apache.org/docs/current/
[13]Date of release: 15th of March, 2016.

2. Document: a file that contains either a *single root object*, or an *array of objects.*
3. Document type: a class of *files* with shared structure and semantics.
4. Document schema: a formal description of a *document type*'s grammar.
5. Base URI: a context-dependent URI[14]; used to resolve relative references throughout the document.
6. Identifier: a URI that identifies a single *document*, or a single *object* within a document.
7. Vocabulary: a map - keys are symbolic *field names* and enumerated *symbols* defined by a *document schema*; values are absolute URIs.

Document context

A document context may be *implicit* or *explicit.*

Explicit contexts concern documents consisting of a single root object. Such a document may contain the following fields:

- `$base`: a string; sets the base URI for resolving relative references. Default: the URI used to load the document.
- `$namespaces`: an *object*; keys and values are strings. Keys consist of *namespace prefixes* used in the document, and values are *prefix expansions.*
- `$schemas`: an array of strings; may list URI references to documents in RDF-XML format. These can be queried for RDF schema data, which may provide additional semantic context, or may be used for document validation.
- `$graph`: an array of objects; if present, holds the primary content of the document.

Implicit contexts apply to documents containing an array of objects; or documents containing a single root object which do not contain the fields mentioned above.

Document graph

A document graph may also be *implicit* or *explicit.* A document consisting of an array of objects is an *implicit graph.* On the other hand, a document containing a single root object and the `$graph` field is an *explicit graph.*

Other

The SALAD specification further defines schemas, to be used for document preprocessing and schema validation. However, we leave these out of our present scope; the specification is new and subject to significant changes in the near future.

[14]RFC 3986: https://tools.ietf.org/html/rfc3986

2.4.3 YAML parsers

Given that we attempt to translate YAML documents into Scala code, we initially considered the option of using an already existing YAML parser. Two candidates were found: scala-yaml [33] and SnakeYAML [34].

The first is an abandoned project, so if any issues were to arise, no external support could be provided. The latter project is active and under ongoing development, but only handles version 1.1 of the YAML language. The current version of the language is 1.2, and was released in 2009, which makes SnakeYAML outdated for our purposes. Moreover, the current YAML version is a first attempt at a merge between the YAML and JSON languages. This particular feature is actively used by the Common Workflow Language specification that the present project is based on.

While more options could have been explored at this point, we decided that the project would include the development of a parser according to our specific needs. The disadvantage of spending additional time on the implementation was outweighed by the potential advantages of having a custom-made parser with internal support. Furthermore, it could then easily be fully integrated with other components of the platform. Finally, the new parser would also be up to date with the current version of the YAML language.

2.4.4 FastParse

As a parser-combinator library for Scala, FastParse facilitates the writing of recursive descent parsers. It is extremely easy to use, as can be seen further below, in a quick overview of the FastParse API. We also note that by employing functions such as `filter` and `map`, FastParse exposes a traditional functional design and makes full use of the flexibility[15] of the Scala language.

Furthermore, it has performed well in benchmark tests (see Figure 2.7); most notably, a parser written using the FastParse library can be 1/10 of the size of a handwritten recursive descent parser, but reaches up to 1/5 of its speed.

Finally, another important characteristic of FastParse is its error reporting system; it automatically provides well-defined and well-structured errors, which is invaluable for debugging purposes.

Informal overview: subset of FastParse API

`a ~ b`: Parses `a`, and then `b`.

`a | b`: Parses `a` or `b`.

`a ~/ b`: If `a` is parsed successfully, performs a cut; when backtracking[16], it does not go back past this point.

`a.rep()`: Parses `a` zero or more times.

`a.?`: Optional parsing.

`a.!`: Captures parsed text as `String`.

[15]The Scala language fuses object-oriented and functional programming.

[16]If the parser attempts to use a certain rule and eventually fails, it will go back to the previous point where it was successful, and attempt a different rule for parsing.

`!(a)`, `&(a)`: Positive/negative lookahead[17].

`a.map(f:A ⟹ B)`: Transforms the parser result with the given function.

`a.filter(f:A ⟹ Boolean)`: Applies the given predicate to the parser, keeping only values returning `true`.

`a.log(s:String)`: Prints information about where the parser was tried and its result. Useful for debugging purposes.

`CharPred(f:Char ⟹ Boolean)`: Parses only characters that satisfy the given predicate (e.g., `isUpper`).

`CharIn(s:Seq[Char]*)`: Parses only characters that are included in the given sequence.

`StringIn(strings:String*)`: Quickly parses strings that are found in the given argument.

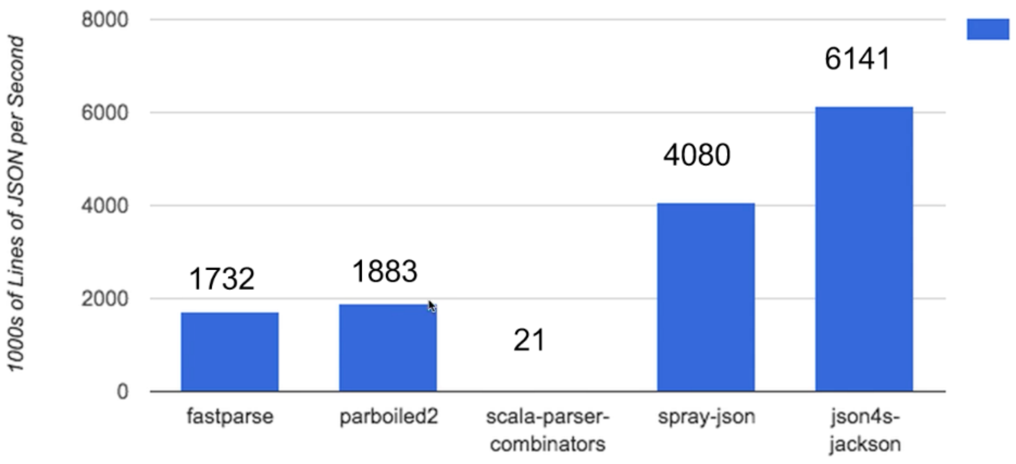


Figure 2.7: FastParse performance compared to: parboiled2, scala-parser-combinators, and two handwritten recursive descent parsers
© Li Haoyi

[17]The lookahead represents the number of tokens that a parser may use to decide which of the possible rules to use for parsing a certain element. For a trivial example, consider `foo` and `far` may be terms of interest with associated rules; when parsing text that begins with `f`, we may use a lookahead of 1 in order to see whether the next character is `a` or `o`, and select the corresponding rule. This is useful to reduce time that may be spent backtracking.

Chapter 3

Comparison of CWL and OpenMOLE DSL

While the Common Workflow Language specification and the OpenMOLE Domain Specific Language present certain similarities when first viewed, a more detailed analysis of their respective capabilities and correlations is necessary. Therefore, we wish to analyse the OpenMOLE platform from a CWL perspective, laying down a foundation for the project's design as well as for the implementation of the CWL specification. A brief overview of the results is presented in Table 3.1, with detailed explanations below.

Features	**CWL**	**OpenMOLE**
Steps (incl. dependencies)	Yes.	Yes.
Labels	Yes.	No.
Type systems	Yes; limited.	Yes; complete.
Natural parallelism	Yes.	Yes.
Exploration	Yes.	Yes.
Aggregation	Yes.	Yes.
Inline Javascript expressions	Yes.	No.
Docker	Yes.	No; short/mid-term plan.
Create output files prior to process execution	Yes.	Yes.
Environment variables	Yes.	Yes.
Shell commands	Yes.	Yes.
User control over hardware resource allocation	Yes.	Yes.

Table 3.1: Features comparison results - CWL and OpenMOLE

3.1 Conditions

In the CWL specification, requirements and hints represent certain conditions that either the runtime environment or the workflow engine must (or may, respectively) meet prior to the execution of the workflow. Failure to meet requirements results into a fatal error whereas a hint failure results into an appropriate warning.

Meanwhile, considering the OpenMOLE platform's use of CARE packaging, some conditions are left solely to the user to consider; the platform will package all user preferences with the workflow to execute.

However, most of the conditions must still be handled by the CWL implementation. All possible conditions in the CWL specification are evaluated from the OpenMOLE perspective below.

3.1.1 Workflow requirements

All of the potential requirements (or hints) for workflows can be satisfied by the OpenMOLE platform, aside from `InlineJavascriptRequirement`.

1. `InlineJavascriptRequirement`: the platform must support inline Javascript expressions;
2. `SchemaDefRequirement`: provides an array of types to be used when interpreting `inputs` and `outputs`;
3. `SubworkflowFeatureRequirement`: the platform must support nested workflows for a given step;
4. `ScatterFeatureRequirement`: the platform must support multiple output streams in a specified step;
5. `MultipleInputFeatureRequirement`: the platform must support multiple input streams in a specified step;
6. `StepInputExpressionRequirement`: the platform must be able to connect the output of a previous step to the input of the given step.

3.1.2 Command Line Tool requirements

All possible conditions for Command Line Tools are presented below. With the exception of `InlineJavascriptRequirement` and `DockerRequirement`, all other requirements (or hints) can be handled by the OpenMOLE engine.

1. `InlineJavascriptRequirement`: the platform must support inline Javascript expressions. OpenMOLE cannot currently handle this requirement.
2. `SchemaDefRequirement`: provides an array of types to be used when interpreting `inputs` and `outputs`.
3. `DockerRequirement`: OpenMOLE does not currently handle execution in Docker containers. However, this feature is being considered for future development.
4. `CreateFileRequirement`: specifies files to be created in the output directory prior to execution of the Command Line Tool.
5. `EnvVarRequirement`: defines a list of environment variables that the platform will set in the runtime environment during execution.
6. `ShellCommandRequirement`: modifies the behaviour of a process such that it generates a string containing a shell command line. It may be quoted to prevent shell interpretation, or unquoted to be added to the execution pipeline.
7. `ResourceRequirement`: specifies a minimum and maximum of hardware resources to be allocated to the job; examples include number of CPU cores, amount of RAM, and file system based storage. OpenMOLE handles this requirement by allowing the user to indicate the preferences within the definition of the execution environment.

3.2 Workflow steps

While the CWL specification divides possible actions into workflow steps and Command Line Tools (CLT), the OpenMOLE DSL does not differentiate between the two. Instead, they are both classified as tasks. Even more accurately, they can be called capsules, but the concepts are essentially the same at a shallow user level. For more details on the differences between tasks and capsules, please see [14].

3.3 Labels

Labels and descriptions are optional concepts described by CWL, referencing a process or a parameter. For example, a label may be used in order to briefly specify the provenance of a sample; a description may be used to further expand how the sample was obtained. These features do not have an equivalent in OpenMOLE at the moment.

However, they appear highly useful from a user perspective as well as for a graphical representation of the workflows. While they could currently be mentioned as comments in the Scala language, this is ultimately an unfeasible solution in the long term.

3.4 Type systems

The type system of CWL is limited, supporting only the following: primitive types (i.e., `null`, `boolean`, `int`, `long`, `float`, `double`, `string`) and files. All other types are covered by the keyword `Any`, which validates any non-null value without further verifying it.

On the other hand, the OpenMOLE platform supports a range of types beyond primitive types and files. This consists of directories as well as all Java and Scala types, but extends even further to user-defined types[1].

We conclude that all of the types provided by the Common Workflow Language specification can be handled accurately by the OpenMOLE engine. However, note that the OpenMOLE user's actions may be severely limited by the lack of available types in CWL.

3.5 Nested workflows

CWL's notion of nested workflow, or subworkflow, is handled by OpenMOLE via the concept of MoleTask. A MoleTask is essentially a workflow step consisting of another workflow.

3.6 Natural parallelism

One of OpenMOLE's principal features is its leverage of natural workflow parallelism, as explained in Section 2.3.2. According to the official specification, a CWL implementation is free to change the order of execution of a workflow's steps [25]. Thus, it may opt not only for a different order, but also for a concurrent execution, upon the condition that all dependencies between steps are met. This correlates directly to OpenMOLE's leverage of natural parallelism within workflows.

[1]New types can be defined by users via plug-ins to the platform.

3.7 Exploration

As specified in Section 2.3.2, key uses of OpenMOLE include model exploration and parameter configuration. In order to perform these actions, the workflow engine was designed to handle the creation of multiple instances of one task for various parameters. This is represented by the exploration transition in OpenMOLE.

CWL did not handle this feature in draft-2 of the specification. However, the current version has added the notions of `scatter` and `scatterMethod`. A scatter operation specifies that the step should execute separately over a list of given input elements. Should multiple input parameters be provided, preferences regarding their selection and repartition to each instance of the task can be provided in a `scatterMethod` field.

While OpenMOLE's capabilities go beyond this via use of samplings, there is a one-to-one correlation between the concepts of exploration and scatter in the two languages.

3.8 Aggregation

OpenMOLE's aggregation transition is complementary to the exploration transition discussed above. The CWL corresponding concept is the newly added `MultipleInput` feature.

Both of these concepts handle the merging of multiple inbound data links. As in the case of exploration/scatter, there is a one-to-one mapping between the notions of aggregation and multiple input in the two languages.

Chapter 4

Product Design

Following the background research presented in Chapter 2 and the side-to-side analysis presented in Chapter 3, we set out to design our proposed contribution to the standard description of big data neuroimaging experiments.

Several factors were taken into account. First of all, the tool needs to be able to transform a CWL document into Scala code that the OpenMOLE platform can execute. For this purpose, it needs to understand the CWL syntax and its relation to the OpenMOLE DSL.

Moreover, given that this tool connects a standard language to a workflow platform, we considered that an intermediate representation for the standard workflows may be used again. Thus, it would allow CWL implementation for different workflow platforms in the future. Therefore, we opted for a modularised system in order to facilitate extensions.

Secondly, we consider the ease of use; thus, the user must be able to run CWL documents on the OpenMOLE platform freely, with a minimum number of actions to take. This condition is in accordance with the higher level purpose of the project, namely allowing scientists to focus on their analyses rather than additional code and execution details.

Finally, our workflow systems analysis revealed the necessity of a visual workflow representation; this must be clear, expandable, providing as much detail as requested, easily accessible, and up-to-date with current visual design standards. This way, it will not only be of great help to the user, but will also feel instantly familiar.

Having briefly mentioned the ideas driving the design of our product, we now present our proposed design.

4.1 System overview

Our project is based on two main technologies: first, there is the Common Workflow Language specification, which provides a portable standard for workflow representation; and the second is the OpenMOLE platform, designed specifically to facilitate model experimentation and distribution of execution to remote environments.

Our proposed tool connects these two technologies, aiming to contribute to an industry standard in the field of neuroimaging experiments and beyond. It is designed as a separate component at the moment, to be later integrated into the OpenMOLE platform as a plugin.

As our project is based on two languages between which we have to translate, we have chosen a modularised approach. Therefore, we have split its design into a parser and a

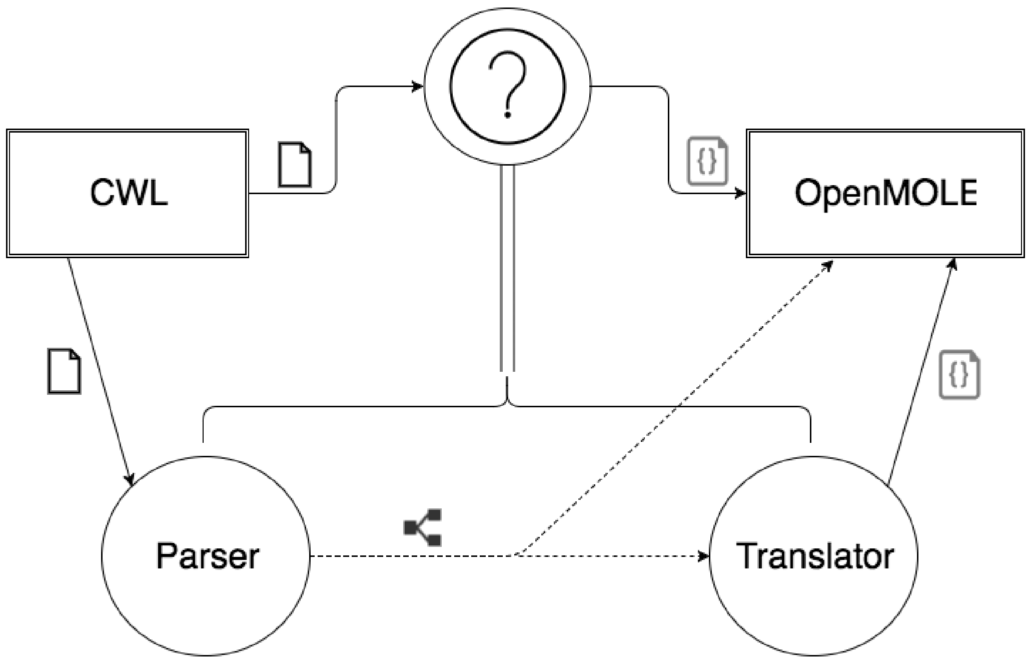


Figure 4.1: Interaction of the system's components

translator. A CWL document would be passed to the parser, which creates an abstract representation to be passed to the translator. The modules' individual designs are presented below in a top-down approach.

Figure 4.1 offers a visual aid to better follow the components' interactions with each other.

4.2 Parser

In order to be able to translate CWL files into Scala code for OpenMOLE, we need to parse the CWL documents we receive. As previously mentioned in 2.4.3, we first explored a couple of already existing parsers for this purpose. We eventually decided to implement our own parser, which would be tailored to the Common Workflow Language.

The next issue concerned the writing of the parser. Having first looked at the Scala Standard Parser Combinator Library[1], we also found FastParse (see 2.4.4), which is 100x faster than the previous option and up to 1/5 the speed of a handwritten parser [35]. We considered this would be an acceptable implementation-time/performance trade-off, and continued the project using the FastParse library.

The parser module's purpose is to translate the CWL document into an abstract intermediate representation. While we only implement translation functionality for the OpenMOLE platform, this intermediate representation could potentially be passed to other packages that would translate the workflow file into a range of languages.

In Chapter 2, we presented both the Common Workflow Language and the SALAD specifications. As SALAD files, CWL documents also need to go through the document preprocessing stage. Document preprocessing consists of URI resolution as well as resolving `import` and `include` directives.

Thus, the parser module has double functionality: parsing and preprocessing the CWL

[1]https://github.com/scala/scala-parser-combinators

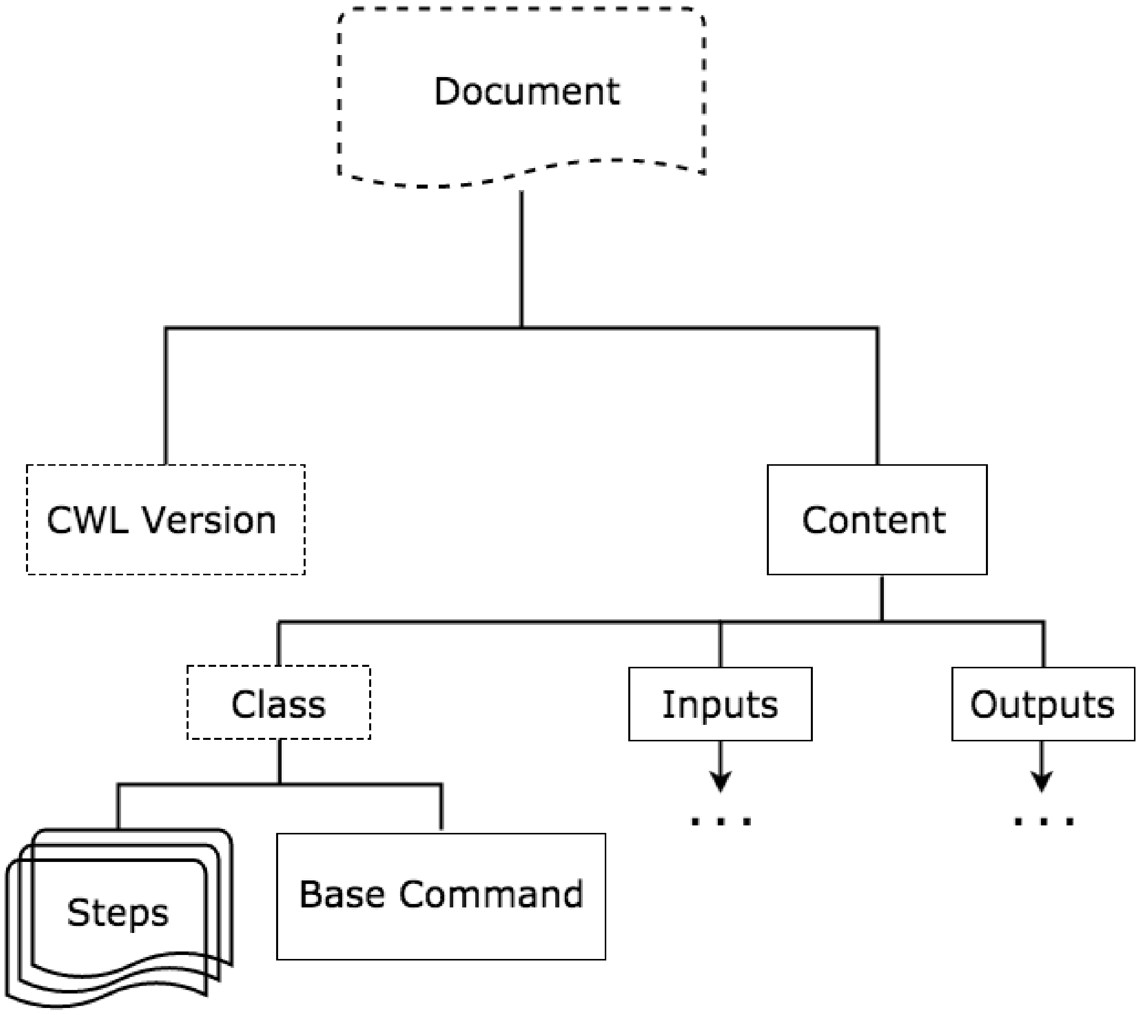


Figure 4.2: From a high-level perspective, our resulting AST follows the parsing tree presented above.

document. We decided these would be implemented within the same module as they can be executed together, in a single document traversal.

While we initially considered the implementation of document preprocessing that is required by the SALAD schema language specification, we found it unfeasible to support it within the current version of the project. Having recently had its first release[2], the SALAD specification is still a draft version, and may be subject to significant changes in the near future. Therefore, we established that only basic functionality would be implemented now, and expanded upon later.

A summary of features is presented in Chapter 6. We note which ones we support, detailing their implementation in Chapter 5. The remaining features are included in the plan presented in Chapter 7.

[2]The first draft of the SALAD schema language specification was released on the 15th of March 2016, at the same time with Version 3 of the Common Workflow Language specification.

4.3 Translator

As follows from the previous section, the translator module receives an abstract intermediate representation of the initial CWL document. For our purposes, this is an Abstract Syntax Tree (AST), roughly following the structure of the continuous line elements in Figure 4.2, which we explain in detail later.

The AST represents the workflow described by the given CWL file in an abstract, language-independent manner. This means that details included in the workflow specification, such as the Common Workflow Language version that was used, or the specific syntax, are left out. However, the AST does keep all details of importance, such as identifiers, values, actions to be taken, and the relations between these.

The translator performs a top-down traversal of the tree, and generates the appropriate Scala code for the OpenMOLE platform. We perform the code generation based on the correspondences found and presented in Chapter 3. It should also be noted that the translator only requires one tree traversal in order to generate the code.

An important issue presented within this module is that the Common Workflow Language lacks support for scheduling process invocations on remote environments. Meanwhile, this is a defining feature for OpenMOLE. While we further explore this incompatibility issue in the Evaluation and Future Work chapters, the CWL-OpenMOLE user will temporarily have to either accept the default execution on their local machine, or manually add task delegation to a remote environment within the OpenMOLE workflow editor.

4.4 Graphical representation

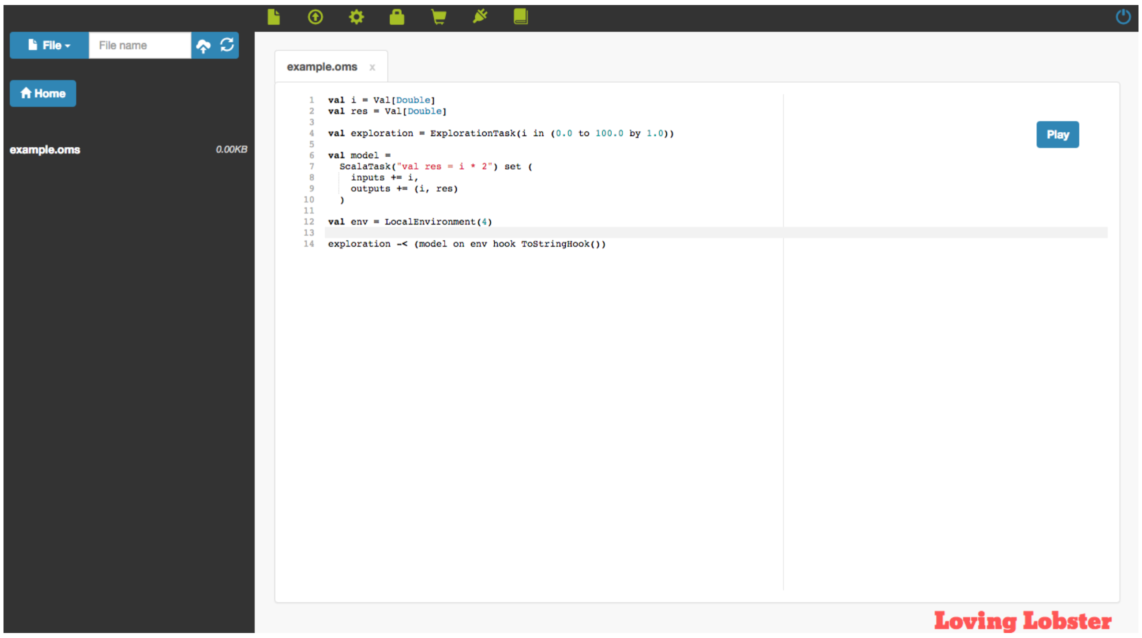
A visual representation of the workflows submitted to OpenMOLE is currently out of scope for our project. In order to implement this feature, additional discussions with the OpenMOLE team would be necessary first, which would be followed by the implementation time itself.

However, we have designed a prototype, which we present in this section. Figure 4.3 presents the current OpenMOLE interface. To this, we would add a button (circled in Figure 4.4) that toggles the graphical representation of the workflow. The button would transform the view into a split-screen: the initial components on the left, and the graphical workflow representation (shown in Figure 4.5) on the right-hand side.

The rectangular parallelepipeds represent exploration tasks, and the rounded rectangles represent other tasks defined within the workflow. This difference in representation exists because the user should be able to differentiate quickly between normal tasks, and tasks with multiple outbound links that will cause the creation of multiple instances of their successor. The blue arrows represent transitions, and the names for the tasks are the identifiers of the variables holding them.

Inputs are circled, and outputs are captured in a rectangle, for an easier differentiation. At the bottom of the screen, a square containing gears is the graphical representation for environment. The representation used as an example in Figure 4.5 is based on the sample workflow in Figure 4.6.

Although the features so far have handled an implication from workflow toward representation, we propose an additional feature in the opposite direction. When selecting a task within the visual representation area, its corresponding code will be highlighted in the workflow editor. While it may appear trivial within the scope of the given example, this



```
val i = Val[Double]
val res = Val[Double]

val exploration = ExplorationTask(i in (0.0 to 100.0 by 1.0))

val model =
  ScalaTask("val res = i * 2") set (
    inputs += i,
    outputs += (i, res)
  )

val env = LocalEnvironment(4)

exploration -< (model on env hook ToStringHook())
```

Figure 4.3: The OpenMOLE interface

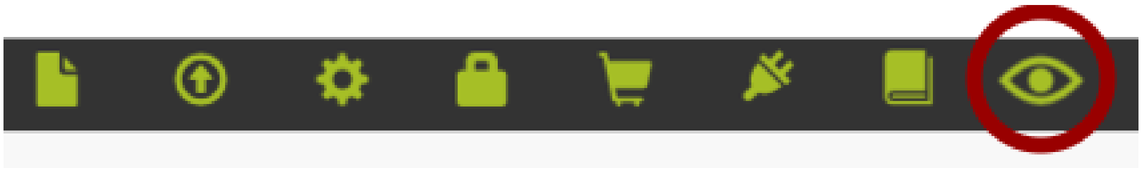


Figure 4.4: A new button

feature would be useful for beginners as well as for more advanced users who may want to deal with longer, more intricate workflows in an efficient manner.

These ideas are subject to further development prior to their implementation. More time would be necessary in order to gather the exact needs of users and provide efficient, easy-to-use solutions to the identified problems.

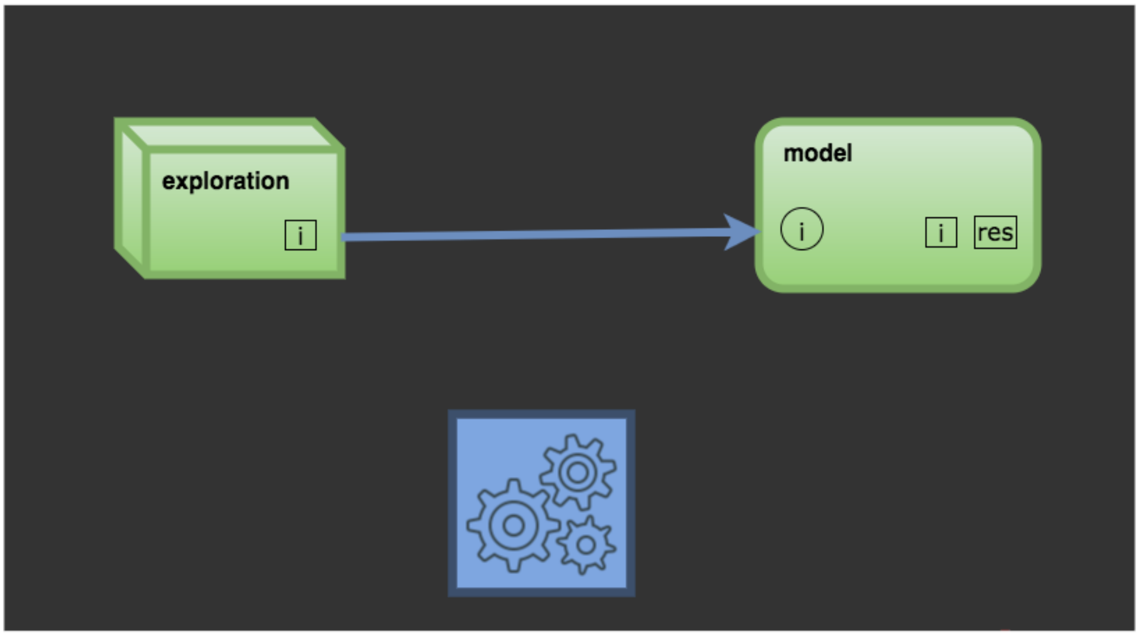
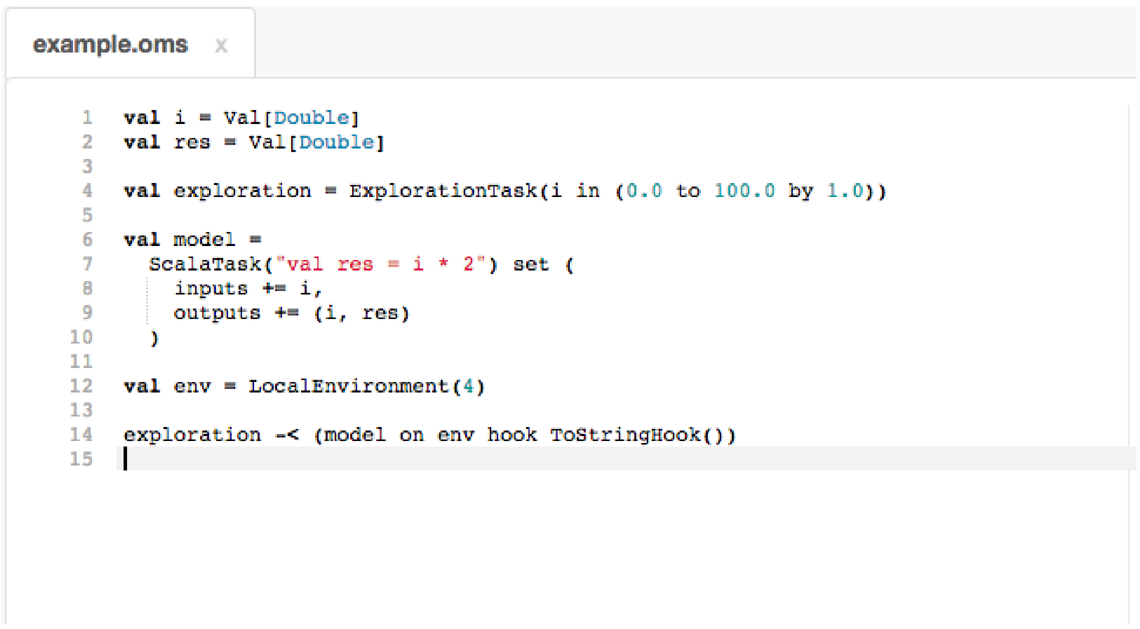


Figure 4.5: An example graph



```
val i = Val[Double]
val res = Val[Double]

val exploration = ExplorationTask(i in (0.0 to 100.0 by 1.0))

val model =
  ScalaTask("val res = i * 2") set (
    inputs += i,
    outputs += (i, res)
  )

val env = LocalEnvironment(4)

exploration -< (model on env hook ToStringHook())
```

Figure 4.6: An example workflow in OpenMOLE

Chapter 5

Implementation

Following the design of the system, we present a basic implementation. Note that we do not handle all previously planned features, although we have presented approaches to be taken in Chapter 4.

5.1 Parser

The parser module has been designed and implemented for the Common Workflow Language specification. It accepts a CWL workflow in input, and produces an Abstract Syntax Tree of the given file.

The input workflow must be a `.cwl` document. Within this file, there may be references to other files for input, parameters, or output. The referenced files must be in `.yml` format. Both CWL and YAML documents follow the syntax used by SALAD documents, i.e., a JSON-compatible subset of the YAML language syntax.

The secondary (`.yml`) files may be referenced directly (i.e., by their path relative to the current directory), or by using `import` or `include` directives. Documents or resources referenced using an `include` directive must be loaded as text data. On the other hand, documents or resources referenced by an `import` directive must be loaded and recursively parsed[1]; the resulting AST will be added to the overall workflow tree in the corresponding location.

Our recursive-descent parser is written using the FastParse library. We have previously presented the FastParse API in 2.4.4. In order to parse a document, we have designed a tree showing how the elements are parsed, which we present in Figure 5.1 and briefly explain below.

[1]This is a temporary solution. In a full implementation, these would have to be recursively parsed and *preprocessed* as SALAD documents. However, we do not currently support this. We further discuss this in the next chapter.

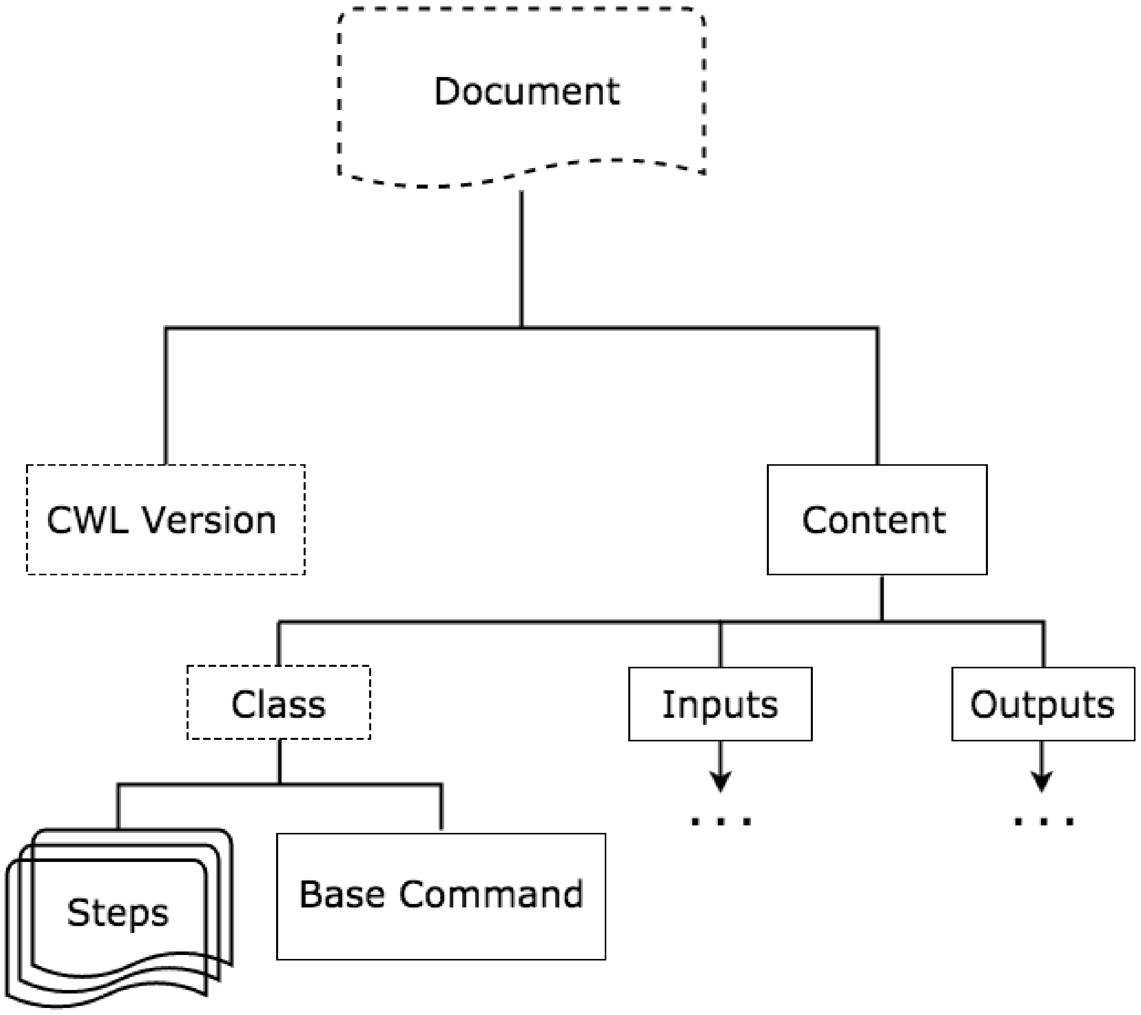


Figure 5.1: Parsing a CWL document

Parsing a document

A CWL document may be split into two parts: version declaration (e.g., `cwlVersion: cwl:draft-3`) and content. The version declaration is only relevant for document validation purposes, and is not included in the AST. The context may be then split into several compulsory components[2]: `inputs`, `outputs`, and `class` are common between Command Line Tools and Workflows; Command Line Tools additionally have a `baseCommand` component, while Workflows have `steps`. The `baseCommand` value will be a standalone, non-interactive program. Meanwhile, `steps` will contain an array of zero or more documents, to be parsed recursively. Similarly to the version declaration, the `class` component is only relevant to parsing purposes, and will not be stored in the AST.

As previously decided, when faced with an `include` or `import` directive, we will also recursively parse the referenced documents. The subtrees created by parsing these additional documents will be added to the main AST in their corresponding locations.

The entire process we have presented produces a single Abstract Syntax Tree. This intermediate, language-independent representation can be passed to the translation module.

[2]For the present project, we chose to only implement the basic components due to time constraints; additional optional ones are planned for future implementation.

5.2 Translator

Following the document parsing, the translation module receives an AST. It performs a single depth-first traversal and generates appropriate code for the OpenMOLE platform (i.e., extended Scala).

First, we note that the AST only contains nodes that are relevant to the workflow actions. The root is in place solely for starting the tree traversal. We perform a depth-first traversal. Therefore, we first generate the inputs declarations according to the `inputs` subtree, containing all the necessary information from the original workflow document. We then proceed to do the same with outputs declarations.

A `baseCommand` node is mapped to the corresponding task sub-type (i.e., `SystemExecTask`), and its children are traversed for information regarding I/O or other variables. A `steps` node in the AST informs us that multiple tasks will be generated, and will have to be composed in their specified order. Thus, we must traverse the children nodes while storing information related to their dependencies[3].

Having finished the AST traversal at this point, we now generate code for composition of the execution pipeline. We use the dependencies information that we have previously stored in order to maintain the user-specified ordering of tasks.

Finally, we remind the reader that the Common Workflow Language specification does not currently support task delegation to remote environments. Therefore, we cannot generate any code for this purpose. If the user does not manually input code for the execution environment (e.g., multi-thread, SSH, EGI), the resulting OpenMOLE workflow will be executed on the local machine by default.

[3]A dependency may occur when a step's input must be provided by a previous step's output.

Chapter 6

Evaluation

Following the implementation of the system, we now evaluate it in accordance with our proposed design as well as our initial goals.

Completeness

In the early stages of the project, we planned to develop a YAML processor as the Common Workflow Language specification only used a subset of the YAML syntax, with no considerable additional features. Evaluation would have been straightforward for the parser module, comparing the output of our parser against of that computed by the reference YAML parser.

However, the release of the SALAD specification imposed some changes to our original plan. Consequently, our first module may be evaluated against the YAML reference parser, but would expectedly not store the same information. Therefore, our evaluation now consists of a feature-by-feature verification.

SALAD preprocessing

Although we initially planned to develop a full SALAD processor, we eventually had to reduce our goal to only parsing the SALAD documents for the time being. Thus, we do not currently support URI resolution; consequently, we only offer partial support for `import` and `include` directives.

While the Common Workflow Language specification is a third version draft, the SALAD documentation was released as a first draft in March 2016, and may be subject to significant changes in the near future. Therefore, our lack of support for SALAD document preprocessing is only temporary, and is included in future plans.

Hints and requirements

Given that the Common Workflow Language is defined independently from any workflow platform, its implementation within various engines may differ widely. Hence, hints and requirements are an integral part of the specification, ensuring that the user is informed when a required or preferred feature is not supported by their platform of choice.

Our present implementation is essentially a proof of concept, handling only basic concepts of the specification. Thus, we do not currently support this feature. However, given Open-

MOLE's present inability to support inline Javascript expressions and Docker containers, hints and requirements must be prioritised for future implementation.

YAML grammars

Considering that the SALAD schema language has been built upon the YAML syntax, support for both the indentation and the flow styles of the YAML language was necessary.

We have succeeded in building a parser handling both grammar styles, but we currently differentiate between which set of rules to use by identifying the file's extension. This is due to conventions that we have identified, where the main document tends to employ an indentation grammar, while secondary files (e.g., describing input, output, parameters) generally use the flow grammar. It is not an ideal approach, and further work will go into handling this issue more elegantly.

Modularised approach

As mentioned in Chapter 1, we contribute to a standard in the description of experiments. In order to achieve this, we planned a modularised approach in our Design chapter.

As a result, our system consists of one module that creates an abstract intermediate representation of a CWL workflow, and a second module that receives this representation and generates the appropriate OpenMOLE DSL code. The intermediate representation from the first module may be used with other translation modules, which halves the implementation work in future Common Workflow Language implementations.

Labels and descriptions

As we noticed in Chapter 3, the OpenMOLE platform does not provide support for labels and descriptions. This hints to a weakness in the platform regarding the ability to track data provenance; however, this disadvantage may also be justified by OpenMOLE's incipient area of work (i.e., model experiments), which did not require data provenance.

Accordingly, we are not able to implement labels and descriptions into the project at the present time. Regardless, the feature was considered in the design stage of the project, resulting in a couple of possible solutions.

A temporary approach would have been the documentation of labels and descriptions from workflows as comments in the OpenMOLE code. However, this does not preserve the features' utility, as they must remain easily accessible and quickly retrievable for the user.

The full solution involves an OpenMOLE DSL extension where labels and descriptions are fully supported. This approach must be further discussed with the OpenMOLE team in order to determine its feasibility and a potential timeline.

Real-time feedback

Among our initial goals, we mentioned the wish to provide an interface that would simplify scientific analysis to some extent. Within this purpose, we considered the implementation of real-time feedback, providing immediate and helpful error messages to the user. This has proven to be unfeasible for the moment, both due to time constraints and the necessity to first integrate the proposed implementation within the OpenMOLE platform. However, such a feature would build directly upon our present parser module.

Ease of use

One of the main factors considered within the system's design was ease of use. A potential user must be able to write a CWL workflow and run it on the OpenMOLE platform with no additional effort.

Due to some incompatibilities that we have identified in Chapter 3, this has not been possible. A notable example of an incompatibility is the lack of support for scheduling workflow execution on remote environments within the Common Workflow Language specification. We note that this infeasibility is temporary, and we are taking steps in order to eliminate the concerns behind it (see Future Work).

Despite the inconvenience that these incompatibilities have imposed, we have succeeded in making our tool as easy to use *as possible* under the given circumstances. Thus, one only needs to run one command in order for their workflow to be processed sequentially by both modules; the result may be run directly on the OpenMOLE platform. While this is currently applicable to only a subset of CWL features, we prove it is possible, and we plan the next steps.

Out of scope

Some features, such as support for Docker containers and inline Javascript expressions, were deemed temporarily unfeasible by the analysis presented in Chapter 3. While they were initially considered, their integration within our project first requires additional implementation within the OpenMOLE engine.

Chapter 7

Future Work

The language and practice standardisation for neuroimaging experiments is a widely spread ongoing effort. The present project lays out the foundations of this effort. In doing so, it provides an in-depth analysis of a standard workflow specification language as well as of one workflow engine for distribution of programs on remote environments. We also attempt to bridge the gap between the two by providing a means of translation from the Common Workflow Language to the OpenMOLE DSL. However, the standardisation effort continues beyond the duration of this project.

With this in mind, we will continue the work started here as a collaborative open-source project. Given the open-source nature of both the Common Workflow Language specification and OpenMOLE, there are no obstacles in doing so. This decision also implies a continuing collaboration with both the Common Workflow Language group and the OpenMOLE team.

Future development plans are presented below, in order of priority.

7.1 Continuing implementation

Both languages have been studied in-depth and compared, as shown in Chapter 3. Besides an extensive analysis, we have also provided an implementation of basic features.

We wish to continue with a full implementation of the Common Workflow Language into the OpenMOLE DSL, starting from the discussion presented in Chapter 6.

We also note that the implementation of some features depends upon actions within either the CWL or the OpenMOLE group. For example, complete preprocessing of SALAD documents depends upon the next SALAD specification release, while implementation of Docker support depends upon its inclusion in the OpenMOLE engine.

7.2 Integration

This project has been building upon OpenMOLE, an already existing workflow platform. However, it has been developed as a separate product for the time being. Therefore, integration with the OpenMOLE web application is still necessary.

Not only will this represent a concrete real-world application of the results, but it will also expand the influence of the Common Workflow Language group's standardisation effort.

Moreover, such an integration will open the OpenMOLE platform to a wider user base, potentially ameliorating the learning curve of new users.

7.3 Real-time feedback

Inspired by FastParse's excellent error reporting system, we feel we could further help users by integrating a real-time feedback system into the OpenMOLE web application.

Users can currently write and edit their workflows within the web app. Therefore, following the integration of the current product into the platform, we plan to implement this feedback system using Scala.js[1]. The new system will report workflow errors to the user in real-time, without asking them to first perform any actions (i.e., compile, run).

We envision such a system for both workflows written directly into the OpenMOLE DSL, or in CWL for automatic translation. The error reporting should happen for either language the user chooses, even prior to any translation necessary for the workflow's execution.

Making use of FastParse's error reporting system is vital as it performs in a highly detailed and well-structured manner.

7.4 Graphical representation

We offered a prototype for a possible graphical representation of workflows (see 4.4). While it is subject to further discussion and future changes, it is an important feature to be implemented. This is primarily in accordance with our goal of facilitating research for scientists, allowing them to quickly view their workflows in an intuitive and familiar representation.

7.5 Collaboration

Throughout the project, differences between the CWL and OpenMOLE DSL have been analysed. It has become evident that extensions are possible within both groups.

Moreover, the Common Workflow Language specification also lists features that it cannot currently handle, but does not mention any final decision on their future status. A notable example is the ability to schedule process invocation on remote environments, which is a representative feature of OpenMOLE. The reasons behind the lack of support for such a feature on CWL's part should be further explored in order to find a solution.

Therefore, we believe it is in the interest of both projects to attempt a closer collaboration. A starting point would be presenting the current findings to the CWL group, and discussing future actions.

7.6 Continuous updates

The Common Workflow Language specification is by no means final. It is undergoing continuous development; a new version was released even during the development of the

[1]https://www.scala-js.org/

present project. It should be noted that the OpenMOLE platform is also under continuing development.

Therefore, we will need to continue developing our tool in order to keep up-to-date on both sides. This is the main reasoning behind the choice of continuing the work as a collaborative open-source project.

7.7 Research

As we previously specified, this project lays down a foundation for language and practice standardisation. However, we are fully aware that the research does not stop here. Therefore, further evaluation of the needs and preferences of researchers within the field is necessary. Identifying additional problems will lead to additional solutions to be integrated.

While we have employed workflow systems analysis as the main research method for the present project, the research can be carried out using other methods (e.g., interviews with the target users).

Chapter 8

Conclusions

We now summarise the work presented in this report.

Firstly, we offered an objective analysis of the most popular or outstanding existing workflow management systems. We continued with an investigation of a workflow language standard side-to-side with our workflow platform of choice. In doing so, we established the foundation of the project.

We then continued with a full system design. This included both technical and visual design of our project. Furthermore, we provided a basic implementation of this system, offering a starting point for future work toward the goal of facilitating research, documentation, and experiment reproducibility. Extensions are further facilitated by the modularised implementation of our system.

The main challenge faced throughout the research and execution presented so far was working with evolving specifications. While the dynamic aspect of the project was expected from the beginning, it was particularly demanding. As the specifications are under ongoing development, we found features we had initially planned to be later removed from the specifications, and novel features added to them.

Finally, we chose to continue the work started here as a collaborative open-source project. This is both inspired and required by the collaborative open-source nature of both foundation components. In order to facilitate the continuation of this work, we will make our results fully available. Furthermore, we will be providing a fully extensible plan based on the list of features in Chapter 6 as a starting point for future work.

In conclusion, we have contributed to a foundation of language standards for experiments, providing a basis for the creation of a standard description within neuroimaging experiments and beyond.

Bibliography

[1] Romano, P., Marra, D., & Milanesi, L. (2005). Web services and workflow management for biological resources. *BMC Bioinformatics*, 6(Suppl 4), S24. Available from: http://doi.org/10.1186/1471-2105-6-S4-S24 [Accessed 11th February 2016]

[2] De Roure, D., Goble, C. and Stevens, R. (2009) The Design and Realisation of the myExperiment Virtual Research Environment for Social Sharing of Workflows. *Future Generation Computer Systems*, 25:561-567. Available from http://doi.org/10.1016/j.future.2008.06.010 [Accessed June 9th 2016]

[3] Wolstencroft, K., Fisher, P., De Roure, D., Goble, C. (2009). Scientific Workflows. *OpenStax CNX*. Available from http://cnx.org/contents/8fedc2ef-93a4-4f55-9bca-52820a2586b7@3 [Accessed June 9th 2016]

[4] Oinn, T., Addis, M., Ferris J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover K., Pocock, M. R., Wipat, A. & Li, P. (2004). Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17), 3045-3054. Available from: http://doi.org/10.1093/bioinformatics/bth361 [Accessed 11th February 2016]

[5] Goecks, J., Nekrutenko, A., Taylor, J., & The Galaxy Team (2010). Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86. Available from: http://doi.org/10.1186/gb-2010-11-8-r86 [Accessed 12th February 2016]

[6] Blankenberg, D., Von Kuster, G., Coraor, N., Ananda, G., Lazarus, R., Mangan, M., Nekrutenko, A., & Taylor, J. (2010). Galaxy: a web-based genome analysis tool for experimentalists. *Current Protocols in Molecular Biology*, Chapter 19:Unit 19.10.1-21. Available from: http://doi.org/10.1002/0471142727.mb1910s89 [Accessed 12th February 2016]

[7] Giardine, B., Riemer, C., Hardison, R. C., Burhans, R., Elnitski, L., Shah, P., Zhang, Y., Blankenberg, D., Albert, I., Taylor, J., Miller, W., Kent, W. J., & Nekrutenko, A. (2005). Galaxy: a platform for interactive large-scale genome analysis. *Genome Research*, 15(10):1451-5. Available from: http://doi.org/10.1101/gr.4086505 [Accessed 12th February 2016]

[8] Hoon, S., Ratnapu, K. K., Chia, J., Kumarasamy, B., Juguang, X., Clamp, M., Stabenau, A., Potter, S., Clarke, L., & Stupka, E. (2003). Biopipe: A Flexible Framework for Protocol-Based Bioinformatics Analysis. *Genome Research*, 13(8), 1904–1915. Available from: http://doi.org/10.1101/gr.1363103 [Accessed 11th February 2016]

[9] Leo, P., Marinelli, C., Pappada, G., Scioscia, G. & Zanchetta, L. (2004). BioWBI: an Integrated Tool for building and executing Bioinformatic Analysis Workflows.

Bioinformatics Italian Society Meeting (BITS 2004), Padova. Available from: http://bioinformatics.hsanmartino.it/bits_library/library/00079.pdf [Accessed 11th February 2016]

[10] Tang, F., Chua, C. L., Ho, L.-Y., Lim, Y. P., Issac, P., & Krishnan, A. (2005). Wildfire: distributed, Grid-enabled workflow construction and execution. *BMC Bioinformatics*, 6, 69. Available from: http://doi.org/10.1186/1471-2105-6-69 [Accessed 12th February 2016]

[11] Lu, Q., Hao, P., Curcin, V., He, W., Li, Y.-Y., Luo, Q.-M., Guo, Y.-K., & Li, Y.-X. (2006). KDE Bioscience: Platform for bioinformatics analysis workflows, *Journal of Biomedical Informatics*, 39(4):440-450, Available from: http://doi.org/10.1016/j.jbi.2005.09.001 [Accessed 12th February 2016]

[12] Romano, P., Bartocci, E., Bertolini, G., De Paoli, F., Marra, D., Mauri, G., Merelli, E., & Milanesi, L. (2007). Biowep: a workflow enactment portal for bioinformatics applications. *BMC Bioinformatics*, 8(Suppl 1), S19. Available from: http://doi.org/10.1186/1471-2105-8-S1-S19 [Accessed 11th February 2016]

[13] Bartocci, E., Corradini, F., Merelli, E., & Scortichini, L. (2007). BioWMS: a web-based Workflow Management System for bioinformatics. *BMC Bioinformatics*, 8(Suppl 1), S2. Available from: http://doi.org/10.1186/1471-2105-8-S1-S2 [Accessed 12th February 2016]

[14] Reuillon, R., Leclaire, M., & Rey-Coyrehourcq, S. (2013). OpenMOLE, a workflow engine specifically tailored for the distributed exploration of simulation models. *Future Generation Computer Systems*, 29(8):1981-90. Available from: http://doi.org/10.1016/j.future.2013.05.003 [Accessed 30th December 2015]

[15] Van Horn, J. D., Toga, A. W. (2009). Neuroimaging workflow design and data-mining: a Frontiers in Neuroinformatics special issue. *Frontiers in Neuroinformatics*, 3(31). Available from: http://doi.org/10.3389/neuro.11.031.2009 [Accessed 8th February 2016]

[16] Wilensky, U. (1999). NetLogo. Available from: http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. [Accessed 7th June 2016]

[17] Wilensky, U. (1997). NetLogo Fire model. Available from: http://ccl.northwestern.edu/netlogo/models/Fire. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. [Accessed 7th June 2016]

[18] Fissell, K. (2007). Workflow-based approaches to neuroimaging analysis. *Methods in Molecular Biology*, 401(235-266). Available from: http://doi.org/10.1007/978-1-59745-520-6_14 [Accessed 21st May 2016]

[19] Oinn, T., Addis, M., Ferris, J., Marvin, D., Greenwood, M., Goble, C., Wipat, A, Li, P., Carver, T. (2004). Delivering web service coordination capability to users. *WWW2004: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, ACM 2004:438-439. Available from: http://doi.org/10.1145/1013367.1013514 [Accessed 1st March 2016]

[20] Freefluo Enactment Engine (2008). [Online], Available from: http://freefluo.sourceforge.net [Accessed 3rd March 2016]

[21] Karasavvas, K., Wolstencroft, K., Mina, E., Cruickshank, D., Williams, A., De Roure, D., Goble, C., & Roos, M. (2012). Opening new gateways to workflows for life scientists. *HealthGrid Applications and Technologies Meet Science Gateways for Life Sciences*, 175:131-141. Available from: http://doi.org/10.3233/978-1-61499-054-3-131 [Accessed 22nd February 2016]

[22] Brooks, C., Lee, E. A. (2010). Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java. Poster presented at the 2010 Berkeley EECS Annual Research Symposium (BEARS).

[23] Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S. (2004). Kepler: An Extensible System for Design and Execution of Scientific Workflows. *SSDBM2004: Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, Available from: http://doi.org/10.1109/SSDM.2004.1311241 [Accessed 10th April 2016]

[24] Ludascher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., Zhao, Y. (2006). Scientific workflow management and the Kepler system: Research articles. *Concurrency and Computation: Practice & Experience*, 18(10): 1039-1065. Available from: http://doi.org/10.1002/cpe.v18:10 [Accessed 16th April 2016]

[25] Common Workflow Language Group (2014). *Common Workflow Language* (Version 3) [Software Specification]. Available from: http://www.commonwl.org/ [Accessed 26th May 2016]

[26] Fisher, R. (1971). *The design of experiments.* Reprinted 8th Edition. New York, Hafner Publishing Company. Available from: http://www.phil.vt.edu/dmayo/PhilStatistics/b%20Fisher%20design%20of%20experiments.pdf [Accessed 5th June 2016]

[27] Madsen, H. (2000). Automatic calibration of a conceptual rainfall-runoff model using multiple objectives. *Journal of Hydrology*, 235:276-288. Available from: https://doi.org/10.1016/S0022-1694(00)00279-1 [Accessed 5th June 2016]

[28] Harp, D. R., Vesselinov, V. V. (2012). An agent-based approach to global uncertainty and sensitivity analysis. *Computers and Geosciences*, 40:19-27. Available from: https://doi.org/10.1016/j.cageo.2011.06.025 [Accessed 5th June 2016]

[29] Mouret, J., Clune, J. (2012). Uncovering phenotype-fitness maps using MOLE. Proceedings of the 13th Conference on Artificial Life, East Lansing, MI. Available from: http://jeffclune.com/publications/MouretClune-MolePoster.pdf [Accessed 5th June 2016]

[30] Aubin, J.-P., Bayen, A. M., Saint-Pierre, P. (2011). *Viability theory: new directions.* 2nd Edition. B Heidelberg, Springer. Available from: https://doi.org/10.1007/978-3-642-16684-6 [Accessed 5th June 2016]

[31] Clark Evans et al (2001) *YAML* (Version 1.2) [Software Specification]. Available from: http://www.yaml.org/spec/1.2/spec.html [Accessed 1st May 2016]

[32] Common Workflow Language Group (2016). *Semantic Annotations for Linked Avro Data* (Version 1) [Software Specification]. Available from: http://www.commonwl.org/ [Accessed 26th May 2016]

[33] Tim Dalton (2009) *scala-yaml* [Software]. Available from: https://github.com/daltontf/scala-yaml [Accessed 11th April 2016]

[34] Andrey Somov (2009) *SnakeYAML* (Version 1.17) [Software]. Available from: https://bitbucket.org/asomov/snakeyaml [Accessed 11th April 2016]

[35] Li Haoyi (2014) *FastParse* [Software]. Available from: https://github.com/lihaoyi/fastparse [Accessed 28th May 2016]