How do users design scientific workflows? The Case of Snakemake

Sebastian Pohl[†], Nourhan Elfaramawy[†], Kedi Cao[†♯], Birte Kehr[♯], and Matthias Weidlich[†]

[†]Humboldt-Universität zu Berlin, Germany [♯]Leibniz Institute for Immunotherapy, Regensburg, Germany
{sebastian.pohl,nourhan.elfaramawy,matthias.weidlich}@hu-berlin.de {kedi.cao,birte.kehr}@klinik.uni-regensburg.de

ABSTRACT

Scientific workflows automate the analysis of large-scale scientific data, fostering reuse of data processing operators as well as reproducibility and traceability of analysis results. In exploratory research, however, workflows are adapted continuously, using a wide range of tools and software libraries, to test scientific hypothesis. Script-based workflow engines cater for the required flexibility through a direct integration of programming primitives, but lack abstractions for an interactive exploration of the workflow design by a user during workflow execution. To derive requirements for such interactive workflows, we conduct an empirical study on the use of Snakemake, a popular Python-based workflow engine. Based on workflows collected from 1602 Github repositories, we present initial insights on common structures of Snakemake workflows, as well as the language features typically adopted in their specification.

KEYWORDS

Scientific workflows, workflow design, user interactions

1 INTRODUCTION

Scientific workflows define series of discrete programs to automate the analysis of large-scale scientific data [3]. Traditionally, models and systems for scientific workflows have been introduced with a focus on the reuse of standardized data processing operators, as well as the reproducibility and traceability of analysis results. Workflow engines such as Kepler [9] and Galaxy [5] provide libraries of standard operators, include collaboration features, and facilitate the execution of workflows on various technical infrastructures. However, they only provide limited support for exploratory research, in which workflows are designed to assess scientific hypotheses. Here, workflows are subject to continuous change and a flexible integration of existing tools and software libraries is important. Script-based workflow engines, such as Snakemake [7] and Nextflow [11], offer the required flexibility, but still focus on the specification of a workflow at design-time and lack the abstractions needed to explore the workflow design at run-time. Although Snakemake enables the integration of Python notebooks, interactions in these notebooks are decoupled from the workflow definition. Hence, users cannot steer the execution of a workflow based on the insights obtained from intermediate or partial results.

In order to design models and systems for interactive workflows, however, we first need to develop an understanding of the *what* and *how* of workflow design in practice:

(1) What are common properties of scientific workflows?
(2) How are the workflows typically specified?

Answers to the first question shed light on the conceptual requirements for a model for interactive workflows, e.g., in terms of actions to apply to a workflow at run-time. The second question, in turn, focuses on the realization of such a model in the context of a particular workflow engine, e.g., in terms of the language constructs that shall be augmented.

Given the importance of the above questions for effective user support, not only for interactive workflows for exploratory research, but for workflow design in general, it is striking that there exist only a few studies that aim at addressing them empirically. Notably, existing work focused on structural properties of traditional workflows based on reusable operators [12, 8] and also derived abstract categorizations of data processing operators and high-level design principles [4]. Yet, there is a research gap, framed by the need to derive conclusions on the structural properties of script-based workflows, as well as the language features adopted in their specification.

In this work, we set out to study the questions of *what* and *how* in the design of scientific workflows for the case of Snakemake [7], a popular Python-based workflow engine. Our starting point has been the Snakemake Workflow Catalog,[1] a listing of more than 2,000 public workflows, mostly in bioinformatics, but also spanning other disciplines, such as astrophysics and Earth-scale infrastructure simulation. Based thereon, we have been able to collect workflows from 1602 Github repositories to analyze the structure of the graphs derived for execution as well as the frequency with which certain language features are used.

Conference'17, July 2017, Washington, DC, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[1] https://snakemake.github.io/snakemake-workflow-catalog/

In the remainder, we first review related work (§2) and provide background information on Snakemake (§3). We then elaborate on how we obtained the collection of workflows (§4), before providing initial insights from its analysis (§5). We conclude with a discussion of our observations (§6).

2 RELATED WORK

Scientific workflow systems, such as Kepler [1], Galaxy [5], Taverna [13], or Pegasus [2] provide rich infrastructures and ecosystems to support users in their data-intensive analysis tasks. Recent script-based workflow engines, such as Snakemake [7] and Nextflow [11], in turn, focus on the flexible combination of existing tools and software libraries, as often required in exploratory research. However, empirical analysis of the use of these systems is scarce. Anecdotal evidence on how users design workflows is available in the form of case studies, e.g., for Kepler workflows in the BioEarth project [10]. Small collections of workflows have also been analyzed to derive common performance characteristics [6].

Only a few studies aimed at more general insights by considering large collections of workflows. Notably, around 400 Taverna workflows from the myExperiment repository have been analyzed in [12], focusing on their structural properties. For instance, it was observed that the majority of operators (57%) were implemented directly by the engine, whereas only (14%) accounted for dedicated scripts. Therefore, the workflows in this collection represent relatively standardized data processing tasks. A similar study based on the same repository at a later point in time revealed an increase in workflow complexity [8]. The authors further noticed that workflows contained a large number of data transformation operators needed to integrate existing tools. This observation may be interpreted as hinting at an increasing use of workflows for less standardized analysis tasks.

Aiming at requirements for an abstract classification of workflows, a collection of 260 workflows has been investigated in [4] with the goal to identify their commonalities. Based on these workflows from various scientific domains, the study devised a collection of motifs, abstract categories for (i) data processing operators (e.g., data retrieval and data visualization) and (ii) the design of workflows (e.g., composition of workflows or manual tasks in workflows).

We conclude that existing studies on collections of workflows did not consider structural properties of script-based workflows as well as the use of particular language features.

3 BACKGROUND ON SNAKEMAKE

Workflow design. Snakemake is a Python-based workflow engine [7]. In Snakemake, workflows are defined in a so-called 'snakefile', which includes rules that capture the logical operators of data processing. A rule typically defines three main parts: the input files, the output files, and the program to derive the output files from the input files. The program referenced in a Snakemake rule can be any `shell` command, a `run` statement with plain Python code, an external `script` (Python, R, Markdown), or a `wrapper` for some script defined in Snakemake's internal repository. This way, Snakemake provides several mechanisms to integrate existing tools or software libraries in the workflow.

To apply a rule to multiple sets of input files, Snakemake offers an `expand` function that produces input file specifications, essentially deriving all combinations of its arguments. More flexible control is achieved by `input functions`, i.e., Python functions to select input files. Such a function takes a `wildcard` as input to guide the selection of files.

To support conditional execution, a rule may be declared to be a `checkpoint`, which means that `input functions` are re-evaluated whenever a physical job created for the `checkpoint` finished execution. A rule may further include execution-related parameters, such as the number of threads and the amount of memory to use, or a path to a Conda environment. The latter enables users to specify a unique software environment per rule.

Snakemake offers four different ways to modularize the design of a workflow: An `include` statement enables the separation of a workflow definition over several files. Workflows may also be combined via a `module` statement, which facilitates the reuse of rules among workflows. Through a `wrapper`, as mentioned above, external scripts may be integrated from a dedicated repository. Finally, the concept of a `subworkflow` enables the specification of preliminary steps in data processing, i.e., a `subworkflow` will run before the parent workflow to prepare files needed for the execution of the latter.

Moreover, users may store the configuration of a workflow in dedicated files (in JSON or YAML format). They are structured as dictionaries of parameter keys and values that can be accessed through a workflow's global variable `config`.

Workflow execution. To execute a workflow, Snakemake needs a target rule, which is given explicitly or assumed to correspond to the first rule in the snakefile. Based thereon, Snakemake derives a set of physical jobs by instantiating each logical rule for each set of `input` files (specified directly or computed by an `input function`) that is needed to eventually compute the input of the target rule. Constraints on the execution of the workflow are captured by a directed acyclic graph (DAG), in which the nodes represent physical jobs and the directed edges model data dependencies.

A key feature of Snakemake is the abstraction provided for the execution of jobs. That is, jobs can be executed locally or using a distributed compute infrastructure. Thus, users may scale up their experiments from a workstation to compute clusters without any modifications to the workflow.

Moreover, Snakemake provides basic support for user interactions during workflow execution through Jupyter notebooks. When executing jobs of a rule that is assigned a `notebook` statement, a notebook is started and opened in a web browser. While this enables users to explore and visualize the available data files, it does not provide abstractions to steer or adapt the execution of the workflow. Once the notebook is closed, the execution simply continues according to the DAG that was constructed initially.

4 A COLLECTION OF WORKFLOWS

Our analysis is based on the Snakemake Workflow Catalog,[2] an automatically generated and continuously growing collection of publicly available Snakemake workflows. Using this catalog, we collected data in two ways:

Run cloned repositories. At the time of data collection, we have been able to extract 1602 Github repositories from the catalog. We have been able to clone 1570 of these 1602 repositories. For each cloned repository we attempted to run Snakemake with a flag to build the respective DAG (*--dag*) in its main folder. Our data collection, therefore, was based on the assumption that each repository corresponds to one workflow and that Snakemake would be able to detect the main snakefile in the root directory of the repository.

For 362 out of 1602 repositories, Snakemake successfully ran with the above flag. In the other cases, it failed, for example due to missing files or specification errors. If successful, Snakemake returned the DAG of the workflow in the DOT graph description language. Our analysis of workflow structures in the remainder, therefore, is based on a collection of 362 DAGs, which we were able to construct in this way.

Query Github repositories. We also collected data regarding the use of Snakemake's language features in these repositories. To this end, we queried Github, collected the source code of Snakemake workflows, and parsed the code line by line to search for specific key words. This part of our analysis exploits the data for 1431 of the 1602 repositories, i.e., all repositories for which the snakefile could be queried directly at Github, i.e., without cloning the repository.

Specifically, our text search starts with the snakefile in the root directory of the repository. To cover as much of the source code as possible, we then also attempted to resolve `include` statements recursively. For 3436 out of the 3550 encountered `include` statements, such a resolution was possible, i.e., we have been able to parse the referenced file.

5 ANALYSIS OF THE WORKFLOWS

Our analysis of the workflow collection focused on the two aforementioned questions on the *what* and *how* of workflow design, exploring (i) structural properties of the workflows (§5.1), and (ii) the used language features (§5.2).

[2] https://github.com/snakemake/snakemake-workflow-catalog

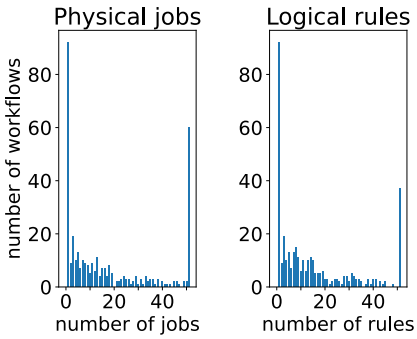

Figure 1: # Rules / jobs per workflow (362 DAGs).

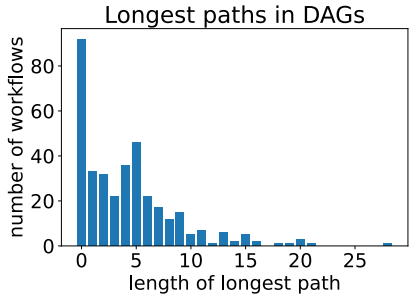

Figure 2: Longest paths in workflows (362 DAGs).

5.1 Structure of the Workflows

To understand the common structure of Snakemake workflows, we study the granularity with which the analysis task is defined and the presence of specific flow patterns.

Granularity. We start with an assessment of the overall size of the workflows in terms of the logical rules and physical jobs, which provides clues on the granularity at which the analysis task is specified in a workflow. For the 362 repositories, and hence workflows, for which a DAG could be generated by Snakemake, the size distributions are given in Fig. 1. Ignoring a considerable number of degenerated workflows with one rule, the results generally suggest that most workflows comprise up to 20 rules. However, there exists also a significant number of very large workflows, with more than 50 rules. Comparing the distributions of logical rules and physical jobs, there is a notable, but not huge shift, which suggests that many rules are instantiated only once (as analyzed in more detail later). However, there are also exceptional cases that yield a relatively high number of workflows with more than 50 jobs.

Flow patterns. Scientific workflows can often be traced back to a few common flow patterns, i.e., sequencing of programs, repetitive behaviour, and parallelism.

Figure 3: DAG of the snakemake workflow from https://github.com/robinmeyers/atac-encode-snakemake.

```
def cromwell_inputs(wildcards):
  inputs={'json': os.path.join("jsons",wildcards.is_grouped+wildcards.condition+".json")}
  if (wildcards.is_grouped):
    inputs['tagalign']=os.path.join("results/groups/",wildcards.condition,wildcards.condition+".grouped.tagAlign.gz")
  return inputs
```

Listing 1: `Input function` from the workflow in Fig. 3.

Sequences: The sequentiality in terms of the longest paths of physical jobs in the workflows is illustrated in Fig. 2. Here, the majority of paths is shorter than 10. Combining this result with the sizes of workflows in terms of the total number of physical jobs, see Fig. 1, this hints at a significant amount of jobs that do not have a direct data dependency and, therefore, could be executed independently.

Repetitions: To analyze repetitions in workflows, we conducted a depth first traversal of the DAGs, from the root to all leaf nodes, and recorded any path that contained at least two physical jobs of the same logical rule. We found six DAGs that showed such repetition; only one of them included a rule that was repeated more than two times.

One example is the workflow in Fig. 3, in which the rules A and B are apparently applied repeatedly, the first four jobs per rule relate to individual data samples, whereas the fifth job per rule processed the merged results of the samples. Interestingly, a closer inspection reveals that only the first rule is in fact repeated. The second one relies on the `input function` defined in Listing 1. It leverages a `wildcard`, which is often used to simplify the application of a rule to a large number of input files, to control the behaviour of rule. That is, depending on the binding of the `wildcard`, the `input function` either selects individual JSON files as inputs or a JSON file of grouped results from an earlier execution of the rule. As such, the construction is used to introduce statefulness to this particular rule, illustrating the high degree of flexibility in script-based workflow specifications.

Parallelism: Physical jobs that are independent may be executed concurrently. However, such independent jobs may represent data parallelism, in which a single logical rule is instantiated for various sets of input files, or task parallelism, in which certain files are taken as input by multiple different logical rules. For instance, turning to Fig. 3, we observe data parallelism for the jobs of rule X and task parallelism for the jobs of rules Y and Z.

To shed light on the general presence of either type of parallelism in our workflow collection, Fig. 4 illustrates the average and maximal in-degrees of logical rules and physical jobs, respectively. Again, neglecting the workflows with an average in-degree of zero (mostly degenerated workflows with a single rule or job), we see that most workflows have an average in-degree around one for rules and jobs. However, a large number of workflows also have a maximal in-degree larger than one for rules, hinting at task parallelism. The distribution for the maximal in-degree for jobs, in turn, is notably right-shifted. This difference provides evidence for the presence of data parallelism, with a few workflows having jobs with an in-degree of up to 1440.

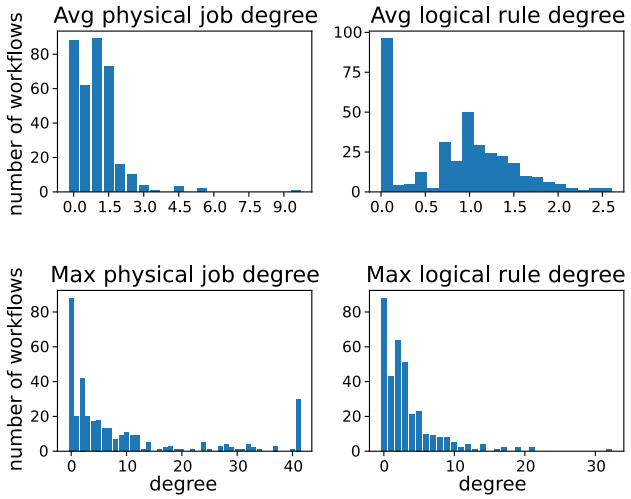

Figure 4: Avg and max in-degrees of rules / jobs (362 DAGs), grouping degrees >10 (avg) and >40 (max).

Figure 5: Ratios of rules and jobs (362 DAGs).

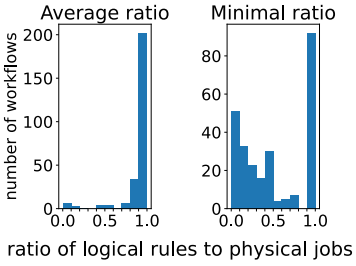

Figure 6: Avg and min ratio of rules and jobs per workflow (362 DAGs).

An alternative view on data-parallelism is provided in Fig. 5, which shows the ratios of logical rules and their corresponding physical jobs, over all rules. While a large number of logical rules is instantiated once, many rules also lead to several jobs. The distribution of these ratios over the workflows is illustrated in Fig. 6 in terms of average and minimal values. Here, dozens of workflows show low minimal ratios, i.e., high data-parallelism for at least one rule.

5.2 Language Usage

Next, we focus on the use of Snakemake language features as observed in the workflow collection.

Modularization primitives. First, the use of the modularization concepts in Snakemake provides clues on the strength of the coupling of parts of a workflow. For the whole collection of 1431 repositories queried at Github, Table 1 illustrates that `include` statements and `wrappers` are very common, occurring in around a third of the workflows and, typically, many times per workflow. Grouping of rules into `modules` is less common, while `subworkflows` are rarely used. We interpret these results such that users often manage complexity through modularization within the context of a single workflow, but rarely encapsulate functionality for explicit reuse.

Table 1: Use of modularization concepts (1431 repos).

	`include`	`wrapper`	`module`	`subworkflow`
Number of repos	679	540	200	10
Number of instances	3550	1769	415	20

Configuration. Snakemake's configuration mechanism turns out to be widely used: 712 of the 1431 repositories contain a total of 1303 non-empty configuration files. The size of these configuration files varies significantly with an average of 64 lines, a median of 32 lines, and a 75th-percentile of 71 lines.

Operators. As explained above, Snakemake supports several ways to define the program of a logical rule. In Table 2, we outline the number of occurrences of the respective keywords in the 1431 repositories, along with the average number of lines of the instructions following these keywords to gauge the average rule complexity. Note that 5676 of the 17330 rules contained more than one of the keywords. In general, most rules define a direct execution of a `shell` command. Yet, all program types are used to some extent.

Table 2: Program types for 17330 rules (1431 repos).

	`shell`	`run`	`script`	`wrapper`
Number of rules	13144	3426	2979	1869
Avg number of lines	5.247	9.531	2.330	2.272

Dynamic execution. Next, we consider the use of the language features for dynamic workflow execution. Here, we first note that the `expand` function to construct sets of input files is used frequently. An extreme case is illustrated in Listing 2, which includes the use of an `expand` function to generate jobs based on the combination of various parameters. As a consequence, a job with an in-degree of 1440 is created,[3] the largest observed among all our DAGs.

[3]Rule *tabulate_sim_scores* from https://github.com/gitter-lab/ssps.

```
input:
  scores=expand(SCORE_DIR+"/{method}/v={v}_r={r}_a={a}_t
    ={t}_replicate={rep}.json",
  method=SIM_METHODS, v=SIM_GRID["V"], r=SIM_GRID["R"],
  a=SIM_GRID["A"], t=SIM_GRID["T"], rep=SIM_REPLICATES)
```

Listing 2: Expand function in one of the workflows.

Turning to the selection of input files via `input functions`, we find a total of 68 occurrences in 42 of the 1431 repositories. A similar observation is made for `checkpoints`, i.e., there are 251 occurrences in 94 of the 1431 repositories. Arguably, input files can be expected to be selected directly in most cases. Yet, our numbers also illustrate a certain need for conditional execution of workflows.

Finally, we observe 47 occurrences of the `notebook` statement in 23 of the 1431 repositories. We conclude that this feature is not (yet) widely used, potentially due to its limited integration in workflow execution.

6 SUMMARY AND DISCUSSION

In this paper, we explored a large collection of Snakemake workflows in order to better understand the *what* and *how* of script-based workflow design. This analysis constitutes a preliminary step to develop models and methods for interactive workflows to effectively support exploratory research. In this regard, our results lead to several conclusions, as follows:

First, our results on workflow sizes, especially in terms of physical tasks, and on the longest paths in workflows illustrate that most workflows split up the analysis task into several rules that at least partially induce jobs that are executed sequentially. These sequences of jobs denote an opportunity for the integration of interaction features that enable users to assess intermediate results and, based thereon, adapt the execution of the workflow.

Second, we observe that many workflows include at least a single logical rule inducing several physical jobs, i.e., the workflow shows data-parallelism. To support the exploration of workflow design, interactions features that enable users to control this data parallelism can be expected to be valuable. For instance, users may choose to first evaluate a certain part of a workflow based on a few sets of input files, before interactively deciding whether, and to which extent, to increase the processed data volume.

Third, several of our observations hint at a need for dynamic workflow steering. Examples include the use of `input functions` and `checkpoints` as well as the number of rules with `script` program types. However, Snakemake provides only limited support for conditional execution so far, as particularly highlighted by the observed use of an `input function` to introduce statefulness to a rule. An integration of user interactions would provide the basis for a richer set of means to adapt and fine-tune the execution of a workflow.

As a next step, to sharpen the requirements for interactive workflows, we intend to study the evolution of the workflows in our collection. For many repositories, a considerable revision history is available, which may lead to valuable insights into common change operations for scientific workflows.

Acknowledgments. Funded by the German Research Foundation (DFG), Project-ID 414984028, SFB 1404 FONDA.

REFERENCES

[1] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludäscher, and Stephen Mock. 2004. Kepler: an extensible system for design and execution of scientific workflows. In vol. 16. (July 2004), 423–424. ISBN: 0-7695-2146-0. DOI: 10.1109/SSDBM.2004.44.

[2] Ewa Deelman et al. 2015. Pegasus, a workflow management system for science automation. *Future Gener. Comput. Syst.*, 46, 17–35. DOI: 10.1016/j.future.2014.10.008.

[3] Ewa Deelman et al. 2018. The future of scientific workflows. *Int. J. High Perf. Comp. Ap.*, 32, 1, 159–175. DOI: 10.1177/1094342017704893.

[4] Daniel Garijo, Pinar Alper, Khalid Belhajjame, Oscar Corcho, Yolanda Gil, and Carole Goble. 2013. Common motifs in scientific workflows: an empirical analysis. *Future Generation Computer Systems*, 36, (Sept. 2013). DOI: 10.1016/j.future.2013.09.018.

[5] Jeremy Goecks, Anton Nekrutenko, James Taylor, and Galaxy Team team@galaxyproject.org. 2010. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11, 1–13.

[6] Gideon Juve, Ann L. Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. 2013. Characterizing and profiling scientific workflows. *Future Gener. Comput. Syst.*, 29, 3, 682–692. DOI: 10.1016/j.future.2012.08.015.

[7] Johannes Köster and Sven Rahmann. 2018. Snakemake - a scalable bioinformatics workflow engine. *Bioinform.*, 34, 20, 3600. DOI: 10.1093/bioinformatics/bty350.

[8] Richard Littauer, Karthik Ram, Bertram Ludäscher, William Michener, and Rebecca Koskela. 2012. Trends in use of scientific workflows: insights from a public repository and recommendations for best practice. *Int. J. Digit. Curation*, 7, 2, 92–100. DOI: 10.2218/ijdc.v7i2.232.

[9] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew B. Jones, Edward A. Lee, Jing Tao, and Yang Zhao. 2006. Scientific workflow management and the kepler system. *Concurr. Comput. Pract. Exp.*, 18, 10, 1039–1065. DOI: 10.1002/cpe.994.

[10] Tristan Mullis, Mingliang Liu, Ananth Kalyanaraman, Joseph Vaughan, Christina Tague, and J. Adam. 2014. Design and implementation of kepler workflows for bioearth. *Procedia Computer Science*, 29, (Dec. 2014), 1722–1732. DOI: 10.1016/j.procs.2014.05.157.

[11] Paolo Di Tommaso, Evan W Floden, Cedrik Magis, Emilio Palumbo, and Cedric Notredame. 2017. Nextflow, an efficient tool to improve computation numerical stability in genomic analysis. 211, 3. DOI: 10.1051/jbio/2017029.

[12] Ingo Wassink, Paul E. van der Vet, Katy Wolstencroft, Pieter B.T. Neerincx, Marco Roos, Han Rauwerda, and Timo M. Breit. 2009. Analysing scientific workflows: why workflows not only connect web services. In *2009 Congress on Services - I*, 314–321. DOI: 10.1109/SERVICES-I.2009.48.

[13] Katherine Wolstencroft et al. 2013. The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Res.*, 41, 557–561. DOI: 10.1093/nar/gkt328.