Retrospective Provenance Without a Runtime Provenance Recorder

Timothy McPhillips[*] *Khalid Belhajjame*[†] *Bertram Ludäscher*[‡]

Abstract

The YesWorkflow (YW) toolkit aims to provide users of scripting languages such as Python, Perl, and R, with many of the benefits of scientific workflow automation. YW requires neither the use of a workflow engine nor the overhead of adapting or instrumenting code to run in such a system. Instead, YW enables scientists to annotate their scripts with special comments that reveal the main computational blocks and dataflow dependencies otherwise implicit in scripts. YW tools extract and analyze these comments, represent scripts in terms of entities based on a typical scientific workflow model, and provide graphical workflow views (i.e., *prospective provenance*) of scripts. In this paper, we present a new extension of YW for inferring *retrospective provenance* from script executions *without* relying on a runtime provenance recorder (such as noWorkflow). Instead we exploit the common practice of scientists to embed important pieces of provenance in directory structures and file names. For such "provenance-friendly" data organizations, we offer a new annotation mechanism based on *URI templates*. YW uses these to link conceptual-level prospective provenance with data files created at runtime, resulting in a powerful, integrated model of prospective and retrospective provenance. We present scientifically meaningful retrospective provenance queries for investigating an execution of a data acquisition workflow implemented as a Python script, and show how these queries can be evaluated using the YW toolkit.

1 Introduction

Despite the advantages that scientific workflow systems offer, many workflows continue to be implemented using scripting languages and executed outside of workflow management systems. This is due in part to the convenience and familiarity of scripting languages (such as Perl, Python, R, and MATLAB), and to the high productivity many scientists experience when using these languages. The YesWorkflow toolkit [YW15, MSK+15] aims to provide such users of scripting languages with many of the benefits of scientific workflow automation. Instead of requiring users to migrate their scripts to a scientific workflow system, YW provides tools for revealing the computational modules and dataflows otherwise implicit in existing scripts. In the short term, this allows scientists to immediately reap some benefits of workflow automation without any change in their programming environment or code. Longer term we envision that the YW approach will lead to a co-evolution of script-based technologies and workflow tools on the one hand and new ways of dataflow thinking by scientists on the other.

In the following, we develop one part of this vision, i.e., the use of YW to reveal prospective *and* retrospective provenance [ZWF06] from scripts, extending prior work that only provided the former [MSK+15]. In order to use YW, an author marks up scripts using simple, keyword-based annotations. These annotations can be placed within any legal comments within a script, so the YW approach and toolkit are language independent.[1] Software in the YW toolkit extract and analyze these comments, represent the scripts in terms of entities based on a common scientific workflow model, and provide graphical workflow views of scripts. In this way, users of YW-annotated scripts may explore and better understand the expected behavior of scripts before running them, using visualizations similar to those provided by workflow systems [LAB+06, WHF+13, FKCS14]. By publishing these workflow views, e.g., alongside the executable script in a software repository, or in the methods section of a scientific article, other potential users can quickly get a conceptual-level overview of the approach implemented by the script, which in turn facilitates reproducibility and reuse [Gan13, SLP14].

From Prospective to Retrospective Provenance

In addition to prospective provenance, which captures the "recipe" of how data products of a workflow or script are produced, retrospective (or *runtime*) provenance is often required to convince scientists that their programs have indeed executed as expected, or to help debug faulty runs. Many scientific workflow systems offer runtime provenance recorders to capture retrospective provenance [ABJF06, DBE+07, BMR+08]. Similarly, a number of approaches have been developed or are

[*]University of Illinois (UIUC), tmcphillips@absoluteflow.org
[†]LAMSADE, Paris Dauphine University, France
[‡]University of Illinois (UIUC), ludaesch@illinois.edu

[1]YW has been applied to Python, R, and MATLAB scripts already.

emerging that capture runtime provenance from scripts, e.g., noWorkflow [MBC+14] and others [Gan13].

Recorder-Free Retrospective Provenance

Although the initial focus of YesWorkflow has been on providing the prospective provenance benefits of workflow modeling, we have begun implementing a new approach for inferring retrospective provenance using YW, while continuing to *avoid* the need to add any runtime provenance recording capabilities to the system. Our approach is based on the observation that in many science domains, data processing scripts use naming conventions for data resources that already record the essential information about key events that occur at runtime. For workflows that take as input existing, staged sets of files and persist their intermediate and final results to new sets of files, a comparison of file names, directory names, and directory structures for workflow inputs and outputs often reveals much of the relevant computational history of the new files. Indeed, this is how many scientists make sense of the products of a script in the first place.

Our approach to reconstructing provenance using YW exploits this common practice, and yields important runtime provenance without any runtime overhead. The key idea is to let scientists link their conceptual-level data elements via YW-annotations to the "provenance-friendly" data organization they already apply in practice. This is achieved using a new *URI template* annotation that enables script authors to declare how inputs and outputs are named and organized. After script execution, YW can then reconstruct runtime provenance by matching these URI templates with the actual names of data files and subdirectories that were created at runtime, thus linking file-level retrospective provenance with conceptual-level prospective provenance.

Related Work

A complementary approach [D+15] combines prospective YW provenance with retrospective provenance from a runtime recorder such as noWorkflow [MBC+14]. A key difference is that our "YW-only" approach presented here does not incur any runtime overhead, since retrospective provenance is reconstructed only after a script has executed and using only those files that were read and written by the scientist's script anyways. Conversely, the noWorkflow provenance approach used in [D+15] logs additional runtime provenance, typically at a much finer level of granularity. Runtime provenance recording may introduce significant execution overhead when not used with caution. In our experience, important retrospective provenance queries are typically well within the scope of the "provenance-recorder free" model afforded by YW. In Section 4 we give such example queries, inspired by a real-world, production-level workflow [TMG+13].

Finally, a distinctive feature of the annotation-based approach employed by YesWorkflow is that it allows scientists to quickly and easily provide a *model* of their workflow that corresponds to the way they *think* about it. This model is typically not apparent from, and difficult to tease out of, a script or fine-grained runtime provenance model that focuses on function calls and program variables.

The use of YW with languages for personal data spaces [VSDK+07, HBF+09] and workflow modeling languages [Bow12, A+15, F+15] is an interesting topic for future research.

2 YesWorkflow Language

The steps required to use YesWorkflow to reveal the prospective and retrospective provenance of the data products of a script are as follows. The script author begins by annotating the computational blocks and dataflows in the script using expressions of the form `@tag␣value`. Here, `@tag` is one of the recognized YW-keywords, after which a `value` follows, separated by one or more whitespace characters. YW then interprets the embedded, structured comments and builds a simple workflow model of the script. This model represents scripts in terms of scientific workflow entities, i.e., programs, workflows, ports, and channels:

- A *program block* (short: *program* or *block*) represents a computational step in the script that receives input data and produces (intermediate or final) output data. A program is designated in a script by bracketing the relevant code between a pair of `@BEGIN` and `@END` comments. Program blocks are usually visualized as boxes.

- A *port* represents a way in which data flows into or out of a program or workflow. Ports are identified by `@PARAM`, `@IN`, and `@OUT` annotations in the comments. Both `@IN` and `@PARAM` represent inputs to a block; the former indicates *data* to be processed, while the latter implies a *parameter* controlling how the block processes incoming data.

- A *channel* is a connection between an `@OUT` port of a program and an `@IN` or `@PARAM` port of another program. YW infers channels by matching the names of `@IN` and `@OUT` ports within the same workflow. A block that contains one or more channels is considered a *workflow*.

```
 1 # @BEGIN collect_data_set
 2 # @PARAM cassette_id @PARAM accepted_sample @PARAM num_images @PARAM energies
 3 # @OUT sample_id @OUT energy @OUT frame_number
 4 # @OUT raw_image_path @AS raw_image
 5 # @URI file:run/raw/{cassette_id}/{sample_id}/e{energy}/image_{frame_number}.raw
 6 run_log.write("Collecting data set for sample {0}".format(accepted_sample))
 7 sample_id = accepted_sample
 8 for energy, frame_number, intensity, raw_image_path in collect_next_image(
 9                                     cassette_id, sample_id, num_images, energies,
10                                     "run/raw/{cassette_id}/{sample_id}/e{energy}/image_{frame_number:03d}.raw"):
11     run_log.write("Collecting image {0}".format(raw_image_path))
12 # @END collect_data_set
13
14 # @BEGIN transform_images
15 # @PARAM sample_id @PARAM energy @PARAM frame_number
16 # @IN raw_image_path @AS raw_image
17 # @IN calibration_image @URI file:calibration.img
18 # @OUT corrected_image
19 # @URI file:run/data/{sample_id}/{sample_id}_{energy}eV_{frame_number}.img
20 # @OUT corrected_image_path @OUT total_intensity @OUT pixel_count
21     corrected_image_path = "run/data/{0}/{0}_{1}eV_{2:03d}.img".format(sample_id, energy, frame_number)
22     (total_intensity, pixel_count) = transform_image(
23                                                      raw_image_path, corrected_image_path, "calibration.img" )
24     run_log.write("Wrote transformed image {0}".format(corrected_image_path))
25 # @END transform_images
```

Figure 1: YW-annotated fragment of a Python script for data collection from protein crystal samples. YW-annotations `@BEGIN` and `@END` delimit code blocks; `@IN` and `@OUT` tags model relevant input and output data elements of a block; `@PARAM` identifies a block's parameters. `@URI` templates for raw images (line 5) and corrected images (line 19) link conceptual-level data elements such as `raw_image` with runtime resources (data files and their file paths). Executable script code is greyed out to emphasize YW-annotations. A program variable (`raw_image_path`) is highlighted in the code (lines 8, 11,23): aliases (lines 4, 16) are used to link such program-level objects to the scientist's concepts (here: `raw_image`). Full example available from [ML15].

3 Reconstructing Runtime Provenance

New retrospective YW capabilities are enabled when the script author qualifies `@IN` and `@OUT` annotations that refer to external data resources using a *URI* [2] *template* that declares the path to the resource via an annotation of the form `@URI␣`*`template`*. If the name of a resource does not vary between runs of the script, this template is just the path to the file. In general though, URI templates will include one or more *template variables* that distinguish the multiple files (or other data resources) written or read by a script in a particular code block. The script in Fig. 1 shows (lines 4, 5) how a `raw_image` data element is linked to both a program variable `raw_image_path` and to the external data organization via a URI template with variables. Such template variables (e.g., `{sample_id}` and `{energy}`) are enclosed in curly braces.

Following a run of a script, YesWorkflow will use these URI templates to discover the actual resources written or read during the run. By matching the actual file paths with the URI templates, the template variables are bound, thus yielding retrospective provenance information that can be queried subsequently.

3.1 Example: Data Collection from a Set of Protein Crystal Samples

We illustrate our approach using a Python script marked up with YW-annotations. The script simulates a portion of a common data collection workflow used by macromolecular crystallographers to collect X-ray diffraction data from a set of samples at a synchrotron radiation beam line [TMG+13]. Fig. 1 depicts a code fragment, while Fig. 2 shows the complete YW workflow graph.

The script loads previously measured data quality statistics for each sample (in Figure 2, see the block `load_screening_results`) from an input spreadsheet file associated with the sample cassette used to store and transport the samples; rejects samples that do not meet a minimum quality criterion; and calculates an optimal data collection strategy (`calculate_strategy`) for each accepted sample (a data collection strategy here comprises a set of data collection energies and a count of

[2] Uniform Resource Identifier

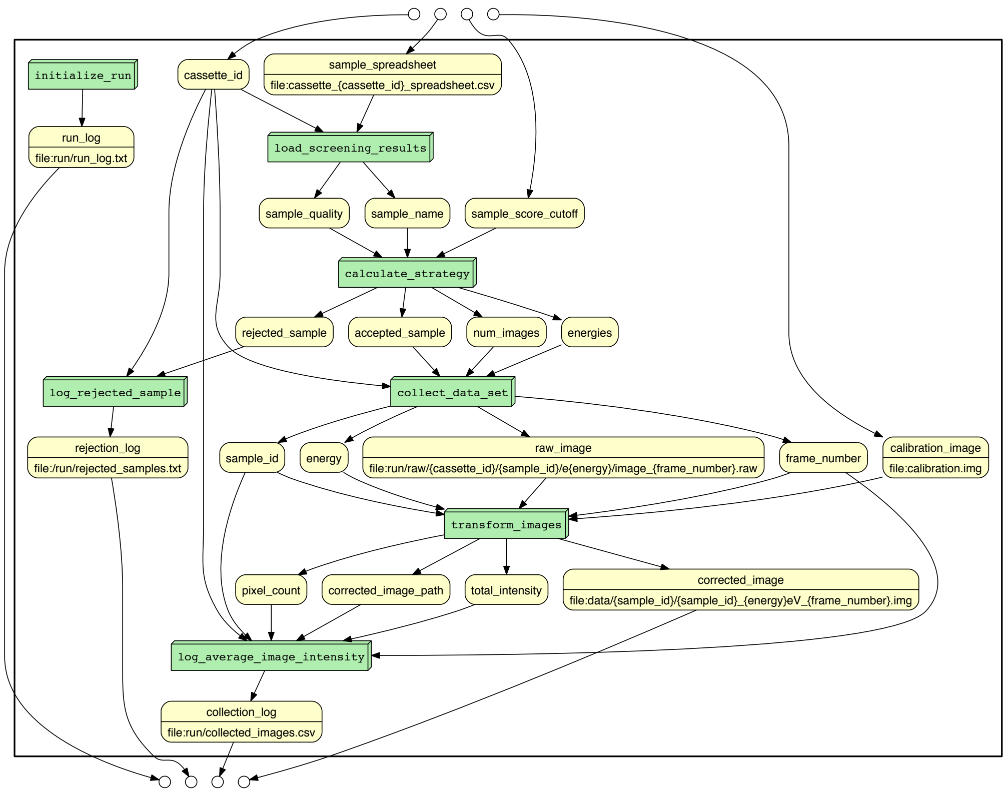

Figure 2: Workflow graph generated from YW-annotations of the data collection script [ML15]. Green boxes represent program blocks annotated in the Python script, while yellow rounded nodes represent data elements that flow between blocks. Available URI templates are depicted in the lower halves of data element nodes in the graph and specify the directory structure and file names for persisting those data.

diffraction images to collect at each energy). The script then collects a series of diffraction images for each accepted sample in turn, saving each raw detector image to the filesystem (`collect_data_set`). The raw images are organized by sample cassette ID, sample name, and X-ray beam energy; images in a sequence collected on the same sample at the same energy are distinguished by a frame number. A subsequent step transforms each raw image to a corrected image using a detector-specific calibration image and saves the resulting corrected images in a different set of output files and directories (`transform_images`).

4 Querying Retrospective Provenance

Our approach to revealing the retrospective provenance of script products in the absence of a provenance recorder is based on a number of observations. Many scientists use directory structures and directory and file names to organize data produced by scripts and to denote their relationships. In particular, when scientists write scripts they often have a set of retrospective provenance queries in mind that are of such high priority that they want to be able to answer them without interpreting the contents of log files. This is especially important when individual log files are independently written by different programs invoked by the single script. The information required to answer these queries is therefore embedded by scientists in filenames, directory names, and in the hierarchical directory structures in which data is organized. Our response to these observations is to let script writers describe this information naturally via URI template expressions. In this way YesWorkflow allows script writers to declare how script inputs and outputs are named and organized based on actual data and metadata values occurring at runtime.

With YesWorkflow we aim to make scientists even more productive by eliminating the need to explore directory structures manually. YW can implement the scientists' high-priority questions as *provenance queries* using the declared URI template information.

We report in this section a number of typical provenance queries that are expressed against our example script. For each query we begin by describing how a scientist would determine the answer by inspecting the names and organization of the files produced by the

```
run/
├── raw
│   └── q55
│       ├── DRT240
│       │   ├── e10000
│       │   │   ├── image_001.raw
...     ... ... ...
│       │   │   └── image_037.raw
│       │   └── e11000
│       │       ├── image_001.raw
...     ...     ...
│       │       └── image_037.raw
│       └── DRT322
│           ├── e10000
│           │   ├── image_001.raw
...         ... ...
│           │   └── image_030.raw
│           └── e11000
│               ├── image_001.raw
...             ...
│               └── image_030.raw
├── data
│   ├── DRT240
│   │   ├── DRT240_10000eV_001.img
... ... ...
│   │   └── DRT240_11000eV_037.img
│   └── DRT322
│       ├── DRT322_10000eV_001.img
...     ...
│       └── DRT322_11000eV_030.img
│
├── collected_images.csv
├── rejected_samples.txt
└── run_log.txt
```

Figure 3: Tree view of the directories and files created by the data collection Python script.

script. We then provide a more general approach to answering each query that does not depend on the exact directory structure and file naming conventions used in the script (while continuing to assume that sufficient information is recorded in such a manner that a scientist could answer the question by hand and in the absence of runtime provenance recording.)

Queries Q_1 and Q_2 represent script run *reports*, i.e., they answer questions that the user may have about the samples used, experimental conditions employed, and results obtained by the script. Queries Q_3 and Q_4, on the other hand, are backward and forward lineage queries, respectively. Q_3 identifies an intermediate data product that a specific final product was derived from; Q_4 determines if there are any intermediate products for which there are no corresponding final products (this is an example of a *why-not* provenance query). Q_5 can be viewed as a data lineage query that reveals the *physical* provenance of a sample (the identity of a cassette that stores a specific sample).

(Q_1) ***What samples did the run of the script collect images from?*** The scientist's solution is to look at the contents of the run/raw/q55 directory (Fig. 3). The names of subdirectories are the names of the samples collected on.

General solution using YW: Assume that 'what samples' means 'what values of sample_id as seen by collect_data_set'. Then the solution is to look for all persisted outputs of collect_data_set that include sample_id in the expanded URI template. Extract the value of sample_id from each and return the set of unique values. Q_1 can be expressed as a Datalog query:

```
samples_used(SampleId) :-
    uri_variable_value(collect_data_set,
                       raw_image_path,
                       sample_id,
                       SampleId).
```

against the provenance facts reconstructed by YW. For our example, this will yield the answers DRT240 and DRT322.

(Q_2): **What energies were used during collection of images from sample DRT322?** The scientist's solution is to look at the contents of the run/raw/q55/DRT322 directory. The names of subdirectories are the values of the energies.

General solution using YW: Assume that 'what energies' and 'from sample DRT322' mean 'what values of energy as seen by collect_data_set when sample_id equals DRT322 as seen by collect_data_set'. Then the solution is to look for all persisted outputs of collect_data_set that include both energy and sample_id in the expanded URI template for the output. Extract the value of energy from each such path for which sample_id equals DRT322 and return the set of unique values.

(Q_3) **Where is the raw image corresponding to corrected image DRT322_11000ev_028.img?** The scientist's solution is to look at the image files nested within the raw directory. Find the image file that contains the "DRT322", "11000", and "028" in the file access path.

General solution using YW: Assume that 'raw image for corrected image' means 'what file output by the port named raw_image with values for URI template variables equal to the matching URI template expansion variables in the path to the file DRT322_11000ev_028.img output by the port named corrected_image'. Then the solution is to extract the URI template variable names and values from the path to DRT322_11000ev_028.img output by the port named corrected_image, look at the paths for all files output by the raw_image port, and return the file whose path includes template variables with names and values matching those for `DRT322_11000ev_028.img` (not all variables need be present in both paths, but where the variable with same name is used the values must match).

(Q_4) **Are there any raw images for which *no* corrected image was written?** This is somewhat similar to Q_3, but follows the lineage in the "forward direction" and (unlike Q_4) asks about the absence of data. In this case the path to each file written by the `raw_image` port is examined, and a corresponding file written by the `corrected_image` port is sought. Return raw images for which no corrected image is found.

(Q_5) **What was the id of the cassette from which sample leading to `DRT240_10000ev_010.img` was taken?** This query shows how the retrospective data lineage information can be used to track the physical provenance of samples. The general solution here is to search the upstream lineage of data provided to `transform_images`, looking for URI templates that include `cassette_id` and `sample_id` as template variables. Return the value of `cassette_id` that occurs in URI expansions where `sample_id` matches `DRT240`.

Example Code. The example Python code, along with the YW-generated prospective provenance shown in Fig. 2, and some of the YW-reconstructed retrospective provenance facts and rules are available from [ML15].

5 Discussion and Conclusions

The idea of using prospective and retrospective provenance for a wide range of applications is not new (see, e.g., [ZWF06, MBZ+08, FMS08]). It is widely known that by employing scientific workflow systems, users can benefit from prospective provenance (through workflow descriptions) and retrospective provenance (from runtime provenance recording) information. On the other hand, it is much less well known that an annotation-based approach such as YesWorkflow can be used to obtain prospective provenance and—as illustrated in this paper—retrospective provenance as well, without the use of a runtime provenance recorder. We have implemented and continue to improve the YW toolkit [MSK+15, YW15]. Using simple YW-annotations, a scientist can use our toolkit to easily and quickly[3] create a high-level dataflow model for a script-based workflow. Since YW-annotations are embedded as comments in the host language, our approach is language-independent, and can be applied to any of the usual scripting or programming languages.

[3]The first author of [BK14] reported that creating the annotations for the YW-model depicted in [MSK+15] (for an R script used in [BK14]) took only half an hour.

In this paper, we also have shown that retrospective provenance from scripts can be easily obtained and combined with prospective provenance using new YW-annotations with URI templates. For this simple approach to work, we make the (realistic) assumption that scientists organize their data using directories for staging and collecting data files: e.g., Bowers *et al.* [BMWL07] includes a detailed account of how a metagenomics researcher organizes his data and results in a nested folder scheme. Moreover, the URI template approach for declaring the layout of data persisted by a workflow run in nested directories has been employed by the RestFlow scientific workflow system [TMG+13]. Despite being a grass-roots effort that was launched only recently, YW has already been successfully applied to real-world, script-based workflows in R, MATLAB, and Python; application domains include climate modeling, bioinformatics, and archeology [MSK+15].

In future work, we will explore other community efforts to workflow modeling such as the Common Workflow Language [A+15] and the Workflow Description Language [F+15] for use in YesWorkflow.

References

[A+15] P. Amstutz et al. Common Workflow Language. github.com/common-workflow-language, 2015.

[ABJF06] I. Altintas, O. Barney, and E. Jaeger-Frank. Provenance collection support in the kepler scientific workflow system. In *IPAW*, 2006.

[BK14] R. K. Bocinsky and T. A. Kohler. A 2,000-Year reconstruction of the rain-fed maize agricultural niche in the US Southwest. *Nature Communications*, 5, 2014. doi:10.1038/ncomms6618.

[BMR+08] S. Bowers, T. McPhillips, S. Riddle, M. K. Anand, and B. Ludäscher. Kepler/pPOD: Scientific workflow and provenance support for assembling the tree of life. In *IPAW*, 2008.

[BMWL07] S. Bowers, T. McPhillips, M. Wu, and B. Ludäscher. Project histories: Managing data provenance across collection-oriented scientific workflow runs. In *Data Integration in the Life Sciences (DILS)*, pp. 122–138, 2007.

[Bow12] S. Bowers. Scientific workflow, provenance, and data modeling challenges and approaches. *Journal on Data Semantics*, 1(1):19–30, 2012.

[D+15] S. Dey et al. Provenance for Scripting Languages. submitted for publication, 2015.

[DBE+07] S. B. Davidson, S. C. Boulakia, A. Eyal, B. Ludäscher, T. M. McPhillips, S. Bowers,

M. K. Anand, and J. Freire. Provenance in Scientific Workflow Systems. *IEEE Data Eng. Bull.*, 30(4):44–50, 2007.

[F+15] S. Frazer et al. Workflow Description Language. github.com/broadinstitute/wdl, 2015.

[FKCS14] J. Freire, D. Koop, F. S. Chirigati, and C. T. Silva. Reproducibility using vistrails. *Implementing Reproducible Research*, page 33, 2014.

[FMS08] J. Frew, D. Metzger, and P. Slaughter. Automatic capture and reconstruction of computational provenance. *Concurrency and Computation: Practice and Experience*, 20(5):485–496, 2008.

[Gan13] C. Gandrud. *Reproducible Research with R and R Studio*. CRC Press, 2013.

[HBF+09] C. Hedeler, K. Belhajjame, A. A. Fernandes, S. M. Embury, and N. W. Paton. Dimensions of Dataspaces. In *Dataspace: The Final Frontier*, pp. 55–66. Springer, 2009.

[LAB+06] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[MBC+14] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noWorkflow: Capturing and Analyzing Provenance of Scripts. In *Intl. Provenance and Annotation Workshop (IPAW)*, 2014.

[MBZ+08] P. Missier, K. Belhajjame, J. Zhao, M. Roos, and C. Goble. Data lineage model for Taverna workflows with lightweight annotation requirements. In *IPAW*, 2008.

[ML15] T. McPhillips and B. Ludäscher. github.com/yesworkflow-org/yw-recon-example, 2015.

[MSK+15] T. M. McPhillips, T. Song, T. Kolisnik, S. Aulenbach, K. Belhajjame, K. Bocinsky, Y. Cao, F. Chirigati, S. C. Dey, J. Freire, D. N. Huntzinger, C. Jones, D. Koop, P. Missier, M. Schildhauer, C. R. Schwalm, Y. Wei, J. Cheney, M. Bieda, and B. Ludäscher. YesWorkflow: A User-Oriented, Language-Independent Tool for Recovering Workflow Information from Scripts. In *10th Intl. Digital Curation Conference (IDCC)*, London, February 2015. to appear in IJDC. Preprint: arxiv.org/abs/1502.02403.

[SLP14] V. Stodden, F. Leisch, and R. D. Peng. *Implementing reproducible research*. CRC Press, 2014.

[TMG+13] Y. Tsai, S. E. McPhillips, A. González, T. M. McPhillips, D. Zinn, A. E. Cohen, M. D. Feese, D. Bushnell, T. Tiefenbrunn, C. D. Stout, et al. AutoDrug: fully automated macromolecular crystallography workflows for fragment-based drug discovery. *Acta Cryst*, 500:69, 2013.

[VSDK+07] M. A. Vaz Salles, J.-P. Dittrich, S. K. Karakashian, O. R. Girard, and L. Blunschi. iTrails: pay-as-you-go information integration in dataspaces. In *VLDB*, pp. 663–674, 2007.

[WHF+13] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, et al. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research*, 2013.

[YW15] YesWorkflow project site and README. yesworkflow.org/yw-prototypes, 2015.

[ZWF06] Y. Zhao, M. Wilde, and I. Foster. Applying the virtual data provenance model. In *Intl. Provenance and Annotation Workshop (IPAW)*, 2006.