Datatrack: An R package for managing data in a multi-stage experimental workflow

Data Versioning and Provenance Considerations in Interactive Scripting

Philip Eichinski, Paul Roe
Science and Engineering Faculty
Queensland University of Technology
Australia
philip.eichinski@qut.edu.au

***Abstract*—In experimental research using computation, a workflow is a sequence of steps involving some data processing or analysis where the output of one step may be used as the input of another. The processing steps may involve user-supplied parameters, that when modified, result in a new version of input to the downstream steps, in turn generating new versions of their own output. As more experimentation is done, the results of these various steps can become numerous. It is important to keep track of which data output is dependent on which other generated data, and which parameters were used. In many situations, scientific workflow management systems solve this problem, but these systems are best suited to collaborative, distributed experiments using a variety of services, possibly batch processing parameter sweeps. This paper presents an R package for managing and navigating a network of interdependent data. It is intended as a lightweight tool that provides some visual data provenance information to the experimenter to allow them to manage their generated data as they run experiments within their familiar scripting environment, where it may not be desirable to commit to a fully-blown comprehensive workflow manager. The package consists of wrapper functions for writing and reading output data that can be called from within the R analysis scripts, as well as a visualization of the data-output dependency graph rendered within the R-studio console. Thus, it offers benefit to the experimenter while requiring minimal commitment for integration in their existing working environment.**

Keywords—computational science; data provenance; R language; R package, workflow;

I. INTRODUCTION

Exploratory data analysis research is often performed in interactive scripting environments such as R, Matlab, Octave or Python. These are popular because they afford the opportunity for researchers who do not necessarily have a strong software engineering or coding background to interactively manipulate data through a read–eval–print loop (REPL). They offer the flexibility of a programming language while allowing swift manipulation, inspection and visualization of data, enabling researchers to quickly roll-out experiments.

Exploratory data analysis in such scripting environments is often performed in a conceptual "workflow" where several distinct steps of data transformation are involved, each taking one or more data objects, and later outputting some result data to be used as input for a subsequent downstream step.

In computational workflows, recording the data provenance - the dependencies, processes, parameters and people responsible for the creation of the data - is important, as it helps interpretation, verification, or tracing the origin of results [2].

When the research undertaken involves designing these analysis processes, inspecting the results of each step, tweaking code and reprocessing, the workflow is often not run in a single end to end execution, but rather, an individual step may be run in isolation, relying on saved intermediate data.

This necessitates that the input data objects be selected by the user, rather than piped in by the preceding step in the workflow. To make this selection, the user must be informed by the provenance metadata associated with all the potential candidates for selection.

Thus, data provenance is not only important from the point of view of reproducibility and debugging, but also to assist online decision-making during the execution of each step of the workflow. Having this provenance information easily available within the interactive scripting environment is of great benefit.

Existing solutions to workflow data provenance tracking normally involve scientific workflow management systems (SWfMS) such as Vistrails [3] or Kepler [4]. These systems are well suited to distributed systems, collaborative research and batch processing across a wide range of parameters, and are hugely advantageous for automating the cycle of moving data to a supercomputer for analysis or simulation, launching the computational processes and managing the storage of the output [5]. When working within a SWfMS, the workflow is formalized using a workflow language, which may be generated through a graphical user interface, and the management system invokes the processes of the workflow. This means that they are best suited where these processes are mature and stable.

However, exploratory research performed in interactive scripting environments, where the code that performs the analysis is being constantly modified, does not always lend itself to integration with a SWfMS. Even in circumstances

where a SWfMS could be used, it may be resisted by the researcher because integration will often draw the researcher out of the environment that they are comfortable working in and require additional effort to learn a new system. When the research calls for frequent modification to the code that runs the data processing steps, working in an SWfMS means effectively working in two environments possibly with two scripting languages.

Under these circumstances, a solution for automated data provenance recording is required for use in interactive scripting environments that do not use a SWfMS. Some tools have been proposed (discussed in section II), however while they address the aspects of data provenance related to verification and reproducibility, they are not aimed at online user decision-making, as discussed above.

This paper introduces **Datatrack**, a prototype R package that [6] manages the dependencies and versioning of data generated in R. It was created to address the problems with data provenance tracking in the interactive exploratory research performed in our lab. The purpose of Datatrack is not to formalize the workflow but rather to provide a way to automate some record-keeping of data generation and assist the experimenter in selecting the correct input data when executing a process. While it doesn't offer many of the features of most scientific workflow management systems, it has the advantage that it has minimal configuration and operates within the R programming environment, reducing the barrier to entry for researchers.

II. Related Work

With the growing importance of e-science and computational science, standards for provenance have emerged. In computationally intensive science, large amounts of data are generated. Data provenance allows the scientist to determine all necessary information about the input data, processes, computing environments and contributors involved in the generation of data output in order to be able to reproduce it. Broadly speaking data provenance captures the identities and relationships between **what** was created, **how** it was created and **who** played a role in its creation.

Standard models for how this information should be structured have emerged, such as the *Open Provenance Model* (OPM) developed following the International Provenance and Annotation Workshop in 2006 [1] and PROV-DM a more recent standard, endorsed by the W3C in 2012 [7]. Both these standards list three types of core 'objects' that represent the what, how and who of provenance, albeit with differing terminology: respectively, *Artefacts*, *Processes*, *Agents* in the case of OPM and *Entities, Activities and Agents* in the case of PROV-DM. They also define a number of causal relationships between them, such as "was derived from", "was controlled by", "was triggered by" amongst others. The standards specify how to model provenance and not how to implement the model.

Most research into the issues of data provenance has been focussed on Scientific Workflow Management Systems (SWfMS) [8]. There is a huge variety of these SWfMS, but generally they have the following functionality: workflow composition, mapping the workflow onto resources or services, executing the workflow and recording the provenance metadata to allow the final output to be reproduced in the future [5]. As discussed in section I, SWfMS are well suited to large, collaborative, distributed analysis and simulations, a situation where good provenance recording is indeed vital.

Until recently, literature in data provenance tracking *outside* of SWfMS has been scarce. Part of the reason for this may be that research on a scale small enough that a SWfMS is not required means that keeping track of data provenance may be assumed to be trivial. Yet, in our experience, simple, user-friendly tools for automatically maintaining a record of dependencies of data outputs has proved hugely useful.

NoWorkflow [9] is a command line tool for recording detailed provenance metadata from python scripts. Its authors note that outside of a SWfMS, provenance capture is challenging due to the fact that the workflow sequence is encoded by the scripts themselves. NoWorkflow works by using software engineering techniques such as abstract syntax tree analysis, to determine this workflow from the scripts themselves and record provenance information during their execution.

The YesWorkflow [10] toolkit is another tool that offers some of the provenance recording functionality of a SWfMS to users of scripting languages such as R and Python. YesWorkflow allows the scientist to insert annotations as code-comments in the scripts that they write. The toolkit can then parse the codebase and interpret and convert these comments into a form that can be queried to answer questions about data objects created.

These tools share with Datatrack the benefit of minimal intrusion into the programming practices that the user is comfortable with. However, Datatrack comes from a slightly different angle, with emphasis put on provenance information for decision-making during invocation of sections of the workflow in isolation.

III. Motivation Context

The development of Datatrack was motivated by needs arising in the research undertaken at the Ecoacoustics Research Group at the Queensland University of Technology, which partly entails designing automated and semi-automated methods of acoustic analysis for environmental monitoring. These methods may involve several data processing steps such as pre-processing, feature extraction, silence removal, clustering and sample ranking, for example. Each processing step performs a series of operations on the output of one or more previous steps, and outputs one or more of its own data files. The R language has often been chosen within the Eco-acoustic Research Group at QUT because it offers the power and flexibility of a programming language but is easy to learn and provides very quick methods for manipulating and visualizing data.

A change at any particular point along the pipeline will generate new versions of all intermediate data downstream to the change. A new idea by the researcher might mean creating a new process that shares some upstream data as a dependency, and creates new branches downstream. A particular step may

take multiple input data objects. For example, a classifier will read in both a model on which to base the classification as well as the data points to classify. Both of these inputs will have been generated by one or more earlier processes.

Although such a sequence of steps comprises a "workflow", it has not been formally specified in a dedicated workflow language. Fig. 1 shows an example of the kind of workflow used.

The workflow is not necessarily run from start to finish: one step of the workflow may be invoked in isolation. This may be because the code for that step may have been modified, or different parameters used. Running a process at a particular point along the pipeline after such changes should not require regeneration of the upstream data, and therefore it is normally desirable to save all intermediate input/output data. Not only does this 'caching' of intermediate data save the time needed to compute it, but it allows inspection of the output of each step in order to evaluate and improve the process that created it.

With every modification to parameters of any process in the workflow, the number of saved data objects in this interdependent network increases, and keeping track of them can become unwieldy. Obviously, in the absence of automated management, diligent manual recordkeeping would be necessary to ensure reliable reproducibility.

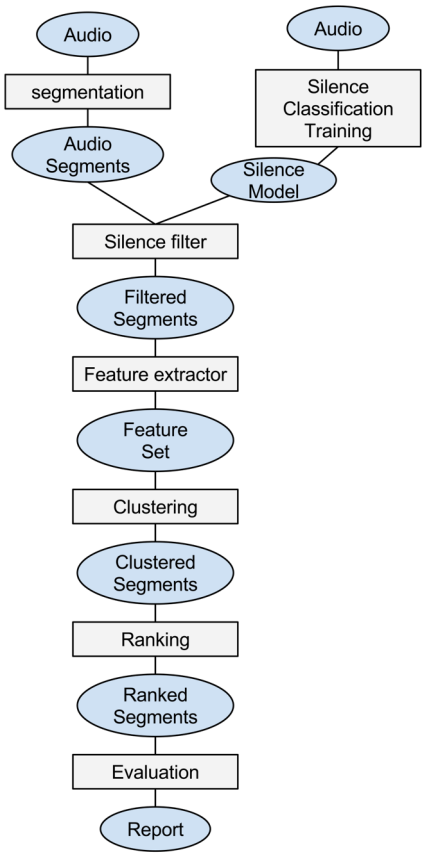

Fig. 1. Example of a workflow for ecological audio analysis experimentation. Rectangles denote processes and ellipses denote data objects that are input and output to these process.

But the primary motivator to the development of Datatrack was to assist selection of input data. When invoking a particular processing step in the workflow, the researcher must decide which data to use as input out of a large number of possible inputs available (generated from variations of upstream processes). Without an easy method for the researcher to ascertain the dependencies of a particular data object, their interaction with their software is less user-friendly. Easily accessible provenance metadata at the time that they are selecting the input data makes this interaction quicker and less error-prone.

Git version control is used to track changes in the codebase over time. While this offers a fair degree of retrospective examination of *how* past results were generated, additional metadata of how the data is related to the code in the repository is required.

IV. Datatrack Framework

In this section, we discuss the prototype package that addresses some of the problems detailed above. The package has been developed iteratively to address specific needs that have arisen during our ecological acoustics research.

A. Overview

From the user's perspective, Datatrack provides wrappers for reading and writing data in R. When these are used instead of the native R functions (such as *write.csv*), additional metadata can be supplied. Datatrack records this supplied metadata as well as some additional information that can be determined from the data and environment. An interactive visual graph representation of the dependencies is available. It is automatically displayed when reading data to allow the user to specify which version of available data objects should be used.

The philosophy behind this simple approach is that the user should not be restricted in their coding style. It does not force the researcher into a particular set of conventions, or to work in an unfamiliar environment.

Notably, there is no formal workflow language or authoring tool. In effect, R is the workflow language as well as the language of the processes that perform the analysis tasks themselves.

Datatrack is completely data-centric. It does not attempt infer the workflow by interpreting source code. Instead it records the data flow, i.e. the dependencies between data objects. This is in line with its goals to provide the necessary information to the user at the time that data is read in.

B. Writing Data

When Datatrack is used to save the output of an analysis process, several pieces of information are supplied to it:

- Name
- Dependencies

- Parameters
- Annotations

The **name** is an arbitrary string used to refer to the data at a later stage. It would typically describe the data object that has been generated or the process that generated it, for example "clustering.result" or "ranked.samples".

Dependencies are a list of names and version-numbers of the data objects that were read in and used in the processing step that created the data being written. This is the mechanism that defines the dependency graph of data output for the workflow.

Parameters are a set of name-value pairs, and refer to the parameters used by the process that created the output data.

Annotations are also name value pairs that the user can supply to be stored as the metadata of the process.

During experimentation, *dependencies* or *parameters* might be altered by the user on separate invocations of a processing step. When writing data of a particular *name*, Datatrack will save a separate version of the output for each unique combination of *dependencies* and *parameters*. The versioning is handled internally to the package. It is not possible to save output data that has the same name, dependencies and parameters as a previous version has, and doing so will result in a user-prompt asking for approval to overwrite the existing file. The distinction between annotations and parameters is that annotations do not affect versioning. Example R code is for writing data is shown in Fig. 2.

C. Reading Data and the Data Dependency Graph

When reading in upstream data in any step of the pipeline, the *name* is specified in a Datatrack function (Fig. 3). There may be many versions of available data objects with a given name, each having been generated with a unique combination of *dependencies* and *parameters*.

Execution is paused until the user inputs the desired *version*, which is chosen from a list of available versions. This is done made with the aid of a visual graph of data dependencies and associated metadata, which is presented to the user at the time data is being read through Datatrack. It is a directed acyclic graph (DAG) where the nodes are data objects, and the edges are dependencies, with the vertical position denoting the direction of the dependencies (dependencies are above the data objects that depend on them). Nodes are grouped by their *name*. The dependency graph is shown for the purpose of assisting the user in their selection. The user can easily see the versions of the potential inputs available, the parameters used when generating them, and the annotations attached to the data, as well their dependencies. The graph also provides the user with convenient access some information that can be derived directly from the data object, such as the column names of saved CSVs, which can also help in their choice of which version to read in.

```
params <- list(k = 240, alg = 'kmeans')

dep <- list(segment_features = seg$version)

datatrack::writeDataobject(mydata,
                     name = 'clustering.1',
                     params= params,
                     dependencies = dep,
                     annotations = list())
```

Fig. 2. Example R code for saving data

```
datatrack::readDataobject('clustering.1')
```

Fig. 3. Example R code for reading data

Because multiple versions of saved data objects are shown simultaneously for comparison, the number of relevant data nodes on the graph can be quite large. This has motivated the design decisions aimed at minimising clutter in the dependency graph.

Fig. 4 shows an example of a graph. In this example, there are three groups of data objects that do not have any dependencies: "weather", "radar.wthr" and "audio". These names are defined by the user as the 'name' argument to the writeDataobject function (Fig. 2). Each of the numbered cells in the groups below the group name is a node of the graph and represents a data object. For instance, in the example shown in Fig. 4 there are six versions of "event.features.1" available, each created with different dependencies and/or parameters. By hovering the mouse over a node, the user can inspect the metadata attached to that version and easily ascertain its dependencies and dependents. In the example, the mouse pointer is hovering over version 1 of "event.features.1" and the direct dependency (version 1 of "events") and indirect dependency (version 1 of "audio") are highlighted. The directly dependent nodes (versions 1 and 3 of "clustering") and indirectly dependent nodes (version 1 of "ranking") are also highlighted. This graph is interactive and is displayed within the viewer of R-Studio (Fig. 5), a popular integrated development for R [11].

Nodes can be filtered by name, hiding nodes that do not have any direct or indirect links to any node in the group with the selected name. Fig. 5 shows the same graph as Fig. 4 but filtered by name "event.features.2". When reading data of a particular name, this filtering is performed to remove nodes that are irrelevant to the choice the user needs to make. Within a group, nodes are ordered by their created date-time, and a range can be specified to filter nodes of the selected group by date-time

Important to note is that the conceptual workflow conceived by the researcher and implemented in R will *resemble* the groups and their relationships in this graph, however the Datatrack data dependency graph is not based directly on any workflow designed by the user, but rather only on the dependencies declared at the time of writing the data through the data object's *name*. No assumptions are made about the workflow, and no attempt at inferring the workflow through code inspection is made. The user is free to choose

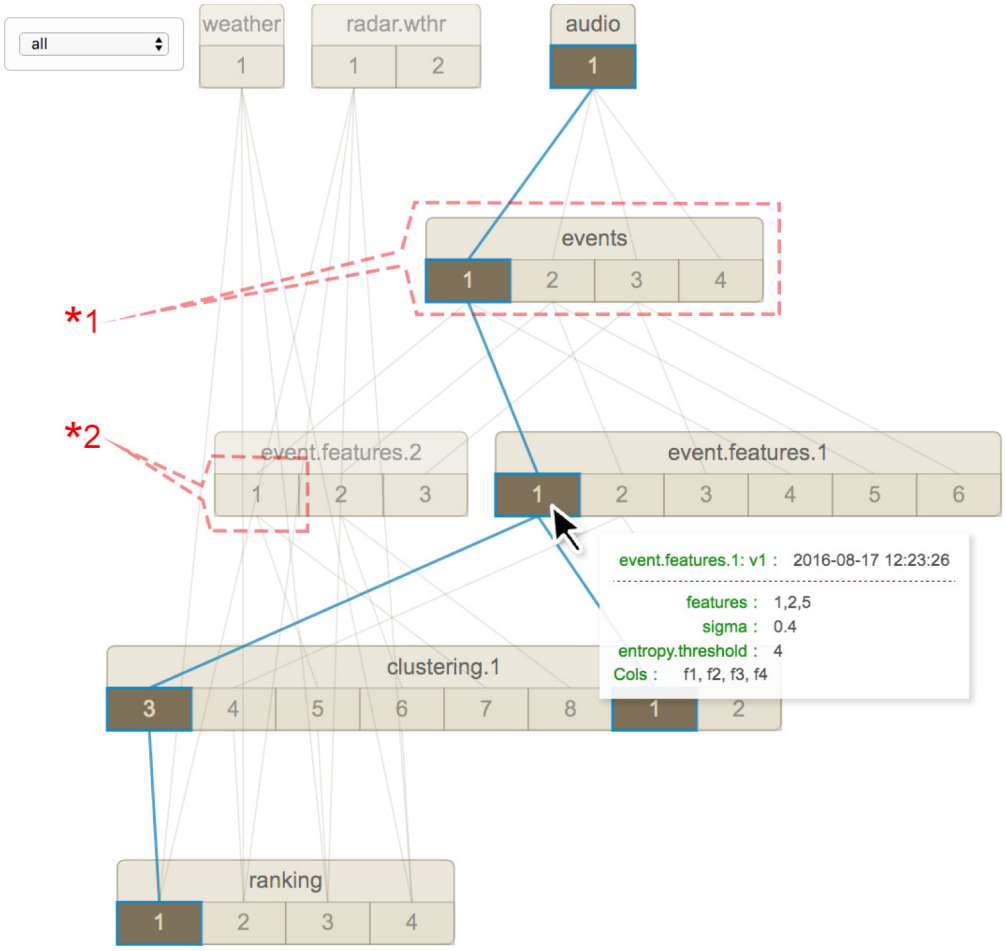

Fig. 4. Example of a graph of data dependencies showing parameters used when generating data.). Each group of nodes (*1) represents a type of data object arbitrarily named by the user. This example shows eight different types of data object. Each node (*2) represents a data object. The nodes are numbered by their version. Dependencies between nodes are represented by the lines. When the mouse hovers over a node, metadata information is for that node is shown, and the dependency chain to and from that node is highlighted. Note: the dashed lines are figure labels and are not part of the graph visualization.

whatever name they like, although normally it would refer in some way to the step of the workflow. This means that the working style of the user is not impacted at all by using Datatrack for reading and writing data.

While giving the user the provenance information at the time that data object is being read in to a process is useful when that step of the workflow is being run in isolation, if multiple steps of the workflow are being executed in sequence or if the workflow is being executed from start to finish, there is no need to prompt the user to select the correct version of data. Unless instructed otherwise, Datatrack will automatically select the last accessed version of a data object of a particular name (read or written), instead of prompting for user selection. This allows an entire workflow of reading and writing data to be run without user input at each step.

D. Considerations and Tradeoffs

Datatrack is not designed to offer all the data provenance features of a Scientific Workflow Management System, because its primary aim is centered around user decision-making assistance and data versioning. In this section, we discuss features and their implications for usability of Datatrack.

1) Tracking of users: the "who" of provenance

The Open Provenance Model and PROV-DM specifications define a number of 'objects' that do not show up as nodes on the Datatrack dependency graph, namely the "who" (which users) and the "how" (which processes), leaving only the dependencies between the "what" (the generated data). This difference can be seen by comparing the toy example given in the OPM specification (Fig. 6) [1]. As well as displaying *artefacts* (e.g. 'cake'), it also shows the *process* "bake" and the *agent* "John".

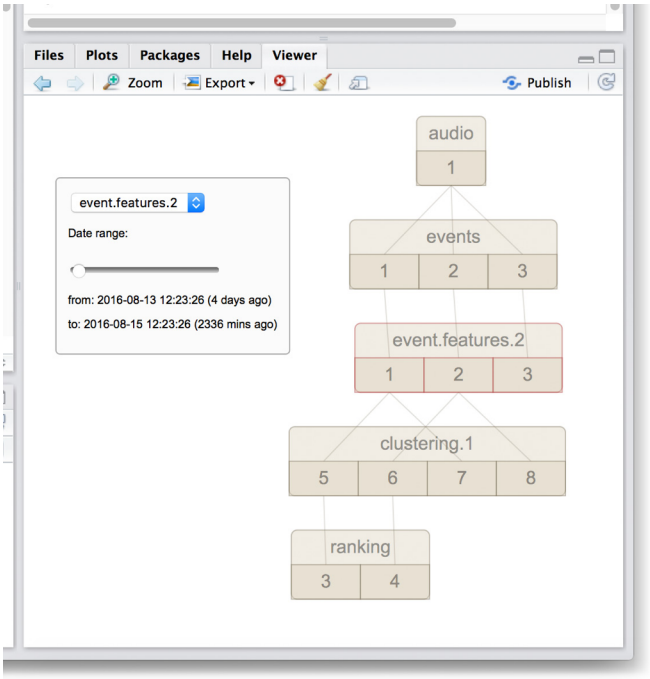

Fig. 5. Data dependency graph within the R-Studio viewer

In making design decisions about what should recorded and what should be present as nodes in the dependency graph, some trade-offs were made between simplicity and completeness. The aim of Datatrack is to only include nodes in the dependency graph that are central to satisfying the needs for which it was built, namely providing on-the-fly assistance in selecting the correct input when running a process. Other provenance metadata is recorded in the background, such that at a later stage it can be recovered for debugging purposes.

Complete provenance tracking includes information regarding who was responsible for a process that generated intermediate data. This is desirable in collaborative environments, however it requires more configuration by the experimenter. By allowing arbitrary user-supplied metadata to be recorded when data is written (passed to Datatrack's write function as *annotations*), this "who" information can be optionally recorded. This metadata may help a user to navigate through the dependency tree by giving extra information, but is not included as nodes in the dependency graph unlike a fully compliant OPM graph. This decision was made to simplify the graph for readability.

2) Tracking of code versions and environment information

For provenance tracking to be complete, the experiment should be completely reproducible. This requires that all the algorithms that played a role in the creation of the output data, as well as the environment in which they were run, should be recorded in the provenance metadata.

Datatrack keeps things simple and lightweight by simply recording the *date that the data object was generated*, and a *stack trace* of function calls leading to the writing of data. In conjunction with a versioning system (in our case Git), this

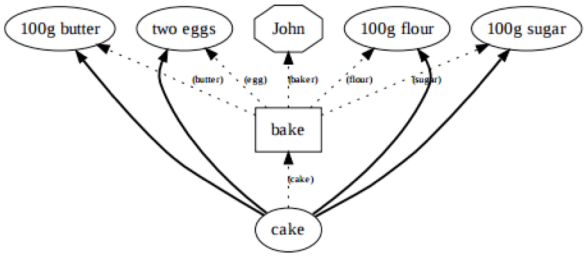

Fig. 6. Example of a workflow graph that adheres to the OPM specification [1]

does reasonably well in allowing the details of the processes responsible for generating output at any later date to be discovered if necessary.

For completeness, Datatrack also records information about the environment including

- The operating system
- The version of R
- The loaded packages and their versions

On the dependency graph, this information is all recorded as annotations on a data node, rather than as their own nodes. Again, while this deviates from provenance graphs favored by the OPM specification, it was primarily done in order to keep the dependency graph as visually uncluttered as possible so as to produce a more useable tool.

The decision to keep the processes (the 'how' of data provenance) as annotations of data nodes, rather as their own nodes, was made because Datatrack is essentially data-centric: the user only interacts with it through reading and writing data. For this reason, while the processes are part of the workflow and would conceptually fit into the dependency graph (connecting two data nodes), there actually is no requirement as far as Datatrack is concerned to formally name the processes.

3) Generating versions and overwriting data.

Core to the aim of Datatrack is assisting the researcher in selecting version of the input data for a process they are running. Numerous versions suitable data might be available, each having been generated by the same earlier process but with different parameters or using different inputs.

As detailed in section B, Datatrack will write multiple versions of the same name, provided they have a unique combination of dependencies parameters. Version numbers are automatically assigned. If the same script is run with different parameters, this parameter information will be passed on to Datatrack and a new version will be saved. Similarly, if the same process is run with different input data, this dependency information will be passed on when writing the output data and a new version of it will be saved.

If modifications are made to the code that generates the data, or if the process is run in a different environment, such as with different versions of packages loaded, then this does not constitute a new version.

This is done for the simple reason that changes to the code that defines the process is most likely done to improve output results. Essentially, this coupling of a workflow model with a constantly evolving codebase of processes creates this situation.

However, this decision is certainly debatable, and it may be desirable incorporate changes to the source code and loaded packages and into the data versioning system. However, this presents a number of implementation challenges and may complicate the user's interaction with the data dependency graph. At this stage of the prototype, we have decided to keep it simple and not have changes to code generate new versions of data. If the user decides that the two versions of a process *should* result in two versions of the output, they can consider representing this information as a parameter or modifying the name of the output.

4) Potential for illogical data dependencies

Datatrack's intentional simplicity is designed to allow it to fit into a wide variety of coding styles, but means that there is no inbuilt checking that the dependencies declared by the user actually make sense. Cyclic data dependencies could in theory be generated. It is up to the user to select the correct data to use as dependencies and to name the output data in a way that conforms to their conceptual workflow. If done in a sensible way, dependency cycles will be absent.

E. Technical implementation of Datatrack package

As the Datatrack package is a prototype, these implementation details are neither set in stone nor as important as the design considerations and motivations behind the packages, and as such this description has been kept brief.

A *Datatrack project* is simply a directory containing the provenance metadata and the data objects themselves. The minimum configuration for a Datatrack project is the path to this Datatrack project directory. Configuration is stored as a json file in the working directory.

The package records all provenance and versioning metadata in a CSV (comma separated values) file on disk. A relational database would also have been appropriate and possibly more efficient, however it was deemed that running and interfacing with the database was an unnecessary complexity, and would require greater user configuration. By storing all metadata available as a text file, the user can manually inspect it if need be, using familiar spreadsheet programs rather than requiring database management software or SQL knowledge.

Data output files are saved in a central location, the path to which is specified through user configuration, with file names automatically generated by the Datatrack version numbering system.

The visualization of the dependency graph is written in Javascript, html and css using the D3 Javascript library [12]. The "HTML widgets" R package [13] was used to display this graph within the RStudio console. For maintainability reasons, the visualization tasks were abstracted to a separate R package, on which Datatrack depends.

V. Future work

A. Improving interactive features of the dependency graph

The visual interface of the graph used to help the user choose between potential inputs is currently interactive to a limited degree:

- The user can select a date range to limit the number of nodes shown.
- Only the data nodes, grouped by name, are shown in the graph. The metadata that actually informs the decision-making is displayed through user interaction (in the form of a mouse rollover).

While this works well, greater control over what is shown would be better. The key is to offer as much information as possible while continuing to allow easy comparison between many data objects displayed side-by-side.

Currently, the user can filter nodes by name, which will exclude any group that does not have at least one node linked directly or indirectly to a node in the group with the given name. Nodes of the selected group can be filtered by created date. Finer-grained control of which nodes are displayed may be useful if the graph become very large, however the need for this has not yet arisen in our use cases.

B. Integration with Git

Currently, in order to inspect the source code used to generate a data object after the code has been modified, the Git repository can be used. This requires taking the stack trace and timestamp available in the Datatrack metadata and examining the Git repository, but this is not integrated and is done manually. This is both somewhat tedious and unreliable, since there could be multiple branches and information about the branch used for generation of a data object is currently not available in the provenance metadata.

Integration with Git version control will be added to solve this problem. By allowing Datatrack attach the Git repository version number, branch and commit status data object's metadata, accessing this retrospective provenance information will be smoother and less ambiguous.

C. Invalidating downstream data

Let us consider the following scenario. The source code of a particular step of the process is modified, for example to fix a bug. Currently, if this step is run after these changes, Datatrack will overwrite its output data objects (if the dependencies and parameters are the same).

However, this presents a problem if there are any existing downstream data objects that depend on these overwritten data objects. A modification to the code of a step in the workflow has an effect not only on its output data objects, but also on everything downstream to that step.

A proposed solution is as follows. Upon overwriting, first check if the new data is the same as the existing. If it is

different, check if there are any downstream dependent data objects. If so, they should be either deleted or flagged as invalid and archived, after confirming with the user.

VI. Conclusion

This paper has discussed some of the provenance tracking and data versioning requirements of research that uses interactive exploratory scripting involving a data workflow, where the nature of the work does not lend itself to integration with a scientific workflow management system.

Although our solution has been developed to cater for the requirements of our specific ecological acoustics research, solving actual problems as they arise, these requirements are reasonably general.

Datatrack addresses these needs by offering a lightweight alternative to using a fully featured scientific workflow management system to researchers whose computational experiments are contained within the single interactive programming environment such as R, and involve constant revisions to the source code.

By automating the generation of provenance metadata in a way that provides almost no modifications to the normal coding practices of the researcher, it succeeds in its goal to be accessible to users from a wide range of computational experimentation practices. Most importantly, it provides a tool to allow the user to select the appropriate input data for their processes quickly and reliably without needing to leave the scripting environment.

References

[1] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, *et al.*, "The open provenance model core specification (v1. 1)," in Future generation computer systems, vol. 27, pp. 743-756, 2011.

[2] S. Miles, P. Groth, M. Branco, and L. Moreau, "The Requirements of Using Provenance in e-Science Experiments," in Journal of Grid Computing, vol. 5, pp. 1-25, 2007.

[3] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "VisTrails: visualization meets data management," in ACM SIGMOD international conference on Management of data, 2006, pp. 745-747.

[4] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in 16th International Conference on Scientific and Statistical Database Management, 2004, pp. 423-424.

[5] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-Science: An overview of workflow system features and capabilities," in Future Generation Computer Systems, vol. 25, pp. 528-540, 2009.

[6] P. Eichinski, "Datatrack," 1.0.0-beta.1 https://github.com/peichins/datatrack http://dx.doi.org/10.5281/zenodo.60582

[7] P. Missier, K. Belhajjame, and J. Cheney, "The W3C PROV family of specifications for modelling provenance metadata," 16th International Conference on Extending Database Technology, Genoa, Italy, 2013.

[8] K. A. C. S. Ocana, V. Silva, D. d. Oliveira, and M. Mattoso, "Data Analytics in Bioinformatics: Data Science in Practice for Genomics Analysis Workflows," in IEEE 11th International Conference on e-Science, 2015, pp. 322-331.

[9] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire, "noworkflow: Capturing and analyzing provenance of scripts," in Provenance and Annotation of Data and Processes, ed: Springer, 2014, pp. 71-83.

[10] T. McPhillips, T. Song, T. Kolisnik, S. Aulenbach, K. Belhajjame, K. Bocinsky, *et al.*, "Yesworkflow: A user-oriented, language-independent tool for recovering workflow information from scripts," in CoRR, vol. abs/1502.02403, 2015.

[11] RStudio-Team, "RStudio: Integrated Development for R. ," http://www.rstudio.com/.

[12] M. Bostock, V. Ogievetsky, and J. Heer, "D3; Data-Driven Documents," in IEEE Transactions on Visualization and Computer Graphics, vol. 17, pp. 2301-2309, 2011.

[13] R. Vaidyanathan, Y. Xie, J. Allaire, J. Cheng, and K. Russell, "htmlwidgets: HTML Widgets for R," R package version 0.6 https://cran.r-project.org/package=htmlwidgets