Automatic generation of provenance metadata during execution of scientific workflows

Felix Bartusch, Maximilian Hanussek, Jens Krüger*
High-Performance and Cloud Computing Group
Zentrum für Datenverarbeitung
University of Tübingen, Germany
*jens.krueger@uni-tuebingen.de

***Abstract*—Data processing in data intensive scientific fields like bioinformatics is automated to a great extent. Among others, automation is achieved with workflow engines that execute an explicitly stated sequence of computations. Scientists can use these workflows through science gateways or they develop them by their own. In both cases they may have to preprocess their raw data and also may want to further process the workflow output. The scientist has to take care about provenance of the whole data processing pipeline. This is not a trivial task due to the diverse set of computational tools and environments used during the transformation of raw data to the final results. Thus we created a metadata schema to provide provenance for data processing pipelines and implemented a tool that creates this metadata during the execution of typical scientific computations.**

***Provenance, Reproducibility, Workflows, Science Gateways*—**

I. Introduction

Originally the term provenance is widely used in the field of arts to ensure the passage of paintings through different owners. Other fields like archeology also uses provenance information to exactly describe the find spot of artifacts. In an analogous way the term data provenance in computational science describes the lineage of data [1]. This lineage encompasses transformations or computations on other data sets [2].

Proper recording of provenance information has many advantages. One important point is the reliability and reproducibility of published results. If there is no or just weak provenance for the result, it is not really reliable [3]

In this work we present our tool *Dataprov* for automatic generation of provenance metadata. It was our goal to develop a tool that can record provenance data for arbitrary command line tools and workflow management systems. A researcher can use our tool to track provenance for computations on resources the researcher has access to. At the same time a provider of ready-to-use scientific pipelines can utilize our tool to create provenance metadata for pipeline runs.

The novelty of *Dataprov* is its extensibility and the support for tracking metadata of used software containers. If a command uses a Docker or Singularity container it is automatically inferred and metadata of the container is incorporated into the resulting provenance file.

The recorded provenance data should be readable for human beings as well as machines. *Dataprov* should be convenient to use without huge changes to existing scripts or workflows.

In Section II we focus on the provenance data format and discuss which questions we want to answer with the recorded provenance data and how we achieve that. Then we describe how the provenance data is recorded with our tool *Dataprov* for arbitrary command line tools as well as for Snakemake workflows [4]. In the last part of the paper we discuss how one could integrate the tool into science gateways that provide access to workflows for scientists of a specific community.

II. Provenance data format

Important information that should be provided by provenance data are already stated in various publications [5], [6]. Given a previously computed result, typical questions that should be answered by provenance data are:

- Who computed this result?
- How was the result computed?
- What input files were used and what is the provenance chain of the input?
- What was the environment on which the computation was executed?
- Was the input or resulting data changed in the meantime?

We chose to represent the provenance data in the XML format. The XML format has the advantage that humans as well as machines can read them and that any produced XML document can be validated against a XML schema. The XML schema files for our provenance data format is available in the Dataprov GitHub repository [1].

Figure 1 shows the first three levels of the XML schema. A data provenance (*Dataprov*) object provides metadata for a file (*target*). We store the URI of the file and the SHA1 hashsum. The checksum can be used to detect changes since file creation. The *history* contains a list of *operations* that lead to the described file.

The XML schema of such an operation is depicted in Figure 1. An *operation* can have several *input files* and can produce several *output files*. As shown in Figure 1 a file is characterized by the triple of filename, URI, and a SHA1 hashsum. The operation is performed by a person (*executor*) on a specific machine (*host*). The *opClass* element describes the operation in more detail and one can provide a *message* in which the purpose of the operation can be noted.

[1] https://github.com/fbartusch/dataprov

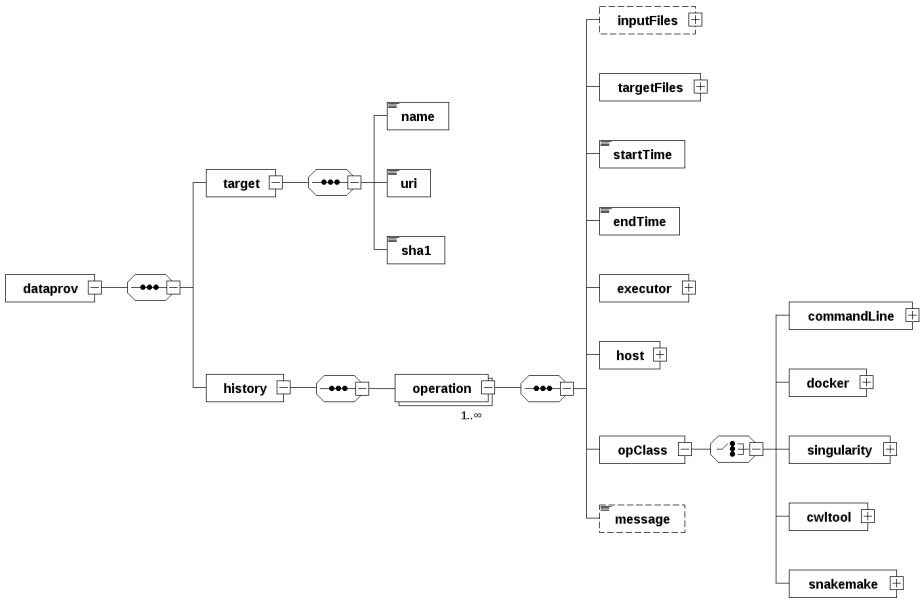

Fig. 1. Top level elements of the data provenance (Dataprov) XML schema. The document contains provenance information for a file object (*target*) and a *history* of operations performed on data objects. The history itself is a list of *operations* performed in the past to create the described data object. The *operations* can have several *input files* and can produce several *output files*. It is performed by a person (*executor*) on a specific machine (*host*). The *opClass* element describes the operation in more detail.

The operation class (*opClass*) element is the extensible part of the schema. Currently our implementation supports shell commands, commands executed in Docker [7] and Singularity [8] containers, Common Workflow Language (CWL) command line tools [9], and Snakemake workflows. Each of the operations has different information to track, hence the different classes. The use cases in Section IV contain examples for some of the supported operations.

For shell commands just the executed tool and its version is of importance. In contrast, for Snakemake workflows also the workflow file, an optional configuration file, and several workflow steps have to be tracked.

Listing 1. XML Element describing the executor of an operation.

```
<executor>
  <title>Dr.</title>
  <firstName>John</firstName>
  <surname>Doe</surname>
  <mail>john.doe@any-uni.edu</mail>
  <affiliation>Group for Awsome Things</affiliation>
</executor>
```

III. Implementation and Extensibility

The implementation of the data provenance schema in Section II was done with Python and was tested with Python 2.7 and Python 3.6. The tool called *Dataprov* is available from GitHub[2]. The general idea is to wrap computations with our tool in the following way:

```
dataprov <options> run <wrapped_command>
```

[2]https://github.com/fbartusch/dataprov

Listing 2. XML Element describing the host machine of an operation.

```
<host>
  <system>Linux</system>
  <dist>CentOS Linux</dist>
  <version>7.3.1611</version>
  <codename>Core</codename>
  <kernelVersion>4.9.34-29.el7.x86_64</kernelVersion>
  <machine>x86_64</machine>
  <processor>x86_64</processor>
  <hostname>anyhostname.any-uni.edu</hostname>
</host>
```

The operation class element of the XML schema was introduced in Section II. Based on the first part of the wrapped command *Dataprov* decides which specific element should be generated. E.g. if the wrapped command starts with 'snakemake', a snakemake workflow element will be generated.

The simplest mode of *Dataprov* wraps shell commands. The user has to define input files and expected output files manually. One example would be

```
dataprov -i FILE_IN -o FILE_OUT run COMMAND
# Example
dataprov -i shakespeare.txt -o wc.txt run \
'wc shakespeare.txt > wc.txt'
```

The tool checks if provenance files for the input files are available. If yes, the *history* of the input provenance data is incorporated into the resulting *history*. Details about the host architecture, running operation system are determined using the Python package *platform*. We show an example of the collected data in Listing 2. Information about the executor are stored in a configuration file and provides among others

Listing 3. XML Element describing a simple command line operation with one input and one output file.

```
<opClass>
  <commandLine>
    <command>samtools index A.bam</command>
    <toolPath>/usr/bin/samtools</toolPath>
    <toolVersion>samtools 1.6, [...]</toolVersion>
    <inputFiles>
      <file>
        <name>A.bam</name>
        <uri>file:///home/johndoe/sorted_reads/A.bam</uri>
        <sha1>a96cd1348e8b1d102bfcc9887055987da80ed7f5</sha1>
      </file>
    </inputFiles>
    <outputFiles>
      <file>
        <name>A.bam.bai</name>
        <uri>file:///home/johndoe/A.bam.bai</uri>
        <sha1>2f5440c28f8931fca56e6c4dd925b23edb949c2f</sha1>
      </file>
    </outputFiles
  </commandLine>
</opClass>
```

name and affiliation as showed in Listing 1. When *Dataprov* is first used this configuration file is created in the home directory and the user is asked to fill in the needed information. When the user wraps an operation with *Dataprov* the executor information from the configuration file is incorporated into the resulting provenance metadata.

For simple command line options we try to infer the location of the executable with the *find_executable* function of the Python *distutils* package and the software version by running a subprocess of the tool with both the *–version/-v* switches and capture the output.

The wrapped commands are executed as subprocesses if possible. The output of the subprocess is also written to the shell such that no important output is masked. After the subprocess has finished, we write the collected provenance data for each output file to an XML file. If an error occurrs during the subprocess no provenance data is written.

In the word count example above one has to specify the input and output files twice. First for *Dataprov*, the other time in the wrapped command. This is tedious and also error-prone, but for a simple wrapper without any alternative.

Specifying the input and output files twice is dropped if one implements support for specific commands into *Dataprov*. An example for this is the implemented CWL command line tool. To use the CWL reference implementation `cwltool` one has to specifiy an cwl-file and an input binding file. *Dataprov* infers information about input and output files by using methods provided by the *cwltool* Python package. Then the wrapped command will be executed.

```
dataprov run cwltool CWL_FILE INPUT_BINDING
```

For each element of the XML schema there is a corresponding class in the source code. The class implements basic functionalities like creating Python objects from XML trees and vice versa. Each element is described in its own schema file, whose path is also set in the corresponding Python class. This greatly simplifies the validation of read or written XML elements.

The fact that every schema element has a corresponding counterpart in the Python code makes it convenient to support new tools or workflow systems. In the first step the new XML element for the new operation class is written. In the second step on creates the corresponding Python class. This class contains functions interacting with the wrapped command in order to fill the XML schema with meaningful information. For the Snakemake case, the constructor of the class performs a dryrun of the workflow and parses the output to determine input, output and command of every workflow step.

IV. Use cases

Based on three use cases we show how the provenance data is generated. The first use case comprises individual computations. These play a decisive role during the development of new workflows when a researcher tries several tools for a specific task. With good provenance the researcher can track which tool and parameter combination yielded a specific result. It plays also a role for individual computations whose results are used by workflows as input data like indices for reference genomes . The second use shows how we handle provenance information for Docker containers whereas the third use case handles whole scientific workflows.

A. Provenance for individual computations

The researcher tries various tools and parameter choices to evaluate what fits best for the workflow. Usually this is done on the command line in an interactive manner and the result files are stored somehow in a directory hierarchy. Hence, one has to keep track from which tool and parameter combination a result file originates. Annotating provenance data for individual computations is also important for data computed once and used often like indices for reference genomes . To know how the genomic index, that is used in a mapping step of a workflow, was generated also increases the reproducibility of the entire workflow. Therefore, provenance for individual computatations is the first use case we want to tackle with our approach.

Assuming that we compute a genomic index of some reference genome with the Burrows-Wheeler Aligner (BWA):

```
bwa index genome.fa
```

This results in a set of files describing the index of the genome. We can use the index in a mapping workflow. But after some time nobody can tell you which release of BWA was used to generate the index or which coworker computed the index. For individual computations one can use our tool in the following way:

```
dataprov -i genome.fa -o genome.fa.bwt run \
    bwa index genome.fa
```

The original command is wrapped by our software *Dataprov*. For single computations you have to tell *Dataprov* the input and output files of the computations because it can not infer it from the command. *Dataprov* generates a XML file

`genome.fa.bwt.prov` whose schema is shown in Figure 1.

The overhead for the researcher consists of specifying input and output files of the operation. This is feasible for manually executing simple computations, but not for automatic workflows.

The second example of an individual computation also shows how provenance information of input files are incorporated into provenance information of output files. We map a set of reads against the reference genome we just created. The structure of the resulting XML file is shown in Listing 4.

```
dataprov -i genome.fa.bwt -i A.fastq \
         -o A.bam run \
         'bwa mem genome.fa A.fastq > A.bam'
```

Listing 4. Dataprov XML files inherit history of input files.

```
<dataprov>
  <target>
    <name>A.bam</name>
    [...]
  </target>
  <history>
    <operation>
      [...
      Inherited from first computation,
      describing creation of genome.fa.bwt
      ...]
    </operation>
    <operation>
      [...
      describing creation of A.bam
      ...]
    </operation>
  </history>
</dataprov>
```

B. Provenance for software containers

This use case is in principle the same as the first one, but this time BWA is provided through a Docker container. We show only the first computation, because it's analogue to the use case before.

```
docker pull biocontainers/bwa
dataprov -i genome.fa -o genome.fa.bwt run \
  docker run -v $PWD/:/tmp/:z -it \
  docker.io/biocontainers/bwa:latest \
    bwa index /tmp/genome.fa
```

The computation takes now place in a Docker container. *Dataprov* interpretes the docker command and extracts the container name. Then it uses the python docker package to inspect the container and creates a XML element describing the operation and the container as shown in Listing 5.

C. Provenance for workflows: Snakemake

We implemented support for Snakemake workflows into our application to demonstrate the main purpose of our work, the automatic generation of provenance metadata during workflow execution. This means you can run Snakemake workflows whilst generating provenance data for the resulting files. The actual use case is composed of the workflow from the snakemake tutorial[3]. The workflow searches for genomic variants in sequencing data. The inputs are two sets of reads which are then mapped to the reference genome using the Burrows-Wheeler Aligner (BWA) [10]. The mapping result is sorted and indexed with samtools [11] and genomic variants are called with bcftools [12].

Listing 5. Docker commands have their own operation class. Not only the wrapped command and details about used Docker version is stored, but also valuable information about the Docker container.

```
<opClass>
  <docker>
    <command>docker run [...]</command>
    <dockerContainer>
      <imageSource>
        <dockerLocal>
          docker.io/biocontainers/bwa:latest
        </dockerLocal>
      </imageSource>
      <imageDetails>
        <imageID>sha256:0b01483[...]</imageID>
        <repoTag>docker.io/biocontainers/bwa:latest</repoTag>
        <repoDigest>[...]</repoDigest>
        <created>2018-02-08T11:04:56.756250089Z</created>
        <labels>
          <item key="software" value="bwa"/>
          <item key="software.version" value="0.7.15"/>
        </labels>
      </imageDetails>
      <dockerPath>/usr/bin/docker</dockerPath>
      <dockerVersion>
        Docker version 1.12.6, build 88a4867/1.12.6
      </dockerVersion>
    </dockerContainer>
  </docker>
</opClass>
```

To run the workflow and call all variants, one would execute the following command:

```
snakemake all.vcf
```

This runs the Snakemake workflow until the specified target file is created. The workflow is described in a so-called Snakefile that specifies rules. Rules have files as targets and Snakemake builds a directed acyclic graph (DAG) from the rules until the specified target file is reached. Then Snakemake executes the commands inferred from the DAG.

For creating the provenance data one has to execute the following command:

```
dataprov run snakemake all.vcf
```

Compared to the execution of individual commands the overhead here is very small. Internally *Dataprov* performs a dryrun of the Snakemake workflow to infer which rules are executed and extract the corresponding commands. Finally the workflow is executed and the resulting provenance files are generated. Because the resulting XML file comprises over 200 lines and can be viewed in the example directory of the GitHub repository[4].

[3]http://snakemake.readthedocs.io/en/stable/tutorial/tutorial.html

[4]https://github.com/fbartusch/dataprov/tree/master/examples/snakemake

V. Integration into Science Gateways

Dataprov is versatile and can be used on a local workstation, a HPC cluster, or a virtual machine in the cloud. The tool does not require any special privileges and can be integrated into tools and workflows offered by science gateways.

Science gateways offer convenient access to tools and workflows. An example for such a science gateway is the MoSGrid portal [13], [14] that provides access for molecular dynamics and docking workflows. A user does not have to install and maintain software suits. Even though this is very desirable for a scientist it renders science gateways more or less a black box. All steps between upload and download are not always verifiable for the user.

If a science gateway integrates the proposed method the gateway can return the provenance metadata together with the computational results. The tools and workflows offered by the science gateway are still convenient to use and in addition all data processing steps are traceable for the scientist.

As shown in the use cases the implementation has almost no overhead for the operator of the science gateway if certain workflow engines are used to perform the computation.

VI. Related Work

The tool noWorkflow [15] captures provenance of Python scripts by wrapping their execution and storing among others a hash of the executed python script, the execution time, called functions, and dependencies. Compared to the simple script support of *Dataprov*, noWorkflow raises more provenance data and provides more functionality. But currently it just supports python scripts, in contrast to *Dataprov* which is agnostic to the used script language.

The tool ReproZip [16] traces the execution of a computational experiment and packs all required dependencies into one archive. ReproZip can unpack the archive and run the experiment again on another Linux-based OS. ReproZip identifies input and output files automatically using a heuristic. If *Dataprov* is used with a reproducible environment like software containers, the reproducibility and provenance of the experiment is comparable to ReproZip. The difference between both approaches is that ReproZip concentrates on Provenance of one experiment and thus one environment, whereas *Dataprov* can incorporate provenance data from previous computations into the provenance of new experiments.

CodaLab [5] offers a collaborative platform for reproducible research. One can upload data and Docker containers to a CodaLab server and run experiments on the data using an environment provided by Docker containers. Data and runs are stored on the platform as immutable bundles. Worksheets written in a custom markdown language present an experimental pipeline and can contain references to the immutable bundles. Because whole research process takes place on the CodaLab platform the experiments are well documented and reproducible. The difference to *Dataprov* is that a researcher moves the computational work to the CodaLab platform whereas *Dataprov* runs in the familiar to the researcher.

Several universal provenance models were created in the past. Two of the most common are the Open Provenance Model (OPM) [5] and the W3C PROV family [17]. OPM emerged from two community provenance challenges and the W3C PROV documents want to set a standard format for the representation of provenance and bases on OPM and other semantic web ontologies. Because the W3C provenance model is a standard for the whole world wide web it is much more comprehensive than our approach. Every information we store with our schema could be also described by the W3C standard, but our method has the advantage that it is especially tailored to the actual need of data scientists. Thus it is not required to acquire full insight into the W3C provenance standard to understand the provenance data our method derives.

Many workflow engines like Snakemake or Galaxy are able to generate a graphical representation of a workflow and return also the commands that were executed during workflow execution. The additional benefit of *Dataprov* is, that the tracked operations can consist of entirely different types. This means for example that the preprocessing of data can be done via a Snakemake workflow whereas the key results are generated by another script or even manual shell commands and the provenance information is still available in a well defined format.

VII. Discussion and Outlook

The presented method is able to wrap single shell commands as well as entire Snakemake workflows and produces a human and machine readable provenance metadata. The provenance metadata describes the transformation of the input data to the output data in detail and thus enhances the reproducibility of the computation massively.

We also discussed how the integration of the proposed metadata schema could improve data provenance of computations executed through science gateways.

Wrapping single computations on the command line has the overhead of specifying input and output of the computation. The support of workflow engines reduce this overhead to the minimum. This has to be implemented for each workflow engine individually. The implementation effort depends heavily on the workflow engine and its capabilities. Engines providing an easy to use API or even a dryrun functionality are easier to support.

The resulting XML metadata file is well defined by a schema and can be validated. For computations with fewer steps the XML document is with not much effort readable by a human. The file tends to be very long and thus not so easy readably if the computation comprises many steps, input files, and output files. Thus a visual representation of the metadata would give more insight into the provenance information. This could be accomplished by generating a simple directed acyclic graph showing data files as nodes and the executed commands as edges between the nodes. A more sophisticated visualization could comprise an interactive graph that shows

[5] https://worksheets.codalab.org/

additional information about files, executors, and environment as mouseover effect.

VIII. Acknowledgements

The authors acknowledge support by the High Performance and Cloud Computing Group at the Zentrum für Datenverarbeitung of the University of Tübingen, the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant no INST 37/935-1 FUGG. Part of the work presented here was also supported through BMBF funded project de.NBI (031 A 534A) and MWK Baden-Württemberg funded project CiTAR ("Zitierbare wissenschaftliche Methoden").

References

[1] Y. L. Simmhan, B. Plale, and D. Gannon, "A Survey of Data Provenance in e-Science," pp. 31–36, 2005.

[2] P. Buneman and S. Davidson, "Data provenance–the foundation of data quality," in *Data provenance–the foundation of data quality'*, 2010.

[3] N. Prat and S. Madnick, "Measuring data believability: A provenance approach," in *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2008.

[4] J. Köster and S. Rahmann, "Snakemake-a scalable bioinformatics workflow engine," *Bioinformatics*, vol. 28, no. 19, pp. 2520–2522, 2012.

[5] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. Den Bussche, "The Open Provenance Model core specification (v1.1)," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 743–756, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.future.2010.07.005

[6] R. Bose and J. Frew, "Lineage retrieval for scientific data processing: a survey," *ACM Computing Surveys*, vol. 37, no. 1, pp. 1–28, 2005.

[7] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, 2014.

[8] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 2017.

[9] P. Amstutz, M. R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Ménager, M. Nedeljkovich, M. Scales, S. Soiland-Reyes, and L. Stojanovic, "Common Workflow Language, v1.0," 2016. [Online]. Available: https://figshare.com/articles/Common_Workflow_Language_draft_3/3115156

[10] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," 2013. [Online]. Available: http://arxiv.org/abs/1303.3997

[11] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin, "The Sequence Alignment/Map format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[12] P. Danecek, S. Schiffels, and R. Durbin, "Multiallelic calling model in bcftools (-m)," pp. 1–2, 2014.

[13] J. Krüger, R. Grunzke, and S. Gesing, "The MoSGrid Science GatewayA Complete Solution for Molecular Simulations," *Journal of Chemical Theory and Computation*, vol. 10, no. 6, pp. 2232–2245, 2014. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ct500159h

[14] L. Zimmermann, R. Grunzke, and J. Krüger, "Maintaining a Science Gateway Lessons Learned from MoSGrid," *Proceedings of the 50th Hawaii International Conference on System Sciences*, pp. 6233–6242, 2017. [Online]. Available: http://hdl.handle.net/10125/41918

[15] L. Murta, V. Braganholo, and J. Freire, "noWorkflow: a Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1841–1844, 2017.

[16] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "ReproZip: Computational Reproducibility With Ease," *Proceedings of the 2016 International Conference on Management of Data*, pp. 2085–2088, 2016.

[17] L. Moreau, P. Groth, J. Cheney, T. Lebo, and S. Miles, "The rationale of PROV," *Journal of Web Semantics*, vol. 35, pp. 235–257, 2015.