CPR - A Comprehensible Provenance Record for Verification Workflows in Whole Tale*

Timothy M. McPhillips[1], Thomas Thelen[2], Craig Willis[1], Kacper Kowalik[3], Matthew B. Jones[2], and Bertram Ludäscher[1,3]

[1] School of Information Sciences, University of Illinois at Urbana–Champaign
[2] NCEAS, University of California at Santa Barbara
[3] NCSA, University of Illinois at Urbana–Champaign

1 Introduction

A growing number of journal publishers verify computational artifacts as part of the peer-review process. Although the problems of defining and achieving computational reproducibility have proved troublesome generally [2], the particular issues publishers aim to detect in this context are well defined. Questions that representative publishers answer via verification workflows include:

- Is the description in the text and supplementary materials sufficient to enable others to repeat the reported computations?
- Does repeating the computations yield the reported results?

Platforms such as Binder [4] and Whole Tale [1] provide environments for assessing reproducibility of computational artifacts by these standards via approaches analogous to *black-box testing* of the reported computational workflow. A *verifier* (i.e. a person carrying out the verification workflow) uses information provided in the paper to (1) set up the required computational environment; (2) stage input data; (3) trigger a sequence of automated computations; and (4) allow these computations to run to completion. The verifier then confirms that the products of the computations match the description in the paper.

Whole Tale further aims to enable verifiers to observe aspects of *how* automated computational workflows produce intermediate and final artifacts. Ultimately this will allow publishers to ask a third general question:

- Is the authors' description of the roles played by various software components consistent with the observed flow of data through those components?

This will provide verifiers with capabilities analogous to *white-box* testing of the computations reported in a paper. Specifically, it will enable a verifier to detect cases where the sequence of computational steps and flow of data between these steps does not conform to the description given in the paper. Here we demonstrate the tools Whole Tale is using or developing to record, store, query, and visualize the flow of data through computational workflows for this purpose.

* Work supported by NSF Award OAC-1541450.

2 The CPR Toolkit

The Comprehensible Provenance Record (CPR) Toolkit is a suite of tools for recording, storing, querying, and visualizing the provenance of artifacts produced by a run of a computational workflow. As the name suggests, a key objective of the toolkit is to make provenance easily comprehensible, not to systems programmers, but to practitioners of a research domain seeking to understand how the computational artifacts associated with a study in that domain were obtained.

While the primary purpose of CPR at present is to automate the monitoring and management of provenance-relevant events and records associated with a Whole Tale *recorded run*, the toolkit can be deployed in any Linux-based computing environment and used to capture, query, and reason about provenance of computational artifacts produced in that environment.

CPR employs ReproZip [5] to observe system calls invoked as part of the recorded run and to record metadata about (1) the operating-system level processes comprising the overall computation; (2) the files accessed by these processes; and (3) the access mode for file accesses, i.e. whether processes opened files for reading, writing, or both. ReproZip captures and records all of this information in a SQLite database with a schema specific to ReproZip.

Once a recorded run is complete, the cpr command-line utility extracts these OS-level records from the ReproZip trace, transform them into RDF triples, and loads the triples into an RDF dataset in an instance of Blazegraph[1]. The triples are expressed using a vocabulary developed to represent provenance information in the context of Whole Tale recorded runs (Figure 1).

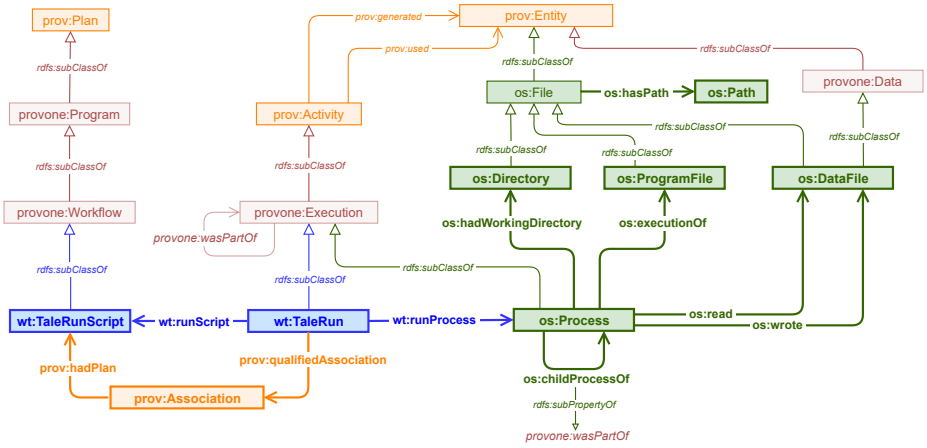


Fig. 1: Relationship of key elements of the CPR vocabulary to classes and properties defined by the PROV and ProvONE vocabularies.

[1] https://github.com/blazegraph/database

The CPR vocabulary extends PROV[2] and ProvONE [3] with subclasses specific to Whole Tale to unambiguously represent run-time provenance records captured from multiple recorded runs and distinct versions of a particular Tale. CPR can represent this vocabulary either as Datalog facts or as RDF triples. Because Blazegraph provides an eager reasoner, all triples implied by the subclass relationships are generated automatically when loading a CPR trace into Blazegraph. Consequently, a CPR trace, asserted using the CPR vocabulary, can be queried in terms of the PROV and ProvONE vocabularies without using a reasoner at query time.

The CPR toolkit and vocabulary recognize the distinct roles played by particular files during a run. A simple YAML file is used to declare a run profile that associates roles with individual files, particular directories, or entire directory trees. Using these declarations while converting a ReproZip trace to the CPR vocabulary, the toolkit is able to distinguish data files of scientific significance from, e.g., shared libraries associated with the operating system or provided by software dependencies, and automatically mask these (often numerous) files in queries and visualization by default.

Finally, the Geist[3] report-templating tool is used to pose SPARQL queries against the Blazegraph instance, to format the query results as reports, and to create visualizations of query results using Graphviz. Geist queries, reports, and visualizations may be parameterized. In Whole Tale we plan to create a predefined set of reports and visualizations following each recorded run.

3 Demonstration

The CPR demo is provided as a Git repository[4] and associated Docker image that enable the examples to be run on Linux, macOS, and Windows-based systems that have Git, Docker, and GNU Make installed. Each example uses the CPR toolkit to record OS-level provenance information from a run of a different computational workflow, to load a Blazegraph instance with the resulting CPR trace, and to produce a set of reports and visualizations via SPARQL queries.

A Makefile in the top directory of the demo repository provides targets for pulling the Docker image from Dockerhub (`pull-image`), building the Docker image locally (`build-image`), for running the examples (`run-examples`), and for deleting all of the reports, visualizations, and other artifacts generated for each example (`clean-examples`). Because the expected results are included in the repository, successful reproduction of the example products is demonstrated by issuing the commands `make clean-examples` and `make run-examples` and confirming that `git diff` reports no differences.

Query results and visualizations for each example provide answers to standard questions including:

[2] https://www.w3.org/TR/prov-dm/
[3] https://github.com/CIRSS/geist
[4] https://github.com/CIRSS/cpr-demo-2021

```
#!/bin/bash
cat inputs/i1.txt inputs/i2.txt > temp/t12.txt
cat inputs/i1.txt inputs/i2.txt inputs/i3.txt > temp/t123.txt
cat inputs/i4.txt > temp/t4.txt
cat temp/t12.txt > outputs/o12.txt
cat temp/t123.txt temp/t4.txt > outputs/o1234.txt
cat temp/t4.txt > outputs/o4.txt
```

Fig. 2: Workflow script.

1. *What programs and script invocations occurred as part of the run?*
2. *What files represent inputs and outputs of the run as a whole?*
3. *What are the input and output data files for each process in the run?*
4. *Which files input to a run are used to produce a particular output file?*
5. *Which run output artifacts are affected by a particular input file?*
6. *What programs contribute to the production of a particular output artifact?*

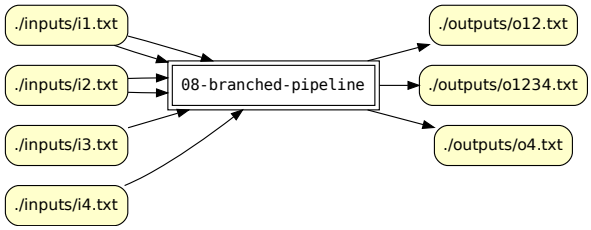
The example computations range from trivial and domain-independent, to relatively complex and domain-specific. An example of minimal complexity that still demonstrates key capabilities of CPR is illustrated in Figures 2 and 3. A simple bash script invokes the `cat` command six times on different combinations of three input files to produce three intermediate files and three final output files. The run profile shown allows CPR to identify the data files (and to ignore system files needed to run the script but irrelevant to the questions a verifier typically asks). The visualizations satisfying queries 2 and 3 are included for a run of this script (Fig. 3b and Fig. 3c) and depict the answers as dataflow graphs. We expect the visualization answering query 3 to be the main CPR artifact a verifier will use to compare the record of execution with the description of the computation in a paper. Visualizations answering queries 4 and 5 can be considered to be subgraphs of the visualization for query 3 limited to nodes and edges relevant to a single output or input file.

4 Observations

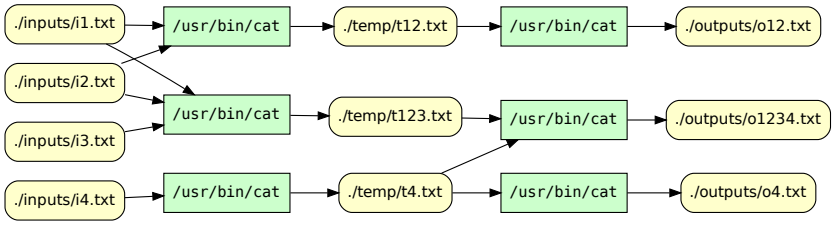
The computations and queries demonstrated here highlight a key challenge in making provenance useful to domain researchers and verifiers: revealing the small subset of recorded events that are of direct relevance to the scientific purpose of an overall computational workflow. At a low level, execution of even a one-line Python 3 script that prints "Hello World" can involve reading tens of different files from disk in addition to the single-line Python file that the user supplied. CPR minimizes such provenance "noise" using SPARQL queries that select files and processes with particular relationships to other files and processes, optionally informed further by a user-provided run profile that assigns distinct roles to files loaded from particular locations on the system. For example, it can be useful to hide processes that do not themselves read or write data files; a bash script that serves only to invoke other programs that do process data files can be masked

```
roles:
  os:
    - /etc
    - /lib
    - /usr/lib
  sw:
    - /usr/bin
  in:
    - ./inputs
  out:
    - ./outputs
  tmp:
    - ./temp
```

(a) Run profile.



(b) Visualization of inputs and outputs of the run as a whole.



(c) Flow of data files through processes comprising the run.

Fig. 3: The run profile (a) indicates that files in the `./temp` directory should be hidden in the "black-box" view (b), but displayed in the "white-box" view (c).

even in the absence of a profile. The bash script listed in Fig. 2 is not depicted graphically in Fig. 3b and Fig. 3c because these queries filter out processes that do not perform I/O on data files.

A second key challenge to making provenance useful to domain specialists is providing vocabularies that convey the significance of particular processes and data artifacts in domain-specific terms. PROV and ProvONE provide essential abstract base classes from which more meaningful provenance vocabularies can be derived. Domain researchers–and the verifiers of computations reported in their papers–likely will find views of provenance employing such specialized vocabularies the most useful. Nevertheless, the base classes are essential for performing general queries that must succeed on traces captured from any domain, e.g. to answer the question, *What are all the files–data files, scripts, executables, shared libraries, etc–that must be archived and restored later to repeat the computation?* By describing computations in terms of files used to store data and processes executed on real computers, the CPR vocabulary provides a set of concepts intermediate to the more general ones comprising PROV and ProvONE, and the more specific concepts of domain-specific vocabularies.

Moreover, deriving the CPR vocabulary from existing standard vocabularies provides multiple options when depositing data and its provenance in public repositories such as DataONE. Because Blazegraph eagerly infers triples implied

by RDF schema declarations, exporting provenance as simply as PROV, or as ProvONE, or as a combination of PROV, ProvONE, and the CPR vocabularies, is as simple as performing a trivial CONSTRUCT query that extracts triples that already exist in the RDF dataset. Finally, much as common base classes in object-oriented programming languages make it convenient to work with collections of objects that are instances of more specialized classes, we expect that access to the PROV, ProvONE, and CPR vocabularies when querying provenance expressed in more specialized vocabularies will in many cases simplify those queries and make them more transparent and reusable.

5 Conclusion

CPR is a toolkit and vocabulary that aims to make the computational provenance of artifacts comprehensible by domain researchers. By specifically highlighting entities researchers in many domains actually think about when planning, describing, and understanding computations–data files, programs, program executions, flow of data between program executions, etc–CPR makes traces of computational workflows transparent and actionable, and enables verifiers and others to judge whether the computations were performed appropriately.

CPR complements existing tools for recording provenance at the operating-system level–including ReproZip and SciUnit which employ execution tracing to identify files that must be packaged to make the computation repeatable on a different system; the CamFlow/CamQuery system which captures whole-system provenance for the purpose of system audit; and the SPADE system which supports auditing in distributed environments. These tools in turn complement provenance-recording and management tools that target specific programming languages and environments, including noWorkflow (for Python), and the Matlab DataONE Toolbox. By observing computational steps that occurs *within* processes, these latter tools provide views of computational provenance that system-level provenance recorders cannot. Making provenance records not just comprehensible but also comprehensive ultimately will require integrating provenance recording tools and vocabularies at multiple levels of abstraction and granularity.

References

1. Brinckman, A., Chard, K., Gaffney, N., Hategan, M., Jones, M.B., Kowalik, K., Kulasekaran, S., Ludäscher, B., Mecum, B.D., Nabrzyski, J., Stodden, V., Taylor, I.J., Turk, M.J., Turner, K.: Computing Environments for Reproducibility: Capturing the "Whole Tale". FGCS **94**, 854–867 (2019)
2. Committee on Reproducibility and Replicability in Science: Reproducibility and Replicability in Science. The National Academies Press (2019)
3. Cuevas-Vicenttín, V., Ludäscher, B., Missier, P., Belhajjame, K., Chirigati, F., Wei, Y., Dey, S., Kianmajd, P., Koop, D., Bowers, S., Altintas, I., Jones, C., Jones, M.B., Walker, L., Slaughter, P., Leinfelder, B.: ProvONE: A PROV Extension Data Model for Scientific Workflow Provenance (2015)

4. Jupyter-Project: Binder 2.0 - Reproducible, Interactive, Sharable Environments for Science at Scale. 17th Python in Science Conference (2018)
5. Rampin, R., Chirigati, F., Shasha, D., Freire, J., Steeves, V.: ReproZip: The Reproducibility Packer. Journal of Open Source Software **1**(8), 107 (2016)