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, 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 [2] 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.

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 rather 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` 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. The triples are expressed using a vocabulary developed to represent provenance information in the context of Whole Tale recorded run executions (Figure 1).

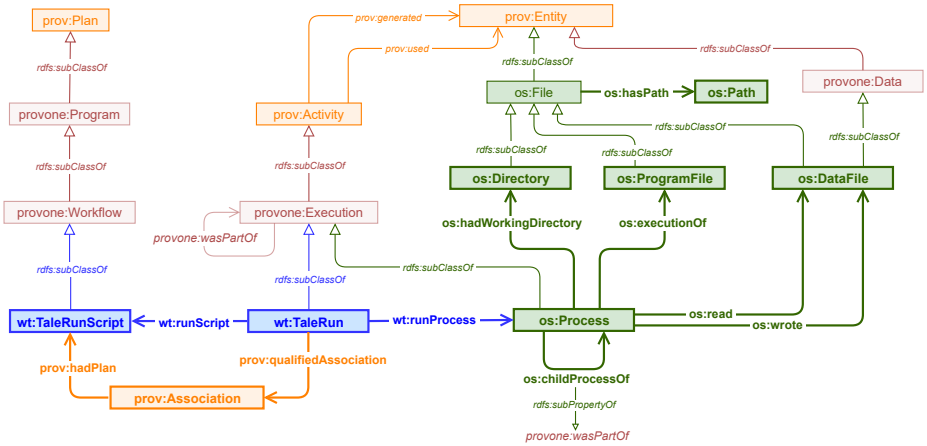


Fig. 1: Relationship of key elements of the CPR vocabulary to classes and properties defined by the PROV and ProvONE vocabularies.

The CPR vocabulary extends PROV and ProvONE with subclasses specific to Whole Tale to unambiguously represeent 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 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 and associated Docker image that enable the examples to be run on any Linux, macOS, or Windows-based system with Git, Docker, and GNU Make installed. Each example uses the CPR toolkit to record OS-level provenance information from a run of different computational workflow, to load a Blazegraph instance with the resulting CPR traces, 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.

The examples range from the trivial and domain-independent, to relatively complex and domain-specific. An example of minimal complexity that still demonstrates key capabalities of CPR is illustrated in Figure **??**. 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 (each a distinct concatenation of the three input files).

3.1 Queries and Visualizations

Queries and visualizations produced for each example included the following. A question phrased in English summarizes the intent of each.

What files are employed as inputs and produced as outputs of the run?

What are the input and output data files for each process in the run?

Which files input to a run are used to produce a particular output file?

Which output artifacts are affected by a particular input file?

What programs and script invocations occur as part of the run?

What programs and script invocations contribute to the production of a particular output artifact?

What constraints on the order of execution of different processes does the observed flow of data imply?

3.2 Observations

A key challenge in making provenance useful to researchers and verifiers alike is highlighting the small subset of recorded events of direct relevance to the scientific purpose of an execution. An execution of a one-line Python 3 script that prints "Hello World" can involve reading as many as X different files from disk in addition to the users' single-line Python file. CPR minimizes such provenance "noise" using a CPR profile that assigns distinct roles to files loaded from particular locations on the system.

It can be useful to hide processes that do not read or write data files. A bash script that invokes other programs that do process data files then becomes invisible.

Exporting a WT trace using the PROV or ProvONE vocabalaries is as simple as a trivial CONSTRUCT query that extracts triples that already exist in the RDF dataset. This is useful for depositing in data repositories, e.g. DataONE which favors provenance expressed using the ProvONE vocabulary exclusively.

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). https://doi.org/10.1016/j.future.2017.12.029, http://www.sciencedirect.com/science/article/pii/S0167739X17310695

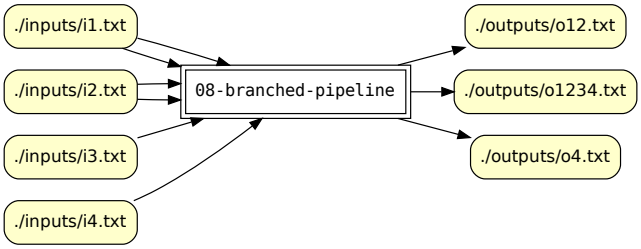
2. Jupyter-Project: Binder 2.0 - Reproducible, Interactive, Sharable Environments for Science at Scale. 17th Python in Science Conference (2018)

```
#!/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
```

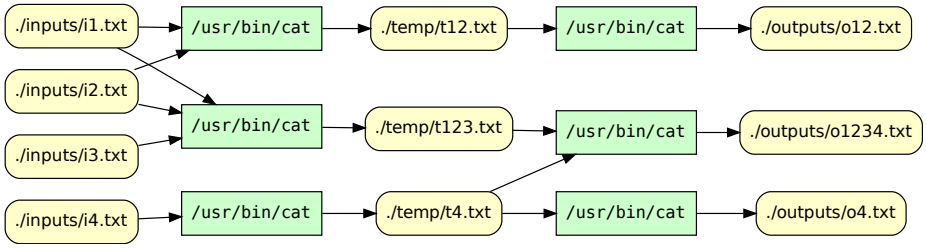
(a) Workflow script.

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

(b) Run profile.



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



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