Write provenance for software experiments

SAMUEL GRAYSON, University of Illinois Urbana-Champaign, USA
DANIEL S. KATZ, University of Illinois Urbana-Champaign Department of Computer Science, USA
REED MILEWICZ, Sandia National Laboratories, USA
DARKO MARINOV, University of Illinois Urbana-Champaign, USA

ACM Reference Format:
Samuel Grayson, Daniel S. Katz, Reed Milewicz, and Darko Marinov. 2023. Write provenance for software experiments. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 GLOSSARY

For the purposes of this paper, we will use the following terms and definitions:

- Software experiment: an experiment defined by software.
  - The inputs may be the outputs of a physical experiment.
  - The experiment may be computing of some statistic from raw data.
  - The output of the experiment may include the performance of the execution.
- Software experimentalist: one who writes software experiments.
- Software researcher: one who studies software engineering aspects of software experiments. Software experimentalists write software experiments, which are studied by software researchers, who write recommendations for software experimentalists.

2 INTRODUCTION

In theory, any Turing-complete computer can emulate the behavior of any other Turing-complete computer. At an abstract level, any software defined for one computer should be runnable on another. Indeed, there are "virtual machine" emulators. One can do even better by defining a high-level software language whose compilers or interpreters can be implemented on multiple machines. However, even with many protability tools, re-executing software experiments still often requires manual labor. Rather than a technical problem with software executors, this is due to there being only nebulous languages for describing how to execute a software experiment.

Such a description language would minimally need to specify the software environment and specify a command which runs the experiment. It is not enough for the language to merely contain this command in a heap of other commands; e.g., a Makefile which defines a rule for executing the experiment alongside rules for compiling intermediate pieces is not sufficient, because there is no machine-readable way to know which of the Make rules executes the experiment. Automatically identifying the "main" command which executes the experiment is critical for:

- Artifact evaluators
- Users seeking to replicate a paper

2023. Manuscript submitted to ACM

- Large-scale re-execution experiments

The former two are accusomted to spending manual effort to find this command, but in the latter case, one might be re-executing thousands of experiments, so if any manual labor is infeasible. The language should describe what the command does, in such a way that a machine can distinguish which command should be run to re-execute the software experiment done in support of a scientific publication.

3 EXISTING STANDARDS FOR EXECUTION DESCRIPTIONS OF SOFTWARE EXPERIMENTS

Of course, there are some existing standards and conventions that are used to describe how to run a computational experiment.

- **Build systems**: While "building" may refer strictly to compiling and linking, many build systems are general enough to be repurposed for describing how to execute a software experiment. E.g., software experimentalists might use GNU Make to execute their software experiment. However, build systems do not fully describe the computational environment (e.g., one might have to install system libraries before running `make`). There is only a loose convention over how to name the targets (e.g., should one run `make all`, `make`, or `make figure_1.png` to run the computational experiment from scratch?).
- **Main script**: Often there is a main script in the project root (`run.sh` or `main.py`). However, In practice main scripts rarely define the software environment; rather they expect to be run from within the proper computational environment. This is especially true if the script depends on a specific interpreter which is not found on all systems seeking to execute the experiment (e.g., Bash, Python 3). There is no automated way to know what exactly this script does and how "deeply" it re-executes the experiment; E.g., if some intermediate data is already written the main script might skip the experiment and only regenerate the figures.
- **Software environment definitions**: Language-neutral (e.g., Spack, Conda, Nix, Guix), and language-specific (e.g., Cargo, Python Virtualenv) environment managers specify the software environment in which the experiment is run. They do not, however, specify the command to run the experiment itself. For Nix and Guix, one might make the experimental output a "package" within the system, but this is quite rare and poorly supported; For example, there is no way to launch a multi-node MPI job from within the Nix or Guix sandbox, and there is no way of describing which package is the experiment.
- **Continuous integration (CI) script**: CI scripts often check an experiment in a defined computational environment as deeply as is feasible given limited CI computational resources. However, there are usually not enough computational resources to execute the experiment, and there is no language for describing which, if any, of the CI script executes the software experiment.
- **Workflow scripts**: A workflow script describes a DAG of tasks for a computational experiment. The tasks can be native or containerized. In practice, workflow scripts may not contain the inputs needed to run the experiment.
  - Case in point, a recent sample of containerized Snakemake workflows showed that these workflows often do not specify example inputs (cite: unpublished work, in submission to ACM REP '23). Even though Snakemake has a standard (https://snakemake.github.io/snakemake-workflow-catalog/?rules=true) for specifying the usage of its workflows, this standard does not have a place

to put example data or an example invocation (https://github.com/snakemake-workflows/dna-seq-varlociraptor/pull/204#issuecomment-1432876029).

A Dockerfile is a combination of the above: a software environment definition composed with an optional main script (in the `CMD`). Sometimes additional context is needed to build the container (e.g., the files referenceed in `ADD` or `COPY` commands may themselves need to be built). While multi-stage containers mitigate this problem, in practice most Dockerfiles cannot be automatically built. Even if they can be built, the `CMD` might require input data or additional arguments, and there is no convention or standard of listing an "example invocation" in a machine-readable way.

4 A NEW STANDARD FOR EXECUTION-DESCRIPTIONS OF SOFTWARE EXPERIMENTS

This is not the final proposal for the complete vocabulary; the peer-review process is not well-suited to iterate on the technical details of the execution description. The point of this document is to argue that the community should spend effort developing this vocabulary.

4.1 Semantic web description

This language could be implemented as an vocabulary for linked data in the semantic web. Linked data is preferrable for these reasons:

1. Linked data is open to extensions.
2. It is possible to link to other resources in linked data.
3. There is already a rich set of ontologies for describing digital and physical resources (RO-crate, wf4prov, software project description, scientific hypotheses, CiTO) in linked data.
4. There is already a rich ecosystem for authoring ontologies and validating documents within those ontologies.

Linked data can be represented in XML format, which is used for other long-term preservation standards. The template of RDF/XML looks like this:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:cito="http://purl.org/spar/cito"
        xmlns:doco="http://purl.org/spar/doco/2015-07-03"
        xmlns:prov="https://www.w3.org/TR/2013/PR-prov-o-20130312/"
        xmlns:wfdesc="http://purl.org/wf4ever/wfdesc#"
        xmlns="http://example.org/execution-description/1.0" >
...
</rdf:RDF>
```

According to the RDF/XML specification, This imports several other vocabularies behind a namespace. E.g., **`rdf:type`** refers to **`type`** in the **`rdf`** namespace, which points to **`http://www.w3.org/1999/02/22-rdf-syntax-ns#`**. XML tags with no namespace are resolved within the default namespace, which is our proposed execution-descriptoin vocabulary.

4.2 Language description

At a very basic level, one could have commands and the purpose that they serve:

```
<process>
  <command>./execute --input data.csv</command>
  <purpose>generates data</purpose>
</process>
```

```
<process>
  <command>python3 main.py</command>
  <purpose>plots figures</purpose>
</process>
```

While we could define conventions around what to name the content of the “purpose” tag, it would be more powerful if the language could link directly to the claims in the publication. The CiTO vocabulary already defines a vocabulary for describing citations.

```
<process>
  <command>./execute --input data.csv</command>
  <purpose>
    <!-- links to an entire publication -->
    <cito:isCitedAsEvidenceBy rdf:resource="https://doi.org/10.1234/123456789" />
  </purpose>
</process>
```

If the publisher hosts an RDF description at the URL “https://doi.org/10.1234/123456789” when the HTTP request content-type header is `application/rdf+xml`, then this creates a web of linked data. The publisher may have the title, authors, date published, and other metadata using Dublin Core metadata terms, for example. This is the dream of linked data: machine-readable data by different authors hosted in different locations linking together seamlessly. Even if the publisher does not have an RDF+XML description, third parties can make claims about “https://doi.org/10.1234/123456789”, although those claims would not be as easily discoverable. One could even reference a Nanopublication, which is a semantic web description of the scientific claim.

The purpose description can be even more granular, using the DoCO vocabulary , which describes documents.

```
<purpose>
  <prov:generated>
    <doco:figure>
      <dc:title>Figure 2b</dc:title>
      <dc:isPartOf rdf:resource="https://doi.org/10.1234/123456789" />
    </doco:figure>
  </prov:generated>
</purpose>
```

With this complete, anyone should be able to execute the experiments which generate figures or claims in the paper if they are labeled by the author in this language.

4.3 Language feature: specified software environment

One can view specifying the software environment as just prerequisite steps in the computational DAG.

```
<process>
  <wfdesc:hasOutput>
    <wfdesc:Output rdf:nodeID="conda-env-out" />
  </wfdesc:hasOutput>
  <command>conda env create --name experiment-123 --file environment.yml</command>
</process>
<process>
  <wfdesc:hasInput>
    <wfdesc:Input rdf:nodeID="conda-env-in" />
  </wfdesc:hasOutput>
  <purpose>figure 4</purpose>
  <command>conda run --name experiment-123 ./plot-figure.py</command>
</process>
<wfdesc:DataLink>
  <wfdesc:hasSource rdf:nodeID="xy-dataset-out" />
  <wfdesc:hasSink rdf:nodeID="xy-dataset-in" />
</wfdesc:DataLink>
```

Now a machine knows that the `conda env create ...` must be run before, and the script should be run within the resulting conda environment.

The purpose of an execution language is not to usurp the build-system or workflow engine, which both already handle task DAGs; there must be some minimal support for DAGs just for the cases where the DAG of tasks is not already encoded in a build-system or workflow engine.

4.4 Language feature: explicit parameters

In addition to specifying the computational environment and command to run, this language is an ideal candidate to also describe the parameters of the experiment, like:

```
<process>
  <command>./generate ${max_resolution} ${rounds}</command>
  <wfdesc:hasInput rdf:resource="max_resolution" />
  <wfdesc:hasInput rdf:resource="rounds" />
</process>
<wfdesc:Parameter>
  <wfdesc:Input rdf:nodeID="max_resolution" description="maximum resolution for convolution" type="fidelity" />
</wfdesc:Parameter>
<wfdesc:Parameter>
  <wfdesc:Input rdf:nodeID="rounds" description="number of repetitions of Guassian blur" type="independent variable" />
</wfdesc:Parameter>
```

One could specify range of valid values or list options.

With this complete, one can even do automated parameter-space search studies, multi-fidelity uncertainty quantification, automated outcome-preserving input minimization, and other automatic experiments.

4.5 Language feature: with retrospective provenance

Retrospective provenance seeks to encode how we got to a specific result. If one expects bit-by-bit reproducibility, the authors can put a hash of the intermediate results into the provenance description; if they only expect approximate reproducibility, they can put summary statistics of intermediate results into the provenance description. A tool might use system-call interposition to learn about a processes reads, writes, and forks. This would be better at identifying and recording intermediate results. When reproducing some software experiment, users can check the intermediate results to see where they begin to differ. Wfprov is one vocabulary for specifying retrospective provenance, and there is already an experimental plugin for Nextflow which targets wfprov.

Users may also want to know how much computational resources (CPU time, disk space, and RAM) the software expeirment require. Provenance is the ideal place to put this. This way, users seeking to reproduce the software experiment know how many resources to request (ahead-of-time allocations are usuually required for batch-scheduled machine).

5 MAKING EASY ON-RAMPS FOR ADOPTION

The execution language should seek to describe existing software frameworks not replace them. In particular, execution should not replace workflow engines. They should instead be wrapped as process-nodes within the execution language. Software experimentalists can continue using their existing build-system and workflow.

A execution description could even be “captured” from an interactive shell session with the user. They would invoke a shell that records every command, its exit status, its read-files, and its write-files (using syscall interposition). The user would exeucte their build-system like normal within this shell. When the user exits, the shell will create a DAG based on the read-files and write-files. For each output file that is not consumed by another command, the shell would prompt the user to describe that command's purpose. Finally, the shell would output a execution description.

In cases where the description cannot be captured by an interactive shell session, one can encode the system of logic that humans would use to deduce the experimental structure. For example,

1. If `shell.nix`, `flake.nix`, or `environment.yml` exists, then use Nix, Nix flakes, or Conda as the environment for future commands.
2. If the environment is not already set, and `requirements.txt` exists, use Pip and Virtualenv as the environment for future commands.
3. If `CMakeLists.txt` exists, set `cmake` and `make` as commands.
4. If `configure.sh` exists, set `./configure` and `make` as command.
5. If `Makefile` exists, set `make` as command. …

A program similar to the one described above could automatically generate a execution description. In the best case, the execution description would be uploaded to the same repository or location that contains the source for the software experiment. This way it can be maintained and used by the original developers, and it is easily discoverable by users. Alternatively, execution descriptions could be placed in a central execution-description repository owned by software researchers. The execution description is linked data, so it can live anywhere, and existing strategies for finding, filtering, and trusting linked data sets would work for execution descriptions.

This is similar to the approach taken by Python for type annotations. Type annotations are easiest to maintain in the original repository, but if the original repository rejects type annotations, there is still a home for them in typeshed.

6 INCENTIVES FOR SOFTWARE EXPERIMENTALISTS TO MAINTAIN EXECUTION DESCRIPTIONS

Software experimentalists are incentized to describe their project this way to benefit from the work of software researchers. When software researchers do a large-scale execution study, it is a "free" reproduction of their work.

To get an artifact evaluation badge, normally authors would have to write a natural language description of what the software environment, what the commands are, how to run them, and where does the data end up. The artifact evaluator has to read, interpret, and execute their description by hand. A execution description could make this nearly automatic; if a execution description exists, the artifact evaluator uses a executor which understands the language and runs all of the commands that reference the manuscript in their `purpose` tag. The only manual labor is comparing these results to those in the paper. Even that comparison can be simplified, if the last step in the execution description outputs a boolean representing "is the hypothesis proven?"; the reviewer just needs to see that all of these output "true".