Wanted: standards for automatic reproducibility of computational experiments

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

1 INTRODUCTION

A computational experiment is reproducible if another team using the same experimental infrastructure can make a measurement that concurs with the original. Investing in reproducibility can improve trustworthiness of scientific results, unlock productivity, and enable reusability and community support[7]. In practice, reproducers often need to manually work with the code to see how to build necessary libraries, configure parameters, find data, and invoke the experiment; it is not *automatic*.

In this work, we consider the use of specification languages that would provide machine-readable metadata on how to interact with a piece of software. It is not enough for the language to merely contain a run 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. Being able to automatically identifying the "main" command which executes the experiment, for instance, would be very useful for those seeking to reproduce results from past experiments or reusing experiments to address new use cases. Moreover, from a research perspective, having a standardized way to run many different codes at scale would open new avenues for data mining research on reproducibility (c.f., [1]).

2 TOWARDS SPECIFICATION STANDARDS FOR AUTOMATED REPRODUCIBILITY ASSESSMENT

At present, there are a diverse range of solutions for expressing how a code should be run, including bash scripts, environment management specifications (e.g., Spack, Nix, Python Virtualenv), continuous integration scripts, workflows, and container specifications. In our own research on reproducibility of scientific codes, as we scale up our studies to include many different codes, keeping track of how to execute each one becomes very complicated. Moreover, when a code fails to run or deliver reproducible results, it is difficult to assess whether there is a fault with the code or whether we failed to run the code in the correct way. While we do not expect (or recommend) that the scientific software community converge on a single solution for executing codes, we do see value in having a standard way of documenting how each code ought to be run. Such a standard could be implemented as a linked-data ontology like the semantic web. This would enable us to build on a rich set of ontologies for describing digital and physical resources (RO-crate [10], Dublin Core metadata terms [11], Description of a Project [12], nanopublications [5], Citation Typing Ontology [8], Documnet Componnets Ontology [2]) and leverage the ecosytem of existing tools for ontology management (Protégé editor, SHACL, SHEX).

2023. Manuscript submitted to ACM

At a very basic level, one could specify available commands and the purpose that they serve (e.g., run make to compile underlying libraries and run main.py to generate figures). But more than just that, a linked data specification standard would enable researchers to link inputs and outputs of codes directly to claims made in publications. With such a specification, any person (or program) should be able to execute the experiments which generate figures or claims in an accompanying paper. For example, the CiTO vocabulary [8] can be used to how the result of a process is used as evidence in a specific publication. These references could be connected to other references of the same publication on the semantic web. The description can be even more granular, such as by using the DoCO vocabulary [2] to point to specific elements (e.g., figures) within a document, or using Nanopublication [5] to reference specific scientific claims. RO-crate [9] has terms for describing dependencies between steps, which can be used to encode dependent steps or specify the computational environment. This would not replace any of the underlying tools, libraries, or frameworks used at present, but it would provide a common vocabulary for explaining how those tools, libraries, and frameworks are employed. Such a specification language could also be used to set bounds on the parameters of the experiment, such as the range of valid values or a list of toggleable parameters; this would enable downstream automated experiments like parameter-space search studies, multi-fidelity uncertainty quantification, and outcome-preserving input minimization.

3 INCENTIVIZING ADOPTION OF SPECIFICATIONS

With a defined specification language in place, tooling could be built to make it as easy as possible to generate a specification document. For example, an execution description could 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); When the user exits, the shell will create a directed acyclic graph based on the read-files and write-files, and for each output file that is not consumed by another command, the shell would prompt the user to describe that command's purpose. Alternatively, in cases where the description cannot be captured by an interactive shell session, it may be possible to encode a system of logic that humans would use to deduce the experimental structure; this is similar to the approach taken by FlaPy [6], a large-scale re-execution study for Python unittests, to install the Python environment.

Meanwhile, conferences and publishers could promote the use of such standard specifications as part of reproducibility requirements for publishing. To get an artifact evaluation badge, normally computational scientists 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; meanwhile, an artifact evaluator has to read, interpret, and execute their description by hand. An execution description could make this nearly automatic; if a execution description exists, the artifact evaluator uses an 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".

4 CONCLUSION

In this position paper, we argue that developing common standards for specifying how computational experiments should be run to get reproducible results would benefit the scientific community. It presents a compromise where different teams can implement their codes in whatever way they see fit while enabling others to easily run them. This would lead to greater productivity in the (re)use of scientific experiments, empower developers to build tools that leverage those common specifications, and enable software engineering researchers to study reproducibility at scale.

5 APPENDIX I: EXAMPLE DOCUMENT

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="https://example.com/"
>
<!--
The RDF tag defines several XML namespaces.

In this document "rdf:label" refers to http://www.w3.org/1999/02/22-rdf-syntax-ns#label
-->
<process rdf:ID="make">
  <command>make libs</command>
  <purpose>compiles libraries</purpose>
</process>
<process rdf:ID="figures">
  <command>python3 main.py</command>
  <purpose>generates figures</purpose>
  <depends_on rdf:resource="#make" />
</process>
</rdf:RDF>
```

6 HOW TO GET AUTOMATIC REPRODUCIBILITY

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

6.1 Language description

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

```
<process rdf:ID="make">
  <command>make libs</command>
  <purpose>compiles libraries</purpose>
</process>
<process rdf:ID="figures">
  <command>python3 main.py</command>
  <purpose>generates figures</purpose>
  <depends_on target="#make" />
</process>
```

The process labeled **figures** references the process labeled make as a prerequisite using the XML RDF reference syntax [4]. The example depects a simple **depends_on** predicate, but one might use the albeit more complex wfdesc vocabulary [9] to describe these dependencies. One can view specifying the software environment as just prerequisite steps in the computational DAG[^define-dag]. 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.

[define-dag]: A computaitonal directed acyclic graph (DAG) is a set of programs and pairs of programs (called links), where each link indicates the output of one program is the input to the other program. This is the basic concept of GNU Make, workflows, and build systems.

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. Purpose blocks might look more like this:

```
<purpose rdf:ID="pub">
  <!-- Links to an entire publication -->
  <cito:isCitedAsEvidenceBy rdf:resource="https://doi.org/10.1234/123456789" />
</purpose>
...
<purpose rdf:ID="fig-pub">
  <!-- Links to a specific figure within a publication -->
  <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>
...
<purpose rdf:ID="claim">
  <!-- Describes a specific assertion -->
  <cito:supports>
    <wikibase:Statement>
      <subject rdf:resource="https://www.wikidata.org/entity/Q12156" /> <!-- Q12156 refers to malaria -->
      <predicate rdf:resource="http://www.wikidata.org/prop/P1060" /> <!-- P1060 refers to disease transmission proce
      <object rdf:resource="https://www.wikidata.org/entity/Q15304532" /> <!-- Q15304532 refers to mosquitoes -->
    </wikibase:Statement>
  </cito:supports>
</purpose>
```

The block labeled **pub** uses the CiTO vocabulary [8] to explain that the result of that process is used as evidence in a specific publication. 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 [11], 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. The purpose description can be even more granular, using the DoCO vocabulary [2], which describes documents, as shown in the block labeled `fig-pub`.

One could even reference a Nanopublication, which is a semantic web description of the scientific claim [5], as in the block labeled `claim`. The claim is itself a subject-predicate-object triple, in this case relating identifiers from Wikidata [3] to express "malaria is transmitted by mosquitoes".

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

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, perhaps using wfdesc [9]. 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.

7 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. Computational experiments 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 execute 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. This is similar to the approach taken by FlaPy [6], a large-scale re-execution study for Python unittests, to install the Python environment. 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. . . .

In the best case, the execution description would be uploaded to the same repository or location that contains the source for the computational 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-engineering 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.

Is this another competing standard? It is something extra users have to do, but it does not attempt to displace their existing practice.

8 INCENTIVES FOR COMPUTATIONAL SCIENTISTS TO MAINTAIN EXECUTION DESCRIPTIONS

Computational scientists are incentivized to describe their project this way to benefit from the work of software-engineering researchers. When software-engineering researchers do a large-scale execution study, it is a “free” reproduction of their work.

To get an artifact evaluation badge, normally computational scientists 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. An execution description could make this nearly automatic; if a execution description exists, the artifact evaluator uses an 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”.

REFERENCES

[1] Christian Collberg and Todd A. Proebsting. Repeatability in computer systems research. *Communications of the ACM*, 59(3):62–69, February 2016.
[2] Alexandru Constantin, Silvio Peroni, Steve Pettifer, David Shotton, and Fabio Vitali. The Document Components Ontology (DoCO). *Semantic Web*, 7(2):167–181, January 2016. Publisher: IOS Press.
[3] Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandečić. Introducing Wikidata to the Linked Data Web. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig Knoblock, Denny Vrandečić, Paul Groth, Natasha Noy, Krzysztof Janowicz, and Carole Goble, editors, *The Semantic Web – ISWC 2014*, Lecture Notes in Computer Science, pages 50–65, Cham, 2014. Springer International Publishing.
[4] Fabian Gandon, Guus Shcreiber, and David Beckett. RDF 1.1 XML Syntax, February 2014.
[5] Paul Groth, Andrew Gibson, and Jan Velterop. The anatomy of a nanopublication. *Information Services & Use*, 30(1-2):51–56, January 2010. Publisher: IOS Press.
[6] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An Empirical Study of Flaky Tests in Python. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 148–158, April 2021. ISSN: 2159-4848.
[7] Peter Ivie and Douglas Thain. Reproducibility in scientific computing. *ACM Computing Surveys (CSUR)*, 51(3):1–36, 2018.
[8] David Shotton. CiTO, the Citation Typing Ontology. *Journal of Biomedical Semantics*, 1(1):S6, June 2010.

[9] Stian Soiland-Reyes, Sean Bechhofer, Khalid Belhajjame, Graham Klyne, Daniel Garijo, Oscar Coricho, Esteban García Cuesta, and Raul Palma. Wf4Ever Research Object Model. November 2013. Publisher: Zenodo.

[10] Stian Soiland-Reyes, Peter Sefton, Mercè Crosas, Leyla Jael Castro, Frederik Coppens, José M. Fernández, Daniel Garijo, Björn Grüning, Marco La Rosa, Simone Leo, Eoghan Ó Carragáin, Marc Portier, Ana Trisovic, RO-Crate Community, Paul Groth, and Carole Goble. Packaging research artefacts with RO-Crate. *Data Science*, 5(2):97–138, January 2022. Publisher: IOS Press.

[11] Stuart L. Weibel and Traugott Koch. The Dublin Core Metadata Initiative: Mission, Current Activities, and Future Directions. *D-Lib Magazine*, 6(12), December 2000.

[12] Edd Wilder-James. Description of a Project wiki, January 2017.