Self-verifiable Experimentation Pipelines

An Alternative to Research Artifact Descriptions

Ivo Jimenez and Carlos Maltzahn (UC Santa Cruz)

We make the case for incorporating self-verifiable experimentation pipelines in journals and conferences as a way of streamlining the review process for code and data associated to academic articles.

1 Introduction

Reproducibility is the cornerstone of the scientific method. Yet, in computer, computational and data science domains, a gap exists between current practices and the ideal of having every new scientific discovery be *easily* reproducible [1]. Advances in computer science (CS) and software engineering slowly and painfully make their way into these domains—even in CS research itself [2,3], paradoxically.

To address some of the reproducibility issues we face today, a growing number of CS conferences and journals incorporate an artifact evaluation process in which authors submit artifacts and descriptions[1] (ADs) that are tested by a committee, in order to verify that experiments presented in a paper can be re-executed by others. In short, an artifact description is a 2-3 page narrative on how to replicate results by making us of the submitted artifacts (e.g. source code and data), including steps that detail how to install software and how to re-execute experiments and analysis contained in a paper.

While an AD can serve as the basis for evaluating the reproducibility of a scientific exploration carried out as part of an article, it gives rise to some shortcomings during the artifact evaluation process [4]:

- Too many ad-hoc scripts.
- Intense schedule and little time for rebuttals.
- Sometimes hard to find evaluators with the appropriate skills or access to proprietary software or custom hardware.
- Lack of common workflows.
- No common formats and APIs (benchmarks, datasets, tools).
- Difficult to reproduce empirical results across a diverse software and hardware, and for distinct inputs.

In this short position paper, we introduce self-verifiable experimentation pipelines (SEP) and make the case for using them to address the challenges outlined above. In short, a SEP is a set of scripts associated to

[1] http://ctuning.org/ae/submission.html

an experiment, that are executed by an automation server. Instead of having authors submit artifacts and ADs, authors submit a link to an automation server that executes a SEP without any manual intervention.

2 Self-verifiable Pipelines

An alternative to the manual creation and verification of an Artifact Description (AD) is to use a continuous integration (CI) service[2] such as Jenkins[3] to ensure that the code (and data) associated to an article can be re-executed by others. If authors use a CI service to automate the execution and validation of their experimentation pipelines, the URL pointing to the project on the CI server that holds execution logs, as well as the repository containing all the automation scripts, can be used as the basis for evaluating the reproducibility of the code and data associated to an article. In other words, the repository containing the code for experimentation pipelines and the associated CI project serve both as a *Self-verifiable Experimentation Pipeline* (SEP). Thus, instead of submitting manually written ADs, authors can submit a link to the code repository where scripts for their pipelines reside, along with a link to the CI server that executes them, and use this as the basis for evaluating the reproducibility of their work.

While automating the execution of a pipeline can be done in many ways, in order for this approach to serve as an alternative to ADs, there are five high-level tasks that SEPs must carry out in every execution:

1. **Code and data dependencies**. Code must reside on a version control system (e.g. Github[4], Gitlab[5], etc.). If datasets are used, then they should reside in a dataset management system (Zenodo[6], CKAN[7], Datapackages[8], etc.). The experimentation pipelines must obtain the code/data from these services on every execution.
2. **Setup**. The pipeline should build and deploy the code under test. For example, if a pipeline is using containers or VMs to package their code, the pipeline should build the container/VM images prior to executing them. The goal of this is to verify that all the code and 3rd party dependencies are available at the time a pipeline runs, and that software can be build correctly.
3. **Resource allocation**. If a pipeline requires a cluster or custom hardware to reproduce results, resource allocation must be done as part of the execution of the pipeline. This allocation can be static or dynamic. For example, if an experiment runs on custom hardware, the pipeline can statically allocate (i.e. hardcode IP/hostnames) the machines where the code under study runs (e.g. GPU/FPGA nodes). Alternatively, a pipeline can dynamically allocate nodes on CloudLab [5], Grid500K [6], Chameleon [7], or a public cloud provider. These services

[2] http://en.wikipedia.org/?curid=22903426
[3] https://jenkins.io
[4] https://github.com
[5] https://gitlab.com
[6] https://zenodo.org
[7] https://ckan.org/
[8] https://frictionlessdata.io/data-packages/

typically offer infrastructure automation tools such as Geni-lib[9] (Cloudlab), Enos [8] (Chameleon), SLURM [9] (HPC centers), or Terraform[10] (AWS, GCP, etc.). All these tools can be used to automate infrastructure-related tasks.

4. **Environment capture**. Capture information about the runtime environment. For example, hardware description, OS, system packages (i.e. software installed by system administrators), information about remote services (e.g. version and state of a batch scheduler), etc. Open-source tools such as SOSReport[11] or Facter[12] can aid in aggregating this information.
5. **Validation**. Scripts must verify that the output corroborates the claims made on the article. For example, the pipeline might check that the throughput of a system is within an expected confidence interval (e.g. defined with respect to a baseline obtained at runtime), or that a numerical computation is within some expected bounds. This can be done with domain-specific tools [10], or generic data analytics stacks such as Pandas [11] or R [12].

A list of Popper [13] pipelines meeting the above criteria are available at Popper's official documentation website[13].

> **NOTE**: A SEP is a set of high-level requirements, rather than a detailed list of low-level requisites. Consequentially, there are many alternatives for creating a SEP. While the examples we link to are of Popper pipelines, this article has the intention to apply, in general, to any type of automated approach.

[9] https://bitbucket.org/barnstorm/geni-lib
[10] https://terraform.io
[11] https://github.com/sosreport/sos
[12] https://github.com/puppetlabs/facter
[13] http://bit.ly/popper-examples.

3 SEP-based Reproducibility Evaluation

We now illustrate how the peer-review process might work for a conference or journal that incorporates SEPs as a way of having authors demonstrate the reproducibility of their results. The process might look like this:

1. Authors create SEPs as part of their work.
2. Authors submit link(s) to the CI service that automatically executes the experimentation pipeline(s) associated to their submission.
3. Reviewers check logs of CIs to verify that the 5 components described in Section 2 are met.

Thus, incorporating SEPs to the peer-review process has several important outcomes:

1. **Better Experimentation Practices**. Authors that consider reproducibility as a first-class goal of their work increase the confidence in their results.

2. **Low-overhead Evaluations**. With SEPs, the burden of ensuring that a pipeline can be re-executed by others relies on the authors, who leverage the

CI service as the third party that is in charge of checking all their scripts and dependencies work as they should. As a consequence, all the experiment-specific knowledge in the scripts that the CI server executes is codified and automated. This in turn results in removing the need for experts that can re-build from scratch what an author originally ran. All a reviewer has to do is to audit that the automation server is indeed going through the steps outlined in Section 2. An optional requirement might be the ability for reviewers to trigger a new execution of the experiment, assuming resources are available, and there are no security or IP restrictions.

3. **Common high-level workflow**. Since the specification of what a pipeline is abstract, both authors and reviewers share the same mental mode when it comes to understanding what a scientific exploration is doing.

References

[1] D.L. Donoho, A. Maleki, I.U. Rahman, M. Shahram, and V. Stodden, "Reproducible Research in Computational Harmonic Analysis," *Computing in Science & Engineering*, vol. 11, Jan. 2009.

[2] G. Fursin, "Collective Mind: Cleaning up the research and experimentation mess in computer engineering using crowdsourcing, big data and machine learning," Aug. 2013. Available at: http://arxiv.org/abs/1308.2410.

[3] C. Collberg and T.A. Proebsting, "Repeatability in Computer Systems Research," *Communications of the ACM*, vol. 59, Feb. 2016.

[4] G. Fursin, "CGO/PPoPP'17 Artifact Evaluation Discussion," Feb. 2017. Available at: https://www.slideshare.net/GrigoriFursin/cgoppopp17-artifact-evaluation-discussion-enabling-open-and-reproducible-research.

[5] R. Ricci and E. Eide, "Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications,";*login:* vol. 39. Available at: http://www.usenix.org/publications/login/dec14/ricci.

[6] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed," *Int. J. High Perform. Comput. Appl.*, vol. 20, Nov. 2006.

[7] J. Mambretti, J. Chen, and F. Yeh, "Next Generation Clouds, the Chameleon Cloud Testbed, and Software Defined Networking (SDN)," *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, 2015.

[8] R. Cherrueau, D. Pertin, A. Simonet, A. Lebre, and M. Simonin, "Toward a Holistic Framework for Conducting Scientific Evaluations of OpenStack," *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[9] A.B. Yoo, M.A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, eds., 2003. Available at: http://link.springer.com/chapter/10.1007/10968987_3.

[10] I. Jimenez, C. Maltzahn, J. Lofstead, A. Moody, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "I Aver: Providing Declarative Experiment Specifications Facilitates the Evaluation of Computer Systems Research," *TinyToCS*, vol. 4, 2016. Available at: http://tinytocs.ece.utexas.edu/papers/tinytocs4_paper_jimenez.pdf.

[11] W. McKinney, "Data structures for statistical computing in python," *Proceedings of the 9th Python in Science Conference*, 2010.

[12] R. Core Team, "R: A language and environment for statistical computing," 2013.

[13] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "The Popper Convention: Making Reproducible Systems Evaluation Practical," *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017.