Considering the Development Workflow to Achieve Reproducibility with Variation

Michael Mercier, Adrien Faure, Olivier Richard

Considering the Development Workflow to Achieve Reproducibility with Variation

Michael Mercier[†⋆], Adrien Faure[†⋆], and Olivier Richard[⋆]

[†]Atos
[⋆]Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

Abstract

The ability to reproduce an experiment is fundamental in computer science. Existing approaches focus on repeatability, but this is only the first step to reproducibility: Continuing a scientific work from a previous experiment requires to be able to modify it. This ability is called reproducibility with Variation.

In this contribution, we show that capturing the environment of execution is necessary but not sufficient; we also need the environment of development. The variation also implies that those environments are subject to evolution, so the whole software development lifecycle needs to be considered. To take into account these evolutions, software environments need to be clearly defined, reconstructible with variation, and easy to share. We propose to leverage functional package managers to achieve this goal.

1 Motivation

Reproducibility is a wide notion that needs to be specified. Feitelson [4] has defined a taxonomy of the different way to reproduce scientific results. In this taxonomy, The first level of reproduction is the **Repetition**, i.e. do exactly the same experimental process to obtain the same results. The second level, **Replication** is similar but the experience's input is changed.

Currently, most of the reproducibility tools only support these two levels by capturing the software environment. Indeed, software environment is hard to reproduce, and without it, it is impossible to run the experiment. Also, experiment software environment tightly depends on the Operating System (OS) distribution it was built on. It is sometimes impossible to install it on an other distribution because of inter dependency issues. One approach to solve this problem is to snapshot the software environment into an image. But, even if an image of the experiment runtime environment is provided by the original author, continuing his work requires more than just repeating the experiment; to be able to corroborate someone's approach, we not only need to rerun experiments, but we also need to modify them: test new variations, add more parameters, and develop new features. This is the next level of reproducibility in the aforementioned taxonomy, called **Variation**.

Enabling scientists to reproduce an experiment with variation requires that the reproducer is able to rebuild experiment software with some modifications (even if the software is unmaintained and the tools necessary for building it, are long gone). It means that the reproducibility with variation can be achieved if the reproducer is able to reproduce not only the experiment "production" environment, but also the "development" environment which is necessary to modify the software. Moreover, when a variation of previous experiment produces new results, this experiment should also be reproducible.

In this context, we are proposing a new way of seeing reproducibility through the scientific software

development lifecycle. Each step in this lifecycle requires a software environment. We define a software environment by a set of applications and libraries, with all their dependencies, and their configurations, required to achieve a step in a scientific workflow.

2 Software Development Workflow and Reproducibility

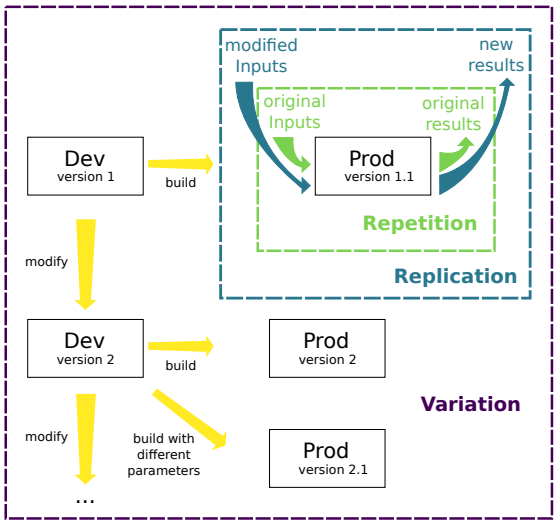

Figure 1: The different level of reproducibility in regard to the development lifecycle: Variation requires to enclose the development environment and to provide a way to modify it while keeping reproducibility.

In computer science, a scientific workflow contains a software development lifecycle that starts by setting up a development environment with build tools and dependencies. Then, this environment is used to build a production environment that will, in turn, be used to run the actual experiment. But, software development is an iterative process: one can produce different versions of the production environment, or even modify the development environment to update or to add tools. This process is in the middle of the scientific workflow and all the software environments produced, for development and production should be captured to enable reproducibility. The Figure 1, exhibits that the first two levels of reproducibility can be achieved with only one environment, but **reproducibility with variation requires taking into account the whole development process.**

To contribute to the scientific workflow, it has to be reproducible by itself: Thus, internal (e.g. colleague, intern) and external (e.g. scientist from other laboratory, reviewer) contributors have the capability to reproduce this workflow.

So, to provide this capability the development environment should be defined entirely, and every changes should be tracked. This statement also holds for others software environments involved in the authors' workflow, like the ones use for input data generation, or output data analysis.

3 Reproducible Software Environments with Nix

Courtès et al. [2] emphasize that functional package managers (FPM), like Guix and Nix, are good candidates to share complex and upgradable environment. In the following of this paper, we will focus on Nix, but most of the assertions also hold for Guix. The FPM are applying the concept of mathematical function to software packaging. Each software building process is described through a function. The dependencies are also functions that are given as inputs. This function, or package definition, allows to precisely describe a package: where and how to gather the source code, which commit to use, the dependencies and their versions, and finally how to build the package. When a package is built, the dependency graph is resolved by a lazy evaluation of the function parameters, and all the necessary piece of software are also built. The result of the evaluation of a package definition is called a derivation. A derivation is concretely a set of files that contains the results of the building process of the software, which placed on the special place that contains all the derivations: the store. Finally installing a software is simply exposing a derivation from the store through symbolic links. Nix packages are written in a functional Domain Spe-

cific Language (DSL). This ensures that each build is pure, i.e. it only depends on its inputs, and the same inputs give the exact same package even on a different machine.

To implement the workflow described in Section 2 with Nix, the critical feature is the capacity to create a software environment without virtualization. This feature is used to create an isolated environment for the package building process. An environment can be seen as a set of derivations and relies on the fact that an FPM can infer all the dependencies of a derivation, and only expose these dependencies on the specified environment. Installing the environment will expose to the user the packages described in the environment. Archiving an environment will extract the whole dependency tree of the environment and create a self-contained archive. The resulting tarball, also called a closure, contains every binary and file necessary to run the packaged applications. Thus, the environment can be installed on another machine without any external download or building process.

To achieve each level of reproducibility with Nix, the first requirement is to create a package definition for each application and its dependencies. Thankfully, Nix is a very active community and more than 40000 applications are already packaged in the main repository called "nixpkgs". It is also possible to maintain a private set of packages that import dependencies from "nixpkgs".

The first level of reproducibility, the Repetition, can be achieved by providing to an external scientist the production closure, that can be installed to repeat the experiment. The environment could also contain all scripts to deploy and run the experiment.

The second level of reproducibility, the Replication, that consists of replaying an experiment while changing its input, have the same requirement as the repetition: Only the production environment is needed.

The third level of reproducibility, the Variation, is where using Nix is the most advantageous. Nix provides the interesting feature, called the "nix-shell", permitting to enter the package build environment. Hence, packaging a software with Nix have the side effect to provide also the build environment for the users. Nix capacity to define software environment and software package in a unified way gives the scientist the ability to share a reproducible production environment, and the associated development environment, with a single definition.

4 Related works

The Popper method, proposed by Jimenez et al. [5], describes a structural framework for dependencies and artifacts. They identified a generic workflow describing an experimental methodology, from source code to the final manuscript of a contribution. Our contribution is compatible with this approach, the popper method proposes a structural organization of the experiment, whereas our approach proposes to implements a part of this workflow using Nix.

Repeatability has been the focus of previous works on reproducibility. The platform presented in [7], has the ability to instantiate an experiment environment in their infrastructure from a previously captured environment. The approach is interesting as it provides a way for a scientist to repeat experiments that requires specific hardware. Our approaches could be complementary to cover both hardware and software to provide a higher level of reproducibility. Boettiger et al. [1] survey how to use docker to do reproducibility, and also introduce the development environment. From our implementation with Nix, the docker approach shows similarities. However, Nix closure is more adapted than the Docker images for application packaging because Docker provides an inappropriate level of abstraction: Docker is about constructing and configuring a complete OS, instead of declaring application dependencies.

Constructing reproducible experiment and workflow with an FPM has already been explored. In [8], they build a toolset upon an FPM (GUIX), to facilitate the usage of bioinformatics common pipelines. They argue that using an FPM is a good foundation for reproducible computational experiment workflow.

The Blue Brain project [3], is a big project, with a complex software stack, that aims to build a mammalian brain with a computer. In addition to a structured development workflow (using git, agile methodologies, code reviewing), they decided to package

their workflow with Nix. They identified nine properties that are facilitated with Nix, from reproducibility to deployment and cross compilation. Their approach is based on internal needs and specific use case, whereas our contribution focuses on the role of the development lifecycle in the reproducibility, with Nix as a possible implementation.

5 Discussion

The proposed workflow is not considering input and output data. One approach is to package the experiment data inside the production environment. It is a viable solution for small amount of data, but most of the experiments have managed data separately with other tools.

The data related to the experiment is not only input and output data, the software environments that are used to develop and run the experiment also have external inputs, like the source code of the experiment, the dependencies, the build tools binaries, and the configuration files, i.e. all the other artefacts necessary to build the environments. With Nix, it is easy to extract these artefacts for the experiment production environment. Nix is capable to export all the dependencies of an environment in a closure that can be imported on any machine where Nix is installed. This feature of Nix is very hard to achieve with other kind of tools because of the lack of clear dependency definition. However, even if it is straightforward for the production environment, how to extract this closure for the development environment is unclear for now.

Capturing and archiving the environments closures is necessary for the Variation reproducibility, because the lifespan of Internet links is only a few months [6]. Even if Nix is capable to rebuild everything from source, the source code repository can be unavailable, breaking the environment reproducibility. The problem that emerges is that closures also have to be safely archived, versioned, and accessible for a long time period. A mid-term solution would be to store those closures (or even replace the closure content) using trusted centralized archives like Internet Archive[1] and Software Heritage[2].

In the case where the experiment results depend on the OS Kernel, (e.g. performance evaluation) this approach is not sufficient. Indeed, Nix packaging handles the whole application software but not the OS Kernel. So, the proposed workflow needs to be supplemented with a building process definition of the entire OS — and not just the application layer — to be able to reconstruct a complete OS image with variation. A Linux distribution based on Nix, called NixOS, is a good candidate.

When a specific hardware is necessary to achieve reproducibility, an additional layer of control is needed. Tesbeds like Grid'5000, Chameleon, and Emulab, are giving this level of control with the capability to create and deploy OS image on the fly on different hardware.

6 Conclusion

The reproducibility with variation is the next level of reproducibility that the Computer Science community should aim at. The Variation requires to take into account the software development workflow, including the capability to modify and rebuild environments. The use of functional package managers is a promising approach. This kind of tool permits to achieve this last mile to the reproducibility with variation, with a unified way to describe environments and packages, and a simple method to backup and to restore them.

References

[1] Carl Boettiger. "An introduction to Docker for reproducible research". In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79.

[1] `https://archive.org`
[2] `https://www.softwareheritage.org/`

[2] Ludovic Courtès and Ricardo Wurmus. "Reproducible and User-Controlled Software Environments in HPC with Guix". In: *Euro-Par 2015: Parallel Processing Workshops*. Ed. by Sascha Hunold et al. Cham: Springer International Publishing, 2015, pp. 579–591. ISBN: 978-3-319-27308-2.

[3] Adrien Devresse, Fabien Delalondre, and Felix Schürmann. "Nix based fully automated workflows and ecosystem to guarantee scientific result reproducibility across software environments and systems". In: *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*. ACM. 2015, pp. 25–31.

[4] Dror G Feitelson. "From repeatability to reproducibility and corroboration". In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 3–11.

[5] Ivo Jimenez et al. "The popper convention: Making reproducible systems evaluation practical". In: *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE. 2017, pp. 1561–1570.

[6] Steve Lawrence et al. "Persistence of web references in scientific research". In: *Computer* 2 (2001), pp. 26–31.

[7] Robert Ricci et al. "Apt: A platform for repeatable research in computer science". In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 100–107.

[8] Ricardo Wurmus et al. "Reproducible genomics analysis pipelines with GNU Guix". In: *bioRxiv* (2018). DOI: 10.1101/298653. eprint: https://www.biorxiv.org/content/early/2018/04/21/298653.full.pdf. URL: https://www.biorxiv.org/content/early/2018/04/21/298653.