ReproZip: Using Provenance to Support Computational Reproducibility

Fernando Chirigati
Polytechnic Institute of NYU
fchirigati@nyu.edu

Dennis Shasha
New York University
shasha@courant.nyu.edu

Juliana Freire
Polytechnic Institute of NYU
juliana.freire@nyu.edu

Abstract

We describe `ReproZip`, a tool that makes it easier for authors to publish reproducible results and for reviewers to validate these results. By tracking operating system calls, `ReproZip` systematically captures detailed provenance of existing experiments, including data dependencies, libraries used, and configuration parameters. This information is combined into a package that can be installed and run on a different environment. An important goal that we have for `ReproZip` is usability. Besides simplifying the creation of reproducible results, the system also helps reviewers. Because the package is self-contained, reviewers need not install any additional software to run the experiments. In addition, `ReproZip` generates a workflow specification for the experiment. This not only enables reviewers to execute this specification within a workflow system to explore the experiment and try different configurations, but also the provenance kept by the workflow system can facilitate communication between reviewers and authors.

1 Introduction

Good science requires reproducibility, not only to discover fraud but also to support the reuse of experimental techniques and software. Computational reproducibility means that a program P running on computational environment E at time T should yield the same answer on environment E' at time T'. Reproducing computational results *should* be easier than laboratory results because all the factors influencing a computational experiment consist of software and data. Unfortunately, few computational experiments are reproducible. Results are often described loosely through tables, plots and figure captions included in publications. Because no details of the computational steps are given, it is difficult to verify and reproduce many of the published results, and this has led to a credibility crisis in computational science [2].

The main difficulty is that authors must generate a compendium that encompasses all the inputs needed to correctly reproduce their experiments. In order to reproduce an experiment, we need detailed *provenance* which includes [5, 3]: (i) a description of the data; (ii) a complete specification of the experiment and its steps, preferably as a workflow in which parameters and computational tasks are explicitly defined; and (iii) information about the originating computational environment E (e.g., OS, hardware architecture, and library dependencies) that may be needed if the experiment is to be re-executed in a new environment E'. These different pieces of information need to be connected so that a complete and executable description can be generated.

Keeping track of this information manually is rarely feasible – it is both time-consuming and error-prone. First, computational environments are complex, consisting of many layers of hardware and software, and the configuration of the OS is often hidden. Second, tracking library dependencies is challenging, especially for large experiments. `ReproZip` aims to address these issues by systematically and automatically capturing the required information. The system does so by tracking operating system calls that originate from an experiment run. The information in these calls is then stitched together into a self-contained *reproducible package*, which include all the binaries, data and dependencies required to run a given command on the author's computational environment E. `ReproZip` also generates a workflow specification for the experiment, which can then be used to help reviewers explore and vary it. A reviewer can extract the files and workflow in another environment E' (e.g., the reviewer's desktop), without interfering with any program or dependency already installed on E'. The experiments and their deterministic processes can then be correctly reproduced and even modified in E'. By using the derived workflow to perform this exploration, provenance of the review process is automatically captured, and can serve not only to document the process but also as a means to support communication between authors and reviewers.

2 Related Work

Reproducibility Tools. A number of tools have been proposed to support the creation of reproducible experiments. Some are aimed at a particular domain, for example: GenePattern (`http://www.broadinstitute.org/cancer/software/genepattern`) is a genomic analysis platform; Madagascar (`http://www.ahay.org`) is used to analyze seismic data and supports multidimensional data analysis; and Sumatra [1] is used for numerical computations. Scientific workflow systems [4, 9, 13], on the other hand, are general and support the specification of arbitrary computational experiments. Because they have full control over the workflow execution, they can capture detailed provenance of the data derivation process. However, they do not capture the provenance of the computational environment. Thus, even though they support reproducibility, they do not support portability to new environments [3]. Another drawback comes from the fact that users must integrate the software they need into the workflow system. This task can be time consuming and there is not much incentive to do so after an experiment is complete, just for publication purposes.

Tools such as Chef (`http://wiki.opscode.com/display/chef`) and Puppet (`http://puppetlabs.com`) help users automate system configuration by creating recipes that can be re-used every time a new machine needs to be configured. Although these can help with reproducibility by reconstructing the required configuration, they can interfere with the reviewer's computational environment, creating a gratuitous disincentive to review. Virtual machines serve a similar purpose with the additional advantage of portability across operating systems, but the overheads of creating, storing and transferring the derived images can be high. CDE [7] offers a lighter-weight alternative to virtual machines. It relies on the *ptrace* call on Linux to identify the files required for running a particular command, and creates a package containing these files. This package can then be copied to different Linux installations where it can be run within the CDE environment: CDE dynamically changes the system calls to point to the correct files included in the package. Although both `ReproZip` and CDE trace operating system calls, they differ in significant ways: (i) with `ReproZip`, users have a greater control over the collected trace and can customize the reproducible package; (ii) `ReproZip` captures *and* stores the provenance in a database, which allows users to query the information, and if the same experiment and configuration need to be packed again, users need not re-run it; (iii) `ReproZip` focuses on *usability* for authors and reviewers – besides simplifying the creation of a reproducible experiment, the system generates a workflow specification for the experiment, which both facilitates the review tasks and makes it possible to capture the provenance of the review process; and (iv) CDE adds run-time overheads for executing a packaged experiment because it dynamically changes system calls during the execution – with `ReproZip`, once the package is extracted, there is no interference in its execution, and therefore no run-time overhead, thus supporting the validation of performance-sensitive experiments.

System-level Provenance Tools. Tools have been proposed that collect system-level provenance to provide a description of how data products were derived. `ReproZip` also captures system-level provenance, but it has a different goal: to attain reproducibility and portability, it uses this provenance to generate a compendium that includes the necessary components required to run a given experiment. Burrito [8] is a Linux-based system that captures OS-level provenance for derived data products and presents this information to users, who may add annotations and generate an HTML report that summarizes the computational activities. ES3 [6] uses *strace* to monitor operating system calls and constructs provenance graphs that resemble workflow specifications. These graphs describe the derivation of data products, but they are not executable. PASS [11] produces audit trails that are stored in a database and that can be queried. The system also generates a script to reproduce a particular object. However, this script can be executed only in the original environment.

3 Creating and Running Experiments

Packing. As illustrated in Figure 1, to create a reproducible experiment in environment E that is invoked by a program P (along with command line arguments), the author simply prepends the command with `ReproZip`. `ReproZip` makes use of two open-source tools to capture the necessary provenance: SystemTap [12] and MongoDB [10]. While the experiment is executed, SystemTap traces system calls (*execve*, *open*, *read*, *write*, *close* and *pipe*, to name a few). Through these system calls, it is possible to capture information such as *command-line arguments*, *environment variables*, *files read* and *files written*. This information is then stored in MongoDB, a NoSQL database, where it can be easily accessed and queried. Our choice of SystemTap and MongoDB was inspired by the Burrito System [8], which successfully used these tools to gather and store provenance for programs run on Linux.

Using the trace data, `ReproZip` creates a *provenance tree* of the experiment, where each node corresponds to an OS process. The tree is built incrementally. The root of the tree represents the main process of the experiment which is specified by the user when `ReproZip` is invoked. When a process corresponding to a node n spawns a process n' a new node is created for n' and an edge is inserted between n and n'. Each node in the

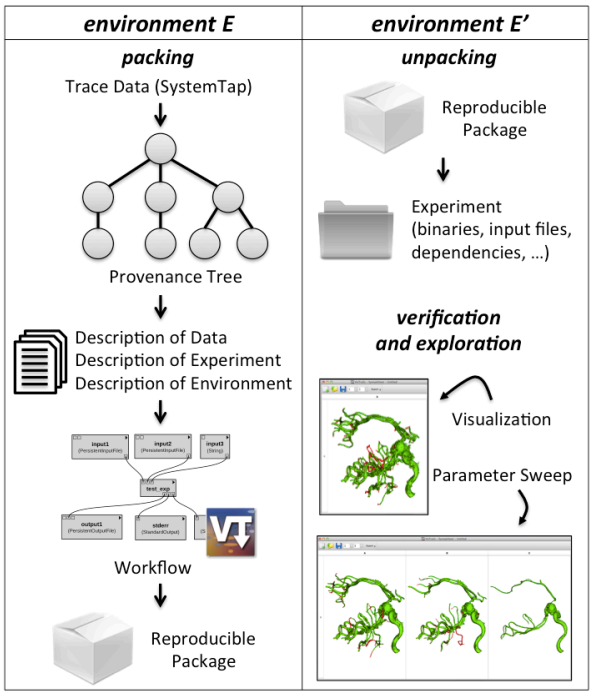


Figure 1: Creating reproducible experiments with `ReproZip`.

tree stores provenance data for the corresponding process, such as *command-line arguments*, *working directory*, *files read*, and *files written*. When the experiment terminates, `ReproZip` traverses the provenance tree to identify all the components involved in the execution and that should be included in the reproducible package. It collects the description of the data (input files and output files), the description of the experiment (executable programs), and the description of the computational environment (environment variables, library dependencies and information about the system/hardware).

Note that SystemTap captures *all* dependencies, some of which may not be necessary. Thus, before copying the files to the reproducible package, `ReproZip` offers the option to output a configuration file that lists all the identified programs, input files and dependencies. Authors may customize the configuration to exclude a specific file or a set of files (e.g., using Unix-shell style wildcards). This step is particularly useful to control the size of the package, for example, by discarding temporary files and omitting large files that can be obtained elsewhere.

The provenance tree, together with the identified input and output files, is also used to derive a workflow specification for the experiment. The main program of the experiment is wrapped in a workflow module that automatically takes the command-line arguments as inputs. By making these arguments explicit in the workflow specification, reviewers can immediately see which parameters can be changed and should be considered for validating the experiment.

`ReproZip` then creates the reproducible package that contains the workflow as well as all the required components from the author's environment E, using the same directory structure. The command-line arguments and the environment variables in the workflow are configured to reference the files that are inside the reproducible package. A mapping between symbolic links and target files is also added to the package, so that these links can be correctly created in the unpacking step.

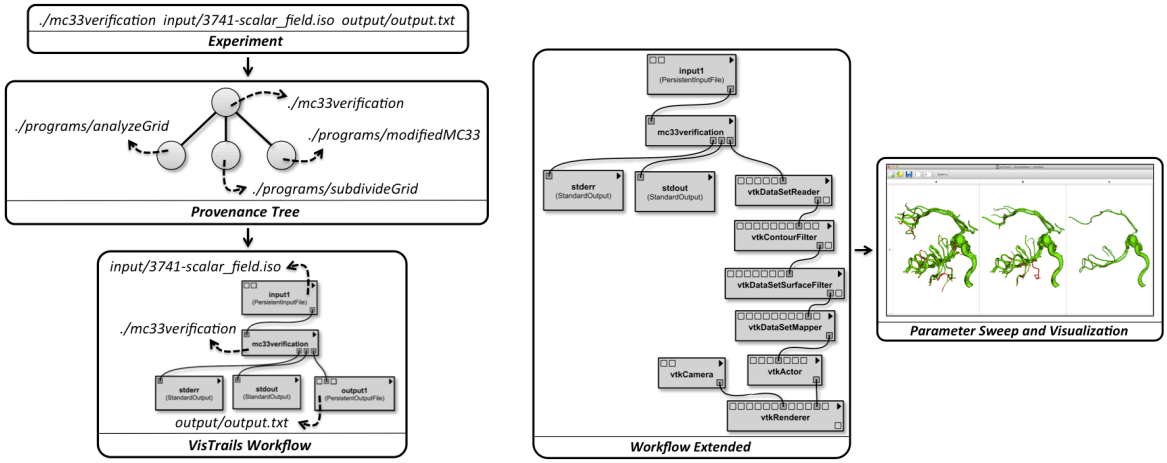
Figure 2a shows how the packing step derives a provenance tree and a workflow specification for a real experiment that verifies the topological correctness of marching cubes algorithms, generating a reproducible package[1]. As shown in the tree, the main program, *mc33verification*, calls three other programs: *analyzeGrid*, *subdivideGrid* and *modifiedMC33*. The provenance information captured for each node is also used to derive the workflow, in particular the input and output files that connect the different programs.

Unpacking. Given an experiment created in environment E, a reviewer can unpack and run it in a new environment E'. All the extracted files are placed in a user-specified directory, i.e., no changes are made to other directories in E'. The workflow is pre-configured by `ReproZip` so that paths to programs, input files, and paths defined in environment variables are adjusted to use the experiment directory in E'. Environment variables are configured *only* for the workflow execution – the original variables remain unchanged to avoid interfering with the normal operating environment of E'.

Verification and Exploration. After unpacking, users can run the experiment and examine the results. In the packing step, `ReproZip` captures what happens on a run, and therefore, the experiment will be reproducible if the process is *deterministic* – in case there is a non-deterministic step (e.g., race conditions on the code that may produce different outputs depending on processor speed and system overhead), it is not possible to guarantee that the original results will be reproduced.

The experiment can be executed from the command line, and users may also run the derived workflow, which can be run by the VisTrails system [4]. By using VisTrails, users can leverage a host of features that simplify validation and exploration. Because VisTrails provides a visual representation of the experiment, where input and output files are explicitly described, the reviewer can visually understand structure of the experiment. VisTrails also provides an interface to perform parameter sweeps and compare results side-by-side in a visual spreadsheet. Reviewers can extend the original workflow to explore different techniques or perform analyses (e.g., generate

[1]The resulting reproducible package had a size of around 40 MB, which is significantly smaller than, for instance, a virtual machine snapshot containing the same experiment (about 3 GB running Ubuntu).



(a) (b)

Figure 2: Making an experiment reproducible. In the packing step, a provenance tree and a workflow specification are derived (a). After unpacking, reviewers can extend the workflow to vary the experiment and view the results (b).

plots and other types of visualization) different from the ones produced by the authors. Finally, because VisTrails captures provenance of the verification process, this provenance can serve as a means of communication between reviewers and authors; for example, if reviewers find an issue with a given parameter combination, they can send the exact configuration back to the authors. Figure 2b illustrates the verification and exploration process performed in the marching cubes experiment. The workflow is extended to derive a visualization, and the parameter sweep feature of VisTrails is also used to compare results for multiple values for the isosurface. Examining the different isosurfaces enables the reviewer to verify the robustness of the marching cubes algorithm being evaluated.

4 Conclusion

Good computational science requires reproducibility, but the effort to achieve this has heretofore been significant. Our system `ReproZip` simplifies this task. By combining features of scientific workflows and tools that transparently and systematically capture the provenance of the execution of the experiments, `ReproZip` not only simplifies the process required to create reproducible experiments, but it also helps reviewers to verify the results and communicate their findings to the authors.

Although our initial evaluation has shown that `ReproZip` is effective for a wide range of experiments in different domains, there are some known limitations. For instance, execution will fail in environment E' if binaries are incompatible with the Linux kernel or hardware architecture or when a given executable uses a hard-coded absolute path. For such situations, our current approach is to `ReproZip` together with virtual machines.

We look forward to the day when reproducibility becomes routine and researchers mix and match the workflows of automatically packed software environments to create entirely new applications. `ReproZip` is a step in that direction.

Acknowledgments. We thank Jesse Lingeman, Lis Custódio and Tiago Etiene for providing their experiments that we have used to test `ReproZip`. This work has been partially funded by the National Science Foundation grants CNS-1229185, IIS-1139832, IIS-1142013, IIS-1050388.

References

[1] A. Davison. Automated capture of experiment context for easier reproducibility in computational research. *CISE*, 14(4):48 –56, 2012.

[2] S. Fomel and J. Claerbout. Reproducible researh. *CISE*, 11(1), 2009.

[3] J. Freire, P. Bonnet, and D. Shasha. Computational reproducibility: state-of-the-art, challenges, and database research opportunities. In *SIGMOD*, pages 593–596, 2012.

[4] J. Freire, D. Koop, E. Santos, C. Scheidegger, C. Silva, and H. T. Vo. *The Architecture of Open Source Applications*, chapter VisTrails. Lulu.com, 2011.

[5] J. Freire and C. T. Silva. Making computations and publications reproducible with vistrails. *CISE*, 14(4):18–25, 2012.

[6] J. Frew, D. Metzger, and P. Slaughter. Automatic capture and reconstruction of computational provenance. *CCPE*, 20(5):485–496, 2008.

[7] P. Guo. CDE: A Tool for Creating Portable Experimental Software Packages. *CISE*, 14(4):32–35, 2012.

[8] P. J. Guo and M. Seltzer. Burrito: wrapping your lab notebook in computational infrastructure. In *TAPP*, pages 7–7, 2012.

[9] B. Ludäscher and et. al. Scientific Workflow Management and the Kepler System. *CCP&E*, 2005.

[10] MongoDB. `http://www.mongodb.org/`.

[11] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *USENIX*, pages 4–4, 2006.

[12] SystemTap. `http://sourceware.org/systemtap/`.

[13] Taverna. `http://www.taverna.org.uk`.