September 2019

Documenting Computing Environments for Reproducible Experiments

Jason CHUAH, Madeline DEEDS, Tanu MALIK [a,1],
Youngdon CHOI, Jonathan L. GOODALL [b]
[a] *School of Computing, DePaul University, Chicago, IL 60637*
[b] *Dept. of Engineering Systems and Environment, Univ. of Virginia, Charlottesville, VA 22904*

Abstract. Establishing the reproducibility of an experiment often requires repeating the experiment in its native computing environment. Containerization tools provide declarative interfaces for documenting native computing environments. Declarative documentation, however, may not precisely recreate the native computing environment because of human errors or dependency conflicts. An alternative is to trace the native computing environment during application execution. Tracing, however, does not generate declarative documentation.

In this paper, we preserve the native computing environment via tracing and and automatically generate declarative documentation using trace logs. Our method distinguishes between inputs, outputs, user and system dependencies for a variety of programming languages. It then maps traced dependencies to standard package names and their versions via querying of standard package repositories. We use standard package names to generate comprehensive declarative documentation of the container. We verify the efficacy of this approach by preserving the native computing environments of several scientific projects submitted on Zenodo and GitHub, and generating their declarative documentation. We measure precision and recall by comparing with author-provided documentation. Our approach highlights over- and under-documentation in scientific experiments.

1. Introduction

Experiments in computational research are vital for establishing and validating an idea. A computational experiment typically has three components: goals, means, and claims [1]. While goals and claims are typically text-based, the means of an experiment, including experimental environment, procedures, and execution, are primarily computational. Sharing the means of an experiment is increasingly being recognized as critical toward establishing the reproducibility of results [2]. Previously sharing the means of an experiment implied sharing code and data relating to an experiment. There is increasing consensus that to establish reproducibility of results, authors must also share a description of computing environments, such as the documentation of used system libraries, configuration files, and parameters. Other users can use documentation about computing environments to build and extend environments.

[1]Corresponding Author: School of Computing, DePaul University, Chicago, IL 60637; E-mail: tanu@cdm.depaul.edu

Documenting computing environments, however, can be challenging. Typically, a user determines primary applications within an experiment's scope. The applications often depend upon complex software packages, which internally depend upon other packages. Documenting a computing environment often requires a user to know all packages and their dependencies including specific release versions. This can be too onerous for users who have not installed or built the application in different computing environments.

Recently, two prominent methods have emerged that document computing environments:

- The *container* method is a declarative method to describe application dependencies using a set of known packages. The known packages are determined either from documentation or from having a general familiarity of the application. If part of an application does not belong to any known package, the user documents this part manually.
- In the *tracing* method, a system observes the execution of an application and tracks direct or indirect references to binaries, input data files, and dependencies. The automatically tracked files comprises of all accessed package dependencies in a computing environment.

The container method is coarse-grained and is useful for documenting computing environments of standard applications, such as database servers, web servers, compilers, where an application builds from well-known packages. Systems such as Docker [3] and Singularity [4] adopt this approach. The tracing method is more fine-grained and is more useful for ad hoc, user-compiled applications where the user has either never built the application from source or does not recollect the complete dependency toolchain. Systems such as Sciunit [5,6], ReproZip [7], and CARE [8] adopt the latter approach for documenting dependencies.

While both methods document computing environments, using the documentation for establishing reproducibility of experiments is a challenge. Container methods rarely specify version numbers of binaries or packages; neither do the container engines (such as Docker) verify if the built container environment is the same as the experiment's native environment. Tracing methods are too fine-grained—at the granularity of each file in the package—and thus lose package-level semantics. Thus, while tracing methods guarantee exact native computing environments, without an accompanying documentation it is difficult to change or extend such environments.

In this paper, we define a reproducible experiment as a shared experiment that is repeated for verification and modified for establishing reproducibility. To repeat an experiment, we preserve its native computing environment via tracing. We document this environment in terms of declarative container-specific instructions. To generate such instructions, we first distinguish between inputs, outputs, user and system dependencies in a trace log for a variety of programming languages. We then map traced dependencies to standard package names and their versions via querying of standard package repositories to generate container-compatible documentation.

The advantage of this approach is that it repeats computational experiments exactly; The container contains only the necessary traced files, and its contents are declaratively documented for extensions and reproducibility of the experiment. We verify the efficacy of this approach by tracing the computing environments of several scientific projects submitted on Zenodo and GitHub. We compare their generated documentation with author-provided documentation and also if the container execution provides similar output. Results show instances of over- and under-documentation in scientific experiments.

We organize the rest of this paper as follows. We describe container and tracing methods to document computing environments in Section 2. We describe our process for generating declarative documentation in Section 3. We present our experiments in Section 4, and conclude in Section 5.

2. Documenting Computing Environments

In this section we describe the container and tracing methods for documenting computing environments. We use Docker [3] and Sciunit [5,6] as representative methods to describe the documentation.

2.1. Containers and Docker

Containers are an OS-level virtualization technique in which the virtual environment shares the OS kernel of the host environment. A container virtual environment isolates processes and files using namespaces, chroot, and cgroups. Docker is a container engine that allows users to create and maintain containers.

A *dockerfile* is a text-based file with declarative instructions defining the contents of a Docker container. The sequential instructions specify the order of execution for creating a desired image. Users typically build an image using a Dockerfile as an argument to the Docker build command. We summarize available instructions that help to document the native computing environment of an application. A Docker container can inherit infrastructure definitions from another container (`FROM` instruction). This can either be an operating system container, such as Ubuntu, but also any other existing container (e.g., with a pre-installed JDK installation). For maintenance, a dockerfile should provide the name and email of an active maintainer (`MAINTAINER` instruction). The `ENV` instruction sets environment variables. `ADD` and `COPY` instruction allow to place files into the container. A user may document a file as a URL, relative to the current path or as a zip file, which unpacks the archive within a container. `RUN` allows to execute any shell command within the container, and is often used to retrieve dependencies, and install and compile software. A container's main running process is the `ENTRYPOINT` and/or `CMD` at the end of the Dockerfile, which may subsequently fork other processes. Each instruction results in a layer in the Docker image. Thus a well documented dockerfile is a programmatic specification of the dependencies of an application.

2.2. Tracing and Sciunit

Application virtualization creates a sandbox in which it copies all files and environment variables referenced by an application. Similar to a container, a sandbox shares the host's kernel but unlike a container processes are not isolated—only files are. The sandbox engine monitors the running application process using *strace* and then copies each referenced file within a sandboxed directory. Strace internally attaches itself to the main application process using the *ptrace* system call, which monitors all the system calls of the running process [5]. Ptrace intercepts each system call to determine the running state of the process. The sandbox engine uses the arguments to file-system specific system calls to copy accessed files. Sciunit is an engine for creating and maintaining sandboxed applications.

The sandboxed creates a log of the traced system calls, which is a file-level documentation of the native computing environment. The log begins with a special "root path" which is where the application directory resides in the host system. The log contains all the dependencies identified during the reference execution audit. The sandbox engine locates the dependencies at the same path within the special root path as it identifies them in the original system. The trace log also contains interactions between processes if they *fork* or *exec* other processes and between processes and files when files are read and written. The log also stores the logical range of times that processes interacted with other processes or with files. Figure 1 shows a sample trace log of an high-energy physics application available on Zenodo [9]. Semantically the log comprises binaries, system libraries, configuration files, input data files, or temporary cache files, which must be distinguished into packages, sub-packages, and inputs to generate declarative documentation.

```
# @agent: maddie
# @machine: Linux cdm 4.15.0-66-generic #75-Ubuntu SMP x86_64 x86_64 x86_64 GNU/Linux
# @namespace: cde-root
# @subns: 1
# @fullns: cde-root.1
# @parentns: (null)
1572557755 14098 EXECVE 14104 /davix-tester /davix/build/test/functional ["./davix-tester", "--help"]
1572557755 14104 EXECVE2 -1
1572557755 14104 MEM 434176
1572557755 14104 READ /etc/ld.so.cache
1572557755 14104 CLOSE /etc/ld.so.cache
1572557755 14104 READ /usr/lib/x86_64-linux-gnu/libgtk3-nocsd.so.0
1572557755 14104 CLOSE /usr/lib/x86_64-linux-gnu/libgtk3-nocsd.so.0
1572557755 14104 READ /davix/build/src/libdavix.so.0
1572557755 14104 CLOSE /davix/build/src/libdavix.so.0
1572557755 14104 READ /usr/lib/x86_64-linux-gnu/libstdc++.so.6
1572557755 14104 CLOSE /usr/lib/x86_64-linux-gnu/libstdc++.so.6
1572557755 14104 READ /lib/x86_64-linux-gnu/libgcc_s.so.1
1572557755 14104 CLOSE /lib/x86_64-linux-gnu/libgcc_s.so.1
1572557755 14104 READ /lib/x86_64-linux-gnu/libc.so.6
1572557755 14104 CLOSE /lib/x86_64-linux-gnu/libc.so.6
1572557755 14104 READ /lib/x86_64-linux-gnu/libdl.so.2
1572557755 14104 CLOSE /lib/x86_64-linux-gnu/libdl.so.2
1572557755 14104 READ /lib/x86_64-linux-gnu/libpthread.so.0
1572557755 14104 CLOSE /lib/x86_64-linux-gnu/libpthread.so.0
1572557755 14104 READ /usr/lib/x86_64-linux-gnu/libssl.so.1.1
1572557755 14104 CLOSE /usr/lib/x86_64-linux-gnu/libssl.so.1.1
1572557755 14104 READ /usr/lib/x86_64-linux-gnu/libcrypto.so.1.1
1572557755 14104 CLOSE /usr/lib/x86_64-linux-gnu/libcrypto.so.1.1
1572557755 14104 READ /usr/lib/x86_64-linux-gnu/libxml2.so.2
1572557755 14104 CLOSE /usr/lib/x86_64-linux-gnu/libxml2.so.2
```

Figure 1. Description of trace logs in column order: (i) timestamp, (ii) process identifier, (iii) type of system call traced, and (iv) accessed file path name

3. Using Trace Logs to Generate Documentation

We describe how to automatically generate a Dockerfile from a trace log.

3.1. Generating Dockerfile Instructions

The workflow to generate a Dockerfile from trace logs is first encapsulating a computational artifact into a sciunit, and then using the trace log to distinguish between various entities namely inputs, outputs, processes, and system/user dependencies. Our method uses a representation of the trace log in terms of lineage graph to determine these entities and, once distinguished, maps them to declarative commands in the Dockerfile. We

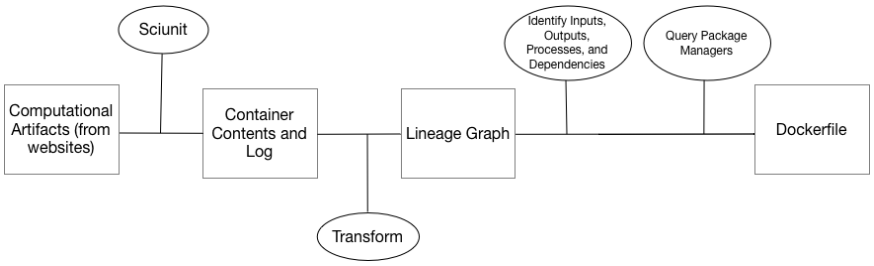

Figure 2. Workflow to generate Dockerfile declarative instructions from trace logs

manually compare generated documentation with author-supplied documentation and by building and executing the container. Figure 2 shows the workflow.

A trace logs maps to lineage graph via logged `READ/WRITE/EXECVE` calls. In particular, files represent entities and processes represent activities. Each `READ/WRITE/EXECVE` represent a data dependency. Given a lineage graph it can be used simply to determine inputs as nodes with no in-going edges, outputs as nodes with no out-going edges, and processes as nodes with process-to-process edges. The log is noisy in that it also contains information about temporary files, outputs, and process memory execution. We filter such files as this is execution-specific information and not relevant for documentation.

Distinguished files are converted to declarative instructions. Since each Dockerfile must begin with a `FROM` command, and the log provides OS information in its header as part of `@machine` command, our method instantiates an @machine specific base image. Identified input datasets and binaries are documented as `ADD` statements; Environment specific information is documented using the `ENV` command.

A bulk of the trace log consists of references to dependencies, which are either part of standard system packages or are user-defined. Distinguishing between system and user dependencies is crucial as the system documents them using different declarations. In particular, user dependencies are copied via the `COPY` command, and system dependencies are documented using the `RUN` command. One may perhaps copy system dependencies too using the `COPY` command. However, if all system dependencies are documented similar to user dependencies it leads to only one layer of the docker image. A one-command Dockerfile provides poor indication about the complexity of the software or the quality of the Dockerfile [10]. Knowing the documentation precisely is helpful if the user wishes to extend the experiment. In such cases a more verbose documentation can indicate if extended dependencies will conflict with the container contents.

Unlike a user dependency which is mentioned directly as part of a `COPY` command, a system dependency cannot be directly mapped to a `RUN` command. We must identify the corresponding package that maps to this dependency. Packages contain more than one dependency and for all identified dependencies only the corresponding package need to be stated in the `RUN` command. For instance, if the trace log specifies a path to *libcrypto.so.1.1*, then the corresponding `RUN` command is to invoke the package manager to install *libssl* as in `RUN apt-get install libssl1.1`. *libssl* also maps to *libssl.so.1.1* and *libpthread.so.0*.

To generate the above `RUN` instruction, we need to map between a dependency and its package name. For this we use programming language-specific package managers. Our method currently works for C/C++, Python and R languages. For instance C/C++

libraries are searched using *apt-get* and *yum* package managers, and Python libraries are searched through *pip*. Package managers provide a search interface to determine a package name from a dependency file[2]. Sometimes the search returns multiple packages, and our current policy is to document the first package obtained as part of the search result. Querying language-specific repository for each package is costly (˜1 min for each package query). For this we create a local database to curate all queries packages and known sub-packages to avoid repeated querying.

The process of mapping between a dependency and its package depends on how the language maintains the packages in the file system. While the most common paths where libraries are installed are /usr/lib and /lib, depending upon the language packages may be found under sub-directories site-packages, dist-packages or site-library (Python and R) or under architecture specific directories (C/C++). Typically a package name follows these paths along with the version information.

```
# information received from:
# https://zenodo.org/record/3379611
# MAINTAINER georgios.bitzes@cern.ch

# this is assumed for now.
FROM ubuntu

# build-essential: contains gcc/g++ compilers
#
# dependendencies identified from:
# libssl1.1 ------> libssl.so.1.1
# libxml2 --------> libxml.so.2
#
RUN apt-get update \
        && apt-get install -y build-essential \
        && apt-get install -y libssl1.1 \
        && apt-get install -y libxml2

# project name is automatically determined from the parent directory
WORKDIR /davix

# places the executed binary and relative dependency in the working directory '/davix'
ADD davix-tester libdavix.so.0 ./

# executes the original program with the same arguments
CMD ["./davix-tester", "--help"]
```

Figure 3. A automaticaly generated Dockerfile for Davix

Detecting packages using path information may result in false positives. In particular, Python and R interpreters check for the existence of several packages during loading. The container log is not able to distinguish between paths in which a package is checked for its existence and a path in which the the content of the package is used (e.g. when libraries are indeed imported into source code). This distinction is possible by tracing the logs at a finer granularity and determining if content was indeed read, but that increases the overhead of tracing. Since the paths of checked packaged is same as the path of a used package, we distinguish them by identifying patterns of usage. For instance, if a package is simply checked then there is an entry in specific paths such as *dist-info*, *egg-info* for Python and R, but no sub-package path or file path within a package is present. Such false positives do not arise with respect to C/C++.

[2]For R, the CRAN repository only provides a GUI-based search interface which required web-scraping to build a local database.

Finally, the sciunit contents are copied into a Docker container but the instructions for creating path directories are not documented as it does not provide any information about the native computing environment. Instead we provide as comments the location of directories which are of relevance to the user. We also use comments to document versions of packages when package managers do not install specific versions of packages. Figure 3 shows an example of a generated Dockerfile from the trace log shown in Figure 1. We would like to highlight that in case of Python applications it is sufficient to use trace logs to generate a setup file instead of a Dockerfile, and the Dockerfile can simply reference to executing the setup file. Such optimizations are outside the current scope.

4. Experiments

We collected scientific computational artifacts from GitHub and Zenodo. GitHub and Zenodo [9] artifacts are not shared using any virtualization. Our experimental setup consists of the following steps: (i) download and manually install a project; (ii) determine if they execute successfully in a new environment, possibly generating an output; (iii) execute the application to containerize under Sciunit; (iv) use the Sciunit log file to generate a Dockerfile that builds a Docker container consisting of necessary dependencies, data, and source code; and finally (v) execute the Docker and Sciunit containers to determine is same output is produced. Since our setup depended on generating a valid result, we downloaded, in total, about 100 repositories from GitHub and Zenodo. However, we could build and successfully execute only 29 repositories. Out of these, 19 are Zenodo repositories and 10 are GitHub repositories. The other repositories reported an error which we did not try to fix. The successful repositories consisted of 19 Python applications, 10 C/C++ applications, and 1 R application. The first three columns of the Table 1 shows the information. We would like to re-emphasize that since we are not original authors of these applications, we assumed the container result as the correct one if it matched with the execution.

Language	Source	# of repositories	# of Dockerfile built	# of Sciunit executed	# of Docker execution
C/C++	Zenodo	10	10	10	7
Python	Zenodo & GitHub	18	18	18	14
R	GitHub	1	1	1	1

Table 1. Repository Description

We measure two kinds of experiments: (i) if the generated Dockerfile generates the same output, and (ii) if the author provided documentation corresponds to the Dockerfile documentation. The former is determined by first building the Dockerfile and then by manually comparing the contents of the output in the container with the native execution. Table 1 shows that we are able to build Dockerfiles for all projects. However some of the C/C++ and R projects did not produce the same output. In our dataset, most C/C++ projects that did not generate a correct output had a networking component which we believe was not sufficiently documented. Currently the trace log does not audit network

events. The single R project that did not run was due to poor mapping of the dependency to a package name. This owes to the poor search interface of the R package manager. Python projects were the most stable in terms of result comparison.

To measure if the author provided documentation corresponds to the Dockerfile documentation. we measure the quality of documentation in terms of precision and recall. Precision is defined as the ratio of the number of user-listed packages identified in the trace log to all that our method can potentially identify, and recall is defined as the ratio of the number of user-listed packages to all those that our method identified. In other words, precision computes the number of application-specific packages identified amongst all identified packages, and recall computes the number of packages identified by our system which are also listed by the author in the documentation. Equations (1) and (2) state them formally.

$$Precision = \frac{Identified\ Packages}{Total\ Identified\ Packages} \tag{1}$$

$$Recall = \frac{Identified\ Packages}{User\ Listed\ Packages} \tag{2}$$

We show how these measure compare with listed dependencies in two scientific projects in Python and R, respectively. pySUMMA is a wrapper for the SUMMA [11,12] computational hydrology model in which authors list seven packages and their versions as were found compatible in their environment. Figure 4 shows the listed dependencies. As available on the GitHub repository [13], pySUMMA is not encapsulated in a container. We downloaded pySUMMA in a virtual environment and installed it with all its dependencies and their versions as specified. Figure 4 shows the five packages and their corresponding versions that were identified as application dependencies. In particular, the model does not identify *seaborn* and *jupyterthemes*: the file paths from the log show that *seaborn* is a sub-package of *matplotlib*, and so it is not a package that the author must explicitly install. *jupyterthemes* on the other hand appeared as a *dist-info* path. So even though *jupyterthemes* was listed it was not actually used. Several other packages are listed in the log, such as NetCDF41.4.2 and geopandas0.4.0/ These packages were identified within the logs and subsequently added by the authors in the setup.py file. Several other packages internal to the Python interpreter are listed as 'Python Built-in Packages'. These identified packages come bundled with standard Python environment. In Figure 4, precision is 0.083 as out of 60 packages identified, the author only listed 7. Recall is 0.714 as out of 7 that the author listed on the provided README file, only 5 were identified.

Our approach works similarly for documenting an R application. Food Inspection Evaluation (FIE) is an R application in Table 3 and includes user-defined, standard and commercial packages. However, in this case, the GitHub repository (not actively maintained) [14] does not list any R packages. It lists only a few C libraries. By downloading and creating a sciunit, we identified all R user-defined and standard packages and C dependencies. For lack of space, we omit the result, and direct the user to our Github repository of analyzed packages.

Table 2 shows the precision and recall for C/C++ and Python Zenodo applications respectively. Tables 3 shows the precision and recall results for some of the Python and R GitHub applications. Most of GitHub applications are poorly documented. For these

Author-listed Packages (5)	Identified Packages (7)
xarray{0.10.7} numpy{1.16.1} matplotlib{3.0.2} hs-restclient{1.3.3} ipyleaflet{0.9.2} seaborn{0.9.0} jupyterthemes{0.20.0}	xarray{0.10.7} numpy{1.16.1} matplotlib{3.0.2} hs-restclient{1.3.3} ipyleaflet{0.9.2}
Other Identified Packages (31)	
Pygments{2.2.0} asyncio backcall{0.1.0} blinker{1.3} certifi{2018.10.15} cftime{1.0.2.1} geopandas{0.4.0} html http ipykernel{5.1.0} ipython-genutils{0.2.0} ipython{7.1.1} ipywidgets{7.4.2} jedi{0.13.1} jupyter-core{4.4.0} netCDF4{1.4.2} pandas{0.23.4} parso{0.3.1} pexpect{4.6.0} prompt-toolkit{2.0.7} ptyprocess{0.6.0} pyparsing{2.3.0} pysumma{0.1} pytz{2018.7} pyzmq{17.1.2} requests-oauthlib{1.0.0} requests-toolbelt{0.8.0} tornado{5.1.1} traitlets{4.3.2} traittypes{0.2.1} wcwidth{0.1.7}	
Python Built-in Packages (24)	
chardet collections concurrent ctypes dateutil distutils email encodings idna importlib jinja2 json logging markupsafe multiprocessing oauthlib pkg_resources pydoc_data requests sqlite3 unittest urllib urllib3 xml	

Figure 4. Listed Vs Identified Packages in pySUMMA. 2 of the Author-listed packages were not identified as they are not used by the program. We also identified 31 other packages that the author does not list.

GiHub/Zenodo Object Name	Precision	Recall
cpp-atlas	0.222	0.66
hdt-CPP	0.4	0.4
simple-web-server	0.33	0.5
research-ocr	0.024	1
c-blockchain	1	1
scram	0.25	0.66
activia	1	1
Dgtal	0	0
Davix	0.28	1
causaltrail	0	0

Zenodo Object Name	Precision	Recall
clam	0.12	0.5
informers	0.208	0.714
pydov: v0.3.0	0.12	0.75
lmfit-py 0.9.14	0.167	0.8
jungleweather	0.02	1.0
pianoplayer	0.069	1
GraSPy 0.1	0.12	1
fbpic	0.156	1
pyBathySfM v4.0	0.208	1

Table 2. Precision/Recall of C/C++ Zenodo applications (left) Precision/Recall of Python Zenodo applications (right)

applications, recall is assumed to be 1 as we consider our manual virtual machine installation as the total number of author-listed packages. All Zenodo applications were documented in that authors do list install requirements. For applications on both repositories, precision value is low since we constantly report many sub-packages that are not reported by the author. However, our recall value is on the high end as we identify most of the author-listed or required packages for the application to run.

GitHub Object Name	Precision	Recall
zagats	0.12	0.5
snake	0.208	0.714
gooselife	0.12	0.75
pySUMMA	0.167	0.8
newmeric	0.02	1.0
asplos	0.069	1
craps	0.12	1
imdb-deeplearn	0.156	1
FIE	0.208	1
Image morphing	0.208	1

Table 3. Precision/Recall of Python & R GitHub applications

5. Conclusions

In this paper we developed a model to interpret container logs and document dependencies of applications. Although precision is low, high recall shows that necessary dependencies captured during tracing can be used to build a comprehensive and verbose dockerfile. We believe this mapping from a trace provides low-overhead for users to create and maintain containers, which is increasingly important for conducting reproducible research. For full documentation of our experiments and artifacts please visit: https://tanum@bitbucket.org/geotrust/trace-descriptions.git

Acknowledgements

This work is supported in part by the Better Scientific Software Fellowship to Malik and by NSF grants ICER-1639759 and CNS-1846418.

References

[1] V. Stodden and *et. al.*, *Implementing reproducible research*. Cambridge University Press, 2013.
[2] National Academies of Sciences, Engineering, and Medicine, *Reproducibility and Replicability in Science.* Washington, DC: The National Academies Press, 2019. https://doi.org/10.17226/25303.
[3] *Docker* https://www.docker.com/, **Online; accessed 8-Jan-2019**, 2019.
[4] G. Kurtzer and *et. al.*, *Singularity: Scientific containers for mobility of compute*, PLoS One,12(5), 2017.
[5] D. H. Ton That and *et. al.*, *Sciunits: Reusable Research Objects*, IEEE eScience, 2017.
[6] Q. Pham, T. Malik, and I. Foster, *Using Provenance for Repeatability*, In USENIX Theory and Practice of Provenance (TaPP), 2013.
[7] F. Chirigati, D. Shasha, and J. Freire, *ReproZip: Using Provenance to Support Computational Reproducibility*, In USENIX Theory and Practice of Provenance (TaPP), 2013.
[8] Y. Janin and *et. al.*, *CARE, the Comprehensive Archiver for Reproducible Execution*, In the Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering, 2014.
[9] *Zenodo* https://zenodo.org, **Online; accessed 8-Jan-2019**, 2019.
[10] J. Cito, G. Schermann, and *et. al.*, *An Empirical Analysis of the Docker Container Ecosystem on GitHub*, In International Conference on Mining Software Repositories (MSR), 2017.
[11] M. P. Clark, B. Nijssen, and et. al., A unified approach to process-based hydrologic modeling, Part 1: Modeling concept. Water Resources Research, 51.
[12] M. P. Clark, B. Nijssen, and et. al., A unified approach for process-based hydrologic modeling, Part 2: Model implementation and example applications. Water Resources Research, 51.
[13] pySUMMA. https://github.com/uva-hydroinformatics/pysumma
[14] Food Inspection Evaluation. https://github.com/Chicago/food-inspections-evaluation