Making computer science results reproducible - A case study using Gradle and Docker

Wilfried Elmenreich Corresp., [1] , **Philipp Moll** [1] , **Sebastian Theuermann** [1] , **Mathias Lux** [1]

[1] Universität Klagenfurt, Klagenfurt, Austria

Corresponding Author: Wilfried Elmenreich
Email address: wilfried.elmenreich@aau.at

This paper addresses two questions related to reproducibility within the context of research related to computer science. First, requirements on reproducibility are analyzed based on a survey addressed to researchers in the academic and private sector. The survey indicates a strong need for open but also easily accessible results, thus reproducing an experiment should not require too much effort. The results from the survey are then used to formulate general guidelines for making research results reproducible. In addition, this paper explores a number of existing software tools that could bring forward reproducibility in research results. After a general analysis of tools a further investigation is done via three case studies based on actual research projects which are used to evaluate the previously introduced tools. Results indicate that due to conflicting requirements, none of the presented solutions fulfills all intended goals perfectly. However, we present requirements and guidelines for making research reproducible. While the main focus of this paper is on reproducibility in computer science, the results of this paper are still valid for other fields using computation as a tool.

Making Computer Science Results Reproducible – A Case Study using Gradle and Docker

Wilfried Elmenreich[1], **Philipp Moll**[1], **Sebastian Theuermann**[1], **and Mathias Lux**[1]

[1]**Alpen-Adria-Universität Klagenfurt, Austria**

Corresponding author:
Wilfried Elmenreich[1]

Email address: wilfried.elmenreich@aau.at

ABSTRACT

This paper addresses two questions related to reproducibility within the context of research related to computer science. First, requirements on reproducibility are analyzed based on a survey addressed to researchers in the academic and private sector. The survey indicates a strong need for open but also easily accessible results, thus reproducing an experiment should not require too much effort. The results from the survey are then used to formulate general guidelines for making research results reproducible. In addition, this paper explores a number of existing software tools that could bring forward reproducibility in research results. After a general analysis of tools a further investigation is done via three case studies based on actual research projects which are used to evaluate the previously introduced tools. Results indicate that due to conflicting requirements, none of the presented solutions fulfills all intended goals perfectly. However, we present requirements and guidelines for making research reproducible. While the main focus of this paper is on reproducibility in computer science, the results of this paper are still valid for other fields using computation as a tool.

INTRODUCTION

Being able to reproduce the results of an experiment is a fundamental principle in science across all disciplines. Reproducing results of published experiments, however, often is a cumbersome and ungrateful task. The reason for this is twofold: First, some fields like for example biology are concerned with complex and chaotic systems which are hard to reproduce (Casadevall and Fang (2010)). At the same time, when approaching the digital world, we would expect software-based experiments to be easily reproducible, because digital data can be easily provided and computer algorithms operating on these data are typically well-described and deterministic. However this is currently often not the case due to a lack of disclosure of relevant software and data that would be necessary to repeat a simulation. Ongoing open science initiatives aim at having researches providing access to data and software together with their publication in order to allow reviewers to make well-informed decisions and to provide other researchers with the information and necessary means to build upon and extend the original research (Ram (2013)).

This paper addresses two questions related to reproducibility using two different methodologies. First the current practice, awareness of the subject and possible concerns have been examined by an online survey, which addressed people at different stages in universities, research institutions and companies. After evaluating the results and elaborating a number of relevant points to be addressed, we identify four tools that can and have been used to support reproducibility. We present three case studies where three different types of software projects are packaged to provide an accurate and easy possibility for reproducing results in an identical environment. Due to conflicting requirements, none of the presented solutions fulfills all intended goals perfectly. The reasons for this are elaborated in Section 6.

1 RELATED WORK

Walters Walters (2013) notes that *it is often difficult to reproduce the work described in molecular modeling and chemoinformatics papers*. A main problem is the absence of a disclosure requirement in many scientific publication venues so far. Morin et al. (2012) reports that in 2010 out of the 20 most cited journals, only three had editorial policies requiring availability of computer source code after publication. Fortunately, this situation is changing to the better, for example *Science* introduced a policy that requires authors to make data and code related to their publication available whenever possible (Witten and Tibshirani (2013); Peng (2011); Hanson et al. (2011)). In a comment to this policy, Shrader-Frechette and Oreskes (2011) brings up the issue that privately funded science, despite it may well be of high quality, is not subject to the same transparency requirements as public science. Another obstacle is the use of closed-source tools and undisclosed software results in publicly funded research software development projects (Morin et al. (2012)).

Several papers report on case studies for data repositories in the context of reproducibility. Examples are from such different domains such as geographic information systems (Steiniger and Hunter (2013)), astrophysics (Allen and Schmidt (2015)), microbiome census (McMurdie and Holmes (2013)) and neuroimaging (Poline et al. (2012)). These approaches are promising, but it cannot be expected that the described approaches are going to be used beyond the field they have been introduced. Simflowny (Arbona et al. (2013)) is another platform for formalizing and managing the main elements of a simulation flow, which is not tied to a field, but a specified simulation architecture. The Whole Tale approach (Brinckman et al. (2018)) aims at linking data, code, and digital scholarly objects to publications and to integrate all parts of the research story. Other works focus on code and data management, such as by Ram (2013) suggesting very general version control systems like Git for transparent data management in order to enable reproducibility of scientific work. The CARE approach (Janin et al. (2014)) extends the archiving concept with an execution system for Linux systems, which also takes software installation and dependencies into account. Docker (Boettiger (2015)), which will be closer examined in this paper, provides an ever more generic approach by utilizing virtualization for providing cross-platform portability.

In 2016, ACM SIGCOMM Computer Communication Review conducted a survey on reproducibility with 77 responses from authors of papers published in CCR and the SIGCOMM sponsored conferences (Bonaventure (2017)). The responses showed that there is a good awareness of the need for reproducibility, and a majority of authors either considered their paper self-contained or have released the software used to perform experiments. However, there was a shortcoming of releases of experimental data or of modifications of existing open source software. An open question part of the survey indicated a need for encouragement for publishing reproducible work or for papers that attempt to reproduce or refute earlier results.

2 SURVEY

In computer science quite a lot of research is backed up by prototypes, implementations of algorithms, benchmarking and evaluation tools, and data generated in the course of research. A critical factor for cutting edge research is to be able to build upon the results of other researchers or groups by extending their ideas, applying them to new domains or by just reflecting them from a new angle. This is easily done with scientific publications, which are nowadays mostly available online. While the hypotheses, findings, models, processes and equations are published, the data generated and the tools used for generating the data and evaluating new approaches are sometimes only pointed out, but have to be found elsewhere.

Our hypothesis in that direction is that there is a gap between scientific publishing on the one hand and the publication of software artifacts and data for making results reproducible for other researchers on the other hand. In that sense we created a survey asking researchers in the computer science field for their approach and opinion. The survey consists of five parts. First, basic demographic information is surveyed, including the type of research, the field of research, the typical research contribution, and the type of organization. Second, the common practice of the participant for publishing software artifacts and data is surveyed, ie. the steps the researchers take to make their work reproducible. Third, we focused on the researchers expectations when they want to reproduce scientific results. Fourth, we asked for opinions on integrating the question of reproducibility in the peer review process. Finally, we collected additional thoughts with open questions.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018

2.1 What Researchers Want

With 125 participants most of the people where from academic research with 74 out of the 125 working or studying at a university and 35 of 125 from research institutes. 13 participants noted that they are mainly working for a company, 2 were private researchers, 1 from school. Within their career 30% of the participants were PhD students, 28% were professors or group leader, 17% worked as researchers within a project, 12% were principal investigators, and 9% were undergraduate students at the time of the study. Three participants were head of departments or organizations, and only two participants indicated that they are postdoctoral researchers. Computer science or computer engineering was the field of research for 72% of the participants. 7.2% of the participants came from electrical engineering and 4% from information systems.

Within the survey we used Likert scales indicating the level of agreement in five steps from 1 (strongly agree) to 5 (strongly disagree). Fig. 1 visualizes the answers in the section on how researchers commit to reproducibility. As can be seen from the chart, the majority of people want to reproduce results from other researchers or groups: 103 out of 125 indicated agreement. Even more (110 out of 125) considered reproducible results as added value for publishing. More than half of the participants (67 out of 125), however, are not willing to pay for open access publishing of their results. There is also a difference in numbers between researchers who typically try to reproduce results from others (53 out of 125) and those who want to do it (103 of 125). Moreover, more than 100 of 125 participants want to publish their software and tools and intend to give detailed documentation on how to produce their results.

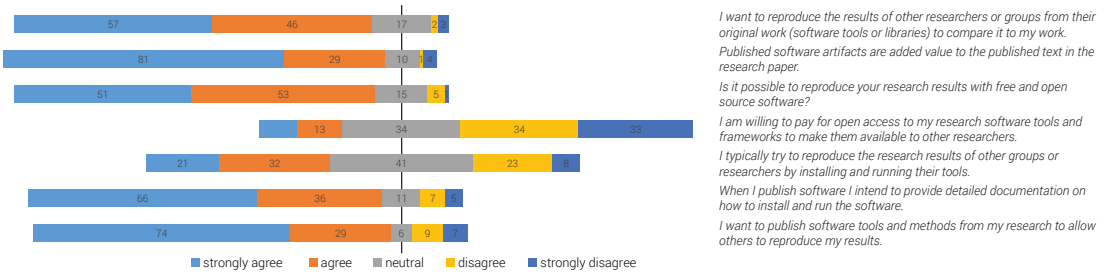

Figure 1. Responses from the second part of the survey cover the common practice of participants.

The question on how many hours researchers wanted to invest into making their results reproducible (excluding three outliers with an answer of 1000 or more hours) led to an average of 24.4 hours, whereas 4 participants indicated 0 hours, 55 participants wanted to invest from 1-16 hours, 37 participants indicated they'd invest 20-80 hours and 6 participants would invest 100-250 hours. 35 participants haven't published any software artifacts at the time of the study. For the other participants, means of making their results reproducible were – multiple means could be specified – detailed instructions (68), make scripts (54), installation scripts (34), virtualization software (29), and container frameworks (15). There were two mentions of hosting web front ends as means of making results available and three mentions of public source code repositories as platforms for distribution. Answering the question if results could be reproduced with free software, participants indicated the use of non-free programming language environments (50 times part of the answer), licensed operating systems (35), copyrighted materials (19), and commercial tools (11).

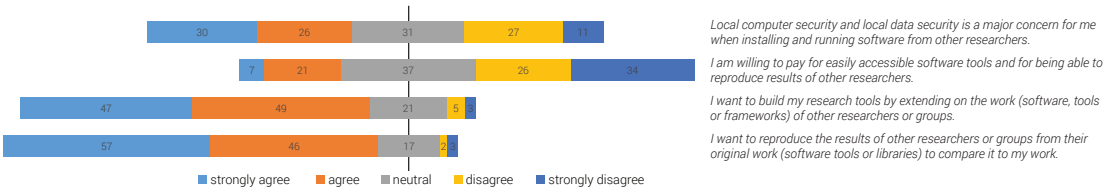

Figure 2. Responses from the third part of the survey cover the expectations of the participants.

Fig. 2 visualizes answers from the survey regarding the researchers' expectations on the reproducibility of the work of others. As can be seen, most of the participants want to use software from others and want to build upon the work of others. However, paying for such a service is not agreeable for nearly half of

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018

the participants. Computer security when installing programs from others is a major concern for 56 out of 125 participants. Leaving aside five outliers (either 0 or greater than 100) participants stated that software for reproducing results should be available for an average of 9.1 years. Whereas 30 participants think it should be up to 0.5-3 years, 55 indicate it should be 4-5, 26 state 8-10, and 9 think it should be more than ten years available. Answering how much time in hours participants would invest to get software of others running to reproduce results, the average amount was 15.94 hours leaving aside 5 outliers with 100 or more hours. 3 participants would invest no time at all, 47 participants would invest 1-8 hours, 32 would invest 10-24 hours, and 18 participants 30-80 hours. Regarding the major concerns on installing and running software from other researchers, participants mentioned that critical factors for them are primarily the ease of installation (104), license issues (72), and hardware requirements (71). Fewer participants noted the size of the download (27).

An open question asking for reasons why software artifacts should be published yielded diverse answers. Main lines of arguments were improvements in credibility and reliability of results, building trust and improving understanding of results of others. Besides that people pointed out the benefit of the practical approach by fostering task based research, spreading your own research by making the tools available and open communication to foster research in general. Another open question was focusing on reservations towards publishing data and software. Most common class of answers noted legal or privacy issues (14). Others pointed out the additional effort needed (8), possible commercial interests (8), missing reward or support for doing so (3), and that it's not part of their job, ie. not supported by the group or organization (2).

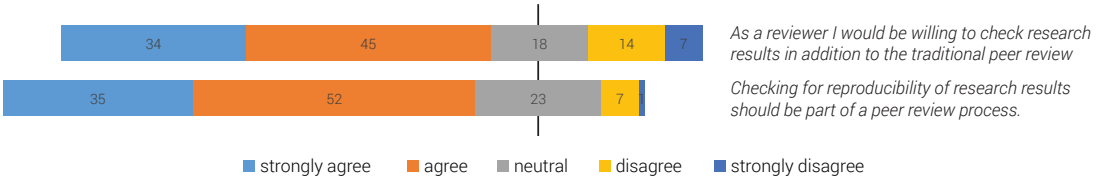

Figure 3. Responses from the fourth part of the survey cover the integration of reproducibility checks in the peer review process. Note that 7 participants did not provide an answer, so there are only 118 responses instead of 125.

Fig. 3 visualizes the researchers opinions on checks for reproducibility in the peer review process. 73.7% of the participants (87 out of 118) noted agreement on additional checks in the review process. 66.9% of the participants (79 out of 118) would be willing to do checks on reproducibility as peer reviewers.

While we assume a minor bias caused by the study's title in the way that participants are attracted by the title if they could identify themselves with the topic of reproducibility in a positive way, it is still valid to create hypothesis from the findings. So while the majority has agreed with reproducibility of results being necessary, major concerns are legal issues, additional effort and possible commercial interests. Regarding legal issues we have to distinguish between privacy concerns, i.e. sensitive data on clinical patients, photos or videos of people, etc. and violations of for instance copyright law, license issues or export restrictions. An interesting hypothesis is that researchers would be willing to share if legal issues and efforts are reduced to a minimum. This may be achieved by license constraints (only licenses others can build upon) or exceptions (leaving license issues aside for research by general agreement) for publishing research, as well as by providing a framework for publishing, sharing and building upon each others research, i.e. a GitHub for researchers.

3 REQUIREMENTS AND GENERAL GUIDELINES

First of all, authors publishing results need to make all information available that is necessary to reproduce the results.

Second, the effort necessary to reproduce the results needs to match the value of doing the work. Work reproducing or refuting previous results is in overall much less appreciated than original work, so the effort a researcher is willing to invest in order to reproduce previous results is much lower than effort willing to spend for new work. On the other hand, when planning to build own research on top of other results the investment can be higher. The most critical case is in reviewing, when reproducibility

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018

is intended to by checked as part of the reviewing process. Reviewers have a strict timeline to perform their review, so there is a need for a straightforward, mostly automated process to reproduce the results. Moreover, despite contributing to verifying the results of a paper, reviewers are nowhere mentioned in connection with the work. Adding that reviewers are doing voluntary work, they are probably the least motivated to reproduce results.

In order to address these issues, the following guidelines should be followed:

- Code, data and information on how to conduct an experiment should be gathered at a single place (a single container) which can be found in connection with the paper.
- The reproduction process should be highly automated (for example by an easy to handle build and execution script).
- To address security issues, the published code should be provided as source code and/or run within a virtual environment.
- Commercial libraries and other components that need the reviewer to pay for its use should be avoided unless absolutely necessary.
- Since research papers tend to create some interest even long after they have been published, it is necessary to ensure that software and environment for the reproduction process stays available, either by packing all necessary components into a container or by referring to well-archived open source tools.
- The time and necessary information to reproduce a result should be tested with an uninvolved person. Unless the size of the project requires it, the reproduction process should take at most one day.

4 EXISTING TOOLS

Most tools for sharing software artifacts are at the same time tools used in the development of software artifacts. This could be either tools for simple tasks like compiling software projects, but also more complex tools for tasks like automated dependency installation and software packaging. To prevent unnecessary complex configuration, it is wise to base the selection of the used tool on the complexity of the software artifact. Software artifacts which are complex to run require a more sophisticated tool with a high level of abstraction, while a simple artifact does not need a complex tool to run it.

In this section we present four commonly used open-source tools for sharing software artifacts. We begin with the simple tool *CMake* which is used as build management solution and continue with tools utilizing a higher level of abstraction. Finally, we summarize the features of the different tools and highlight benefits of each tool.

4.1 CMake

CMake is a cross-platform build-tool based on C++. It is designed to be used with native build environments such as *make*. Platform-independent build files are used to generate compiler-specific build instructions, which define the actual build process. Main features of CMake are tools for testing, building and packaging software, as well as the support of hierarchical structures, the automatic resolution of dependencies and parallel builds.

One drawback of CMake is that required libraries or other dependencies of software artifacts must be available and installed in the required version on the host system in order to successfully build the project. This could lead to extensive preparations for a build which is mandatory for executing software artifacts.

Tools with similar functionality are *configure scripts*, the *GNU Build System* and the *WAF* build automation system.

4.2 Gradle

Gradle is a general purpose build tool based on Java and Groovy. Gradle integrates the build tools Maven as well as Ant and can replace or extend both systems. Main features of Gradle are the support for Maven repositories for dependency resolution and installation and the out of the box support for common tasks, i.e. building, packaging and reporting. Gradle supports multiple programming languages, but has a strong

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018

focus on Java, especially as it is the recommended build system for Android development. An integrated wrapper system allows to use Gradle for building software artifacts without installing Gradle. Dependency installations and versions are maintained automatically. If a build requires a new version of a library, it is downloaded and installed automatically.

The automated dependency installation is a great benefit of Gradle, although there are still some challenges to overcome. One issue is that automated dependency installation only works, if the required libraries are offered in an online repository. If the required dependency is removed from the online repository, building any software depending on this library becomes impossible.

For other programming languages tools with similar functionality are available, i.e. the *Node Package Manager* (NPM) for JavaScript projects.

4.3 Docker

The open-source software *Docker* allows packaging software applications including code, runtime, system tools, and system libraries into a single container. The resulting container can be published, downloaded and executed on various systems without operating system restrictions in a virtualized environment. That way, an application embedded in a Docker container will execute in a predefined way, independently of the installed software environment on the host computer. The only requirement for the host system is the installed Docker engine.

A Docker container is a kind of lightweight virtual machine. A running Docker container is accessible via terminal or graphical user interface. Thereby, Docker has a broad range of application. A container could contain the runtime environment for a single application with graphical user interface, but it could also contain a ready to deploy server application for web services or even environments for heavy calculations or simulations.

The major difference between Docker and the previously presented tools is that Docker is usually not used for the development of an artifact. In most cases, a Docker container is created for sharing a predefined environment in a team. This means, that the container is created and the software artifact is deployed in the container afterwards.

4.4 VirtualBox

VirtualBox is an open-source software for the virtualization of an entire operating system. VirtualBox emulates a predefined hardware environment, where multiple operation systems, like Windows, Mac OS and most Unix Systems can be installed. The installed operating system is stored as persistent image, which allows the installation and configuration of software. Once the image is created, it can be shared and executed on multiple machines.

As mentioned before, VirtualBox emulates the entire hardware of a computer resulting in higher execution overhead as well as higher setup effort. Before the scientific software artifact can be installed in an VirtualBox container, an operating system and all dependencies have to be installed.

4.5 Comparison of Analyzed Tools

After the presentation of selected tools in the last section, we now want to compare their features relevant for sharing scientific software artifacts. Table 1 briefly summarizes our findings.

Security: Some software artifacts require administrator access rights on the local machine in order to be executed. These access rights allow malicious behavior, which could lead to unwanted consequences on the local machine or even on the local network.

VirtualBox and Docker execute software artifacts in sandboxed environments and therefore allow the secure execution of software artifacts. Tools like CMake and Gradle do not offer this security mechanism. When executing a shared software artifact from untrusted sources, a sandboxed environment is recommended.

Required Software: Gradle has the lowest requirements regarding required software for the execution of software artifacts. The only requirement for building an artifact with Gradle is the Java Virtual Machine, which is nowadays installed on most hosts. The Gradle Wrapper allows the dependency installation and the build of artifacts without installing Gradle itself. VirtualBox and Docker have no additional software requirements. CMake is comparatively cumbersome. It is required to install all dependencies of the desired software artifact, which can be time-consuming.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018

Table 1. Comparison of tools for sharing scientific software artifacts

Tool	CMake	Gradle	Docker	VirtualBox
Security	no security mechanisms	no security mechanisms	sandboxed environment	sandboxed environment
Required Software	CMake, required dependencies	Java, (Gradle)	Docker Platform or Docker Tools	VirtualBox
Supported platforms	Linux, MacOS, Windows	Java VM	Linux, MacOS, Windows	Linux, MacOS, Windows
Required knowledge for sharing	little	little	moderate	little
Required knowledge for installation and execution	moderate	moderate	little	little
Effort for sharing	little	little	moderate	high
Effort for installation and execution	moderate/high	little	little	little
Size of shared object	small	small	up to multiple GBs	up to multiple GBs
Limitations	Installation could be exhausting	Specific Gradle project structure required	GUI requires extra effort	Images always include the entire operating system

Supported platforms: All tools work on various platforms. CMake, Docker and VirtualBox are compatible to most Linux platforms, Windows and MacOS. Gradle is working everywhere, where the Java Virtual Machine is available. Besides this great platform support it has to be kept in mind that the software artifacts itself could require a certain operating system. Through the virtualization of Docker and VirtualBox this problem can be eliminated.

Required knowledge for sharing: If a build management tool is used in the development of a scientific software artifact, we assume that the researchers become familiar with the build management tool during the development phase. Therefore, no additional knowledge for the researcher who is sharing the artifact is required.

VirtualBox also does not require a lot of additional background information. Everybody who is able to install an operating system is able to share a software artifact embedded in a VirtualBox image.

Docker, especially the terminology of Docker seems to be confusing on the first sight. It is necessary to become familiar with the terminology.

Required knowledge for installation and execution: Researchers are often not familiar with the tools used for the creation of software artifacts. Reading the documentation of build management tools can be exhausting and not time-efficient for a short test of an artifact. CMake and Gradle require some knowledge in order to build a software artifact, especially if errors appear.

VirtualBox and Docker are easier to use. If a Docker image is hosted on DockerHub, a single command is sufficient for downloading and running the image. If this command is provided, no additional knowledge is required. Due to a graphical user interface, running a virtual box image is even easier.

Effort for sharing: CMake, Gradle and other build management systems are intended to define a standardized build process. If a build management system is used during the development of the scientific software artifact, no additional effort arises for sharing. The configuration file for the build management system can be shared along the source code of the software artifact.

Docker and VirtualBox are usually not directly involved in software development. In most cases, a Docker or VirtualBox image has to be created explicitly for sharing the software artifact. The structured process of building a Docker container allows easy reuse of already existing Docker containers for other software artifacts. In case of VirtualBox the whole VirtualBox image has to be shared on a file server. Docker containers can be shared on the free to use Docker Hub or on a file server. Alternatively, a

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018

Dockerfile, which contains the building instructions for a Docker container, can be created and shared as a text file.

Effort for installation and execution: The installation effort is low for all discussed tools. The highest effort regarding the execution of software artifacts arises when using CMake, where required dependencies have to be installed manually. For building and executing software artifacts with Gradle only a few commands are required. Docker and VirtualBox require the least effort; the shared image only needs to be executed.

Size of shared object: When using CMake or Gradle, only the source code of the software artifact and the configuration file of the build management tool have to be shared which usually leads to small shared objects.

The shared container of Docker or VirtualBox has to contain the source code and all other tools which are required for the execution, such as the operating system. This results in large shared objects, in some cases the size of a Docker container exceeds a Gigabyte or more.

Alternatively, Docker provides an option allowing for smaller shared objects – Dockerfiles. A Dockerfile contains only text instructions for building a Docker image. Therefore, the size of a Dockerfile is only a few kilobytes, but once it is executed Docker automatically pulls and builds the source code of the software artifact, which results in a large Docker image on the local machine.

Limitations: All analyzed tools show limitations. CMake is a lightweight tool for software development, but the effort for installing the dependencies of a software artifact could be extensive. Furthermore, it is only applicable for a handful of programming languages like C or C++.

When Gradle is chosen as build system early in the development phase, it is perfectly suited for Java projects. Unfortunately, Gradle requires a certain project structure which makes it hard to configure Gradle for existing projects which are not structured in the Gradle way.

Docker is perfectly suited for command-line or web applications, which is the case for a huge amount of scientific software artifacts. Additional configuration is required to support GUIs of desktop applications. Frevo (see section 5.2), used in one of our case studies, demonstrates GUI support for desktop applications with Docker.

VirtualBox is applicable for all types of software artifacts, but the overhead of creating and sharing a VirtualBox image could be huge. For sharing an artifact, independent of its size and complexity, a complete operating system has to be installed and shared.

5 USE CASES

After introducing the theoretical background in the last sections, we now present case studies where we analyzed the applicability of various tools for sharing software artifacts. Therefore, we selected three scientific artifacts from different computer science research areas, which allows a broad view on sharing scientific software artifacts.

5.1 Stochastic Adaptive Forwarding

Stochastic Adaptive Forwarding (SAF) (Posch et al. (2017)) is a forwarding strategy for the novel Internet architecture Named Data Networking (NDN) (Zhang et al. (2014)). Forwarding strategies in NDN are responsible for forwarding packets to neighboring nodes and therefore select the paths of traffic flows in the network.

The Network Forwarding Daemon (NFD) implements the networking functionalities of NDN. It is written in C++ and uses the WAF build automation system. For testing purposes, the network simulator ns-3/ndnSIM (Mastorakis et al. (2016)) is utilized, which also uses the WAF build system. For testing SAF in the simulation environment three steps are required: i) The installation of the NFD; ii) the installation of the network simulator ns-3/ndnSIM and finally iii) patching SAF into a compatible version of the NFD. The installation of SAF was tested and analyzed in the standard way by using WAF and by using Docker.

SAF with WAF: The standard way of developing the NFD is by using the WAF build system. The functionality of the WAF build system is similar to the functionality of CMake. This means that WAF automatically resolves dependencies, but the installation of dependencies must be done manually.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018

Although extensive installation instructions were published[1], it is quite difficult to install the simulator and it's dependencies. Furthermore, there are slight undocumented differences when installing the NDN framework on different Unix versions. Once the NDN framework is compiled in the correct version, it is easy to patch SAF. Nevertheless, it takes up to multiple hours to compile the NDN framework with SAF the first time.

SAF with Docker: NDN and SAF are licensed under GPL V3, which means that packaging the software is not a legal issue. Technically, Docker provides two possibilities for creating and sharing images. The first variant is to check out a preconfigured image like Ubuntu Linux from the Docker website and connect to it via terminal. All required changes can be done in the terminal and finally persisted with a commit. The changed image can be shared via the Docker website or as binary file. The second possibility to create the image is by using Dockerfiles. These files contain simple creation instructions for images and can be easily shared due to their small size. To build an image, the Dockerfile can be executed on any host with the Docker framework installed. Both variants were tested for SAF. The resulting images, containing all dependencies and the compiled software artifacts, have a size of about 4.6 GB; the size of the Dockerfile is only about 2 KB. When using the precompiled image[2], running the image only takes an instant. The execution of the Dockerfile takes, depending on the Internet connection and the computing power of the host system, between 15 and 60 minutes. Once the image is running, the results of the paper can be reproduced or new experiments with SAF can be done by conducting simulations using the provided network simulator.

5.2 FREVO

FREVO (Sobe et al. (2012)) is an open-source framework to help engineers and scientists in evolutionary design or optimization tasks to create a desired swarm behavior. The major feature of FREVO is the component-wise decomposition and separation of the key building blocks for an optimization task. This structure enables the components to be designed separately allowing the user to easily swap and evaluate different configurations and methods or to connect an external simulation tool. FREVO is typically used for evolving swarm behavior for a given simulation (Fehervari and Elmenreich (2010); Monacchi et al. (2014)). FREVO is a mid-sized project with 50k lines of mostly Java code, having a graphical interface as well as a mode for pure command line operation, e.g. on a simulation server. The component-based structure allows to easily extend and remove components (e.g., a simulation, a type of neural network, an optimization algorithm), which sometimes creates some effort in newly setting up FREVO.

FREVO was tested and analyzed with the following three tools:

FREVO with build script: Until recently, FREVO was provided as a download zip file[3] that included sources of the main program and additional components together with an ant build file. However, there had been problems in the past with different language versions of Java. A further problem can be dependencies on third party tools or libraries, which are not automatically maintained by this type of build script.

FREVO with Gradle: An analysis showed that the current structure of FREVO, especially because of its component-plug-in-architecture, conflicts with the expected and possible project structure for Gradle.

FREVO with Docker: Since FREVO and its components are open source under GPL V3, there was neither a legal nor a technical problem to put it into a virtual Docker container. We used an Ubuntu Linux system that was provided by Docker. Openjdk8 was installed as Java Runtime environment. After installing FREVO in the Docker system, it was pushed onto the free Dockerhub fileserver.[4] To reproduce a result made with Frevo it thus possible to (given that Docker is installed) download the respective Docker container and start it. In general, the result was easily usable, apart from some effort to get a graphical display working. The parallelization of simulation, which is a natural ability of FREVO, works fine as well inside a Docker container. The FREVO container has a compressed size of 223 MB, which is mostly due to the files of Ubuntu Linux.

[1] `https://github.com/danposch/SAF`, last visited 2018-04-25
[2] `https://hub.docker.com/r/phmoll/saf-prebuild/`, last visited 2018-04-25
[3] `http://frevo.sourceforge.net/`
[4] `https://hub.docker.com/r/frodewin/frevo/`

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018

5.3 LireSolr

LireSolr (Lux and Macstravic (2014)) is an extension for the popular Apache Solr[5] text retrieval server to add functionality for visual information retrieval. It adds support for indexing and searching images based on image features and is for instance in use by the World Intellectual Property Organisation, a UN agency, within the Global Brand DB[6] for retrieval of similar visual trademarks.

LireSolr brings the functionality of the LIRE library (Lux and Marques (2013)) to the popular search server. While LIRE is a library for visual information retrieval based on inverted indexes, it's research driven and to be integrated in local Java applications. Apache Solr on the other hand is more popular than the underlying inverted index system Lucene as it allows to modularize retrieval functionality by providing a specific retrieval server with cloud functionality and multiple APIs to access it for practical use.

LireSolr is intended for people who need out of the box visual retrieval methods without the need of integrating a library in their applications. It can be called from any mobile, server or desktop platform and runs on systems with a Java 8 runtime. This flexibility is valued among researchers as well as practitioners. LireSolr is hosted on Github[7]. Gradle and Docker build files are part of the repository.

LireSolr with Gradle: The standard way for building LireSolr is by using Gradle. Current IDEs can import Gradle build files; any task can be done from within the IDE. While Gradle makes sure that the right version for each library is downloaded and everything is ready to build, installing the new features to the Solr server has to be done manually. The supporting task in Gradle just exports the necessary JAR files. The user or developer has to install Solr, then create a Solr core, change two configuration files, copy the JARs and restart the server to complete the installation. While these steps are extensively described in the documentation, it's still major effort for new users, who do not have prior experience with retrieval in general or Apache Solr in particular.

LireSolr with Docker: As LireSolr is extending Solr by adding additional functionality, the intuitive way to create a Docker container is to extend the Solr Docker container. The *Dockerfile* defining the build of the Docker container is part of the LireSolr repository, where a specific Gradle task is building and preparing relevant files for the creation of the image. This includes the aforementioned JARs and config files as well as a pre prepared Solr core and a small web application as a client. The Docker container can easily be run and provides basic functionality for digital image search. Developers who just want to test LireSolr can get it running within seconds using Docker Hub: `https://hub.docker.com/r/dermotte/liresolr/`.

6 ONE TOOL TO REPRODUCE THEM ALL

In the previous sections, we presented tools for sharing software artifacts and a case study showing how the tools can be applied in order to share scientific software artifacts. In this section, we now reflect on the advantages and shortcomings of the tools with respect to the results from our survey presented in Section 2.

Each of the presented tools has its advantages and shortcomings. For instance, the additional effort for sharing an artifact when using a build management tool is very low because in most cases a build management tool is already used during the creation of the artifact. In contrast, it can be challenging and time-consuming for other researchers to get the build management tool up and running because required dependencies or the installation process may not be documented in detail. Software artifacts, which are provided as virtualized containers are easy to run and provide a high degree of security but are cumbersome in case a researcher wants to build upon or extend previously published software artifacts.

When weighing these advantages and shortcomings we quickly see that *the one tool to reproduce* all our scientific results does not exist. Nevertheless, based on our findings from the survey we now want to give recommendations for creating reproducible results and scientific software artifacts which can be easily used by other researchers.

The survey clearly showed that many researchers are interested in building their research on the work of others, which gets much easier, when published software artifacts can be reused. Furthermore, we

[5] `http://lucene.apache.org/solr/`, last visited 2018-04-13
[6] `http://www.wipo.int/branddb/en/`, last visited 2018-04-13
[7] `https://github.com/dermotte/liresolr`, last visited 2018-04-13

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018

saw that the average time researchers are willing to invest to get artifacts running is only about two workdays. Thus, we assume that it is very important for researchers to get the artifact running in a short time, otherwise, researchers lose interest in using the artifact and start developing their own solution. When taking the demand for security into account as well, we see that virtualized containers appear to be a good choice. The provided software artifact can be executed without the overhead of installing it, by simply running the container. Furthermore, it is possible to become familiar with the artifact in the virtualized environment and check if the artifact is suitable to base own work on it.

When researchers decide to build on the artifact, it may be cumbersome to continue using a virtualized container, because altering a software artifact is more convenient on a local system. This means that the researcher has to install the artifact locally, without virtualized container. According to our study, researchers currently prefer providing detailed instructions and build tools. Solely relying on this information, it could be challenging to install the artifact, as already discussed.

Dockerfiles are one solution to overcome this issue. As already explained, a Dockerfile is a kind of a construction guideline for Docker containers. It contains all command line directives, which are required to build a Docker container and can therefore be seen as exact procedure for the local installation of an artifact. Following the commands listed in the Dockerfile, local installation of a sofware artifact is a breeze. These commands ensure that all dependencies are installed correctly, otherwise it would not be possible to create a Docker container. This means by providing a Dockerfile both options become possible, the software artifact can be executed in a secure container, but it can also be easily installed by following the instructions from the Dockerfile.

Another finding of our survey is that the long-term availability of software artifacts is important for researchers and should be about 10 years. In addition, the ACM Artifact Reviewing and Badging guideline[8] emphasizes the importance of the possibility to reproduce results after a long time, by providing a separate badge for artifacts which are archived in archival repositories. When looking at our presented tools, we can see technical, as well as legal issues on the way to achieve long term availability. Although services, such as Code Ocean[9] or Dryad[10], for archiving software artifacts exist, the following things should be kept in mind. Tools, such as Gradle, rely on online repositories for downloading required dependencies. If one of the required dependencies becomes unavailable, the build is not possible any more. This means that all dependencies as well as all required tools have to be included when the artifact is archived. This leads to technical issues, because the amount of required tools to reproduce a result could be tremendously high. For instance, if a required operating system or compiler is not available any more, the results can not be reproduced, which means that even such tools need to be archived. Besides this technical issue, packaging these tools could lead to legal issues as well when tools with limiting licenses are used. Furthermore, also the platform for archiving software is operated by someone. If it's operator decides to discontinue service, all artifacts archived by this provider are lost.

7 CONCLUSION

In this paper we focused on the reproducibility of research results in computer science. We collected the opinions and requirements of 125 researchers by means of a survey. The analysis of the survey's results confirmed our initial assumption that the reproducibility of research results is an important concern in computer science research. In addition, researchers not only want to reproduce results, but also want to base their own work on the results of others. Main reasons for the importance of reproducibility are improved credibility and improved understanding of results. Based on the researchers opinions, we created guidelines which aid researchers in making their research reproducible. The applicability of various tools for publishing software artifacts was discussed while keeping our guidelines in mind. Scientific artifacts of different research areas in computer science were used to test the applicability of discussed tools for sharing reproducible research. Finally, we discussed our findings and concerns on the process of publishing reproducible research. According to our study, the long-term availability of reproducible results is of great importance to many researchers, but we identified open issues in achieving availability for longer periods. Even if reproducibility of research is currently not the common case, we recognized a strong positive shift towards reproducible research, backed not only by individual researchers, but also by renowned scientific journals and publishers.

[8] `https://www.acm.org/publications/policies/artifact-review-badging`, last visited 2018-04-06
[9] `https://codeocean.com/`, last visited 2018-04-06
[10] `https://datadryad.org/`, last visited 2018-04-06

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018

ACKNOWLEDGMENTS

We would like to thank all participants of the survey for their valuable input and all colleagues who helped us by sharing their practical experience and discussion.

This work was supported in part by the Austrian Science Fund (FWF) under the CHIST-ERA project CONCERT (project no. I1402).

REFERENCES

Allen, A. and Schmidt, J. (2015). Looking before Leaping: Creating a Software Registry. *Journal of Open Research Software, 3 (1):e15*, 3.

Arbona, A., Artigues, A., Bona-Casas, C., Massó, J., Miñano, B., Rigo, A., Trias, M., and Bona, C. (2013). Simflowny: A general-purpose platform for the management of physical models and simulation problems. *Computer Physics Communications*, 184(10):2321 – 2331.

Boettiger, C. (2015). An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79.

Bonaventure, O. (2017). The january 2017 issue. *SIGCOMM Comput. Commun. Rev.*, 47(1):1–3.

Brinckman, A., Chard, K., Gaffney, N., Hategan, M., Jones, M. B., Kowalik, K., Kulasekaran, S., Ludäscher, B., Mecum, B. D., Nabrzyski, J., Stodden, V., Taylor, I. J., Turk, M. J., and Turner, K. (2018). Computing environments for reproducibility: Capturing the "whole tale". *Future Generation Computer Systems*.

Casadevall, A. and Fang, F. C. (2010). Reproducible science? *Infection and Immunity*, 78(12):4972–4975.

Fehervari, I. and Elmenreich, W. (2010). Evolving neural network controllers for a team of self-organizing robots. *Journal of Robotics*.

Hanson, B., Sugden, A., and Alberts, B. (2011). Making data maximally available. *Science*, 331(6018):649.

Janin, Y., Vincent, C., and Duraffort, R. (2014). Care, the comprehensive archiver for reproducible execution. In *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*, TRUST '14, pages 1:1–1:7, New York, NY, USA. ACM.

Lux, M. and Macstravic, G. (2014). *The LIRE Request Handler: A Solr Plug-In for Large Scale Content Based Image Retrieval*, pages 374–377. Springer International Publishing, Cham.

Lux, M. and Marques, O. (2013). Visual information retrieval using java and lire. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 5(1):1–112.

Mastorakis, S., Afanasyev, A., Moiseenko, I., and Zhang, L. (2016). ndnSIM 2: An updated NDN simulator for NS-3. Technical Report NDN-0028, Revision 2, NDN.

McMurdie, P. J. and Holmes, S. (2013). phyloseq: An r package for reproducible interactive analysis and graphics of microbiome census data. *PLOS ONE*, 8(4):1–11.

Monacchi, A., Zhevzhyk, S., and Elmenreich, W. (2014). HEMS: A home energy market simulator. *Computer Science – Research and Development*.

Morin, A., Urban, J., Adams, P. D., Foster, I., Sali, A., Baker, D., and Sliz, P. (2012). Shining light into black boxes. *Science*, 336(6078):159–160.

Peng, R. D. (2011). Reproducible research in computational science. *Science*, 334(6060):1226–1227.

Poline, J.-B., Breeze, J. L., Ghosh, S., Gorgolewski, K., Halchenko, Y. O., Hanke, M., Haselgrove, C., Helmer, K. G., Keator, D. B., Marcus, D. S., Poldrack, R. A., Schwartz, Y., Ashburner, J., and Kennedy, D. N. (2012). Data sharing in neuroimaging research. *Front Neuroinform*, 6:9.

Posch, D., Rainer, B., and Hellwagner, H. (2017). Saf: Stochastic adaptive forwarding in named data networking. *IEEE/ACM Transactions on Networking*, 25(2):14.

Ram, K. (2013). Git can facilitate greater reproducibility and increased transparency in science. *Source Code for Biology and Medicine*, 8(1):7.

Shrader-Frechette, K. and Oreskes, N. (2011). Symmetrical transparency in science. *Science*, 332(6030):663–664.

Sobe, A., Fehérvári, I., and Elmenreich, W. (2012). FREVO: A tool for evolving and evaluating self-organizing systems. In *Proceedings of the 1st International Workshop on Evaluation for Self-Adaptive and Self-Organizing Systems*, Lyon, France.

Steiniger, S. and Hunter, A. J. (2013). The 2012 free and open source {GIS} software map – a guide to

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018

facilitate research, development, and adoption. *Computers, Environment and Urban Systems*, 39(0):136 – 150.

Walters, W. P. (2013). Modeling, informatics, and the quest for reproducibility. *Journal of Chemical Information and Modeling*, 53(7):1529–1530.

Witten, D. M. and Tibshirani, R. (2013). Scientific research in the age of omics: the good, the bad, and the sloppy. *J Am Med Inform Assoc*, 20(1):125–127. amiajnl-2012-000972[PII].

Zhang, L., Afanasyev, A., Burke, J., Jacobson, V., claffy, k., Crowley, P., Papadopoulos, C., Wang, L., and Zhang, B. (2014). Named data networking. *SIGCOMM Comput. Commun. Rev.*, 44(3):66–73.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27082v1 | CC BY 4.0 Open Access | rec: 1 Aug 2018, publ: 1 Aug 2018