The present work was submitted to
the Research Group
Software Construction

of the Faculty of Mathematics,
Computer Science, and
Natural Sciences

Bachelor Thesis

Reproducibility of Computational Environments for Software Development

presented by

Marvin Strangfeld

Aachen, August 01, 2022

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Christian Plewnia, M.Sc.

Alex Sabau, M.Sc.

Abstract

A computational environment is a representation of everything that can influence computations done inside them. This includes computer hardware, operating systems and software libraries. Computational Environments are being used in every step of the software development process. On the workstations of developers for performing development tasks, the CI pipeline that builds and tests the software and on production servers or customer's machines that run the released product.

These environments can be heterogeneous and change over time. This can cause problems, for example when the software application depends on a certain version of a software library to be available. Therefore it is possible that the application runs properly in one computational environment but does not work in another. To solve this problem, computational environments can be made reproducible to a certain degree.

In my thesis I provide a theoretical definition of computational environments that is more precise than preceding ones and can be used to create descriptions of real-world computational environments. I also provide a definition of reproducibility of computational environments that helps to argue about the degrees to which an environment can be made reproducible.

Following these theoretical foundations I present a catalogue with various approaches to create reproducible computational environments and compare their use cases. These approaches can be used like building blocks to meet different requirements for reproducible computational environments.

In the end I present an explorative case study of how a reproducible computational environment can be implemented using a real-world software project at my industry partner *Open-Xchange*. In this case study I use a subset of the presented approaches that has been selected by leveraging the theoretical definitions to analyze the requirements for this environment.

Contents

List of Figures

List of Listings

1 Introduction

Contents

A computational environment is a representation of everything that can influence computations done inside them. This includes computer hardware, operating systems and software libraries. When developing a software application, there are often multiple computational environments that are used to perform various tasks. Each developer for example has a workstation to write and test code on. A continuous integration (CI) pipeline might run on some kind of server. The deployed application might run on a customer's machine or on some production servers. In any case, there is often more than one environment where the developed software application is required to run on. These environments can be heterogeneous and also change over time.

As an example, consider two developers, Alice and Bob, each working on the same software application with their own workstation. Alice has the software library *libExample* in version `1.0` installed on her machine. Bob has the same library in version `2.0` installed on his machine. Now it might happen that the application under development works on Alice's workstation but due to some changes of the *libExample* library in the version `2.0`, it does not work on Bob's workstation. This problem is also known as the "It works on my machine" problem [6]. To mitigate this problem, the two developers would need to reproduce the same library version on each of their systems to guarantee that this particular problem does not occur again.

This example describes just one manifestation of the many problems of diverging computational environments. The software development process could benefit in many ways from reproducible environments.

To illustrate this, let us assume there was a way to perfectly and automatically reproduce the computational environment needed for every task in the development lifecycle. If every developer had exactly the same computational environment available, every bug occurring in one environment could be easily reproduced in another environment. Thus it can ease the synchronization across the development team. It also enables developers to quickly reproduce any failure that occurred in a production environment, making the debugging process much more simple. CI pipelines and local workstations could share the same environment as the deployment servers, reducing the need to go back and fourth between these environments to debug a new feature. Having an environment

that is truly reproducible can also provide a way to revisit older projects because the development environment could just be recreated. Additionally, the onboarding process for new developers could be simplified as setting up the development environment on a new machine is an automated and reproducible process. Of course, in practice there are no perfectly reproducible computational environments, but it is likely that even some form of reproducibility can improve the development process in some way.

The same problems occurring in software development due to diverging environments can also be found in scientific computing where researchers try to reproduce computational experiments in order to verify published results.

1.1 Reproducibility in Scientific Computing

"Reproducibility is widely considered to be an essential requirement of the scientific process." [19] It enables independent researchers to recreate an experiment and verify or disprove the experimental findings. This characteristic of making an experiment reproducible is therefore desirable for all scientific fields, including scientific computing where experiments are carried out by using computations that analyze digital data. At first glance, reproducing a computational experiment seems to be a straightforward task, as only the program instructions and the input data for the computation need to be exchanged. Every researcher should then be able to "just run" these instructions on their own computer and verify the results. But in practice, there are more barriers that prevent this easy sharing of computational experimental setups.

To illustrate this, let us have a look at very simplistic example. In this example we use a program that we have written ourselves and that performs its analysis steps deterministically, meaning the same input to the program should always produce the same output. One might therefore assume that sharing the source code of the program or a precompiled binary file along with the original input dataset should allow anyone to repeat the analysis steps themselves. However, even if these artifacts are provided, it is not guaranteed that the execution of the program behaves the same when executed in different environments. To give a very simplistic example of such a computational experiment, here is the source code of a program written in the *Python* programming language:

```
print "Some reproducible result: 42"
```

Listing 1: A simple computational experiment written in Python

While this program will run fine with the Python `2` interpreter, it will not run with a version `3` interpreter because there have been breaking changes between those two versions. In this case, changing `print`, which was a statement in version `2`, to a function in version `3`, breaks the code for the newer Python versions [42]. To be able to run with Python `3`, the code would have to be changed like this:

```
print("Some reproducible result: 42")
```

Listing 2: Alternative version of the experiement for Python 3

This admittedly trivial example gets to the heart of the problem, namely: in order to reproduce the execution of a program, we must have at least some information about the requirements of the computational environment [35]. In this case it would have been sufficient to write down the intended Python version alongside with the source code. Then the correct Python version could have been installed manually before trying to run the program. However, for more complex experimental setups, it might be beneficial to use other techniques to capture and share the computational environment.

1.2 Software Development as a Scientific Experiment

The expectations in developing software are no different in their basic assumptions from those in conducting a scientific computing experiment. The difference is that instead of running a single deterministic program to generate a result, there are multiple programs for each task of the development lifecycle.

Let us take the process of source code compilation as an example. In this process the input data is the source code of the application under development. The analysis steps are performed by the compiler, which generates a result in form of one or more binary files. If other developers can recreate the same result using the same input with the same compiler, the experiment of compiling the source code can be called reproducible. It should be noted that we are free to define what exactly we consider to be "the same result". In some cases it may be sufficient to obtain a working binary file, in other cases we may want to achieve bit-by-bit binary reproducibility.

In addition to the development lifecycle tasks, the execution of the application under development can also be interpreted as an experiment. The input data is the input to the application. This might be user input from the command line, files from the file system or network packets from a network card. The application then performs its analysis steps and generates output in form of, for example, writing to the standard output (stdout), writing to the file system, or sending network packets. Also note that the application does not have to terminate and the stream of output data can be seen as the result of the analysis. When attempting to reproduce the behavior of the application, it is again possible to freely define what is considered to be the "same result" as well as the "same input data".

To achieve reproducibility of application behavior in different environments, the same principles apply as in scientific computing experiments. We need to share the dataset, which in the case of software development tasks could be the source code and in operational tasks such as deployments it could be compiled binaries. Additionally, we need to share the relevant information about the computational environment needed to execute the applications. Therefore, the computational environment itself becomes the subject

of reproducibility.

Let us illustrate this again with the previous example of a Python program that is compatible with version `2` of the interpreter but incompatible with version `3`. To be able to execute the program, the intended Python version has to be reproduced by the end-user and every developer. To be able to perform other tasks of the development lifecycle such as testing the code, the required tools to test the code also have to be reproduced. Therefore each task in the development lifecycle requires a reproducible environment [24].

1.3 Research Questions

While reproducibility in the scientific context has already been researched, the topic of reproducibility specifically in the context of software development has not yet been covered. There are already existing definitions of "reproducibility" and "reproducible research" as well as practical proposals on how to implement reproducibility for scientific computing. Although the basic assumptions are the same, both fields have different requirements when it comes to implementing reproducibility.

For example, the requirement for accurately reproducing the same results are necessary in scientific computing, where as they might be optional in software development. Developers might choose speed and convenience of their workflow over accuracy when it comes to reproducing some results. This also has to do with the different workflows. Scientific papers are usually published once and being reproduced by few individuals, where as software development is an ongoing and highly distributed process where speed is essential.

These somewhat contradicting goals for reproducibility open the following research questions that I will address in this thesis:

1. What is a definition of reproducibility of computational environments beyond the context of scientific experiments that also covers environments for software development?

   I provide a fundamental theoretical definition of computational environments and reproducibility of computational environments. The definitions are compatible to the definition of reproducibility in scientific computing but provide a new level of formalism that help us to describe the overall mechanism. The definitions will further be used to describe the requirements of reproducible computational environments.

2. What are the available approaches to achieve reproducibility and how do they compare?

   I create a catalogue of approaches to reproduce different categories of computational environmental properties. The approaches are building blocks that can be used to systematically create a reproducible computational environment based on the individual needs of an application.

3. How can a reproducible computational environment be implemented in a software development project?

   I provide an illustration of an explorative transformation to create a reproducible computational environment for a real-world software project at my industry partner *Open-Xchange*.

2 Defining Computational Environments and Reproducibility of Computational Environments

Contents

There is some ongoing debate on how to define "reproducibility" and "reproducible research" with sometimes contradicting definitions. Efforts have been made to collect and compare these definitions by Heroux et al. [18] and Barba [2]. What all these definitions have in common is, that the subject of reproducibility is a scientific experiment with a certain result. While this could be translated to the context of software development as explained before, it is rather impractical. Additionally, the subject of interest shifts in software development from reproducing the same static end-result to a more dynamic reproducibility of the development environment. This is due to the fact that the software under development continuously evolves by adding new features or fixing bugs.

Therefore, I will provide a new definition for reproducibility of computational environments that can be used to describe reproducible software development environments. In addition to this, the definition should also be able to describe experimental setups for scientific computing including the reproduction of scientific results. Due to the contradicting definitions of reproducibility in scientific computing I will adopt a condensed definition by Barba [2] which is based on the most widely used definition of "reproducible research" in the field of computer science by Claerbout and Karrenbach [7]:

"Reproducible research: Authors provide all the necessary data and the computer codes to run the analysis again, re-creating the results." [2]

The Turing Way, a project focused to promote reproducible research, furthermore defines a reproducible scientific result: "A result is reproducible when the same analysis steps performed on the same dataset consistently produces the same answer." [8] It differs from the term *replication* which means that an equivalent result has been obtained by collecting new data and using other experimental setups. Replication also plays an important role in science but in this thesis I will solely focus on reproducibility.

A precise definition of a concept can be used as a basis for communication. Therefore, I will define computational environments and reproducibility of them as precisely as possible. The goal is not only to give an intuition of the concept but also to create formal

representations that can be used to describe a specific computational environment and whether or not it is reproducible. While there are existing definitions, they mostly lack the precision and formalism needed.

2.1 Defining Computational Environments

The Alan Turing Institute defines *computational environment* in its project *The Turing Way* as the "Features of a computer which can impact the behaviour of work done on it, such as its operating system, what software it has installed, and what versions of software packages are installed." [8] Other definitions of computational environments do not only include software artifacts and system configurations but also the computer hardware, network and other peripherals [34, 39, 19, 24]. While these definitions give an intuition of what a computational environment might include, they all do not claim to be exhaustive. The definitions are built in this way because it is impossible to generalize which properties are part of a computational environment and which are not.

To explain this, let us have a look at what might influence a computation. System architecture, operating system (OS) and version of software libraries are obvious properties that can influence the behavior of a program executed in the environment. But also properties like current system time, specific hardware, and network status could impact the behavior of computations. Every program might have different inputs which again can depend on arbitrary other variables.

One could even argue that the only way to describe a computational environment accurately enough to capture anything that might affect the computations performed within, it would be to take a snapshot of the entire universe. To illustrate this, let us assume a program that uses a specific hardware to count photons of the physical surroundings. Not only would the hardware affect the behavior of the program, but all the photons in the surroundings have a direct impact on the computation. Another example would be an hurricane rolling over your server location causing the destruction of all your servers. Surely, it will have an impact on the computations in this environment. For the sake of creating a usable definition and not to drift too much into philosophical details about whether the universe behaves deterministically or not, I will focus on a more practical example in the following.

Let us have a look at the Python program in listing 3 that uses multiple inputs for its computation. The program gets the current year of the system time in line 6 using the `datetime` module which comes with the Python standard library. In line 8 it sends an `HTTP GET` request to an external web server using the `requests` library. The current year is embedded in the URL of the request. After that, it outputs the HTTP status code of the web server's response.

The result of the execution of this program depends on several characteristics of the computational environment. To be able to even execute the program, Python `3` and the `requests` library have to be present on the system. The URL of the HTTP request changes based on the current year of the system time, so the environment's system time directly influences the computation. The external web server to which the request is send

```python
#! /usr/bin/env python3
import requests
from datetime import datetime

# Get the current year
year = datetime.now().year
# Execute an HTTP GET request
response = requests.get(f"https://swc.rwth-aachen.de/{year}")

print(response.status_code)
```

Listing 3: A simple Python program to illustrate various sources of inputs

can reply with any HTTP status code based on numerous other inputs that therefore also influence our computation. Furthermore, the entire Internet infrastructure used to communicate with the server (DNS, router, …) decide what the output of the program will look like. If the network fails and the web server cannot be reached, the result will be different than if the connection is working.

Based on this example we could start trying to describe the computational environment of this program. But based on the existing definitions of computational environments it is not clear what description would be sufficient. Do we have to include every little detail of the network infrastructure between our machine and the web server? Or is it sufficient to just state: "A working connection to the web server"? Do we also need to specify every input that might influence the web server's computation? And what about if these inputs are programs themselves with other inputs?

Due to this lack of precision the need for the following more formal and therefore more precise definition emerges.

2.1.1 Definition (Computational Environment)

A set of ordered tuples where each tuple (x, y) describes a specific property of the environment using its unique id $x \in \mathbb{N}$ and a specification with the matching domain $y \in D_x$.

$$\{(x, y) \mid x \in \mathbb{N},\ y \in D_x\}$$

I will name the class of all computational environments $\mathbb{CE}$ where each description of a computational environment is $c \in \mathbb{CE}$.

Using this definition we can describe any computational environment on any abstraction level independent from the programs executed inside the environment. A computational environment according to this definition is just a collection of tuples of arbitrary properties. A property is identified by its unique id x and a qualitative or quantitative specification y. The id is just a natural number but for readability any string can be used and encoded into a unique natural number. The domain of the specification can be

freely chosen. It could be booleans, integers, strings or any other data type, including nested computational environments.

To illustrate this concept let us describe a computational environment where the CPU architecture is *x86*, the OS is *Linux* and the year of the system time is 2022. We name the ids of the properties: "cpu.arch", "system.os" and "system.time.year". For the domains we choose sensible options like strings for the first two properties and natural numbers for the year. We could also define the domain based on an enumeration like $D_{\text{cpu.arch}} = \{\text{x86}, \text{AArch64}, \text{MIPS}\}$. A valid description of our environment could then look like this:

$$\{(\text{“cpu.arch”}, \text{x86}), (\text{“system.os”}, \text{“Linux”}), (\text{“system.time.year”}, 2022)\}$$

Alternatively we could even encode this into a more readable *JSON* format [5] as shown in listing 4.

```
{
  "cpu.arch": "x86",
  "system": {
    "os": "Linux",
    "time.year": 2022
  }
}
```

Listing 4: A *JSON* representation of a computational environment

Using this definition enables us to compare two environments and their relation to another. First, two computational environments $A, B \in \mathbb{CE}$ are *equal* when all of their properties are equal:

$$A = B \iff (A \setminus B) = (B \setminus A) = \emptyset$$

We can also form equivalence relations on $\mathbb{CE}$. Two environments $A, B \in \mathbb{CE}$ are *C-equivalent* ($\sim_C$) to another, when $C \in \mathbb{CE}$ is a subset of both of the environments.

$$A \sim_C B \iff (C \subseteq A) \land (C \subseteq B)$$

The relation $\sim_C$ creates an equivalence class $[C] \in \mathbb{CE}/\sim_C$ containing all computational environments that are equivalent to another in the context of C.

For example, let $C = \{(\text{“cpu.arch”}, \text{“x86”})\}$, then all computational environments with a x86 CPU architecture would be C-equivalent to another, regardless of any other properties they might have.

To avoid always having to define a new equivalence relation when just comparing two computational environments with each other, I suggest that the term: "A is equivalent to B" means the same as $A \sim_B B$. Or put differently, the computational environment described with the properties A could also be described with the properties B. Keep in

mind, that these sets of properties are just descriptions of a real-world computational environment that can be described with an infinite amount of properties in theory.

To illustrate this equivalence in a more practical example, let us take the computational environment from listing 4 and add an additional property "python.versions" which represents all available python versions on that system. A representation of this environment can be seen in listing 5.

```
{
  "cpu.arch": "x86",
  "system": {
    "os": "Linux",
    "time.year": 2022
  },
  "python.versions": [ "3.8.8", "2.7.18" ]
}
```

Listing 5: The same computational environment with the additional property "python.versions"

This second description of a computational environment is different from the first one because not every environment that could be described with the first one could also be described with the second one. They are therefore not equal. On the other hand, every computational environment that could be described with the second description could also be described with the first one, as we could choose to ignore the installed Python versions. Therefore, the description of the second environment is equivalent to the description of first environment but not equal.

These characteristics of the definition enable us to specify an environment with an infinite amount of equivalent environments possible. The description abstracts from any particular technology or physical location where the environment lives in. The description can match to a server, a developer's workstation, a virtual machine, a container, or any other possible instantiation. Furthermore, we can describe any real physical environment with as much detail as we need.

2.2 Defining Reproducibility of Computational Environments

A reproducible computational environment is an environment that can be recreated in different contexts. Existing definitions of reproducibility either focus on scientific computational experiments [7, 4] or define reproducibility in terms of computational results [27]. But I want to examine what it means to reproduce a computational environment, independent from experiments or program executions. This gives us the freedom to use the definition for more dynamic scenarios such as the software development lifecycle. Therefore I propose the following definition for reproducibility of computational environments:

2.2.1 Definition (Reproducibility of a Computational Environment)

A computational environment $c \in \mathbb{CE}$ is reproducible, if there is a deterministic function $f_{c,r}$ that can derive c-equivalent environments $c_b \in [c]$ from an r-equivalent input environment $b \in [r]$ and a recipe $x \in \mathbb{N}$.

$$f_{c,r} \colon \mathbb{CE} \times \mathbb{N} \to \mathbb{CE}, \quad (b, x) \mapsto c_b$$

If we can find such a deterministic function $f_{c,r}$, we can reproduce c-equivalent environments from any environment that is equivalent to a minimal requirement r using the recipe x. The parameters c and r define the main characteristics of the function. The properties that are defined in c determine what properties we are able to reproduce with this function. The properties that are defined in the minimal requirement r determine how portable to other environments this function is.

To give an intuitive example, let us look at the two extremes of portability. The function $f_{c,c}(b, \emptyset) = c_b$ expects as a minimal requirement an input environment b that is equivalent to c and it reproduces an output environment c_b that is also equivalent to c. It is the least portable function, as to reproduce the environment, it already requires the environment to exist. Therefore it can only be applied to environments that are already equivalent to the environment we want to reproduce. Note, that we do not need the recipe here, as no additional information is necessary on how to derive the output environment.

In contrast to this would be a function $f_{c,\emptyset}(b, x) = c_b$. In this case the environment can be created in the context of any other environment b, as there are no minimal requirements. It is therefore as portable as it gets because every computational environment b is equivalent to a computational environment $\emptyset$. But in this case all information on how the environment c_b is derived from b has to be encoded in the recipe x.

In a real-world scenario we would probably try to find a function $f_{c,r}$ to reproduce our environment c as portable as possible, with as little information as necessary encoded in the recipe x. In detail this means that we try to minimize the amount of properties specified in the requirements while also minimizing the amount of information encoded in the recipe.

Let us implement this definition in a short example. Assume we need to reproduce an environment c as seen in listing 6.

```
{
  "python.versions": [ "3.8" ],
  "python.3.libraries": [
      { "name": "requests", "version": "2.27" }
  ]
}
```

Listing 6: A computational environment with Python 3 and the `requests` library

```
{
  "python.versions": [ "3.8" ],
}
```

Listing 7: A computational environment with Python 3

For now it is sufficient for us to reproduce this environment from any other environment that has already `Python 3.8.8` installed. This is our minimal requirement r to the input environment and could be described like in listing 7. Now we need to find a deterministic function $f_{c,r}$ and a recipe x, such that we can derive c-equivalent environments from any input environment b that is equivalent to r.

A possible implementation of such a function could be provided by the package manager *pip* for the Python ecosystem. Pip can manage python software libraries in an environment. We can specify the required packages in a `requirements.txt` file like in listing 8.

```
requests==2.27.1
```

Listing 8: A `requirements.txt` file for pip

Running the command `pip install -r requirements.txt` will install the `requests` library in the version `2.27.1` into the current environment. Thus `pip install` can be interpreted as our function $f_{c,r}$ with the recipe $x = encode(\texttt{requirements.txt})$.

However, the command to install a library may fail due to various reasons. To get the Python packages, pip uses a repository that is accessible via the internet. If there is a problem with the internet connection to the server hosting the repository, the installation of the package fails. Furthermore, it might be that the packages are simply not available anymore, or at least not in the specified version. Also other things like running out of disk space can make the installation fail. This is a direct violation of calling the function $f_{c,r}$ deterministic. To mitigate this, we have to set up further requirements for the input environment to account for the availability of repositories and disk space for example.

Another approach would be to define the function $f_{c,r}$ as quasi-deterministic [21], allowing the function to either fail or to create a deterministic result. This means of course that we cannot guarantee that the environment c can always be reproduced, as the function can just output an error. It is a trade-off between the guarantee of always being able to reproduce an environment and more portability, as we just "rely" on properties that we do not list explicitly in the minimal requirements for the input environment.

2.3 Dependencies of Reproducibility

The previous sections discussed how computational environments can be described by a set of properties and how such properties can be reproduced from other environments.

But what is the actual meaning of a property like `system.os = "Linux"` ? The semantic that we established is that the computational environment has a *Linux* OS installed. But this leaves several details unspecified. For example, there are various distributions of Linux available such as Debian, Arch or openSUSE. Additionally, all those Linux distributions have multiple versions available.

So, when specifying a property like `system.os`, we are actually specifying an abstraction or an interface that can be implemented in various ways. This also means that we can interpret two computational environments with two different Linux distributions installed as equal. While this may seem counterintuitive at first glance, it can help us avoid over-specifying environments to the point where they are no longer portable.

The question is: How much do we need to specify to achieve our initial goal of making development tasks reproducible? To understand this we need to look at what is necessary to reproduce a task in the first place. Every task of the development process uses programs that need to execute reproducibly. We are now assuming that we do not use any programs that intentionally behave irreproducibly by using random numbers for example. To create the same output of the program over and over, we need to give the program the same input. But a program has more inputs than just the user input it processes. It may also have dependencies on other programs or other properties of the computational environment it is executed in.

For example, the Python interpreter depends on the *glibc* library when used on a Linux OS. But when it is used on Windows it uses the *MSVC* runtime library instead. While both of these libraries are an implementation of the *C standard library*, they may behave differently. The expectation of the Python interpreter is, of course, that it abstracts these two different implementations and provides a single abstraction that behaves the same on all operating systems. But this unified behavior across different dependencies may also fail [10].

So, to ensure a program behaves exactly the same on all environments we would need to reproduce not only the program itself but all of it's dependencies. Those dependencies itself can also depend on other dependencies that can modify the behavior of our program. In figure 2.1 we can see a directed acyclic graph (DAG) that illustrates all the software library dependencies of the Python interpreter on a Linux system. All those libraries can be used during the runtime of a Python program that is executed with the Python interpreter. Therefore all of these libraries can also change the behavior of our program.

But software libraries and user inputs are not the only inputs to our program. For example, the thread scheduler of our operating system is responsible to manage the execution of threads. In some cases it might be possible that our program behaves differently when the threads are executed in a different order. Another example are semantic dependencies in our program such as file system paths. When referencing a path like `/usr/bin` we implicitly require a UNIX file system hierarchy which is not present on a Windows OS.

To identify all of the inputs that may change the behavior of our program is very complex. Even more complex would it be to reproduce all of these inputs in a compu-

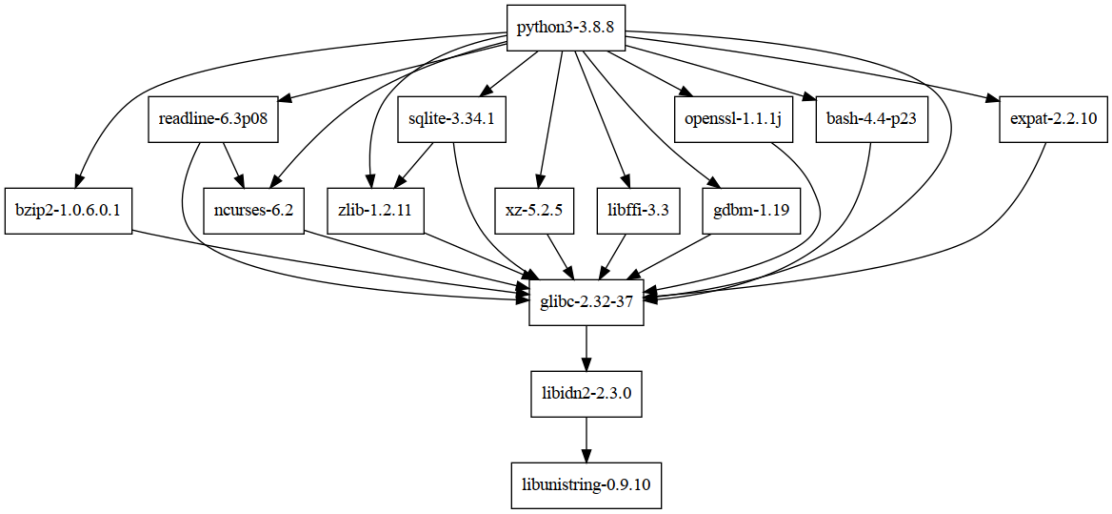

Figure 2.1: Dependencies of the Python interpreter in a Linux environment

tational environment, up to the point of loosing portability to other environments. So, for practical reasons we need to choose which sources of non-determinism we are willing to accept and which we want to cover. This is a decision that needs to be evaluated by each individual software project depending on the requirements as well as the trust one is willing to give an abstraction in behaving the same across different implementations.

3 Approaches for implementing Reproducibility

Contents

There are multiple approaches available that can be used to create reproducible computational environments on different abstraction levels. Each of these approaches introduces a different set of features and trade-offs regarding the properties that can be reproduced. The usage of one approach does not exclude the usage of another one, so rather than seeing them as competitors they can bee seen as possible building-blocks to combine. In this chapter I will describe the conceptual mechanics of each technique as well as the gains and costs regarding the goal of creating reproducible environments for software development purposes.

3.1 Hardware Requirements

Some properties of computational environments cannot be reproduced by a program that implements a deterministic function. This includes hardware related properties, as the hardware needs to be physically available and cannot be created by just running a program. However, for some software development projects we might want to enforce that every reproduced computational environment satisfies a certain set of hardware properties. This can be implemented by specifying a certain set of hardware properties that are required for the computational input environment. In this case the function $f_{c,r}$ is more of a check that fails if the input environment is not equivalent to the requirements. The requirement can for example be as simple as the specification of the CPU architecture or a minimum amount of RAM capacity. But the requirement can also be much more restrictive up to the point of specifying one physical machine that has to be used.

This approach provides multiple gains depending on the field of software development. In the field of embedded software it is necessary to define certain properties of the hard-

ware, as the applications are often tightly coupled with the electronics. The applications therefore directly depend on some properties of the physical hardware environment. For development and deployment purposes it therefore makes sense to restrict the hardware as much as possible to avoid unexpected behavior.

However, this restriction comes at the cost of portability because every developer needs access to a certain piece of hardware. Although it is possible to abstract from the physical level by emulating hardware, the emulator can behave differently compared to the physical hardware. [20] It therefore is a question of whether one is willing to trust the emulator to gain more flexibility and portability across different environments. For development purposes this can be a good trade-off where all development environments can be reproducible by using emulators, avoiding the need of giving access to the physical hardware to each developer. It is furthermore possible to create dedicated testing environments using exactly the same hardware that is being used in deployments. This way development can happen on the emulated hardware but the software is additionally tested on the real hardware before it gets released. [15]

To illustrate this, let us assume we are trying to develop an application for an *AVR* [1] based microcontroller like the *Arduino* [2]. For developers to be able to write and test their code we would need to distribute a microcontroller to each developer. Alternatively, we could also setup a single microcontroller on which each developer has to test their code. Either solution is not very flexible because either we have to distribute a lot of hardware or developers need to coordinate access to the single hardware. As an alternative we could use a system emulator like QEMU [36], so every developer can test code on their machine. QEMU can run on almost any system architecture and all major operating systems as host and is able to create a virtual *AVR* hardware that can execute code written for it. The portability and flexibility is therefore much better than using real hardware. On the other hand, the cost is that we have to assume that the emulation might be inaccurate or behaves differently to the real hardware. Therefore we still test the developed code on a real microcontroller before it gets released. The access to this test device still needs to be coordinated but because it is just needed in the end of a development cycle, the amount of access requests is decreased.

3.2 Disk Images

A disk image is a file containing all the contents of a data storage medium. In this case I want to look at disk images that contain all necessary data like the operating system (OS) and kernel to boot and use a computer. They can additionally contain all other kinds of data like software applications and configuration files. Disk images can therefore be used to capture and reproduce a lot of software related properties of a computational environment.

There are two main ways to use a disk image. First, it can be transferred to a physical device like a hard disk to enable a computer to boot from it. While this provides the

[1] AVR is is a family of microcontrollers developed by Atmel.
[2] Arduino is an open source single-board microcontroller project.

best computational performance of the two alternatives, it is not practical for developers to constantly change the boot device and reboot the computer when the environment changes. Additionally it can be necessary to restrict the physical hardware because the OS on the image might be configured to expect a certain hardware like a *x86* CPU for example.

The second alternative is to create a virtual machine (VM). "A Virtual Machine (VM) is a compute resource that uses software instead of a physical computer to run programs and deploy apps." [41] A VM runs on a physical computer (host) but uses its own operating system independent from the host. VMs can abstract from the underlying hardware of the host by creating virtual devices such as displays, hard disks and other peripherals. Additionally, they can be combined with hardware emulators like described in section 3.1. Using disk images with a VM instead of using them directly with hardware therefore provides a better portability.

Apart from the dependency on hardware, disk images offer great portability, as they place hardly any requirements on the input environment b. This is because all of the software related information about the target environment c is encoded in the disk image itself. The disk image can be interpreted as our recipe x. A deterministic function $f_{c,r}$ can be provided by a *hypervisor* that creates and hosts virtual machines using disk images.

However, using disk images for software development comes at a cost. As described before in section 2.2, having almost no requirements to the input environment does mean that x has to encode all necessary information on how to build the target environment. This encoded information needs space, in our case disk space at rest and bandwidth when transferred. Because the disk image needs to include the complete OS with all its configurations, it can get very large. This can be problematic when trying to distribute the file among developers as it can be slow. Especially for fast changing environments where dependencies get frequently updated the size of disk images can be a major drawback. Using disk images also does not allow multiple environments to share common software components because each image ships its own copy which further increases bandwidth to download and disk space to store the images. [9] Additionally, it is not possible to easily merge two disk images. So for example, if Alice changes a file in her copy of the disk image and Bob changes another file in his copy, it is not a trivial task to combine these two changes into a new disk image containing both changes.

To tackle some of these problems there are tools that can create disk images and VMs based on programmatic descriptions of the desired environment c. So instead of seeing the image itself as the recipe x for our function $f_{c,r}$ we use another function f_{build} with input x_{build} such that $f_{c,r}(b, f_{\text{build}}(b, x_{\text{build}})) = c$. This way $f_{c,r}$ is still the hypervisor creating the VM but the disk image is created by f_{build}.

One example of a tool providing a function f_{build} is *Packer* by HashiCorp [33]. It can create disk images based on a configuration where one can define which OS alongside which software applications and configurations should be installed inside the disk image. In theory this can reduce the bandwidth needed to share disk images as it is only necessary to share the programmatic description alongside with a base image to build

upon. Every developer can then build a disk image equivalent to the environment c based on this base image and the description. The base image can be shared among multiple environments which further decreases bandwidth. When a developer wants to make a change to the environment, they put it in the description and rebuild the image. The description can then be shared in a version control system and easily merged with changes from other developers.

However, this approach does come with a caveat. Because the final disk image is dynamically build on top of the base image, it either cannot reproduce as many properties as a static disk image do or it needs to add more requirements to the input environment. For example, we need the software library *libExample* in version `1.0` installed in our final environment c. When working only with static disk images the library is included in the image and the environment can be easily reproduced. But when we need to create this image dynamically, the library somehow needs to be installed from somewhere. Therefore it either needs to come encoded in the programmatic description x_{build} of the image or we need to add a requirement to the input environment that we can download the library in this exact version from somewhere.

So, the trade-off of using these intermediate tools like *Packer* compared to just using disk images is gained ease of synchronization of the environments for the cost of less reproducible properties and/or more input environment restrictions meaning less portability.

3.3 Container Images

Another approach to reproduce computational environments are container images. Container images are essentially like disk images as it is possible to put arbitrary files and executable programs inside a container image. But unlike disk images they can only include one file system and not an entire disk with all its partitions. A running computational environment that is created from a container image is called a container. To create a container, a container image has to be loaded by a container runtime on a host system. This is similar to VMs but instead of using a separated kernel to interact with the hardware, containers use the kernel of the host's OS. "To put it simply, containers virtualize at the operating system level, whereas hypervisor-based solutions virtualize at the hardware level." [25]

The advantages of using containers over VMs are that they are more lightweight, as they do not require to run their own operating system with kernel, system services, etc. Containers have isolated user spaces from the host's OS and therefore cannot see other processes or files outside their own container. So for a process inside a container it looks like there are only the container's processes on the system. This enables us to create multiple containers on a host that do not interfere with each other, similar to VMs. Additionally, container images can be much smaller than bootable disk images for VMs. Container images can even just include one single binary that gets executed.

The disadvantages of using container images to reproduce computational environments are mainly the same as using disk images. Although the image sizes can be significantly

smaller compared to images for VMs, the synchronization issues of the images are the same.

To tackle this issue there is a similar approach to the one building disk images. Instead of modifying and distributing container images directly, we can build container images based on descriptions. A popular tool that implements this approach is called *Docker* [12]. Docker can be used to build, run and manage containers and container images. But it comes with the same limitations regarding reproducibility as *Packer* for disk images. Namely, we either need to reduce the amount of properties in c that can be reproduced or increase the requirements to the input environment. This is also because the recipes to build container images can execute arbitrary non-deterministic commands depending on various inputs such as package repositories to install software into the image.

Compared to VMs, containers have another drawback. Because they depend on the host's kernel they can only run on hosts with similar kernels. [19] This means that a container image containing binaries that were compiled for a Linux kernel cannot run on macOS, which uses the *XNU* kernel for example. To use the same containers across different operating systems it is therefore necessary to run the containers in a VM when the host's OS does not provide the necessary kernel.

3.4 Package Management Systems

A package management system (alternative: *package manager*) is a tool that manages the state of software installed on an OS. Package managers can be used to reproduce the available applications and software libraries in a computational environment. Software artifacts that form a coherent component are bundled in so called *software packages*. A software package can require the installation of other software packages in order to guarantee that the included applications and libraries can be used correctly. These dependencies between software packages can then be resolved by the package manager.

Let us have a look at a simple example. We want to have the software package `Python 3` installed in our reproducible environment c. The software package itself includes multiple files, such as the binary executable for the interpreter but also the Python standard library as source files for example. In section 2.3 we have already seen the software dependencies of Python `3`. The direct dependencies are attached to the `Python 3` package as some kind of meta information. The packages that provide the direct dependencies can also have their own dependencies. When we install the `Python 3` package with our package manager of choice, it will go through the dependencies recursively, trying to resolve and install all software artifacts that are necessary to use Python `3`.

While this eases the management of software packages, there are multiple challenges to ensure reproducibility of the software installed by a package manager on a system. First, the software packages and their dependencies have to be available to the package manager. This is in practice often achieved by using repositories of software packages that are hosted online. Using these online repositories makes installing new packages very accessible as we do not need to manually collect all information about a package to install it. However, this also means that the reproducibility of the environment directly

depends on their availability and whether they still host the software package we wish to install. Depending on the granularity to which we define our reproducible environment c this can be a problem. Package repositories often update the packages they host so you can only get the latest version of the package. But if we define our environment with an older version of a package, it might not be available anymore.

3.5 Reproducible Build Systems

Build systems (aka build-tools) are general purpose task execution managers. [26] A common use case for build systems is the building of software from its source. This process can include multiple tasks like the compilation of source code, the linking to libraries and the creation of software packages for package managers. All these tasks and their dependencies on each other can be defined and get automatically executed in the right order by the build system.

A reproducible build system is designed to do the same thing with the additional feature of guaranteeing the exact same result for each build. This means for example, that a build process for the same source code executed in two different environments will result in two bit-by-bit identical binaries. This quality is also known under the term *reproducible builds* [38]. The reproducible build system achieves this by eliminating all sources of non-determinism that may cause differences in the result of the build. This includes environment variables, system time and timestamps of the source files. Also things like ensuring that all dependencies that get downloaded during the build process are always the same and do not change.

An example of a reproducible build system is the open source build-tool *Bazel* [3] which is developed by Google.

Having reproducible builds can be very useful as it is ensured that every developer can always recreate exactly the same binary to check for bugs for example. It can also act as a building block for a chain of trust because third parties can verify that a distributed binary is in fact the result of a build process with given input [22]. Additionally, reproducible builds can make caching of build artifacts more reliable because it is guaranteed that a build process with a certain input will always generate the same output. When a build artifact for the input is already present in the cache it can therefore be used without running the build process again. This can improve the speed of incremental builds because only the source code modules that have been changed need to be rebuild.

On the other hand, using a reproducible build system can result in a lot of operational overhead. This is because every input to the build process as well as every dependency need to be exactly defined in order to guarantee that the result will be the same. This overhead even led big projects like *Kubernetes* to delete their reproducible build system as it was too hard to maintain. [14]

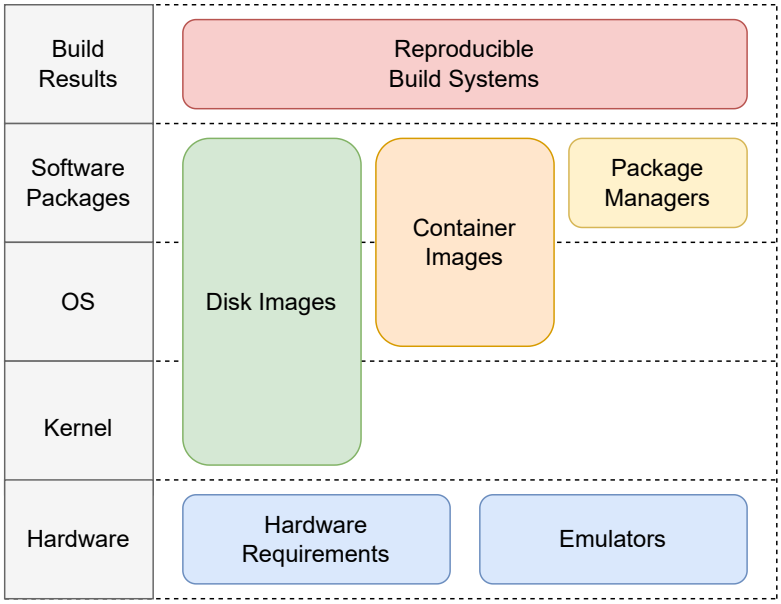

Figure 3.1: Overview of the available approaches to create a reproducible computational environment

3.6 Building Blocks

All the presented approaches to reproducible environments provide their own set of properties they can reproduce in exchange for portability. In figure 4.1 we can see an overview of all the approaches with the category of properties they are able to reproduce. The approaches can be freely combined with another like building blocks to create the best solution for a given software development project.

For example, if the project includes the development of OS kernel drivers for a custom hardware, it might be beneficial to use disk images on VMs together with hardware emulators. The disk images can reproduce the same kernel version for every developer and the hardware emulators eliminate the need of equipping every developer with a real piece of the custom hardware.

In case of developing a C++ web server you might decide that a container image is sufficient to provide the same software dependencies to every environment.

When developing an application in a high-level language like JavaScript and the end result that is shipped to the customer needs to be bit-by-bit reproducible for compliance reasons, you might combine a package manager with a reproducible build system.

4 Case Study: Reproducible Environment for Cross-Project Integration Testing

Contents

Up to this point I have defined what a reproducible computational environment is and described several approaches to create reproducibility of such environments. But for the task of writing software we also need to know how well these concepts apply to the real world. Therefore I transformed a real-world development project at my industry partner *Open-Xchange* to use a reproducible computational environment.

In this chapter I will describe the different phases of how I implemented this environment. First I will go over the project's requirements and the goals I try to achieve with the reproducible environment. Then I will explain which technologies I chose to use and how they fit the requirements. After this I will go into more details about the actual implementation and what steps were necessary to transform the project. In the end I will evaluate whether I could achieve the goals for the project and what problems still need to be solved.

4.1 Project Requirements

Open-Xchange is a company that creates open source software products. One of the main products is the *OX App Suite* [32] which is a web-based communication, collaboration and office productivity software suite with groupware features.

The backend of this software is written in *Java* using the *OSGi* framework [40]. *OSGi* enables *Java* applications to be modularized into components that can be developed and deployed independently from each other. The components for the backend are developed by different teams each with their own *Git* repository for version control. For the development process we use a simple *GitHub flow* [16] branching strategy in the repositories, where a feature gets implemented in a short lived *feature* branch and

merged back into the *main* branch when ready.

While this separation allows the development teams to develop their component independent from each other, it can also lead to problems when APIs change. For example, there is a *core* component which encapsulates all the base features of the backend and another component handling interactions with documents. The *documents* component uses the API of the *core* component, so when the team of the *core* component changes their API, they might not notice that they are also breaking functionality of the *documents* component. While the *core* team does not notice any problem when they build their component, the build process of the *documents* team might fail with the new *core* API as dependency. To avoid this problem there is a central integration project that can be used to collect all components from their Git repositories and build them together. This way one can check if a change in one component causes an error in the build process of another component. This testing is also automated in a CI pipeline such that the teams are only allowed to merge their changes back into the *main* branch if this pipeline succeeds.

The integration project is therefore a crucial part of the daily development workflow. A lot of developers depend on it to work properly and fast. If there was an error in the CI pipeline of the integration project, it would stop many developers from working. Additionally, the project needs to run in different environments like the mentioned CI pipeline but also on a lot of developer's workstations. These environments need to behave the same, so when an error occurs in the CI pipeline we are able to reproduce it on another machine to fix the problem.

This already gives us some objectives for the reproducible environment:

1. Portability - across different workstations and the CI pipeline
2. Developer Experience - as a lot of developers interact with the environment, it should be as easily usable as possible and integrate with the individual development workflows of each developer
3. Reliability - because it gets used for every merge on every component and errors might delay the release of features and bug-fixes
4. Performance - because otherwise it can be a bottleneck for the whole development process

Let us examine a little bit further the process that gets executed when the CI pipeline runs and what properties of the environment are necessary to reproduce. First, the components are cloned from their Git repository. Therefore we need a Git client to be installed. All components and their dependencies will occupy around 3Gb of disk space, so it is a requirement to have enough free space. Next, the build process for the components gets executed. As a build system we use *Gradle* [17] and *Apache Ant* [1] both of which also require a Java runtime environment (JRE) to run. Because we want to regularly update the versions of these build tools to benefit from continuous speed improvements, we chose to run them with the newest Java runtime in version 17. The

build system is configured to run different tasks such as compiling the Java source code. For the compilation of the source code we also need another Java runtime including the development source files and the Java compiler. This collection of runtime, source files and development tools is called the Java development kit (JDK). The backend components do not use the newest version of Java, instead they need a JDK in version 8. After the compilation of the source code, the compiled artifacts need to be bundled using a *tar* archive. Additionally, user manual files are automatically created using a tool named *pandoc* [23]. In the end a container image should be build to deploy the backend with all components. For this purpose we chose to use the tool *Docker* [12].

```
{
  "freeDiskSpace": ">3Gb",
  "accessible": [
    "Git repositories of all components"
  ],
  "system.os": "Linux OR macOS",
  "software": [
    { "name": "coreutils", "version": "9.1" },
    { "name": "git", "version": "2.36.1" },
    { "name": "jre", "version": "17.0.3" },
    { "name": "jdk", "version": "8.0.322" },
    { "name": "gradle", "version": "7.5.0" },
    { "name": "ant", "version": "1.10.11" },
    { "name": "pandoc", "version": "2.17.1.1" },
    { "name": "gnutar", "version": "1.34" },
    { "name": "gzip", "version": "1.12" },
    { "name": "docker", "version": "20.10.17" }
  ]
}
```

Listing 9: The computational environment for the integration project

In listing 9 you see a representation of the final environment c we need to reproduce. It includes all properties that are necessary to start the build process. Of course, having such an environment does not imply that the build process will produce a working result as this also depends on the source code. Furthermore, these properties are not complete in the sense that every error that occurs on one machine can be reproduced on another machine. It is more a baseline which can be expanded in the future when we notice irreproducible behavior across implementations of this environment.

4.2 Choice of Technologies

To implement the environment there are multiple approaches available as we have seen in chapter 3. Each of these approaches also offers multiple implementations in the form of software applications. While it is possible to create the reproducible computational environment in many different ways, each one comes with its own trade-offs.

First of all we have two dependencies that are directly related to the hardware and which cannot be reproduced any other than to require them from the input environment. This is of course for one the available disk space. The other one is the accessibility of the Git repositories of the components. The environment needs a network connection to the servers hosting the Git repositories. For the sake of having a lightweight solution we will just require that all these servers hosting the Git repositories do work as expected and are reachable via the available network.

Other requirements to the used hardware only arise from the availability of the needed software for different CPU architectures. However, we only use `x86-64` and `AArch64` CPUs in all our servers and workstations. All the required software applications are available or can at least be cross-compiled for both architectures. Of course, running the applications on different architectures could potentially introduce irreproducible behavior but for the first iteration of this implementation I will accept this calculated risk and mitigate it if I feel it becomes necessary in the future.

When it comes to the OS we do require a UNIX like OS such as Linux or macOS. All the servers for the CI pipelines run on Linux and most of the developers at Open-Xchange run Linux or macOS on their workstations. However, there are some developers that use a Windows OS. For these developers it is necessary to create a UNIX like environment on their workstation. As we have seen, VMs can be a useful approach to do this. Because this step is only necessary for machines that run on Windows, we can rely on the *Windows Subsystem for Linux* (WSL) [43]. It is a feature of recent Windows versions that lets you run a very lightweight VM with a Linux kernel that is highly integrated with the host's OS. This way the computational performance overhead is minimal and developers can easily integrate the environment into their development workflows.

Up until now the selected approaches can be used to reproduce everything of our environment except for the installed software. To achieve having the same software in exactly the same version installed in all environments there are multiple approaches we could use.

We could use disk images to create VMs with all the software applications pre-installed. But this has multiple drawbacks such as reduced computational performance for the build process. Additionally, the environment is frequently updated to benefit from improved speed and stability of the newer application versions. Nevertheless, we still want to keep the ability to quickly rollback to older versions of software if an update introduced a bug. These updates and eventual rollbacks would mean that developers need to download new images frequently which is time and disk space consuming.

Alternatively, we could use container images to give every developer access to the needed applications. This would have the advantage that the image sizes would be

smaller and the computational overhead would be more lightweight.

However, there might be a different approach that is even better than using container images. Let us step a bit back and think about how we would create these container images. The most forward approach would be to take a base image of a Linux distribution like Debian and use the package manager of that distribution to install the necessary software on top of the base image. The newly created image would then be used by the developers and the CI servers. But when someone wants to change the environment, they would need to create a new container image, upload it to a central server where every developer and CI server can access it and then inform everyone that this uploaded image is the new environment for the project. This can get confusing, especially when multiple developers would like to change properties of the environment in separate branches that later get merged back into the *main* branch. Each of the container images from the feature branches would then be incorrect to further base the development on as they only contain one of the two changes to the environment. A new container image which contains both changes would need to be created and published.

The other option for using containers would be to let every developer and CI system build the container image themselves from a recipe like a *Dockerfile*. The downside to this approach is that for using a package manager to install the necessary software packages into the image, we need to use a package repository. There are two options for this. Either we use a public repository that is managed by the maintainers of the Linux distribution for example, or we need to run our own package repository. The first option does imply a dependency on an external service and we cannot verify that the needed software packages are still available in the specified version when we need them. So for example, when Alice creates a container image from the *Dockerfile* and Bob creates a container image an hour later, the two images might have completely different versions of packages installed. Running our own package repository could solve this issue as we would be able to keep all versions of packages that we have ever used. But it imposes a lot of additional operational overhead to run the package repository hosting all necessary packages in all the needed versions with all the transitive dependencies for every version.

Fortunately, there is a solution to this problem. I chose to implement the reproducibility of the software packages with *Nix*. *Nix* is an open source, purely functional package manager and reproducible build system [28]. It emerged from the work of Dolstra, de Jonge, and Visser [13] and was first released in 2003.

But what exactly does a purely functional package manager do? “Functional package management is a discipline that transcribes the functional programming paradigm to software deployment: build and installation processes are viewed as pure functions in the mathematical sense – whose result depends exclusively on the inputs –, and their result is a value – that is, an immutable directory.” [9] So instead of installing software packages like other package managers do, by placing and replacing the contents of the package directly into the file system of the OS, in Nix each package has its own dedicated path and it cannot overwrite any files outside this path. The directory holding all the software packages is called the nix store and by default it is located at `/nix/store`. For example, the Java development kit (JDK) could be installed into this directory:

`/nix/store/y9h42qm7mn9x29awfzfzga16diilb7a1-openjdk-17.0.3+7`

Inside this directory are all the contents of the package like the Java compiler in this example. The name of the directory consists of two elements. A hash value that is derived from all inputs to this package and an human readable name of the package. The inputs to a package consist of the source code of the application as well as all dependencies for the package. Therefore even changing the version of a dependency, changes the hash of the final package. This way we can pin the entire dependency tree of the software package with just one hash value, as every change of a transitive dependency would change the hash value of a direct dependency, which would result in a different hash value of our software package.

This approach gives us the ability to install multiple versions of a package on the same system because even if the version number of the software did not change but the source files from which the package is build did, it would result in a different hash value thus a different directory where it gets installed to. This is different from more traditional package managers where all executable binaries for example are installed into one directory like `/usr/bin`. When two packages provide binaries with the same name they cannot be installed together without a conflict in this case.

To make the contents of the nix store accessible to the user, nix uses so called *profiles*. A profile is basically just a collection of symbolic links that construct an environment with a subset of the packages in the nix store. Therefore it is possible for two users of a system to use completely different versions of a software package. These profiles can also be created on a per-project basis. This feature is called nix-shell. It enables the developer to quickly instantiate and load a predefined profile with all the necessary software packages for a development project. Each project can also use different versions of packages and once the packages are present in the nix store, the switching to a different profile is just a matter of creating some symbolic links and loading some environment variables. Additionally, unlike other approaches to create software environments, it does not use any virtualization and can therefore integrate with other software running on the system like the IDE for example.

Software packages do need to be defined using Nix's own expression language which is also called *Nix*. In listing 10 we can see such a definition of an example package. It is essentially a single function with input arguments and one output. The input arguments provide the nix *stdenv*, which is used to pull in basic tools such as the GNU core utilities. The other arguments provide the *fetchurl* function to download artifacts from the internet as well as the software packages *ghc*, which is the Glasgow Haskell compiler and a library *zlib*. The output of the function is a *derivation* which is basically a standardized build script that describes how exactly the package has to be build. In the definition of the derivation we can see that we are defining the name and the version of our package. The source code is downloaded from an external server. Note, that the hash value of the file has to be specified in order to guarantee that the downloaded file is always the same. There is a build and an install phase to build the software from the downloaded source code and install it into the `$out` directory which is a variable for the final directory in the nix store.

```
{ stdenv, fetchurl, ghc, zlib }:
stdenv.mkDerivation rec {
  pname = "myPackage";
  version = "1.0.0";

  src = fetchurl {
    url = "http://git.rwth-aachen.de/${pname}-${version}.tar.gz";
    sha256 = "010b25bb234f747973351e6..."
  };

  buildInputs = [ ghc zlib ];

  buildPhase = ''
    ./Setup build
  '';

  installPhase = ''
    mkdir -p $out/bin
    cp build_output/${pname} $out/bin
  '';

  meta = {
    description = "My super awesome package";
  };
}
```

Listing 10: A simple Nix expression for an example package

When the derivation of the nix expression is build, the result is called a *realisation*. These realisations can then be cached and even shared, so other users do not have to execute the build steps again.

The Nix package manager also has an official repository called *Nixpkgs* [30] which has such derivations for a lot of software packages. Different from other package repositories, the Nixpkgs are just a Git repository hosting only the Nix expressions to build the packages. This way the repository can be pinned to a specific Git commit, making it always available. But note, that most derivations depend on external inputs such as the Git repositories of the source code. So in order to build these derivations it is necessary that the source is still available under the specified location.

Because we are only using well established open source software applications in our environment, we will assume that these source code repositories will be available and continue to host the specified commits in the nix expression files.

Using the Nix package manager rather than using virtualization techniques such as container images has the big advantage that there is no barrier between applications executed inside and outside a nix-shell. Developers can use the IDEs they have installed on their OS along with other tools and configurations in their system. Process isolation might be a useful feature when using software on a server to avoid certain clashes in the namespace but when it comes to tasks such as debugging on a workstation it is more a barrier than being a help.

Nevertheless, our CI pipeline is build on containers to enable restricted and controlled environments for the builds and tests of software. Additionally it provides a good abstraction to freely distribute the CI pipeline workload across multiple heterogeneous servers using Kubernetes. To be able to use the same Nix environment as developers do with the nix-shell in the CI, we have to instantiate the nix-shell inside a container. For this case I am using the official NixOS container image, as it has already the Nix package manager pre-installed.

To sum up the choice of technologies that are being used, figure 4.1 shows an overview of the used building blocks and how they build upon each other. Linux and macOS workstations can use the nix-shell directly. Developers using Windows have to use the WSL to create a Linux VM. The CI runs on Linux servers, that run a Kubernetes that orchestrates NixOS containers running the nix-shell.

4.3 Implementing the Reproducible Environment

Now that we have decided which technologies we should use, we need to actually implement them for our project's environment.

4.3.1 Creating the Nix Environment

The first task is to install Nix on our workstation. Nix can be easily installed on any Linux or macOS system including the WSL for Windows by executing a simple bash

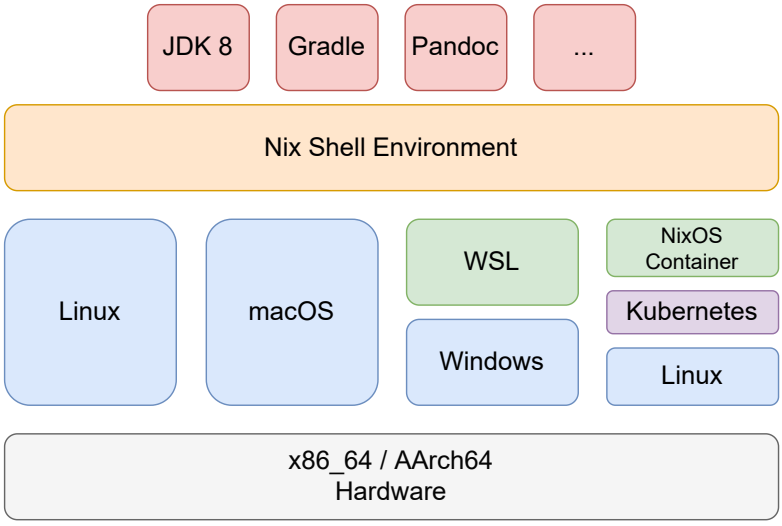

Figure 4.1: An abstract overview of the implementation of the reproducible environment

script as installer. After the installation we can use the `nix` command line utilities to interact with the system.

The next task is to define an environment for our project. It should be noted that there are multiple ways to do this. In the past there have been some competing solutions on how to define a Nix environment for a project. In our case we will use the new API called *flakes* [29] that has been recently released and provides a more unified and officially supported approach. To create an environment for our project using the *flakes* API we need to create a `flake.nix` file in the root directory of our project.

In listing 11 you can see a simplified version of this `flake.nix` file. A `flake.nix` file consists of an input and an output section. In the input section every input that gets used to create the environment must be declared. In our case we are using the official Nix package repository *Nixpkgs* as well as an utility library to help us write the expression more concise. In line 3 you can see that the *Nixpkgs* get declared as an input from their official GitHub repository. We are referencing the Git branch `nixpkgs-unstable` as it includes the newest versions of the needed software packages. In line 4 you can see the same thing for the utility library *flake-utils* where by default the main branch is used.

Declaring the inputs like this does not pin the applications to a fixed version as the branches are constantly updated. To use one specific commit of the Git repository we could either use the commit's hash value in the input section of the `flake.nix` file or we use another file called `flake.lock`. The advantage of using this lock file is that the update process can be automated by simply running the command `nix flake update`. This automatically creates or updates the lock file which then contains the hash value of the commit where the branch points at. When the environment is build, only the commit referenced in this lock file will be used to get the derivations from the Nixpkgs. This means that as long as this lock file stays the same, the versions of the applications

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, flake-utils, ... }:
    flake-utils.lib.eachDefaultSystem
      (system:
        let
          pkgs = nixpkgs.legacyPackages.${system};
        in
        {
          devShell = pkgs.mkShell {
            packages = with pkgs; [
              coreutils
              git
              jdk8
              gradle
              ant
              pandoc
              gnutar
              gzip
              docker
            ];
          };
        }
      );
}
```

Listing 11: Nix environment configuration

in the environment stay the same.

In the output section we define a `devShell` which defines a standard nix-shell of this project. It would also be possible to define multiple named nix-shells each with a different set of applications for different use cases. From line 15 to line 25 you can see that we are specifying all the needed applications for creating the environment. Note, that we do not need to specify the JDK 17 version as the Gradle package already includes it to by default.

The rest of the file is just the necessary boilerplate to use the same flake file across different environments. For example, in line 8 the same outputs get created for "each default system". By looking into the source code of the flake-utils library we can see that the list of default systems include `AArch64` and `x86_64` architectures with Linux and macOS / Darwin as OS.

4.3.2 Direnv for Automatic Loading of the Nix Shell

To be able to interact with the specified environment developers can checkout the Git repository of the integration project and execute the command `nix develop` inside the project's directory. This will build and install all the necessary derivations into the nix store of the workstation and create a shell environment with the specified packages available using symbolic links. This process can even be automated using a tool called *direnv* [11]. It is an extension for the shell that automatically looks for `.envrc` files in directories and evaluates them when entering a directory containing such a file. This `.envrc` file can contain a simple instruction to automatically load the nix shell when entering the directory and when exiting the directory it will exit the environment. This makes interacting with the project very simple as the environment is automatically available anytime the developer interacts with the project.

4.3.3 Using the Nix Environment in the CI Pipeline

While using the nix-shell feature of Nix is great for local development, it is not as simple for using the environment in our CI pipelines. Our CI system is based on the *Jenkins* automation server which executes all the workload in containers running on a Kubernetes cluster. Therefore we also need to instantiate our environment inside a container. To do this there are several options.

First, we could simply build a container image from the definition of our environment. Nix includes a feature called *dockertools* that does exactly this. The problem with this approach is that the image needs to be available in the beginning of CI pipeline's execution. But when the environment changes the image needs to be build at first.

A very simple solution that I chose to implement is to use the official *NixOS* container image that can be downloaded from public container repositories like DockerHub. NixOS is a Linux distribution that builds upon the concepts of the Nix package manager to configure the entire system. It ships with the Nix package manager by default which enables us to instantiate the nix-shell inside the container.

Another more complex solution would be to run a meta pipeline on every execution of the pipeline which checks first if the environment has changed and then rebuilds the container image and triggers the main pipeline with the correct image. This approach would have the advantage that the complete environment could be cached by the container runtimes of the CI servers which would probably result in faster startup times of the environment.

4.3.4 Binary Caching

Until now there is one major drawback to this setup. Every time someone instantiates the environment on a new workstation and every time the CI pipeline runs, all the derivations of our environment need to be build from their source code. This of course is not very efficient and therefore the Nix package manager has a mechanism called *binary substitution*. It enables us to build derivations once and upload them to a binary cache server. Then every time we would need to build the derivation again Nix first checks the cache server for a substitution and downloads it when available. The Nix project hosts a public binary cache providing prebuilt binaries for the derivations in the Nixpkgs repository. However, we would like to have a caching server that is inside our network and can therefore serve the substitutions at much higher speeds.

To implement a caching server there are again multiple solutions available. The most simplest of all would be to dedicate one host to run a program called *nix-serve* that makes all the contents of the host's nix store available via HTTP. To populate the cache all the built derivations have to be copied to the host using SSH for example. This approach has the drawback that it has a single point of failure and bottleneck relying on one single host.

Another approach that I chose to use is an object storage through a S3 API. The advantage of this is that we can use our already existing highly available storage cluster and do not need to run any other dedicated machine or server program. To automatically populate the cache I configured the CI pipeline to upload all derivations of the environment after it has been instantiated. This way new derivations are only build once in the CI environment and can then be used through the binary cache by following pipeline runs and also other developers.

5 Evaluation

In this chapter, I will discuss the results of my case study and evaluate whether I succeeded in answering my research questions. I will go through the research questions in descending order starting with the evaluation of the case study.

5.1 Evaluation of the Case Study

My third research question was: "How can a reproducible computational environment be implemented in a software development project?" To give an illustration of how this might be possible, I implemented a reproducible environment for a real-world software development project at Open-Xchange. For the implementation of the reproducible environment I had multiple goals in the beginning. I will go through each of the goals and evaluate whether I could achieve it with the current implementation of the reproducible environment and what improvements are possible for the future.

5.1.1 Portability

I tested the Nix environment across all the supported operating systems: Linux, macOS and Windows.

In the case of Linux and macOS I installed Nix directly to the system with the official installer. The Linux distribution I used for the testing were Manjaro and NixOS with a Linux kernel in version `5.15`. The macOS version was *macOS Monterey* `12.4`.

In the case of the Windows OS I used Windows `10`, enabled the WSL in version `2` and used an Ubuntu `22.04` Linux distribution as the OS of the WSL. Then I installed Nix into the WSL like on the other Linux distributions.

The CI system uses the Nix shell in NixOS `22.05` containers that run on a Kubernetes.

Due to lack of access to `AArch64` hardware, I could only test the implementation on `x86_64` hardware.

I assume that an environment successfully instantiated by Nix always contains software packages with exactly one version of the source code. This assumption is due to the fact that Nix compares the hashes of every input to the build process, including the source code. The only divergence in the binary files of the packages is due to different CPU architectures and either a Linux or Darwin kernel.

Executing the build process of the project works across all tested environments. I could not observe any irreproducible behavior although the binary applications provided by Nix are not identical in every environment, as explained before. If irreproducible behavior occurs in the future, I would have to reconsider if the environment needs further

restrictions to a certain CPU architecture or OS. But these restrictions would mean of course, that more developers would need to run a VM and CPU architecture emulator. This makes it harder for these developers to integrate the environment into their typical development workflow or use other tools on their system in combination with the reproducible environment. I will therefore first wait to evaluate the amount of issues due to divergence of the environments in contrast to the cost of convenience for developers.

5.1.2 Developer Experience

The environment integrates well with other tools like IDEs on the host system. I tested using the environment in Visual Studio Code and emacs. Both IDEs have plugins available that make it possible to load `.envrc` files from *direnv*. This way they have access to exactly the same applications as when you manually load the environment in the terminal. Nevertheless, other tools from the system are still available and can be used in the development process. This is due to the fact that no process isolation is necessary for creating the environment and programs outside the Nix environment can easily access the programs Nix is providing. Furthermore, the CI environment now has the identical packages installed as the local development environments, making it easier to fix bugs in the CI pipeline.

A possible problem for future maintenance could be the Nix expression language as it is rather unintuitive for developers not familiar with the concept of functional programming. There is already an approach to abstract this language for the nix-shell called *devshell* [31]. Instead of using the `flake.nix` file to change the environment it uses a very simple `TOML` file. Using this could therefore make the maintenance of the environment more accessible.

To further evaluate the developer experience, surveys among the developers at OpenXchange who are using this environment could be conducted.

5.1.3 Reliability

Due to the fact that the current solution only uses Nix, the WSL and container images for the environment, it is fairly stable. This is because Nix has been around since 2003 and has an active community of maintainers that work on making the system more stable. The official Nixpkgs repository is also at the time of writing the largest most up-to-date package repository of all Linux package distributions. [37] Nevertheless, there have been some issues with broken packages after an update of the environment, since we are using the unstable channel to always get the newest versions of the packages. But this is not really an issue as a rollback to an old version of the environment just requires to restore the old lock file from the version control.

The reliability of the WSL and container runtimes have not been further explored by me.

5.1.4 Performance

The performance of the instantiation of the environment depends on multiple factors. First, the initial building of the environment on a developer's workstation can be rather slow. In most cases the official binary caches of the Nixpkgs are already populated with the needed derivations eliminating the need to build the software from its source code. Nevertheless, the derivations need to be downloaded from the binary cache server over the internet which can be slow depending on the internet connection. Once the environment has been build on one workstation, the next instantiation of the environment on the same workstation is very fast as it only needs to create some symbolic links to the nix store.

For the CI pipelines it is very similar. The first run with an updated environment needs to use the official binary caches over the internet. After this initial run the internal S3 binary cache is populated which makes the next instantiation faster as the derivations can be loaded from the company-internal network storage. However, it still is not as fast as using pre-build container images which can be completely cached on the hosts that run the containers. Therefore I will create a meta-pipeline in the future that creates the container images when the nix environment changes. This is likely to increase the speed of instantiation for following CI pipeline runs as the nix environment can be completely cached by the hosts that run the containers.

The computational runtime performance is not affected by the Nix approach as the binaries are directly executed on the underlying system as if they were installed as normal packages by the OS. For developers using a Windows OS there is the additional runtime overhead of running the applications on the WSL which I did not benchmark.

5.2 Evaluation of the Theoretical Concepts

My second research question was: "What are the available approaches to achieve reproducibility and how do they compare?"

I created a catalogue of different approaches that can be used as building-blocks to implement reproducible environments. All the approaches can reproduce different properties of the environment, while having different requirements for the input environment. Using these characteristics as trade-offs I was able to compare them in my case study to find a matching solution to my requirements. I combined the approach of reproducible package management with Nix to install the required software packages reproducibly. I used virtual machines in form of the WSL to be able to use Nix on a Windows host. Additionally, I used containers to be able to use Nix in the CI environment.

My first research question was: "What is a definition of reproducibility of computational environments beyond the context of scientific experiments that also covers environments for software development?"

I provided a definition based on mathematical models to describe computational environments as well as reproducibility of computational environments. In the catalogue of approaches I was able to argue about the properties that can be reproduced by an

approach using the definition of computational environments. Additionally, I was able to argue about trade-offs between the properties of the output environment, the complexity of the recipe and the requirements to the input environment using the definition of reproducibility of computational environments. In the case study I was also able to use the *JSON* representation of the expected environment to argue about possible approaches. Therefore I can conclude that the provided definitions were sufficient to specify and argue about the reproducibility of computational environments in this thesis.

6 Conclusion

In this thesis I explained what a reproducible computational environment is and what benefits it can provide in the context of software development. The main argument for having reproducible environments is to reduce "it works on my machine" problems and unify the computational environments of developers and other systems like the CI pipeline.

I presented a definition of computational environments that can be used to describe any computational environment in various details. The definition uses a structure that can be visualized in a *JSON* syntax with arbitrary nested attributes. With this definition, two computational environments can be compared with each other in equivalence. The definition also refines other existing definitions of computational environments while still being compatible with them. I also presented a definition of reproducibility of computational environments. It uses a deterministic function that, applied to an input environment and a recipe, produces an output environment that is equivalent to the desired reproducible environment. The definition can be used to help arguing about the different approaches to reproducibility.

I created a catalogue of approaches to reproducible computational environments. The different approaches each have their own set of properties they are able to reproduce under certain conditions. Each approach therefore has its own trad-offs regarding the portability of environments and the amount of properties one can reproduce. The approaches can be combined like building-blocks in different ways to fit the needs of a project.

In the case study I implemented a reproducible environment for a real-world software project. I compared the trade-offs of the different approaches in the context of the given project and decided on a subset of approaches to use. I showed how it is possible to reproduce software packages across different environments using Nix.

6.1 Future Work

My definition of computational environments creates a framework for describing computational environments with arbitrarily named properties. However, there is no standardization of identifiers for these properties. Therefore, anyone can use their own identifiers to refer to a particular property of their environment. In order to use the definition for more practical implementations, it might be interesting to define a standard set of identifiers. This schema could then be used to automatically create descriptions of environments and share them with others. This allows for other use cases such as automatically selecting the "best" approaches to reproducing the environment.

As discussed in section 2.3, software applications always have dependencies on the environment. These dependencies can be for example other software packages or the CPU architecture to run on. It is already possible to automatically scan for these dependencies and create a dependency graph. However, as already mentioned there can also be other more semantic dependencies like the structure of the file system that is expected or the dependency on a certain input device. To be able to automatically detect these dependencies would require a deep understanding of the semantics of the source code. It could be interesting to explore this further in order to create fully automated and complete scans of the dependencies of a program.

In this thesis I implied that using reproducible environments for software development might improve the development workflow. While I showed that the divergence of environments can be minimized, it is not yet clear if the benefit actually outweighs the effort. It might be possible that maintaining the reproducible environments imposes a bigger workload than "just dealing with diverging environments". To be able to explore this further, long-term studies are needed that compare development teams that use some form of reproducible environments with development teams not using reproducible environments.

As for using Nix in a containerized CI pipeline, the best approach for doing so has yet to be evaluated. The mentioned meta-pipeline to dynamically create container images might be one possible solution that performs better than the current one in this case study. However, there might be other more elegant solutions out there.

Bibliography

[1] *Apache Ant*. Apache Software Foundation. URL: https://ant.apache.org/ (cit. on p. 26).

[2] L. A. Barba. *Terminologies for Reproducible Research*. Feb. 9, 2018. arXiv: 1802.03311 [cs]. URL: http://arxiv.org/abs/1802.03311 (visited on 07/03/2022) (cit. on p. 7).

[3] *Bazel*. Google. URL: https://bazel.build/ (cit. on p. 22).

[4] F. C. Y. Benureau and N. P. Rougier. "Re-Run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions." In: *Frontiers in Neuroinformatics* 11 (Jan. 4, 2018), p. 69. ISSN: 1662-5196. DOI: 10.3389/fninf.2017.00069. URL: http://journal.frontiersin.org/article/10.3389/fninf.2017.00069/full (visited on 07/06/2022) (cit. on p. 11).

[5] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Dec. 2017. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/info/rfc8259 (cit. on p. 10).

[6] G. Brown. "It Works on My Machine! How Container Technologies Like Docker Can Revolutionize Continuous Integration." 2014. URL: https://www.usenix.org/conference/ures14/technical-sessions/presentation/it-works-my-machine-how-container-technologies (cit. on p. 1).

[7] J. F. Claerbout and M. Karrenbach. "Electronic Documents Give Reproducible Research a New Meaning." In: *SEG Technical Program Expanded Abstracts 1992*. SEG Technical Program Expanded Abstracts 1992. Society of Exploration Geophysicists, Jan. 1992, pp. 601–604. DOI: 10.1190/1.1822162. URL: http://library.seg.org/doi/abs/10.1190/1.1822162 (visited on 07/03/2022) (cit. on pp. 7, 11).

[8] T. T. W. Community. *The Turing Way: A Handbook for Reproducible, Ethical and Collaborative Research*. Version 1.0.1. Zenodo, Nov. 10, 2021. DOI: 10.5281/ZENODO.5671094. URL: https://zenodo.org/record/5671094 (visited on 05/04/2022) (cit. on pp. 7, 8).

[9] L. Courtès and R. Wurmus. "Reproducible and User-Controlled Software Environments in HPC with Guix." In: *Euro-Par 2015: Parallel Processing Workshops*. Ed. by S. Hunold et al. Vol. 9523. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 579–591. ISBN: 978-3-319-27307-5 978-3-319-27308-2. DOI: 10.1007/978-3-319-27308-2_47. URL: http://link.springer.

com/10.1007/978-3-319-27308-2_47 (visited on 05/03/2022) (cit. on pp. 19, 29).

[10] *CPython Bug Report on Windows*. GitHub. Mar. 10, 2015. URL: https://github.com/python/cpython/issues/67822 (visited on 07/31/2022) (cit. on p. 14).

[11] *Direnv*. URL: https://direnv.net/ (cit. on p. 35).

[12] *Docker*. Docker, Inc. URL: https://www.docker.com/ (cit. on pp. 21, 27).

[13] E. Dolstra, M. de Jonge, and E. Visser. "Nix: A Safe and Policy-Free System for Software Deployment." In: *Proceedings of the 18th USENIX Conference on System Administration* (Atlanta, GA). LISA '04. USA: USENIX Association, 2004, pp. 79–92 (cit. on p. 29).

[14] B. Elder. *Kubernetes Pull Request: Remove Bazel*. GitHub. Mar. 1, 2021. URL: https://github.com/kubernetes/kubernetes/pull/99561 (visited on 07/31/2022) (cit. on p. 22).

[15] J. Engblom. "Continuous Integration for Embedded Systems Using Simulation." In: *Embedded World 2015 Congress*. 2015 (cit. on p. 18).

[16] *GitHub Flow*. URL: https://docs.github.com/en/get-started/quickstart/github-flow (visited on 07/16/2022) (cit. on p. 25).

[17] *Gradle*. Gradle Inc. URL: https://gradle.org/ (cit. on p. 26).

[18] M. A. Heroux et al. *Toward a Compatible Reproducibility Taxonomy for Computational and Computing Sciences*. SAND2018-11186, 1481626. Oct. 1, 2018, SAND2018–11186, 1481626. DOI: 10.2172/1481626. URL: http://www.osti.gov/servlets/purl/1481626/ (visited on 07/03/2022) (cit. on p. 7).

[19] P. Ivie and D. Thain. "Reproducibility in Scientific Computing." In: *ACM Computing Surveys* 51.3 (May 31, 2019), pp. 1–36. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3186266. URL: https://dl.acm.org/doi/10.1145/3186266 (visited on 07/03/2022) (cit. on pp. 2, 8, 21).

[20] M. Jiang et al. *Automatically Locating ARM Instructions Deviation between Real Devices and CPU Emulators*. Aug. 12, 2021. arXiv: 2105.14273 [cs]. URL: http://arxiv.org/abs/2105.14273 (visited on 07/31/2022) (cit. on p. 18).

[21] L. Kuper et al. "Freeze after Writing: Quasi-Deterministic Parallel Programming with LVars." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Diego California USA: ACM, Jan. 8, 2014, pp. 257–270. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535842. URL: https://dl.acm.org/doi/10.1145/2535838.2535842 (visited on 07/06/2022) (cit. on p. 13).

[22] C. Lamb and S. Zacchiroli. "Reproducible Builds: Increasing the Integrity of Software Supply Chains." In: *IEEE Software* 39.2 (Mar. 2022), pp. 62–70. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2021.3073045. URL: https://ieeexplore.ieee.org/document/9403390/ (visited on 08/01/2022) (cit. on p. 22).

[23] J. MacFarlane. *Pandoc*. URL: https://pandoc.org/ (cit. on p. 27).

[24] M. Mercier, A. Faure, and O. Richard. "Considering the Development Workflow to Achieve Reproducibility with Variation." In: *SC 2018 - Workshop: ResCuE-HPC*. Dallas, United States, Nov. 2018, pp. 1–5. URL: https://hal.inria.fr/hal-01891084 (cit. on pp. 4, 8).

[25] D. Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583 (cit. on p. 20).

[26] A. Mokhov, N. Mitchell, and S. Peyton Jones. "Build Systems à La Carte." In: *Proceedings of the ACM on Programming Languages* 2 (ICFP July 30, 2018), pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3236774. URL: https://dl.acm.org/doi/10.1145/3236774 (visited on 07/16/2022) (cit. on p. 22).

[27] O. S. Navarro Leija et al. "Reproducible Containers." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20: Architectural Support for Programming Languages and Operating Systems. Lausanne Switzerland: ACM, Mar. 9, 2020, pp. 167–182. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378519. URL: https://dl.acm.org/doi/10.1145/3373376.3378519 (visited on 05/03/2022) (cit. on p. 11).

[28] *NixOS / Nix Package Manager*. URL: https://nixos.org/ (cit. on p. 29).

[29] *NixOS RFC 0049 - Flakes*. URL: https://github.com/NixOS/rfcs/pull/49 (cit. on p. 33).

[30] *Nixpkgs*. URL: https://github.com/NixOS/nixpkgs (cit. on p. 32).

[31] *Numtide Devshell*. Numtide. URL: https://github.com/numtide/devshell (cit. on p. 38).

[32] *OX AppSuite*. URL: https://www.open-xchange.com/products/ox-app-suite/ (visited on 07/16/2022) (cit. on p. 25).

[33] *Packer*. HashiCorp. URL: https://www.packer.io/ (cit. on p. 19).

[34] C. Paulsen and R. Byers. *Glossary of Key Information Security Terms*. NIST IR 7298r3. Gaithersburg, MD: National Institute of Standards and Technology, July 2019, NIST IR 7298r3. DOI: 10.6028/NIST.IR.7298r3. URL: https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.7298r3.pdf (visited on 05/04/2022) (cit. on p. 8).

[35] R. D. Peng. "Reproducible Research in Computational Science." In: *Science* 334.6060 (Dec. 2, 2011), pp. 1226–1227. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.1213847. URL: https://www.science.org/doi/10.1126/science.1213847 (visited on 07/03/2022) (cit. on p. 3).

[36] *QEMU*. Version 7.0.0. URL: https://www.qemu.org/ (cit. on p. 18).

[37] *Repology: Repository Statistics*. URL: https://repology.org/repositories/statistics/newest (visited on 07/27/2022) (cit. on p. 38).

[38] *Reproducible Builds*. URL: https://reproducible-builds.org/ (cit. on p. 22).

[39] R. F. Schmidt. "Software Architecture." In: *Software Engineering*. Elsevier, 2013, pp. 43–54. ISBN: 978-0-12-407768-3. DOI: 10.1016/B978-0-12-407768-3.00003-3. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780124077683000033 (visited on 05/04/2022) (cit. on p. 8).

[40] A. L. Tavares and M. T. Valente. "A Gentle Introduction to OSGi." In: *ACM SIGSOFT Software Engineering Notes* 33.5 (Aug. 31, 2008), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/1402521.1402526. URL: https://dl.acm.org/doi/10.1145/1402521.1402526 (visited on 07/16/2022) (cit. on p. 25).

[41] VMware. *What Is a Virtual Machine?* URL: https://www.vmware.com/topics/glossary/content/virtual-machine.html (visited on 06/13/2022) (cit. on p. 19).

[42] *What's New In Python 3.0*. Python 3 Documentation. URL: https://docs.python.org/3/whatsnew/3.0.html (visited on 07/30/2022) (cit. on p. 2).

[43] *Windows Subsystem for Linux (WSL)*. Microsoft. URL: https://docs.microsoft.com/en-us/windows/wsl/ (cit. on p. 28).