Universität Bremen

Fachbereich 3: Mathematik und Informatik

Master's Thesis

Long-term analysis and visualization reproducibility of heterogeneous robotic experience data in a continuously developed knowledge processing service

German title: Langzeit-Reproduzierbarkeit von Analyse und Visualisierung heterogener Robotikexperimentdaten in einem ständig weiterentwickelten Wissenverarbeitungsdienst

Moritz Horstmann

Matriculation No. 2590074

28th January 2019

Examiner: Prof. Michael Beetz PhD

Supervisor: Dr. Karsten Sohr

Advisor: Daniel Beßler

Moritz Horstmann

Long-term analysis and visualization reproducibility of heterogeneous robotic experience data in a continuously developed knowledge processing service

German title: Langzeit-Reproduzierbarkeit von Analyse und Visualisierung heterogener Robotikexperimentdaten in einem ständig weiterentwickelten Wissenverarbeitungsdienst

Master's Thesis, Fachbereich 3: Mathematik und Informatik

Universität Bremen, January 2019

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 28th January 2019

Moritz Horstmann

Contents

Chapter 1

Introduction

More and more researchers profit from the use of computational power to advance in their field. Computer simulations, statistics and the possibility to collect and process huge amounts of data (big data) offers advantages to sciences like biology, physics and engineering. Basically any field of research can leverage digital data collection and the processing of it. The results of Big data science enabling the gain of information from largely unstructured datasets and the proliferation of powerful computing hardware lowered the entry bar to benefit from this technology.

When using digital data and software to publish research results, often only the interpreted and processed results are included into the publication. The replication of such results is next to impossible, as the foundation of results gained through computational science often lies within the used data, and not in the description published along with the result. The importance of result replication can be seen in the 'replication crisis' of scientific studies and research results in psychology, where many of the result could not be reproduced and are thus considered to be flawed [Sch14].

To prevent this flaws in scientific research results, the concept of Open Science gained popularity in the research community, defining a framework to conduct transparent research with reproducible results. The goal of this thesis is to establish the requirements to reproduce results obtained by the application of research software, which is a sub-category of the Open Science movement, and the transfer of those results to the cloud-based knowledge processing service openEASE. The analysis of openEASE regarding its suitability to conduct reproducible research with it as a research toolkit, as well as the reproducibility of the openEASE software state itself.

The analysis then is used to suggest amendments to the architecture of openEASE following the established requirements, followed by the implementation of some of those suggestions. During this research, a data management and build artifact archiving software was implemented as an experimental case study, which is then evaluated for its suitability to assist in efficient archiving artifacts used during a software build process, and the repetition of such a build process just by using the archived artifacts. Together with the implemented architectural changes of ope-

nEASE, this thesis presents a guideline and takes the first steps towards software reproducibility of an existing research software.

Chapter 2

Motivation

Many researchers of various research fields leverage the collection and analysis of data. Their effort can either be directed at working on collection processes and analysis algorithms directly as part of their research (e.g. data science), or be directed at applying established computing and data processing methods to aid them in advancing their topic (e.g. statistics, computational science). This chapter discusses the importance of reproducibility of all data related actions in research activities, and the different challenges in achieving it.

2.1 Reproducibility in research

When referring to reproducibility in research, this work focuses only on the reproducibility of computer assisted data processing, as it already offers a wide range of interesting questions and challenges to solve. How is reproducibility of data processing defined, how can it be achieved and what are the benefits of it?

In typical scenarios (not limited to research), there are many steps involved in performing data processing tasks. At first, there is the collection of data. Data sources have to be found and exploited or the data has to be recorded, imported or even typed in manually. When the data is digitally available and accessible by the researcher, it then needs to be preprocessed further based on the format and quality of the data, and the required target format of the software designated to process the data. The preprocessed data is then passed to either existing software, custom made software or a hybrid of both, to achieve the actual processing of the data. As a result of this process, new data or an alteration of the original data is produced, followed by post-processing for aggregation, evaluation or visualization, which is then ultimately used for deriving or validating results. In varying degrees, some of the described steps might already take significant time and effort on their own, even such that they can be the main goal of a research topic, or the sole purpose of a commercial company.

Ideally, when research is executed with extra care for the reproducibility of its data processing, the complete chain of processing steps is transparent, meaning that it is documented in great detail how the involved steps can be replicated. At least the raw source data should be published so the (possible not easily reproducible) task of data collection does not has to be repeated - in the best case, all data results from each processing step is also made available. Apart from data itself, the environment in which the data was processed is an important factor to be considered for successful reproduction, too. The emphasis here does not lay on the sharing of locations or hardware involved in the data processing, but rather the software and its configuration used in the process.

Being able to reproduce all or parts of the data processing steps from research gives the research community many advantages. It fosters the peer-review of publications, as reviewers are able to validate the progress from the research basis to the final result, in contrast to validating a publication based on just the reported result. Follow up research can profit from reproducible research by using both algorithm implementations and used datasets: Improved processing algorithms could be used to extract information with higher quality and diversity from the gathered data, which is a fundamental principle of big data analytics, but generally applies to all data intensive research areas (machine learning/artificial intelligence, biology or medical research to name a few). Or, new and improved processing algorithms could be applied to existing datasets outside the scope of the original authors, without the need of re-implementing algorithms in software. Making newly created research software open source and available can lead to enhanced collaboration between researchers, sharing of software revisions and consequently improving the results it produces.

However, there are some challenges in conducting reproducible research and critique against the openness of this methodology. To get a better understanding of the challenges faced in the involved fields, the following section will provide an examination of them in the context of the open science movement, before gathering individual issues subsequently.

2.2 Open science

Research reproducibility is part of the open science movement. Open science is the description of a methodology framework defined by the principles of "transparency, universal accessibility and reusability of the scientific information disseminated via online tools" [Pon+15, p. 1]. The European Commission funded project FOSTER has compiled a taxonomy for open science, from which three categories applicable to this thesis can be extracted: "Open Data", "Open Reproducible Research" (specifically "Open Source in Open Science") and "Open Science Tools" (figure 2.1 on page 5). The first two categories describe the publication of research software and

ancillary data, the latter mainly describes the toolkit to support the former.

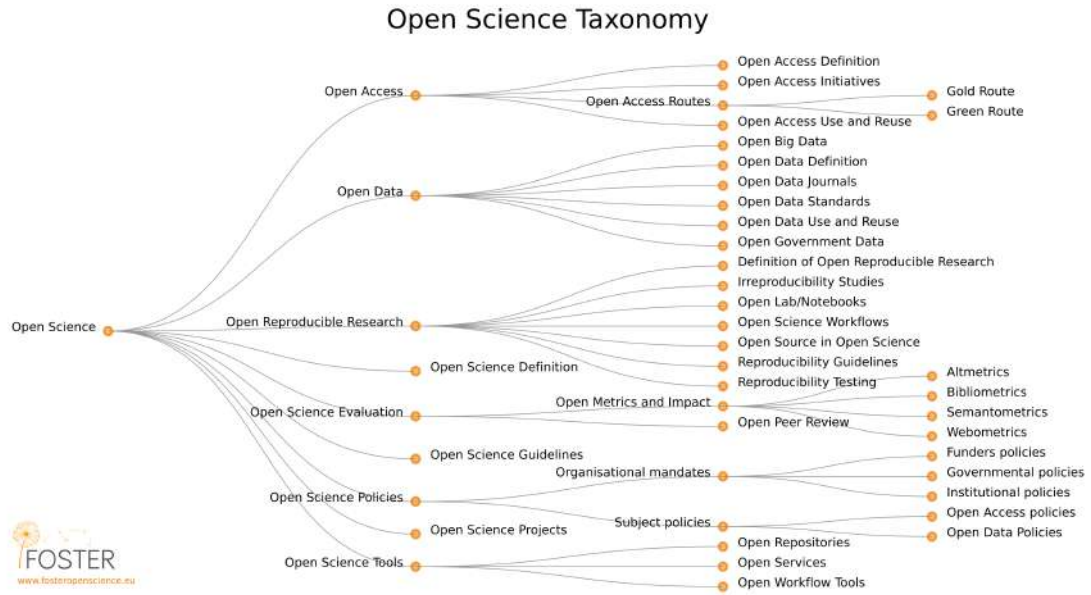

Figure 2.1 A taxonomy of open science [Pon+15]

While sharing the basic general motive, different groups of scholars follow slightly different goals with being more open in their research. Fecher et al. identified two objectives of practicing open science that are endorsed by the first motivation section: Collaboration between scientists and the availability of collaboration platforms and tools, as well as free availability of data, publications and software/code. The other identified objectives are improved citation/impact metrics, peer reviews, and broader science accessibility for citizens [FF14].

With the previous section already covering some supportive arguments, critical opinions about open science do also exist. In his book "Reinventing Discovery" from 2012, Nielsen brings up the remarkable opinion of a scientist on Paul Ginsparg (the creator of arXiv, one of the first instance of open access media [Gin91]): "[He] had 'wasted his talent' for physics by creating the arXiv, and [...] what Ginsparg was doing was like 'garbage collecting'" [Nie11, p. 182]. At least some parts of the research community seem to show a great disregard for the creation of assistant tools for science, any effort invested into tool development is considered a waste of time. When this disregard is shared by research institute leaders, member researchers would also lack the funds and time to spend on innovating the toolkit in their field. And reservation against creating software instead of writing publications is not unwarranted at first glance, because when institute funding depends on the visibility of its research or publication count, potentially resource intensive software development would not immediately contribute to that.

Another aspect brought up by critical scholars is the possibility of being overwhelmed by shared

research, which could apply to both mass publication of research data and early or excessive collaboration [Nie11, p. 198]. As publishing data and source code alongside with papers would greatly increase the volume of information to review, it is also imaginable that either the review expenditure rises, or more information is published unreviewed. The issue with vast amounts of information with varying quality can be seen in the field of big data, where making sense and finding answers from huge datasets is an ongoing challenge [SS13] [Har14]. Publishing too much data could eventually introduce dependencies to similar querying and analysis technologies, making further research dependent on the quality of such technology.

Not only the publishing of data and software is found to be problematic, the reliance on complex data and software in research is accused of making the verification of the research itself harder [Nie11, p. 202f]. On the other hand, at least the reproducibility of research would definitely benefit from this, as those artifacts are required in the first place to recreate it.

Finally, Lancaster finds in his article "Open Science and its Discontents": "Other aspects of the open science movement, including building tools for reproducibility, [...] code, infrastructure and raw data [...] are still largely unrewarded by the current academic system" [Lan16]. This confirms the assumption about the perception of tool development in science and could further explain the lack of effort on other categories of open science, as well.

Moving on from theoretical advantages and disadvantages to practical challenges, the next sections assess individual open science categories, and offer related examples.

2.2.1 Open Data

Sharing research data offers a wide range of benefits for the research community. Not needing to set up and execute expensive and/or time-consuming experiments for data acquisition lowers the initial hurdle for new researchers and institutes with less funding to contribute to that field.

However, the handling, publication and usage of such data needs consideration on some issues in order to be useful to others.

- *Data format*: The data format is a key factor when it comes to re-usability and reproducibility. It determines the requirements and preparation time necessary to use the data. As data usually needs to be interpreted by software, choosing a proprietary data format requiring proprietary and closed-source software should be only done when absolutely necessary, as it can hinder the later use of the data. Licensing requirements can impose issues for researchers and institutes with less financial resources. Newer software versions could be incompatible, compatible versions could be made unavailable or require deprecated hardware and operating system environments. The software development company could no longer exist and copyright laws could prevent the access to working copies of the software.

Another recommendation is the use of standardized and well established data formats. Those formats have the advantage of offering a wide variety of existing and possibly maintained software libraries and tools, allowing researchers to operate with the development environment and programming language they prefer.
The 5-star open data initiative defines similar guidelines and metrics about the quality of data formats and their presentation, which support these considerations [KH].

- *Meta data*: Third party data can only be useful if it can be found in the first place and has proper descriptions available. Meta data helps search engines and data management software to provide researchers with relevant data according to their queries and filters. For identifying and referencing datasets, a unique identifier that is globally recognized and established should be assigned.
- *Storage*: For data to be published, the established publishing instruments have limited suitability. A suitable data hosting service has to be commissioned to publish the data, either with the own institute if it offers one, or with a third party institution or company. For the choice of storage, the requirements of service level has to be clarified: How much data storage is required? How and when is the data transferred to the service? Is the storage required to be available before the publication and are additional resources such as processing power for data analysis etc. desired? How long is the retention time of the data? Is the service protected against data loss?
  If the dataset is small (up to three-figure megabyte range), this may not seem like a difficult task, as cheap and ubiquitous consumer-grade storage solutions can be used. On the contrary, a simple change in the infrastructure of such a solution could for example render published and printed hyper links to data invalid. At least when the dataset exceeds the gigabyte bound, different solutions are required that are capable of handling large data volumes and guarantee long term access to the data [CER].

These considerations do come at a cost for the publisher. Preparation and conversion of open data formats, chunking data into individually addressable parts and writing meta data for it is a complex and time-consuming task. Limited resources and hosting costs can be a major obstacle for researchers to publish their data alongside their regular publications. The success of open data is coupled to the availability of tools and services to support researchers and ultimately the acceptance and valuation of open data contribution in the research community.

For reference, there are some noteworthy open data sources and services for research data publication available. An extensive list of open data sources is the Awesome Public Datasets list, comprising of references to over 600 publicly availably datasets [Awe19]. Examples of research data hosting services include Zenodo[1] and Dryad[2]. Zenodo is a data storage service for research data of all sciences and humanities. It provides 50 Gb of storage free of charge, with more capacity being availably by request. The data is kept indefinitely on the same petabyte scale

[1] *https://zenodo.org*
[2] *https://datadryad.org*

infrastructure as the data from the Large Hadron Collider. It offers a public search engine for the hosted datasets and assigns a Digital Object Identifier (DOI) for uploaded data [OC]. It is operated by OpenAIRE[3], a service for accessing European Commission funded research. Dryad also offers curated data storage services, providing additional release and embargo management capabilities, supporting the coordination of publication with journals and assigning a DOI for datasets. It charges $120 in fees per publication, which include 20 Gb storage space and is operated by a nonprofit organization [Isa+07].

Further services do exist which offer a complete data storage and analysis suite, some of which are covered in the next section about the challenges of using and providing software in the context of reproducible research.

2.2.2 Software in research

When existing standard software is used to process research data, the most important considerations for reproducibility are to be made in the realm of Open Data covered in the previous section. The only additional requirement for research to be reproducible in that case is the availability of the software and its configuration (if applicable). But if custom made software modules are used for data processing, extra care has to be taken in order for the process to be reproducible, and the reproducibility of the software might decay over time without countermeasures.

For software to function properly in general, it is required to at least possess its binary executable alongside with the third party libraries and modules it depends on. Depending on the technology the software was created with, a certain version of a software runtime and/or a certain operating system with a certain version is also required. For both commercial and most free/open-source software projects, these requirements are managed and controlled by maintainers, who provide documentation, installation packages and updates in case of updated or changed dependencies (from software libraries to operating systems).

Properly maintaining a software again requires time and effort, especially for software with a large and complex code base, and there are some reasons imaginable why researchers might chose not to publish their custom-made software, which consequently impedes the reproducibility of their research:

Software artifacts might not be published by a researcher if they were only used for evaluation purposes. While the results of a software can be aggregated comprehensively for evaluation using tables or graphs, the software itself may not be feasible for publication. An issue preventing the publication of such software can be missing documentation: To set up and execute a software, extensive instructions about the required operating environment (such as hardware- or operating

[3] *https://www.openaire.eu/*

system requirements) and instructions on how to build the software from source-code or how to acquire and install binaries is required. This effort can be reduced by using public source-code hosting services (e.g. github[4] or bitbucket[5]) and standard software development frameworks, such as common build tools (CMake, Maven, Gradle, MSBuild) or self contained bundling formats (e.g. static linking, container images or software installers). However, the development of software that employs those frameworks requires conformance to the frameworks' guidelines and ultimately still needs additional effort by the researcher.

When software is not designed to be released to the public, but is only intended to be run on a researchers' machine, the quality of the software may hinder the public release of it. Imminent deadlines, inexperience in software development or indifference can lead a researcher to disregard existing or suitable software architecture, design patterns and coding styles, adding and changing functionality where it does not belong and inflating complexity. Combined with the circumstance that software used to yield or verify a research result is not valuated equally as the result itself, a researcher may have very little motivation to adhere to software development quality standards and include software artifacts in publications [Boe15, p. 71f].

If software artifacts are released alongside the research publication, the documentation is sufficient to execute it on another machine and all required dependencies are present, reproducibility of the software result is still not guaranteed. Software can perform differently on machines with deviating hardware specifications resulting in mismatching performance, different outcomes or failures. To prevent this, the software would have to undergo extensive testing on different operating environments - or the system environment of the developer must become the only authorized configuration. When software dependencies (e.g third party libraries) are not pinned to a specific version, any major update might break compatibility or introduce behavioral changes. Dependencies from unreliable or user-provided sources can become unavailable or be deleted by their owners, resulting in software build breakage [Wil16]. Measures against these types of failures (such as software tests or quality assurance) are usually not prevalent in research software, as they are very resource intensive. Also they do not prevent incompatibilities introduced by software the developer can not control (e.g. a browser), no passive or preventive countermeasures exist against this issue, and therefore active maintenance is required.

For most of the data- and computational-science related tasks which require custom software modules, there are a range of web-based services offering a complete research data hosting and analysis development suite. Researchers who can fit their custom data processing algorithms into a framework offered by one of these services can simply reference those in their publication - setup, maintenance, hosting and publication of the software is taken care of by the service. One example is the data science community kaggle. It features a collection of publicly accessible datasets, provides hosting for datasets (10 Gb maximum each) and a Python-based develop-

[4] *https://github.com*
[5] *https://bitbucket.org*

ment environment for machine learning, deep learning, visualization and statistical analysis of uploaded datasets. The software environment is based on container technology and runs in their data center, thus no installation effort is required. They host data science competitions where institutions and companies can offer rewards for solving their data science problem. kaggle is free to use at the time of writing and owned by Alphabet Inc. [GH]. Another service offering a platform for the development, operation and publishing of data science workload is the Open Science Data cloud. It provides researchers compute resources for a time frame of three months, comprising of access to virtual machines and storage. From those resources, it is possible to access one petabyte of publicly available datasets. Users are assigned individually negotiated usage quotas at no to cost-covering expense [Gro+10] [Gro+12]. The last featured example is OpenML, a data science community similar to kaggle but with a focus on machine learning tasks. It hosts datasets and offers the possibility to construct own machine learning processing pipelines, which can be applied to said datasets. The supported dataset format choice is very limited (only the ARFF format [PTF08]) and the service is free to use at the time of writing [Rij+13].

Accomplishing research software sustainability is actively being discussed by members of research institutes and research infrastructure providers. In the "Knowledge Exchange Workshop on Research Software Sustainability", attendants put strong emphasis on the importance of extending the lifecycle of research software. They identified currently lived measures applied by the research community regarding the software lifecycle, ranging from productive examples such as using virtual machines and containers, as well as constant renewal of software, to destructive examples, like deprecation of software and ignorance ("7. Procrastination. Do nothing") [Het16, p. 9-10]. Aerts et al. suggest a formal framework to foster the introduction of software valuation and sustainability guidelines to be embraced by top level hierarchy research institutions and governments [AD16]. This seems like a sustainable approach, because requiring all researchers receiving public funding to put emphasis on software re-usability could break the cycle of irresponsible research software handling.

All discussed concerns and obstacles in achieving reproducibility in research software until now only covers short-lived software created and used for the a specific, time-constrained research matter, such as writing a paper, a thesis etc. Additional challenges arise when building longer-lived tools to be used by other researchers, as covered in the next section.

2.2.3 Open Science Tools

An Open Science Tool (or any software tool for that matter) is a special instance of research software, as its purpose is not limited to producing publishable results in a constrained data domain, but rather to assist multiple researchers during their research activities, possibly in diverse fields of research, over an extended amount of time and spanning multiple publications

and projects.

Besides the aforementioned considerations about research software reproducibility in general, it is especially important for research tools to focus on compatibility, reliability, longevity and a good user experience. The lifespan of a tool does not end with the scientific publication like many of the regular research software artifacts, it really just starts there. This increases the requirements to maintain reproducibility considerably: Was it sufficient to just 'keep the software alive' after publication before, it is now necessary to keep it in a state where new research can be done reliably while using it. Wishes for new functionality and updated requirements due to advancements in the research field the tool is targeting add to the existing workload of keeping the tool compatible with the current typical operating environment. Unless there is enough funding to employ staff dedicated to maintenance and support, the software authors need to balance the amount of work they invest in advancing the tool, responding to support requests from users and continuing their original research they designed the tool for in the first place.

Properly maintaining software is also time-consuming, studies estimate that proper software maintenance takes 40% [Wes93] to 75% [McK84] of the time spent on software development in total. Maintenance comprises of correcting errors found in the software, reacting to changed environments (both system environment and the environment of the domain the software operates in) and preventing future software issues (e.g. migrating to a faster storage technology when the old one will struggle with the work load in the foreseeable future).

Different system designs of research tools come with different challenges: Delivering the tool as an individually installable software package requires effort in data migration, different operating system support and easing the installation process, whereas providing the tool as a service might reduce said expenditure. Hosting tools as web-service on the other hand introduces the need for user data separation and management and the provision of resources for each user either for free, or for a payment with additional need for accounting. Operation and monitoring effort must also not be underestimated.

Summing up the difficulties achieving reproducible research data and software results, a general and cheap solution is unlikely to be found anytime soon. A good start would be for each individual researcher dealing with either valuable data or software to invest as much time publishing, preserving and maintaining it as comfortably deductible from the schedule and trying to promote the idea of reproducibility to the superior to earn additional time.

2.3 Open Science, reproducibility and openEASE

Leaving the general analysis of reproducibility requirements and returning to the objective of this thesis, it is left to inspect the subject for its position in the realm of Open Science, and what

areas could possibly conflict with the goal of making it sustainable for future use in research.

openEASE is a knowledge processing service for robot experience data developed by members of the Institute for Artificial Intelligence (IAI)[6] at the University of Bremen. It is designed to allow recording a robots episodic memory, comprising of data from the robots' sensors and most importantly of a symbolic representation of the plan data executed by the robot during its operation. Combined with the reasoning capabilities of the integrated knowledge base KnowRob , it enables semantic querying about why and how an action was performed by the robot, as well as the state of the environment and related objects inside before and after the action. A web-based workbench for interaction and querying of the recorded data is also provided, delivering visualization of the data and the results of the issued queries. Queries itself can be issued freely as a Prolog query (called goal), which is then passed to the KnowRob knowledge base providing the reasoning functionality. Additionally, a list of predefined queries may be presented with a description of its meaning to assist in usage of the system and to emphasize interesting reasoned results from an experience dataset [Ten+15]. openEASE is not only designed to be used by humans for evaluation of robotic experiences, but also the other way around. By recording human interaction (e.g. with a virtual pizza making game as demonstrated in the paper) and feeding it into openEASE , and also from re-using previously recorded robotic experience data, robotic agents can be trained to extract information about complex movements and repeat them [Bee+16].

openEASE offers a solution for the difficulty in reproducing complex robotic experiments. Research comprising a robotic experiment is usually bound to an enormous amount of highly integrated software components for sensor-data processing, movement planning, action planning and knowledge processing, interacting with many sensors and motors in a specific and carefully set up environment. Recreating an identical setup with the same parameters as used in the original experiment for verification and follow-up research requires similar or identical hardware components, is very resource consuming at best and thus often infeasible for other researchers. With its extensive reasoning capabilities, openEASE can be used to reconstruct the exact course of a robotic experiment, allowing detailed querying of algorithmic decisions and their impact on an experiment. As an Open Science Tool, openEASE would be able to foster reproducibility of research about algorithms used to control a robot. Any researcher could just upload a recording of the complete robotic sensor data, together with the internal belief-state of the robot and a digital description of the environment, generate a reference to the storage location in openEASE and publish it for other researchers to track, verify and possibly re-use the data effortlessly.

Even though openEASE is designed to be offered as a web-service for users and robotic agents, there are also some instances of openEASE already running at different organizations outside the IAI. One goal of openEASE is the possibility for other institutes to host their own instance, so this option is expected to be used continuously with increasing intensity. In this way, the

[6] *https://ai.uni-bremen.de*

maintenance effort of the software is the sum of both a centralized service and a distributed toolkit.

This software takes a special role in the context of Open Science being classified as both a research software and a research tool. The cutting-edge technology of knowledge processing is still researched with great intensity and frequency, with the knowledge processing engine and semantical framework KnowRob being a particularly active research target [BPB18] [Hai+18] [Yaz+18]. Thus, the code-bases of KnowRob and openEASE are constantly evolving. At the same time, openEASE is striving to become a knowledge processing service to be used outside the IAI by both researchers and automated systems like robotic agents. It invites researchers to import datasets of their robotic or human experience data, providing storage space for every registered account in the system and an Application Programming Interface (API) for accessing the reasoning service. It therefore is both an important contribution to the research field of knowledge processing and a valuable contribution to the Open Science movement.

For the reproducibility of openEASE itself however, this means that the expectations about its reliability and longevity are very high. New versions of openEASE must not interfere with the functionality of visualizing and analyzing existing datasets. It shall be possible to upload a dataset, and safely assume that it would be possible to successfully reload it and run at least the same queries on the dataset that were available at the time of its upload 10 years later. Despite the long-term reliable access to datasets and functions, it is still desired to regularly be able to add new features to openEASE , restructure its architecture to fit future requirements or to deprecate and remove features. Given the high research activity around openEASE and the software it comprises of, amending the software modules doing the knowledge processing and related activities should not be hindered by the preservation and recreation measures.

The next chapter analyses the current state of openEASE , if and how it is affected by the previously established issues and which requirements it currently fulfills and lacks to be considered a reproducible software with reproducible outcome.

Chapter 3

Sustainability analysis of openEASE

To assess the suitability of openEASE to be both a dynamically developed research software and to offer its users a safe and reliable way to access their datasets and retain the software capabilities they were offered at the time of upload to execute on their datasets, an in-depth analysis of the current state of openEASE needs to be carried out.

This chapter provides a description of the components openEASE consists of and its architecture, and examines its components for structural and quality issues threatening the goal of functionality reproduction with continued development-

3.1 Introduction of openEASE

The provision of a knowledge processing service is a complex endeavor, requiring a lot of involved software and systems. openEASE accomplishes this task through two software parts, which are involved in providing the actual reasoning service and robotic experience analysis together with an in-browser visualization and query console.

The first part, the knowledge processing platform KnowRob which is also developed at IAI, provides reasoning and analysis capabilities for recorded data. It contains an extensive ontology, a fine-grained and detailed representation of object classes and general actions. KnowRob contains "the vocabulary for describing knowledge about actions, events, objects, spatial and temporal information", allowing semantic interpretation of data having no semantic annotation of its own [BTW15]. The predefined classes and actions are sufficient to semantically describe robotic experience data, like logged information from perception systems [Bee+15] or high-level task and action planning software [BMT10]. Combined with a description of the environment and location information of the participating entities (robot and perceived/manipulated objects), which can be obtained from sensor or internal state data (laser scanners, depth-camera, robot coordinate frame data), KnowRob allows semantic querying on the episodic memory.

The queries, as well as large parts of KnowRob itself are written in the Prolog logic programming language, which represents program logic as rules. A rule is a declaration of a predicate that evaluates to true when the given condition is also true. When the condition of a rule is always true, it is called a fact. A Prolog interpreter (like SWI-Prolog, also used in KnowRob [Wie+12]) then uses the set of rules and facts to evaluate input statements. When the statement contains no variables, the interpreter tries to verify the statement and outputs true if the statement is conforming to the set of known rules, or false if it violates a rule. If the statement contains a variable, the interpreter backtracks the set of known rules, trying to find matching candidates of the variable for the statement to become true. KnowRob defines Prolog rules to represent various data types from input data in Prolog (e.g. time or location), accessing external data from Prolog and to reason on this data with the defined ontology. Leveraging the Prolog to Java bridge of SWI-Prolog, KnowRob also provides a set of companion modules written in Java to perform tasks lacking native Prolog library support or which are easier (or only) implementable with an imperative programming style (e.g. accessing a MongoDB[1] NoSQL database).

KnowRob is integrated into the Robot Operating System (ROS) infrastructure, a system designed to cater the needs of robotic agents, providing a common framework and inter-process communication for the diverse required software to operate a robot. This allows live reasoning of data captured live during operation of the robot and to let the robot decide its next action based on inferred knowledge. This advantage however turns into a burden when running KnowRob outside a robotic agent, requiring a ROS installation for correct operation. ROS imposes very strict preconditions for the installation of a specific release version, as it requires a specific version of the Ubuntu[2] GNU/Linux long time supported operating system.

The other part of openEASE , comprises of a rich client to the KnowRob reasoning engine running in the web browser. It implements a workbench solution for users to view, analyze and visualize previously recorded robotic experience data. A Graphical User Interface (GUI) allows users to issue Prolog queries to KnowRob, either by freely typing them, or by selecting them from a list of predefined queries, which are described in plain English language. The predefined queries are curated by the maintainers of openEASE and enable users to quickly explore interesting data points in the experience data. The inferred results of the Prolog queries are returned to the user as raw Prolog response, but also visualized in a freely explorable 3D rendering of the experience setting, fed by semantic environment data from KnowRob (see figure 3.1 on page 17), and a textured 3D model of the robot, if present. Images or videos of selected episodes are displayable in a designated area of the GUI, as well as statistical data gathered by KnowRob through a graph rendering. Technologically, the client leverages the RobotWebTools[3] stack, a collection of ROS modules and JavaScript libraries to enable access to various ROS resources from a browser (or any remote system, that is). Normally, the ROS

[1] *https://www.mongodb.com*
[2] *https://www.ubuntu.com/*
[3] *https://github.com/RobotWebTools*

infrastructure only allows inter-process communication over a local area network with bidirectional communication, prohibiting the use of firewalls and routing/address-translating facilities. RobotWebTools overcome this limitation by tunneling ROS communication data through common and widely used protocols (like WebSocket[4]) over the Internet. Additionally, it offers a set of visualization libraries for common data formats used in ROS software and communication, which is also used by openEASE.

Further features of the web front-end of openEASE are user management to supply individual users their own KnowRob knowledge processing instance with separated data, an editor for user-supplied ontologies and Prolog programs and administrative options to maintain the content of the episodic memory database.

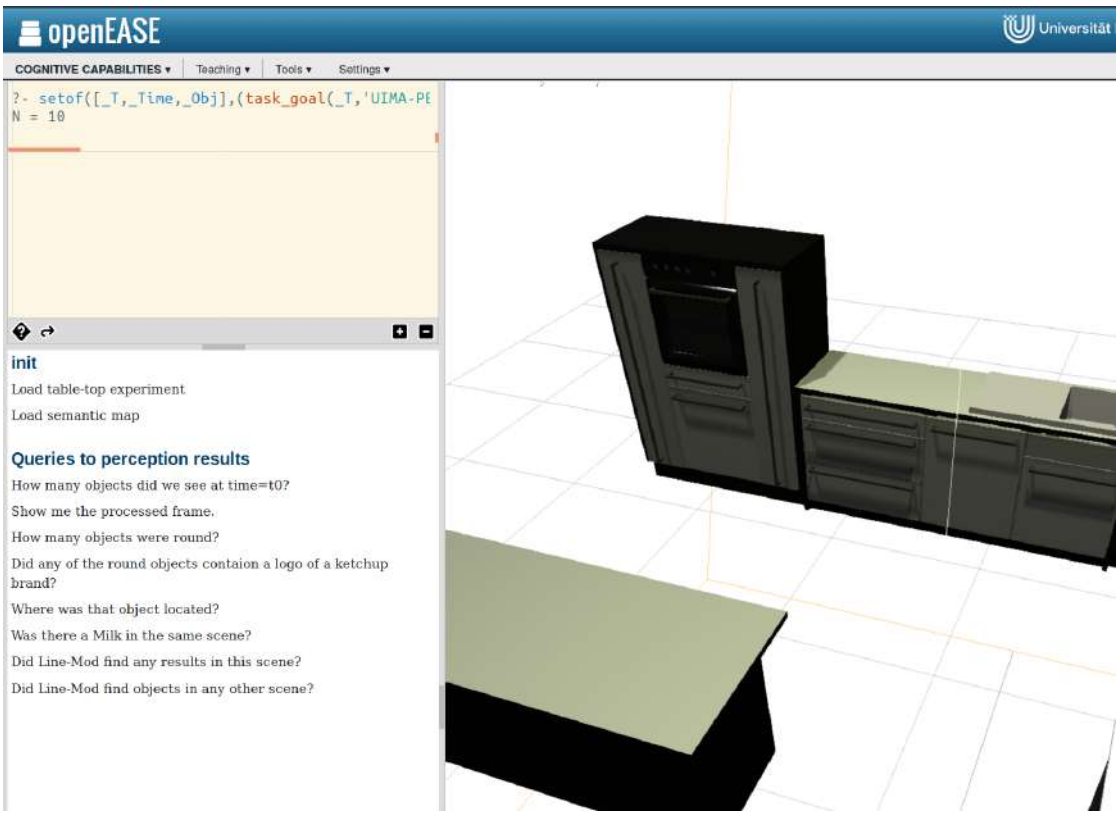

Figure 3.1 openEASE GUI

[4] *https://tools.ietf.org/html/rfc6455*

3.2 Architecture

Another prerequisite to properly assessing openEASE according to the rules of reproducible research is to gain an overview about its architecture, and the purposes of each individual component.

openEASE operates on Docker [Mer14], a containerization technology which allows deploying and running software with their required libraries in isolation from the rest of the operating system, but without introducing significant overhead (e.g. through emulation of hardware or by running an extra operating system instance on a virtual machine). Docker containers run directly inside the operating system of the host, making them dependent on the API/system-calls of the operating systems kernel and its ability to isolate processes from each other. Docker itself originally supported containerization for GNU/Linux only, using the cgroups and namespaces features offered by the GNU/Linux kernel [BN06] [Men07], alongside with separate stacked file-system images containing the isolated software with libraries. Although Docker recently supported native containerization for Microsoft Windows operating systems, running containers based on different kernel technology is not possible, so virtualization is required for interoperable container execution.

To cater the diverse requirements of involved software components in openEASE , several containers need to be provisioned. Through defined shared folders (called volumes in the context of Docker) and inter-container networks, these containers are orchestrated to cooperate in serving the necessary functionality of openEASE

- *webrob*: Contains the web interface of openEASE written with the Python based web-framework flask[5]. It comprises of both the core KnowRob query client and visualization interface and the administrative and service operation management functions, such as the user management and login feature, and administrative access to experience data synchronization (currently only from IAI) and content editing. Furthermore, it offers a video recording service for the visualization of knowledge processing results, an editor for user-editable Prolog programs and knowledge bases (ontologies), and a log viewer for the session-associated KnowRob instance. Supporting the use of openEASE in teaching courses, a tutorial and an option to add course material and exercises are also built-in.
- *postgres_db*: Hosts a PostgreSQL relational database[6]. Manages user registration information and openEASE tutorial and course information, which can be added and edited dynamically through the administrative interface from the web interface.
- *knowrob*: Spawned for each user visiting the openEASE website, either anonymously without data persistence, or with persistent user data for registered users. The container-image holds a fully functional ROS and KnowRob installation. Upon start, a ROS master in-

[5] *http://flask.pocoo.org/*
[6] *https://www.postgresql.org/*

stance, a registry service used for tracking and advertising other active ROS components (called ROS nodes), is launched. Alongside with it, an instance of a Prolog interpreter capable of being queried in the JSON format is executed and initialized with KnowRob Prolog software, as well as openEASE specific Prolog add-ons. For enabling access of JavaScript clients through a WebSocket connection, an instance of rosbridge_server accompanied with a rosauth authentication system (both originating from the aforementioned RobotWebTools stack[7]) is launched finally.

- *mongo_db*: Holds all episodic memory data integrated into openEASE inside a MongoDB[8] database. Both the *webrob* container (for administrative maintenance) and all user-individual *knowrob* containers (to provide the reasoning service) have access to this database.
- *nginx*: Hosts an instance of the nginx[9] web-server/load-balancer. It is the only container allowing incoming requests to openEASE , enabling easy central Transport Layer Security (TLS)-deployment for securing connections and operates as a central communication hub to both the web-interface and the KnowRob knowledge processing containers in openEASE.
- *dockerbridge*: A Remote Procedure Call (RPC) service written in Python to control and interact with *knowrob* user-containers. It provides means to create, start, stop and remove user-containers through the host Docker API. Data-transfers from and to containers can be initiated for controlled bypass of the container isolation.
- *docker-gen*: Monitors the Docker management system on the host for updates in the container constellation and updates the route configuration inside the *nginx* container when necessary[10]. This is required to provide access to dynamically spawned containers, such as the user-individual *knowrob* containers, or to remove access to terminated containers.

Missing in the list of containers are the auxiliary containers created for data storage. Docker containers are designed to be short-lived, updating a container is neither directly supported nor recommended. Instead, the container is deleted and re-created. Persistent data is handled through the concept of volumes. Volumes are directories managed separately from containers, that can be linked into container instances and survives their re-creation. Historically, volumes were tied to a specific container itself, which required the concept of long-lived data containers: Containers whose sole purpose is to store data in a directory, which they exposed to other short-lived application containers. Starting with Docker v17.06, volumes are not tied to a container anymore[11], eliminating the deviation from the short-lived container paradigm.

For a complete analysis of openEASE , knowledge about the location of associated source code and used data, along with information about the build process of openEASE and its components are of relevance, and is the subject of the following sub-sections.

[7] *https://github.com/RobotWebTools*
[8] *https://www.mongodb.com*
[9] *https://www.nginx.com/*
[10] *https://github.com/jwilder/docker-gen*
[11] *https://docs.docker.com/v17.06/engine/admin/volumes/volumes/*

3.2.1 Source code and data

openEASE requires a wide selection of data both during build and operation.

The main source and entry point for openEASE is the *knowrob/docker*[12] git repository. It should be noted that this repository is not the official source code repository for openEASE but a copy of the main developer to account for the latest changes in openEASE , which are not yet published to the official repository. It contains the latest source code of the *webrob* and *dockerbridge* containers in the sub-directories `flask` and `dockerbridge`. Furthermore, build descriptions for all Docker containers either containing software directly developed by the IAI or for third-party software requiring amendments to the official released form. This includes the KnowRob knowledge processing implementation with separate build descriptions for each version corresponding to the ROS distribution it was developed against (`hydro`, `indigo`, `kinetic` subdirectories).

Looking at the dependencies of the openEASE components, the KnowRob software has the highest amount of external dependencies from diverse sources during the build process. Based on a bare Ubuntu 16.04 GNU/Linux image, a KnowRob installation suitable for the use in openEASE additionally needs 1381 software packages from Ubuntu and ROS APT servers, 121 Java library files from the Maven repositories JCenter and Spring and the download of 8 git repositories when compiling from source. The build-time dependencies of the *webrob* container is mostly limited to those of the Python interpreter and a Node.js environment, both of which are satisfied with 305 software packages from the Ubuntu APT server in total, as well as the installation of 38 Python dependencies from pip and 216 Node.js modules from npm. Other containers required well below 100 packages from APT servers and no other dependencies.

As for the runtime dependencies of openEASE , there are two very large datasets required for operation. One being the 3D robot model and texture data hosted at two git repositories and one svn repository, totaling in 1.4 Gb of data. The other one being the previously recorded episodic memory and robotic experience data hosted on an File Transfer Protocol (FTP) server, totaling in 39 Gb of data.

The investigated dependencies, required datasets and their sources have to be carefully considered when analyzing and optimizing openEASE for software reproducibility, because it has to be ensured that all data is still accessible at the time of reproduction. For an overview of the used sources by openEASE, refer to the table 3.1 on page 22.

[12] *https://github.com/daniel86/docker*

Type	**Name**	**Content description**
G	docker[13]	Source code of *webrob* and *dockerbridge*, Docker build description and auxiliary scripts
G	knowrob[14]	Source code of KnowRob
G	knowrob_addons[15]	Additional packages for KnowRob not required for standard operation (e.g. openEASE extensions)
G	iai_maps[16]	KnowRob dependency
G	iai_common_msgs[17]	KnowRob dependency
G	iai_cad_tools[18]	KnowRob dependency
G	mjpeg_server[19]	openEASE dependency
G	tf2_web_republisher[20]	openEASE dependency
G	knowrob_robcog[21]	KnowRob source package
G	ros3djs[22]	JavaScript 3D rendering library for ROS
A	Ubuntu APT	Basic software dependencies, required by all containers
A	ROS APT[23]	Dependencies for the ROS environment
M	JCenter[24]	Dependencies for ROS-Java packages
M	Spring[25]	Dependencies for ROS-Java packages
-	npm	Node.js module dependencies
-	pip	Python dependencies
G	iai_robots[26]	Robot 3D model data

[13] *https://github.com/daniel86/docker*
[14] *https://github.com/knowrob/knowrob*
[15] *https://github.com/knowrob/knowrob_addons*
[16] *https://github.com/code-iai/iai_maps*
[17] *https://github.com/code-iai/iai_common_msgs*
[18] *https://github.com/code-iai/iai_cad_tools*
[19] *https://github.com/RobotWebTools/mjpeg_server*
[20] *https://github.com/RobotWebTools/tf2_web_republisher*
[21] *https://github.com/andreihaidu/knowrob_robcog*
[22] *https://github.com/RobotWebTools/ros3djs*
[23] *http://packages.ros.org*
[24] *https://bintray.com/bintray/jcenter*
[25] *https://repo.spring.io*
[26] *https://github.com/code-iai/iai_robots*

Type	Name	Content description
G	pr2_common[27]	Robot 3D model data
S	cad_models[28]	Robot 3D model data
F	episodes[29]	Episodic memory

Table 3.1 Data sources used during openEASE build, deployment and operation

Type legend: G = git, A = APT, M = Maven, S = svn, F = FTP

3.2.2 Build process

For software reproducibility in research, it is important to have either long-term available archived copies of the exact executable software versions used for the research, or to be able to build the software from its source code and produce executables which behave identical to the research version.

openEASE consists of a collection of software modules packaged in containers, this means that either the full images of all custom built containers for all released versions of openEASE have to be archived, or it must be possible to repeat the construction of the container images. The Docker software contains a container image download and upload mechanism to a service called called 'registry'. The company behind Docker, Docker Inc. offers the platform DockerHub[30], being such a registry, for sharing Docker images at no cost for open source projects. They impose no limitations on how many images can be uploaded per user, how many versions of the same container are uploaded, how large the uploaded images are and how long the images are preserved. Actually, this publishing method has been used previously for openEASE, there exists an account on DockerHub on which required images were published[31]. Moreover, a script file in the *knowrob/docker* main git repository exists to download images from this exact account. The significance and acceptance of this method is shown by inspecting last modification date of the images in that account: The last update was uploaded more than two years ago.

Apart from the infrequent update on the DockerHub account, and despite the seemingly very suitable conditions for archiving openEASE containers, other technical issues could advise against this solution. First of all, the image sizes of the openEASE related containers are very large

[27] *https://github.com/daniel86/pr2_common*
[28] *http://svn.ai.uni-bremen.de/svn/cad_models*
[29] *ftp://open-ease-stor.informatik.uni-bremen.de*
[30] *https://hub.docker.com/*
[31] *https://hub.docker.com/u/knowrob/*

to download, amounting to 4.62 Gb for the *knowrob* container, 453 Mb for the *dockerbridge* container and 1.04 Gb for the *webrob* container. Due to the way the containers are constructed, the data would have to be downloaded again for the smallest amendment to the dependencies, which might be feasible for fast Internet connections available in research institutions, but not for a standard household connection, taking around three quarter of an hour to download for a 15.3 Mbits connection, the average Internet connection speed for Germany in 2017 [Tec17]. Another issue arising from this solution is the decreased flexibility in assigning versions for openEASE . As openEASE continues to be actively developed with new functions added regularly, and has been used and continues to be used as a tool in teaching, it is desirable to build individual versions running in parallel (e.g. using a new version with added reasoning capabilities for newly captured experience data). A host system running openEASE would have to store the full size of all currently selectable container images, even if differ minimally. To explain this limitation, it is necessary to understand how Docker normally handles the build process of containers and how openEASE uses it.

The build process for a Docker container is described in a file called 'Dockerfile'. A Dockerfile describes the content of an image, by declaring commands to be executed sequentially. A typical command to be executed in a Dockerfile is the installation of a software package with a packet manager or the invocation of a build process for a software. It is also possible to add files and folders from the machine building the image. The modification each addition or command execution does to the image is saved as an intermediate image, with only the added and changed files being stored physically (copy-on-write). Thus, a container image is really a stack of intermediate images, whose cumulated changes result in the file system content of the final image. Dockerfiles can refer to existing Docker images and start executing its commands with the file system state of the referenced image. This allows to re-use images and start the build of a container with a baseline (e.g. commonly used libraries), which can either be built beforehand on the same machine, or be downloaded from a Docker registry. Furthermore, a Docker image referenced in the same version in multiple Dockerfiles only has to be downloaded and stored once, which should prevent the repeated download of complete images on minor changes to a container, as criticized earlier.

The special use of build files for the openEASE containers however prevents this optimization, as its Dockerfiles contain commands changing large portions of the image at once. The installation of dependencies for example, is done in a single command. The reason for this design decision is the reduction of overhead and improved container performance. Docker recommends not to use too many image layers (it had a limitation of 42 layers when openEASE development started), because the excessive stacking of images would degrade performance. If the installation of dependencies were done for each package individually, 20 command executions would be necessary instead of one. Also the installation of packages through the package manager fills up cache directories in the file system of the created image, so it is considered best practice

to update the package manager, install all required packages and clear the cache directory in the same Dockerfile command. Another factor that prevents the optimization is the size of KnowRob. The delta size of the intermediate image where just the source code is downloaded amounts to 450 Mb at the time of writing, the delta size of intermediate image directly after invoking the build process of KnowRob amounts to 890 Mb.

Being able to preserve just the dependencies of each container for openEASE should be evaluated, as the archival of all dependencies needed to build the openEASE containers would consume just under one gigabyte of storage space, with changed or added dependencies only adding their actual size to the consumed space. If this option proves to be unpractical or unreliable, the archival of full Docker images could still be an option, because with regard to reproducibility, inconvenience is trumped by the possible inability to reconstruct an earlier state of a software. One argument in favor of full image archival is the fact that the build processes of both KnowRob and the *webrob* container with the web interface of openEASE have references to bleeding edge dependencies, i.e. always targeting and requesting the latest change of the source code repository of a dependency. When dependencies are archived individually, it must be ensured that the exact same change is used in the repeated build process when the software is rebuilt.

Continuing with an overview of the used technology for building containers openEASE comprises of, there are many build systems and technologies involved and to consider for the reproducibility analysis. KnowRob, being placed in the ROS ecosystem, comprises of many individual software components called packages. ROS uses the build system *catkin*, which supports dependency resolution in the realm of ROS packages (i.e. it is able to resolve dependencies to other software used and published in the ROS ecosystem) and building ROS packages written in C++, Lisp, Python or Java. The way the KnowRob Dockerfile is written, no dependencies are actually necessary to resolve using *catkin* directly. However, for building ROS packages written in Java, *catkin* invokes the Java build system Gradle, which uses the Maven repositories referenced in chapter 3.2.1 on page 20 for dependency resolution.

The container with the web interface of openEASE , *webrob*, is made of a Python web application serving both dynamic content for user/session management, management of container instances and for dynamically reading from and writing to database backed data, as well as static content, made out of the JavaScript libraries necessary for communication with the KnowRob containers and the GUI and visualization portions, and also images, Cascading Style Sheets (CSS) and web fonts. For the resolution of Python dependencies, the package manager pip is used, which is backed by the Python Package Index (PyPI). Currently, dependencies of the JavaScript components are resolved through the npm package manager. As the dependencies resolved through that manager are designed to run with Node.js in a standalone environment, they are converted to be used within a browser by the Node.js module *browserify*[32]. The package

[32] *http://browserify.org/*

manager npm is backed by the npm-registry.

Other containers built by openEASE require significantly less (< 100) dependencies from the Ubuntu APT and pip package managers (*dockerbridge*), or no dependencies at all (data containers, *postgres_db*).

All Docker containers which are no data containers reference fixed image versions available from DockerHub or other container images built in the same openEASE build process, so there is no risk in having suddenly non-functional containers due to an image update.

The build of openEASE is invoked through the `build` shell script in the `scripts` directory of the main *knowrob/docker* git repository. It allows the selection of container images to be built (all, or any combination of *knowrob*, *dockerbridge* and *webrob* containers) and offers options to both disable the Docker build cache and to replace the APT and Maven repository sources used during the container builds with caching services. The first option is suitable when Docker asserts that the outcome of a command execution is unchanged and skips its execution by re-using an intermediate image gained from previous build executions. When the use of an updated package from APT is desired, this option can force Docker to execute the command regardless of the cache status. The second option, the injection of caching services, helps accelerate the container build by preventing the repeated download of the same packages. The services are provided by two Docker containers, running a *apt-cacher-ng*[33] service and a *nexus*[34] repository manager.

3.3 System review

With information about the architecture of openEASE , the environment openEASE is built in and which sources it uses during build and normal operation, all prerequisites are met to begin the in-depth analysis of openEASE. To reiterate, the goal of this analysis is to assess the possibility of reproducing previously obtained results of openEASE at a future date. Issues hindering or preventing this shall be addressed, to be able to propose a solution in the following chapters. The build and setup procedure of openEASE, its software architecture and its source code will be considered in the analysis. Note that only openEASE itself is considered in this analysis, the source code of software and containers only built and/or used within openEASE (e.g. KnowRob) is not considered here and is assumed to be an immutable dependency. KnowRob in particular could be exempted from this supposition, given that it is developed in the same institute as openEASE, however KnowRob is not used exclusively for openEASE, thus any possible issues might not be directly amendable just to fix reproducibility of openEASE. Another

[33] *https://wiki.debian.org/AptCacherNg*
[34] *https://www.sonatype.com/nexus-repository-sonatype*

aspect is that the analysis and later amendment of KnowRob would increase the effort of this work beyond its reasonable limit.

Starting with the setup procedure of openEASE, by enumerating the requirements and steps to be executed to operate an instance of openEASE with a given target version (e.g. a version used by a researcher in a publication), an analysis baseline can be established. Any issues found during this process will immediately and directly prevent reproduction of research results, as the openEASE system hosted by the IAI does not offer a way to execute experience data analysis with an earlier software version at the time of writing.

The first positive results of the analysis are the presence of a setup documentation[35], which describes the setup procedure of some of the openEASE dependencies (web-browser, Docker). As outlined in the chapter 2 on page 3, documentation is one of the most important elements in achieving research reproducibility. Besides the written documentation, the usage of Docker itself supports achieving software reproducibility, as the Dockerfile build descriptions of containers essentially are part of the system documentation. They describe step by step the construction of the target system, starting from a common and often publicly available baseline (e.g. a public base image from DockerHub). The use of Docker for software reproducibility is being discussed and recommended by other authors, as well [Boe15] [CFG16].

Following the mentioned installation documentation, the reader is advised to execute the script downloading the outdated Docker images from DockerHub (mentioned in chapter 3.2.2 on page 22). This step reveals two major issues: The first being the outdated documentation or the outdated Docker images (depending on whether the DockerHub image download method shall be retained to obtain a version of openEASE). The second issue is revealed when closely inspecting the DockerHub images. It can be seen that only one version is published: 'latest'. Further investigation of the `scripts` directory inside the main openEASE repository shows the script responsible for publishing versions to DockerHub, and that the version used to publish images is always 'latest'. The openEASE repository itself also does not contain any reference to explicitly defined or released versions: No git tags are present, the development happens on the 'master' branch (except for some branches containing modifications for a few events where a modified openEASE instance was used, `robohow` and `fallschool`) and the source code inside the repository also contains no references to a central openEASE version. Consequently, the containers whose source code is hosted in the main openEASE repository are not versioned, affecting both the *webrob* and *dockerbridge* container.

The lack of a versioning continues throughout the software built in openEASE: KnowRob, which at least has branches in its git repository for every ROS distribution it was released for, does not receive a version during the build of its container. As for the required data to operate openEASE (Episodic memory and robot model data for 3D robot visualization), looking back at section 3.2.1 on page 20 shows an FTP server as the source for the episodic memory, offering

[35] *http://www.knowrob.org/doc/docker*, accessed on 23th January 2019

no support for versioning or for retrieving deleted or changed versions of the episodic memories at all. In the previous chapter, it was established that in research, software used together with data must be present at exactly the same combination they were had during research for them to be reproducible.

Generally, through the usage of version control system (vcs)s, such as git or svn, a mechanism for automatic versioning is already present: Simply taking the revision or commit ID as version is sufficient to restore the exact state of this version through the use of the check-out mechanism of the version control system. In the case of openEASE though, because there are so many repositories of different vcs involved, which are referenced in multiple Dockerfiles and npm package dependency declarations, the restoration of an old version of openEASE (knowing all revision IDs and commit IDs of the openEASE installation used in research) is infeasible and only achievable by modifying said references to refer to the correct IDs.

Because continuing to follow the documentation will not assist the analysis further due to the missing selection of the target version, the build process of openEASE is examined next. Building openEASE from source to reproduce a published research experiment is the only viable option currently, despite being inconvenient due to the lack of documentation. The missing versioning on the episodic memory data can still cause issues, but it seems more likely that the software is changed due to continuous development, rather than that the episodic memory data is changed after recording.

The inspection the build files revealed another issue linked to versioning, which is the reference of bleeding-edge software dependencies in the Dockerfile and the npm package dependencies. The build files of the container images for *webrob*, *knowrob* and *dockerbridge* have in common, that they install software packages from APT repositories as one of their first build step. While maintainers of the Ubuntu GNU/Linux APT repositories generally are very careful with updating software versions, only fixing software bugs or security issues and not add functionality breaking older functions, it can happen there as well. The ROS APT repository, although requiring extensive testing before a release, can be populated with packages from any developer targeting the ROS platform[36] and might be more likely to have breaking features due to package updates. The container image builds continue with instances of this issue, referencing development branches of software dependencies' git repositories and unpinned versions in pip and npm package dependency declarations. And while the maintainers of APT repository generally ensure that no package is deleted from the repository, an owner of a dependencies' source code repository may rename, restructure or delete it at any time. The lack of version pinning does not only theoretically cause problems, openEASE already experienced issues linked to that cause before [Beß16] [Bal19]. The first experienced issue is particularly interesting, because the fix for the issue was to pin the version of the affected library being incompatible, but the version has not changed for over two years. While not directly affecting reproducibility of the software, the use

[36] *http://wiki.ros.org/ROS/ReleasingAPackage*, accessed on 23th January 2019

of very old software libraries together with other, bleeding-edge libraries indicates a quality issue and might cause problems when developing new features or when new versions of the library are released with security fixes.

Other issues with the build process of openEASE were not encountered, by using the provided `build` script it was possible to build the latest version of openEASE with basically just having Docker as a required dependency.

Regarding the architecture of openEASE, the use of Docker containers providing the functionality of very heterogeneous software while isolating them from each other and the host is a good measure to assist in achieving reproducibility. But when the purposes of the single containers, or rather the contained software is inspected closely, a sincere issue emerges: Due to the fact that openEASE is both a research tool providing services to other researchers over a long time period, and contains heavily developed and changed research software itself (KnowRob), it must be ensured that the part of openEASE offering the service functionality is strictly separated from the actively developed and regularly changed KnowRob part. As mentioned earlier, the *knowrob* container isolates the KnowRob software itself from the rest of the system, requiring just a network connection to communicate with it, so the KnowRob part presents no immediate architectural reproducibility issue.

The *webrob* container nonetheless, is affected, as it provides both management functionality for the operation of openEASE as a service, and hosts the client web interface communicating directly with KnowRob, and thus is tightly coupled with the API of the KnowRob ROS-services and the format of ROS-messages transmitted from and to the *knowrob* container. This architectural design prevents an independent operation of the service functionality of openEASE and the client interface, imposing consequences both for the reproducibility of the software and for continuous development of openEASE. Moreover, if the KnowRob code is updated causing an incompatibility with the current client code in the *webrob* container, no entanglement exists which binds the client code to a specific KnowRob revision. The independent operation of the KnowRob web client and visualization modules would allow having multiple versions of such client being active on the same openEASE host system together with their respective KnowRob version (it is already separate and user-individual in the *knowrob* container anyway) and episodic memory data. If openEASE would allow the user to select the KnowRob and web client version combination, and every version combination ever used is preserved and runnable on demand, it would shift the effort of having to install a historic version of openEASE (possibly from source code) from the individual to the provided instance of openEASE itself.

Another issue in the architecture which could affect the endeavor of reproducibility is the lack of data management for user-provided data. Users can use the editor functionality to contribute ontology and Prolog data to their KnowRob instance. With no management features for this data, the only facility to keep track of the data is the Docker software stack knowing about the users' data container stored on the host running openEASE. Because the paradigm of a

container is to be short-lived, there is no special protection against accidental deletion by the Docker command line interface. Regardless of whether this is a theoretical issue (because it never occurred so far on any productive openEASE systems to the knowledge of the author), when it happens, it will affect reproducibility, because users might use their storage for their research and refer to this data in their publications.

The analysis of the source code of openEASE shows one issue interfering with software reproducibility: On startup of openEASE, the mesh and texture data for robot model visualization is downloaded using the value of an environment variable provided by the user. While the variable is documented in the documentation mentioned in the beginning of this section, it is data belonging to the episodic memory data and the KnowRob version, as those data determine when and which robot shall be displayed. When choosing to use the Docker complete image archival method with DockerHub, the mesh data is missing from the image and could prevent reproduction partially, as robot models might not render correctly in visualization reproduction scenarios.

Is worth noting that some functional separation issues and quality issues in general can be also found on the source code level. An example is the duplication of large chunks of code, like in the directory `flask/webrob/templates` [Boz18]. The html files `knowrob_simple.html`, `knowrob_teaching.html` and `knowrob_tutorial.html` share almost identical script dependency declarations, and some identical JavaScript method declarations. Such quality issues are a sign of increasing complexity [Leh80] or no sufficient design phase before implementation [Bro+98] and could suggest a re-engineering of openEASE [Mül97]. This, however, does not affect reproducibility directly, because even though future development might be hindered by such issues, the reconstruction of current and previous states of the software are not affected.

Based on the findings about openEASE summed up in the table 3.2 on page 30, a strategy needs to be found on how to solve the most grave pitfalls in reproducing results and software development states and how to mitigate the others as best as possible. A suggestion for such a strategy is developed in the following chapter.

No.	Description
1.	Outdated documentation on how to set up openEASE
2.	Missing versioning of openEASE
3.	Missing versioning of dependencies
4.	Missing versioning of required data (episodic memory/3D meshes and textures)
5.	Reference of bleeding-edge software dependencies (latest change in a repository, or 'HEAD', unpinned versions)
6.	KnowRob web client and visualization code is not separated from the openEASE service implementation
7.	KnowRob web client and visualization code, KnowRob and the required data are not entangled
8.	No management features for user-provided data
9.	Mesh data is downloaded during runtime, not during build

Table 3.2 Issues interfering or preventing software reproducibility in openEASE

Chapter 4

Achieving software reproducibility in openEASE

In the previous chapter, several issues were found in openEASE which need to be addressed to make it a reliable and long-term dependable tool for fostering research in the field of robotics and knowledge processing. In this chapter, a strategy to resolve these issues is developed.

The findings in table 3.2 on page 30 lead to both obvious and debatable solutions. In the following section, a solution to issue 6 from the referenced table is proposed and its implementation is documented afterwards. It is the first prerequisite for having a reproducible software, as a tight coupling between a software component that is changed often due to active development and might break sometimes, and a software component required to work reliably is not compatible.

4.1 Combined KnowRob client and openEASE management

For the issue of having no separation between the client web interface to the research software KnowRob and the rest of openEASE , an obvious solution is to separate both parts from each other. At first, the present structure of the web interface implementation is explained, followed by the description of the separation of said parts and the moving of the KnowRob web-client code with its visualization and GUI libraries into an own Docker container. The analysis of the current state and the modification executed was done based on the commit `b52b1536c80518ad948a23735a92ed2dd50a52ae` in the git repository attached to this thesis as `openease.zip`. All paths and files referenced in this section refer to this repository as well.

4.1.1 Current state

The files related to the openEASE web-interface can be found in the `flask/webrob`. Three folders inside that directories are relevant to the examination: The `pages` directory containing Python files using the web-framework flask to serve web pages dynamically, the `templates` folder containing the HTML code for the dynamic (and static) web-pages with placeholders to be replaced by flask and the `static` folder containing JavaScript libraries, CSS files and images.

When a user opens the openEASE web-page, he is directly redirected to the GUI from which he can select episodic memories to load, an action the user is always prompted for when he first visits the site. Apart from opening the user login and registration page, the user has no option other than selecting an episodic memory (figure 4.1 on page 32). After the user selects the desired episode, a KnowRob user-container is launched and the web-interface waits until this container is fully loaded, polling the state of the container in a fixed interval. An overlay displaying a progress circle prevents any user interaction during this time.

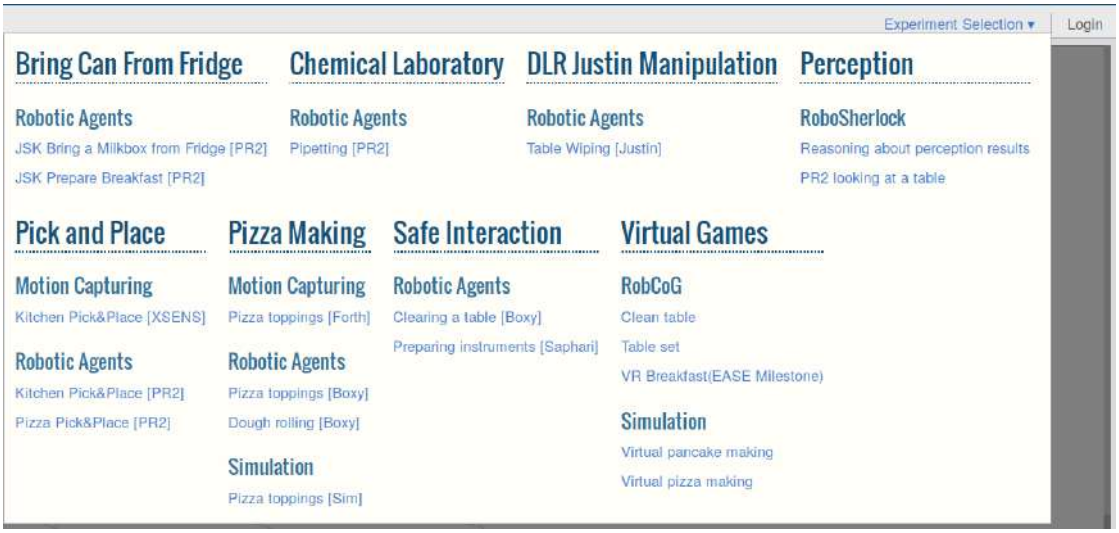

Figure 4.1 Right menu of openEASE with episodic memory selection

When the container is completely started, meaning the ROS system with the KnowRob software and the WebSocket transport are usable, the overlay disappears, and the full GUI is displayed to the user. The user then has the ability to either use the query console or the predefined query list to query the KnowRob knowledge base and visualizing the episodic memory data, or to open one of the following views: A episode replay page capable of producing videos from episodic memory visualization, a tutorial page teaching the basic usage of openEASE with a live walk-through of the Prolog query and visualization capabilities, an editor for adding user-supplied content and a log view showing the log messages of the user-assigned KnowRob container (figure 4.2 on page 33). All views selectable are pre-loaded in the background inside an invisible frame, and

are brought to the front upon selection, allowing fast switching without load times.

Figure 4.2 Left menu of openEASE with different selectable views

The user can also select another episode from the right menu, which terminates and removes the currently running *knowrob* container and launches a fresh one, because the existing container might have been tainted by previous user queries and it is non-trivial to unload experience data from a running KnowRob instance.

All enumerated actions require access to the JavaScript KnowRob client instance and the visualization libraries and auxiliary scripts - other interactions with openEASE like the display of a user details page or administrative pages when logged in as administrator do not involve the use of the KnowRob client code or visualization.

Unfortunately, the code for initialization and display of the KnowRob client and visualization is mixed with all of the other code responsible for the rest of openEASE functions. Moreover, the KnowRob client initialization code residing in the `templates/main.html` file and some of the JavaScript libraries depend on having user information and information on how to authenticate against the *knowrob* container injected through the template system of flask. In return, some of the JavaScript code responsible for managing the frames for faster view-switching is found inside the library managing the KnowRob client connection.

4.1.2 Separation into openEASE and openEASE-webclient

The resolution for these findings is to establish a clear separation between web-pages and code responsible for displaying the outer GUI, frames and menu items, and the code responsible for communication with KnowRob and the visualization. Challenging is to handle the various callback notifications necessary in this system: The aforementioned overlay for example must be hidden when the KnowRob client is ready to use. Also, the client needs to retrieve the information on which Uniform Resource Locator (URL) the *knowrob* container is reachable and which authentication data to use. To implement a clean, low-complexity and future-proof solution, the separated KnowRob related code should be placed inside its own container. For clear reference, when referring to the 'webclient' from now on, both the client KnowRob and ROS libraries and

the HTML and JavaScript parts responsible for displaying the visualization and Prolog console options are meant. 'openEASE web interface' refers to the old, now client-independent, web pages of openEASE . The webclient container should be completely static (i.e. no Python or other framework should be required during operation) and be served through a simple nginx web-server instance.

All amendments described in this subsection can be viewed in the file tree of the commit `9a933448e8c66410bc1ad0100c391f8486f6258a` in the git repository attached to this thesis as `openease.zip`.

As first refactoring step, all static content clearly identifiable as being part of either the KnowRob and ROS client libraries, or the query console, visualization and 3D navigation code, was moved to the directory `openease-webclient`. A directory structure similar to the originating `flask/webrob` is established inside: `openease-webclient/static` for JavaScript libraries, images and CSS files, and `openease-webclient/html` for HTML files.

The dependency resolution for JavaScript, which was previously handled by one npm `package.json` file in the `flask/webrob/static` folder, was split into dependencies required by the webclient in the `openease-webclient/package.json` file, and the remaining dependencies which are left in the original `flask/webrob/static/package.json` file. Also, an own Dockerfile was added to the webclient directory, which invokes the npm dependency resolution and browserify conversion, as well as a static templating tool selected for this task to minimize duplicate HTML code: jekyll[1]. Jekyll allows the generation of static HTML pages from templates and is the template engine behind the Github pages[2] feature.

With these preconditions, a lightweight container serving static content can be built and launched. To reach the nginx web-server, the webclient container had to be added to the main *nginx* container configuration of openEASE, so that it is made available under the `https://openease-host/webclient/$WEBCLIENT_NAME` URL, with $WEBCLIENT_NAME being the name of the webclient container. This allows the parallel operation of multiple webclient instances, for when the startup of an older *knowrob* container is requested. Additionally, the latest webclient version is built and launched per default, using the name 'default'.

Remaining for this restructuring is the separation of JavaScript code and the adjustment of the existing HTML files the client GUI comprises of to work with jekyll.

The openEASE web interface was stripped of any code specific to tasks to be executed by the webclient. It now handles the management and handling of the frame switching previously located in the KnowRob webclient library and the menu content control. For the frame switching control, a new JavaScript library was created `flask/webrob/static/lib/framecontrol.js`, containing old frame control functions now stripped from the webclient library. The web inter-

[1] *https://jekyllrb.com/*
[2] *https://pages.github.com/*

face was redesigned in such a way that the webclient is loaded into a single frame, in which the KnowRob client instance is initialized. Inside this frame, the frame control library creates the view frames previously located directly on the openEASE web interface pages. The available view frames a webclient provides is now declared via a JSON file placed in the root directory of every webclient instance: `openease-webclient/html/webclient-description.json`. When loading the webclient, the openEASE web interface queries this file to determine the content of the menu and the view frames to spawn. The library responsible for loading the webclient is the newly created `flask/webrob/static/lib/controller.js`, along with the initialization code in `flask/webrob/templates/main.html`. These files gather necessary information for the webclient to connect to the *knowrob* container instance, set up the surrounding GUI elements and load the selected webclient instance.

The webclient itself now needs to obtain initialization information from the openEASE web interface, as it does not have access to such information anymore as a static web page. For that matter, a new HTML page is created in the webclient, which is loaded into the frame controlled by the openEASE web interface (`openease-webclient/html/client_frame.html`). The new page then accesses the controller instance of the parent web page to retrieve the KnowRob connection and authentication information, gain access to some GUI controls such as the overlay and initializes the KnowRob web client. The inter-frame access is possible through the Same-Origin-Policy modern browsers employ, which grants frame-embedded pages access to the parent page and its JavaScript context when both the frame and its parent originate from the same server.

The KnowRob client JavaScript library in the webclient (`openease-webclient/static/lib/knowrobClient.js`) was rewritten in a way, that where it directly tried to access resources now located in the openEASE web interface, it would use the newly introduced controller classes. The required connection and authentication information are loaded into the client library from the `client_frame.html` page. All references to the client library, mostly present in the `knowrob_*.html` main webclient pages in the `openease-webclient/html` directory, were updated to obtain their controller instance through the `client_frame.html` parent frame, more specifically the new client controller `openease-webclient/static/lib/knowrobController.js`. It keeps track about the state of the client when it is initialized, and fires a callback on initialization, ensuring all webclient pages retrieve only active and initialized KnowRob client instances.

The editor page in this scenario shares both domains of the webclient and the openEASE web interface, making the decision where to put it challenging. As it requires both dynamic access to the user-data container and thus requiring a dynamic web-framework such as flask, but also requires access to the KnowRob client to propagate changes to edited packets back to the running *knowrob* container, it was decided to leave it in the openEASE web interface (`flask/webrob/templates/editor.html`). The propagation of package changes was solved with a special callback, which is registered by the webclient on initialization. The controller of openEASE web interface invokes this callback, accessing the `knowrobController.js` on the

webclient frame. On another occasion, the changing of the currently active episode, the controller is also accessed by the openEASE web interface to signal the client the change of the episode.

While this creates a reference to the webclient requiring knowledge about the API of the controller inside, it is limited to two functions, which is tolerable (see lines 60-70 in `flask/webrob/static/lib/controller.js`).

The main webclient interface pages, namely `knowrob_simple.html`, `knowrob_tutorial.html` and `knowrob_teaching.html` pages for displaying the visualization, query console and library, statistics and video GUI, as well as the `video.html` episode replay and video generation page were amended to fit the jekyll template language. Common JavaScript dependencies were externalized into `openease-webclient/html/_includes/knowrob_base.html`, and common JavaScript initialization code required by most of the interface pages was moved to `openease-webclient/html/_includes/knowrob_commonjs.html`, fixing most of the code-duplication encountered in the HTML code.

With these changes, the webclient for KnowRob is completely separated from the *webrob* container, now operating as *openease-webclient* container. As a bonus, many issues in the amended JavaScript libraries being caused by a race condition (like the KnowRob client being loaded to early or too late, the view frames not being completely loaded etc.) are solved through the introduction of the aforementioned various callback facilities.

4.2 Versioning and dependency management

One of the major problems being present in the openEASE software project is the lack of versioning in individual components, and the whole system as well. It is therefore the next logical step to evaluate possible solutions how to support adding versions to openEASE, including its dependencies, and the required datasets, as well. A solution for this problem could solve issues 2, 3, 4 and 7 from table 3.2 on page 30, adding not only versions to every dependency currently un-versioned, but would also enable the coupling of the KnowRob version to the newly created *openease-webclient* container introduced in the previous section and the dataset of episodic memories.

Furthermore, if the solution could also assist in pinning bleeding-edge software dependencies, issue 5 from table 3.2 on page 30 would at least be mitigated.

Remembering the requirements established in chapters 2 on page 3 and 3 on page 15, this solution is required to gain the following features:

- The coupling of KnowRob, the *openease-webclient* interface and the associated episodic memory data.
- Assigning stable versions of the above 3-tuple combination to existing experiment data, allowing experiments to be migrated to new KnowRob developments gradually, not all at once requiring immense compatibility testing effort or the risk of breaking a previously working experiment.
- Having the freedom of not migrating old experiments to work with a new KnowRob version, because it is unmaintained.
- Having the freedom of using bleeding-edge software dependencies (a feature which is desired in actively developed and constantly evolving research software) AND long-term stability of release iterations.
- Providing the option to load older versions of the above 3-tuple software combination for reproducing (historic) analysis and visualization results.

Firstly, the option discussed in 3.3 on page 25 to archive Docker images shall be reiterated. When compiling a Docker image of a container, all dependencies in their current state are fixed inside the image. No artificial version number would have to be introduced for currently un-pinned dependency versions and bleeding-edge development versions. The only issue with this solution is the omission of the episodic memory, which totals to over 39 Gb of data. A Docker Inc. (the operators of DockerHub) employee states, that image sizes are not limited, but problems might be encountered with image sizes above 10 or 20 Gb[3]. Thus, the upload of the episodic memories to DockerHub must be tested before employing this solution, or it has to be split to multiple image to reduce the maximum single image size.

The scripts to upload the Docker images to DockerHub would have to be amended to pass a unique version number instead of 'latest' to prevent overwriting the previous image versions. openEASE then could be changed to remember previously built versions of its container images, load these versioned images of KnowRob , the client libraries and the related episode files from DockerHub on demand and every release would be totally reproducible, given that the last issue 9 of the mesh data being downloaded at runtime is also solved by uploading it to DockerHub in an own container.

This option is inconvenient, as it requires large amounts of storage space both on DockerHub and on a host system running openEASE (Docker caches images downloaded from DockerHub). With the *knowrob* container image being 4.6 Gb large at the moment of writing, the storage requirements could grow dramatically, especially when many combinations of openEASE are released in the future. Also, the episodic data needs to be imported into a MongoDB database before it is accessible by KnowRob , effectively doubling the required amount of data to store (image with the raw episodic memory data, and the containers holding the MongoDB data). Additionally, as mentioned in the previous chapter, the option of downloading large images is

[3] *https://forums.docker.com/t/does-docker-hub-have-a-size-limitation-on-repos-or-images/10154*

not optimal for individuals with slow, household Internet connections.

Therefore, another solution is proposed and shall be implemented: Rather than archiving complete Docker images, the archival of each dependency individually should be attempted. This would keep the required amount of storage to a minimum, as only the changed and added dependency data would have to be archived when a new combination of KnowRob , its client and the episodic data (or the openEASE service itself) is released. When a specific version combination of openEASE is requested by a user, this solution would require an on-the-fly build of that container image, which could be faster than the download of a complete image, given that all dependencies are available locally. If a version combination is requested frequently, it could be kept in the Docker image cache, reducing the waiting time for those versions to a minimum.

Like the solution of archiving complete Docker images, implementing this solution would archive required data sources and dependencies, but with the ability to store everything separately either locally or in the same network. It should collect every necessary artifact required to build a version of openEASE during the build process and be able to repeat the build process based on the collected artifacts. If any upstream dependency breaks due to a bad update or a newly introduced incompatibility, the centrally archived and versioned artifacts should remain unaffected. This measure should not hinder standard development methods, though. It should still be possible to declare dependencies in a standard following fashion, because adhering to a standard helps the understanding of external developers, fosters contribution and ultimately reproducibility, too. In other words, this solution should not require to change the way dependencies like the installation of packages from APT, pip, npm or Maven repositories are declared. They should still be declarable through a command execution declaration in a Dockerfile, or the standard package dependency declaration of the various build systems of the programming languages used in openEASE.

Those demands result in the following requirements of the solution:

- Being able to store potentially large binary files efficiently (for episodic memory and binary dependencies)
- Providing a reference to the archived binary files
- Offering means to record and repeat a container building process
- Offering a mechanism to add a version to each recorded building session
- Issuing a combined version to recorded building sessions, stored binary files and source code, which is sufficient to describe a complete openEASE system including all related containers and data.

The design and implementation of such a solution is documented in the next chapter.

Chapter 5

Development of a heterogeneous data management system

In the last chapter, the requirements for a data management solution capable of handling the storage of both large binary files, as well as text files (such as software source code) have been set-up. The system should be capable of efficiently storing heterogeneous data, assigning version information to a collection of heterogeneous data and distributing such data. In this chapter, the design and implementation of such a solution is documented.

5.1 Design and quality requirements

During the analysis of openEASE, the need for an advanced and reliable data management tool emerged, resulting in the creation of 'MultiRepo'. It is designed to cater the needs of archiving dependencies required by the build processes executed to construct Docker images, as well as the repetition of said build process based on the archived artifacts. Another requirement was establishment of being able to create single versions describing a system comprising of heterogeneous data, such as openEASE, to be able to reference the state of the system by using a single identifier. A researcher using openEASE should be enabled to describe the system state his research was executed in with just the identifier, allowing other researchers to repeat the analysis and visualization results obtained from openEASE.

As MultiRepo should be able to reliably support the operation of openEASE , the implementation is accompanied with unit- and function tests. Furthermore, the consequent use of code comments of non-obvious implementation details assists in future maintenance and extension of MultiRepo.

MultiRepo is implemented using the Java programming language and is structured into four

software modules: The software core, a command line interface, a server implementation and the artifact-recorder.

5.2 Core

In the MultiRepo core, main functionality of MultiRepo is implemented. It offers the creation and the management of a collection of binary file repositories for large binary data and git repositories for source code and text files in a MultiRepo repository. A MultiRepo repository consists of a directory with a set of sub-repositories being either a git or binary file system repository.

Git was chosen to be supported as sub-repository for text-based files, because it offers the best performance for management of small binary and text-based content. It is the standard for collaboration in development projects, as it was designed exactly for this purpose. The widespread usage of git, observable for example on the code-hosting platform Github, means that developers are likely to be familiar with the use of git in software development. MultiRepo does not try to re-implement git or the collaborative hosting git repositories. Instead it is designed to silently archive read-only copies of the git repositories registered in a MultiRepo repository. The publication of changes to a git repository to an upstream collaborative git host happens through the official git command line interface, as well as the update of a git repository inside a MultiRepo repository.

Binary repositories on the other hand are completely managed within MultiRepo. Even though a MultiRepo repository may contain multiple binary repository instances, they are stored internally in the same data structure to enable cross-repository data de-duplication. Data added to a binary repository is automatically chunked into small parts using the rolling hash-algorithm Adler32 algorithm originally designed by Mark Adler for the deflate compression algorithm [GD]. To find the end of a chunk, the value of the rolling hash-algorithm is calculated with a window size of 2048 bytes, meaning that for each byte read from a file, the checksum for the last 2048 bytes since the current byte is calculated. Due to the nature of the Adler32 algorithm, this can be done very efficiently with one addition and one subtraction for each newly read byte. The result of the hash-value calculation is evaluated after each byte, whether or not it end with a specific bit-pattern. For MultiRepo binary repositories, the bit pattern `1001111111111111` (or `9FFF` in hexadecimal style) was selected. Knowing that the Adler32 checksum is relatively uniformly distributed, it statistically results in a bit-pattern matching that of `9FFF` every 40 kilobyte on average ($9FFF = 40959$). This algorithm has been adapted from the backup repository software [18], which in turn adapted the algorithm from the efficient network file transfer software [TJG98]. MultiRepo uses the chunking algorithm to deterministically split the files added to a binary repository, which results in the same chunk bounds being calculated when the content of the

chunk is identical. This trick allows de-duplicating file contents without knowing if and where a file with the same content has been added to the MultiRepo repository previously. MultiRepo just verifies whether the checksum of a chunk is already known to any of the binary file systems. If it is, a reference to the already stored chunk is used.

This technique along with the compression of chunks with the deflate algorithm [GD] allows the efficient storage of even large files, employing multi threaded chunking and parallel file writing capabilities. Parallel writing is done on purpose to leverage the parallel writing speed offered by modern solid state drives, two of which are built into the host system designated to host openEASE.

A binary repository preserves the file system structure of the files and directories added to it by encoding it into an efficient representation of the structure. For that purpose, an own implementation for describing, parsing and encoding objects into a byte representation has been written (*de.mh0rst.multirepo.core.binary* package), using the CBOR format [BH].

MultiRepo offers basic operations for binary repositories through an API: Showing the commit log, committing the current changes to the file system, calculating the difference between two revisions, as well as switching the checked out repository to a named revision. Every commit to a binary repository creates a new revision, which is simply the Secure Hash Algorithm SHA-256 hash value for the binary data representation of said revision. Binary repositories are one-dimensional, meaning that currently no branching or having multiple parent/child revisions are supported. This decision was made on purpose to keep the structure and the maintenance of such repositories simple. It is not expected that binary repositories are often updated by multiple developers, only on the release of a new software version built with the assistance of MultiRepo.

A MultiRepo repository has three different types of sub-repository collections. The first type is the collection of all known sub-repositories. It contains information about both binary and git repositories, like the name and (for git repositories) the upstream repository host and the tracked branch. If a user decides to remove a sub-repository from a MultiRepo, this collection still keeps a copy of the deleted repository. The deletion and addition of repositories is represented in the collection of working-copy repositories. It contains a list of currently checked-out sub-repositories, meaning that their contents are present directly in its assigned sub-directory inside the MultiRepo repository directory. The third collection of repositories is a compilation. A MultiRepo repository may contain any amount of repository compilations, which contain information about the sub-repositories being part of the compilation, as well as the revision ID of said repositories. A compilation can be checked out in a MultiRepo repository, which replaces all sub-repositories (both git and binary) in the working copy with the exact state represented by the compilation. Because a compilation also has an identifier, this facility can be used as a global version number to identify a combined openEASE software state.

An example of such a compilation for openEASE can be seen in the following enumeration:

- docker (git) - Repository containing build descriptions (Dockerfiles)
- dockerbridge (git) - Repository containing the source code of the *dockerbridge* container
- openease-webclient (git) - Repository containing the source code of the *openease-webclient* container with the KnowRob web client
- knowrob (git) - Repository of KnowRob (as example, KnowRob needs more repositories as established previously)
- dependencies (binary) - Repository of dependencies required for building openEASE containers
- episodes (binary) - Episodic memory data
- meshes (binary) - 3D robot model data

Note that the main openEASE repository is not part of this example compilation, as it is independent from the KnowRob client. It can still be part of the MultiRepo repository though, even part of the working copy, thus allowing the collection of all data related to openEASE in one place.

Additionally, apart from the actual transport itself, the client and server implementation is prepared in the core module. It offers an abstract implementation of pushing and pulling a complete MultiRepo repository including all sub-repositories, the transfer of one new revision to or from the server and the publication of a new default working-copy, a new sub-repository or the update of a git repository stored on the server from its upstream host (e.g. Github).

5.3 CLI

For easy interaction with a MultiRepo repository, a command line interface is provided. It offers operations like the initialization of a MultiRepo, operations on single sub-repositories (commit, diff, log) and the interaction with a MultiRepo server. To ease the interaction with the command line interface, the library picocli[1] is used. It allows the definition of commands, options and positioned parameters and generates help text from said definitions. As a bonus feature, command line TAB completion is offered by this library, providing the means to generate the required shell scripts.

The CLI implements a HTTP client suitable for communication with the MultiRepo server described in the next section. The CLI module itself only utilizes API offered by the MultiRepo core module, to ensure the coverage of its functionality by the unit- and function tests in the core module. In addition, some unit tests are installed to check the usage of the command line. For this to resemble a realistic test, the unit test launches the CLI Java software as a new process and asserts the expected output on the standard input/output.

[1] *https://picocli.info/*

5.4 Server

The server, as the command line interface, mostly utilizes abstract transport implementations from the MultiRepo core. It uses the JAX-RS[2] standard to provide a REST HTTP interface.

The server supports all transport operations implemented and described in the core module. For increased accessibility It also uses the milton2 library[3] to serve MultiRepo repository contents via WebDAV. Also, a simple HTTP based repository access service is implemented, supporting the retrieval of repository (again both git and binary repository) contents by repository revision ID, compilation ID or the latest (HEAD) version.

Currently, no security is implemented into the server, like a role based access control preventing unauthorized read and write access. This is easily integrateable using the JAX-RS security extension, supported by many libraries implementing authentication and role support.

5.5 Artifact-Recorder

The last module of MultiRepo is the artifact-recorder. It is responsible for the recording and replication of dependencies used in various build- and dependency systems. It can be imagined as serving as an HTTP cache, as all of the used dependency resolution mechanisms used in openEASE use a HTTP transport. Actually, the artifact-recorder can also act as a cache, accelerating the repeated access of dependencies (for example for subsequent builds of the same container, or two containers sharing the same dependency).

It is currently configurable with a JSON file, in which various repository types and their original host sources can be configured. Supported repository types are APT, Maven and npm repositories, as well as the PyPI pip repository. For these repository types, the artifact-recorder contains special caching and response filtering rules to support the dynamic rewriting of returned absolute host addresses to point to the artifact-repository service, as well as marking files likely to never change as immutable, always serving them from cache regardless of the client request.

All recorded dependencies are stored in a configurable directory, which can be the working-directory of a MultiRepo. With this method, all recorded dependencies can be stored and restored in a binary repository, using either a revision ID or a compilation ID.

The biggest advantage of the artifact-recorder is the transparent helper script injection functionality. It can supply a shell script through its HTTP service, which prepares the system it

[2] *https://github.com/jax-rs*
[3] *https://github.com/miltonio/milton2/*

is executed on to use the artifact-recorder instance the script was downloaded from as dependency source. The supported and replaced services to prepare can be dynamically selected using query parameters. For example the URL *http://localhost:8081/helper/cache/?with-service=apt&with-service=gradle&with-service=sh* provides a script which replaces all APT services configured in the running artifact-server on the system it is executed on with references to the artifact-recorder.

Another trick employed by this helper script is the replacement of the standard shell executable `/bin/sh` for Docker container builds. With this facility, the helper script intercepts calls made by the command execution function in a Dockerscript (`RUN command`), which is translated to a call to (`/bin/sh -c command`). Only when the Docker build is active, this replacement shell searches the supplied command for the occurrence of configured dependency services, and replaces them with artifact-recorder service references. This allows the replacement of commands like `RUN wget http://dependency/to/install.deb`, a command often used in Dockerfiles to obtain dependencies not present in any package repository.

This comes with an advantage and a disadvantage: The advantage of such a helper script is the easy integration into Dockerfiles. In the following evaluation chapter, an example listing of such an integration is provided, which is enough to intercept the dependency resolution for all dependencies used in openEASE. The disadvantage of this method is the possibility of missing a dependency, whose source host is not configured in the artifact-recorder, or whose dependency resolution system is not yet known to artifact-recorder. Whether or not this limitation hinders the use of artifact-recorder and consequently the MultiRepo stack for building openEASE shall be subject to further investigation conducted over a longer time period.

The efficiency and functionality of this MultiRepo stack is evaluated partially in the next chapter.

Chapter 6

Evaluation

After having completed both an analysis on the subject of reproducible results in research, especially for reproducible software, and the implementation of a new software tool designed to achieve the goal of software reproducibility, the following sections shall provide an assessment on the success of the documented work.

6.1 Quantitative evaluation of MultiRepo

With the implementation of the MultiRepo software suite, a tool has been created with the target of managing and efficiently storing both binary data and text-based source code files used and produced during the assembly of a complex software system, such as openEASE.

The evaluation of a system designed to offer long-time reproducibility of its data is particularly difficult, as the term long-time surpasses the timespan of this thesis by far. However, testing the realization of single goals of the software can help in evaluating the suitability for this task.

Beginning with the quantitative evaluation of MultiRepo, this section documents the archival of the episodic memory directory, totaling in 39 Gb of size, into a binary file repository hosted by MultiRepo. This dataset is chosen for evaluation because it is the largest binary dataset to be archived by MultiRepo when used to archive openEASE , and it offers a diverse assortment of data types, ranging from easily compressible but large JSON files, to badly-compressible video files. For comparison, the same task is executed with three other data archival solutions. The first being a plain git repository [TPH], the second being Data Science Version Control (dvc), a data management system combining git repositories with the ability to externally store large data [DVC] and the third being [PBZ19], a data backup software advertising its efficient storage capabilities.

During the addition of the episode data to those systems, the amount of time (both wall time and CPU time) was measured, as well as snapshots of the memory consumption during the execution. After the execution, the total storage size needed to store the episodic memory data internally was measured. The evaluation was executed on a system with the following specification:

Component	**Specification**
CPU	2x Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
RAM	256GB
Hard drive	2x 1.5 TB SSD storage
Operating system	Ubuntu GNU/Linux 16.04.5 LTS

Table 6.1 Specification of the evaluation test system

Starting with git version 2.7.4, adding the episode directory to a repository took 13 minutes and 23 seconds of wall time and exactly the same amount of CPU time, requiring 12,9 Gb of storage. Git utilizes file compression during the add process, thus reducing the size of the episodic memory data to a third of the original size. No significant memory usage was detected during this process.

```
$ time git add .

real    13m22.907s
user    13m3.088s
sys 0m19.788s
$ du -d0 .git/
12939364    .git/
```

An interesting behavior can be observed however when the garbage collection process of git is started afterwards. The garbage collector of git runs periodically, to execute de-duplication tasks and to efficiently store data in the git pack format. It is also executed when uploading git repository data to a remote repository. The garbage collection process takes 37 minutes and 36 seconds to complete, requiring 16.5Gb of memory during peak consumption. Over 2 hours and 54 minutes of CPU time was consumed during that process.

```
$ time git gc
Counting objects: 1956, done.
Delta compression using up to 72 threads.
Compressing objects: 100% (1945/1945), done.)
Writing objects: 100% (1956/1956), done.
Total 1956 (delta 1080), reused 10 (delta 0)

```

```
real    37m35.884s
user    167m4.376s
sys 7m20.444s
$ du -d0 .git/
12380936    .git
```

Through garbage collection, the size of the internally stored episodic memory data was further reduced to 95% of the size before the garbage collection. Both execution times combined amount to a total CPU time consumption of 3 hours 7 minutes and 28 seconds, and a wall time of 50 minutes and 58 seconds.

The next candidate is dvc in version 0.23.2 . It allows the management of large data inside a git repository by storing the data itself outside the git repository and keeps only references to the data. Adding the episode data to dvc required less time than git, totaling to 24 minutes and 48 seconds, but offered no compression or de-duplication savings:

```
/episodes$ time dvc add -R .
real    23m22.487s
user    8m50.216s
sys 14m8.080s

$ du -d0 .dvc
39096676    .dvc

$ time dvc push
real    1m26.740s
user    1m11.336s
sys 3m29.008s

$ du -d0 /tmp/dvc-storage
39095944    /tmp/dvc-storage
```

The amount of storage required by dvc equals the size of the episode directory, 39 Gb, showing no significant memory usage or CPU usage during the storage process.

The last candidate for comparison, bup, was tested in version 0.29-3. Originally advertised as a backup solution for binary data, the comparison against MultiRpo is especially interesting, because MultiRepo uses a similar de-duplication algorithm as bup, although with different parameters. It took 15 minutes and 10 seconds to add the episodes directory to the bup repository, requiring 14.9 Gb of storage. The CPU time totals to 14 minutes and 45 seconds, showing the single-threaded implementation of bup.

```
$ time bup save -n episode-image /episodes
Reading index: 2112, done.
```

```
bloom: creating from 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: creating from 3 files (600000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: creating from 9 files (1800000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (181062 objects).
bloom: adding 1 file (132299 objects).
bloom: adding 1 file (132581 objects).
bloom: adding 1 file (131621 objects).
bloom: creating from 19 files (3510177 objects).
bloom: adding 1 file (132097 objects).
bloom: adding 1 file (131334 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
bloom: adding 1 file (200000 objects).
Saving: 100.00% (39096530/39096530k, 2112/2112 files), done.
bloom: adding 1 file (176771 objects).

real    15m10.327s
user    14m2.348s
sys 0m43.232s

$ du -d0 .bup
14919044    .bup
```

Finally, the episode directory was added to a MultiRepo repository. The used Java runtime version for this test was the OpenJDK 11.0.2 version from Oracle, and the used version MultiRepo matches that of the latest commit from the source code repository `multirepo.zip` attached to this work. The test shows that MultiRepo completes the addition of the episode data in the least amount of time, requiring only 10 minutes and 17 seconds to save the data to the internal storage directory. The required CPU time of 58 minutes and 49 seconds proves that the optimization of

MultiRepo for multi threaded operation is working, occupying five threads on average. 13.9 Gb of storage are required by MultiRepo to store the episode data.

```
$ time mr rc -m "initial commit episode directory"
[Repo episodes initial revision] bd48b73d043d1d49

real    10m17.052s
user    54m3.544s
sys 4m45.988s
$ du -d0 .mr/fsBin/
13949480    .mr/fsBin/
```

On a note about memory utilization of MultiRepo: As no limitations were configured for the Java runtime, the memory usage peaked to 20 Gb during the test. This can be explained with the garbage collection mechanism of Java, allocating as much memory it is allowed to consume. The default memory limit for the Java runtime used on the test machine was 32 Gb, which was selected based on the total memory size available in the system (ca. 12.5% of the total size). To verify whether MultiRepo can archive the episodes directory with less available memory, the test was repeated while limiting the available memory to the Java runtime to 2.4 Gb:

```
$ export MR_OPTS=-Xmx2G
$ time mr rc -m "initial commit episode directory"
[Repo episodes initial revision] b9277983d30eaf24

real    10m20.822s
user    67m32.824s
sys 4m16.264s
```

The repeated test shows an increase in consumed CPU time to 1 hour and 7 minutes, which amounts to the average concurrent occupation of around 7 threads. The elapsed wall time only increased slightly by 3 seconds compared to the previous run.

Summing up the evaluation of the different data storage programs, MultiRepo ranks on the second position regarding space efficiency: The least amount of required storage is used by git with 12.3 Gb of occupied space, at the price of needing the same amount of memory as used by the internal data storage format. This behavior is explained by the fact that the deduplication and difference-building algorithm of git, xdelta, requires all data to be present in memory, as it is is optimized for operation on small, text-based files. The other data storage solutions required one gigabyte (bup) and 25.1 Gb (dvc) additional space. Referring to the required wall time to archive the episode data, MultiRepo was the fastest system to complete the storage process, with all compared systems requiring at least five additional minutes - clearly showing the unsuitability of git to store large amounts of binary data, such as the episodic memory, with over 50 minutes of runtime. This test shows that MultiRepo is suitable for the current worst case storage scenario

of openEASE.

Another measurement relevant for this evaluation is the test coverage for MultiRepo. Using the EclEmma code coverage tool it can be shown, that the unit tests of MultiRepo cover over 82% of the instructions present in the core module. As the uncovered lines mostly consist of unreachable exception blocks and code added for evaluation, it can be considered a well tested and high quality software.

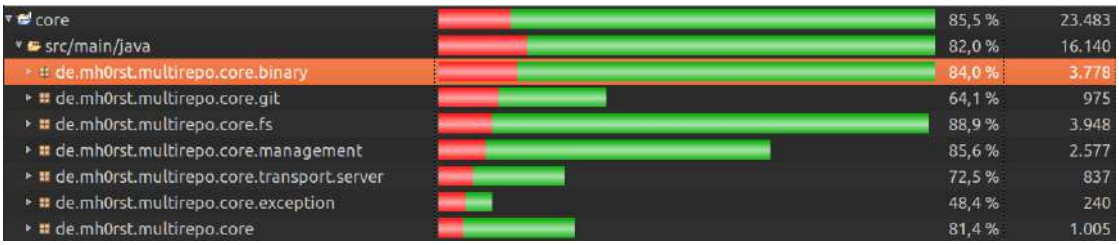

Figure 6.1 MultiRepo unit test code coverage

6.2 Improvements for openEASE reproducibility

The standalone storage performance of MultiRepo is just one aspect of the evaluation, as there are is also the implemented architecture improvement of the *openease-webclient*, providing a clean separation between the openEASE as a service and the used research software components KnowRobband the web-client. The loose coupling between the openEASE web interface and the *openease-webclient* container is an important step towards achieving research result reproducibility, while being independent from the MultiRepo implementation. MultiRepo is an experimental tool to produce efficient recordings of container build processes and the repetition of such recorded build processes, with no long-term experience. Therefore, the standalone architectural improvement of openEASE components is a notable advantage.

For a demonstration of the MultiRepo software to record and reconstruct container build processes, the build process of openEASE was modified such that the resolution of all dependencies was done with the artifact-recorder from MultiRepo. The required changes to the build scripts are documented in the attached repository `openease.zip` in the file tree of the commit `0ba9dbbe307a3ac8bb7efed66f8d5ab043fccdd5`. All Dockerfiles of openEASE were modified to download the documented helper script from the artifact-recorder service using the same three lines:

```
1 ARG AR=""
2 ADD $AR /tmp/ar-helper
```

```
3 RUN test -f /tmp/ar-helper && chmod +x /tmp/ar-helper && /tmp/ar-helper ||
      true
```

These lines define the argument `AR` as build parameter, which defaults to an empty string. When this parameter is omitted, the artifact-recorder service is not used. When the parameter is specified, the build process will attempt to download the dynamically generated helper script from the artifact-recorder service and execute it, which transparently modifies the build environment to request all specified dependencies from all supported dependency management systems from artifact-recorder.

Furthermore, the build script `scripts/build` was modified to pass both the URL of the artifact-recorder and the operation mode (cache, archive or rebuild) to the Dockerfiles, as well as the required dependency providers needed by each container. With a running instance of the artifact-recorder server using the provided `demoConfig.json` configuration file, a complete build of openEASE could be performed:

```
1 ./build --ar-url http://192.168.122.1:8081 --ar-mode cache nocache all
2 [...]
```

During the build, the artifact-recorder saved the dependencies from all openEASE containers into the configured directory (`arcache`), which then could be archived inside a MultiRepo binary repository.

```
1 $ du -hd0 arcache/
2 902M    arcache/
3 $ find arcache -type f | wc
4 5645    5645  463048
```

The artifact recorder was able to record 5645 dependencies from APT, pip, npm and Maven repositories, as well as a few plain HTTP resources, totaling to 902 megabytes of used storage space. This is less than a fourth of the *knowrob* container image size alone required for the complete build of openEASE . The results indicate the feasibility of the MultiRepo software stack to operate as central dependency archive, and with the collection feature which combines multiple repository resources, the assignment of a referencable openEASE version across all required dependencies brings the openEASE system a step closer to offer it as a valuable research service tool.

6.3 Conclusion

During the creation of this thesis, an existing research software was analyzed for its suitability to conduct reproducible research with it, working out the value of research reproducibility in general,

and the value of reproducible research software. The analysis showed both design decisions fostering that goal (like the use of Docker for the service organization and build description) and properties that prevented such an endeavor. The first steps to allow results of openEASE to be repeated have been completed with this thesis, but there is still a lot of work to be done.

MultiRepo still needs some improvement regarding its performance and security properties. A long-term test phase evaluating the suitability of MultiRepo to serve as the sole data archival system for openEASE is recommended before deploying it in productive systems. Regarding the user experience of MultiRepo, an integration of the functions currently only available through its command line interface into the administrative web interface should be considered. With the introduction of role based access control in the MultiRepo server implementation, it is feasible to use MultiRepo for the hosting and management of user-provided data (even large experience data), as well.

If the MultiRepo suite proves itself to be insufficient to reproduce research results during further tests, the archival of Docker images as explained in chapter 4 on page 31 could serve as a last resort.

Appendix A

Appendix

A.1 List of Figures

A.2 List of Tables

A.3 Bibliography

[18] *bup: Very efficient backup system based on the git packfile format, providing fast incremental saves and global deduplication (among and within files, including virtual machine images). Current re..* original-date: 2012-09-03T22:51:40Z. Feb. 14, 2018. URL: *https://github.com/bup/bup* (visited on 01/12/2019).

[AD16] Patrick Aerts and Peter Doorn. "Responsibilities - Data Stewardship and Software Sustainability". Research Software Sustainability: Report on a Knowledge Exchange Workshop. DFB Berlin, 2016. URL: *https://pure.knaw.nl/portal/en/publications/a-conceptual-approach-to-data-stewardship-and-software-sustainability(59c24848-9cf7-437c-b2d5-e943e9e4a35e).html* (visited on 01/12/2019).

[Awe19] AwesomeData. *A topic-centric list of HQ open datasets in public domains. PR : awesomedata/awesome-public-datasets*. original-date: 2014-11-20T06:20:50Z. Jan. 23, 2019. URL: *https://github.com/awesomedata/awesome-public-datasets* (visited on 01/12/2019).

[Bal19] Ferenc Balint-Benczedi. *Kinetic gradlew fix by bbferka · Pull Request #210 · knowrob/knowrob*. GitHub. Jan. 4, 2019. URL: *https://github.com/knowrob/knowrob/pull/210* (visited on 01/12/2019).

[Bee+15] M. Beetz, F. Bálint-Benczédi, N. Blodow, D. Nyga, T. Wiedemeyer, and Z. Márton. "RoboSherlock: Unstructured information processing for robot perception". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015 IEEE International Conference on Robotics and Automation (ICRA). May 2015, pp. 1549–1556. DOI: *10.1109/ICRA.2015.7139395*.

[Bee+16] M. Beetz, D. Beßler, J. Winkler, J. Worch, F. Bálint-Benczédi, G. Bartels, A. Billard, A. K. Bozcuoğlu, Zhou Fang, N. Figueroa, A. Haidu, H. Langer, A. Maldonado, A. L. P. Ureche, M. Tenorth, and T. Wiedemeyer. "Open robotics research using web-based knowledge services". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016 IEEE International Conference on Robotics and Automation (ICRA). May 2016, pp. 5380–5387. DOI: *10.1109/ICRA.2016.7487749*.

[Beß16] Daniel Beßler. *knowrob/docker git repository, file flask/Dockerfile, line 11*. GitHub. Oct. 29, 2016. URL: *https://github.com/knowrob/docker/blob/528f433/flask/Dockerfile#L11* (visited on 01/12/2019).

[BH] Carsten Bormann and Paul Hoffman. *Concise Binary Object Representation (CBOR)*. URL: *https://tools.ietf.org/html/rfc7049* (visited on 01/12/2019).

[BMT10] M. Beetz, L. Mösenlechner, and M. Tenorth. "CRAM — A Cognitive Robot Abstract Machine for everyday manipulation in human environments". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems. Oct. 2010, pp. 1012–1017. DOI: *10.1109/IROS.2010.5650146*.

[BN06] Eric W. Biederman and Linux Networx. "Multiple instances of the global linux namespaces". In: *Proceedings of the Linux Symposium*. Vol. 1. Citeseer, 2006, pp. 101–112.

[Boe15] Carl Boettiger. "An Introduction to Docker for Reproducible Research". In: *SIGOPS Oper. Syst. Rev.* 49.1 (Jan. 2015), pp. 71–79. ISSN: 0163-5980. DOI: *10.1145/2723872.2723882*. URL: *http://doi.acm.org/10.1145/2723872.2723882* (visited on 01/12/2019).

[Boz18] Asil Kaan Bozcuoglu. *knowrob/docker git repository, directory flask/webrob/templates*. original-date: 2014-06-24T15:41:02Z. Apr. 11, 2018. URL: *https://github.com/knowrob/docker/tree/fe8659e/flask/webrob/templates* (visited on 01/12/2019).

[BPB18] Daniel Beßler, Mihai Pomarlan, and Michael Beetz. "OWL-enabled Assembly Planning for Robotic Agents". In: *Proceedings of the 2018 International Conference on Autonomous Agents*. 2018.

[Bro+98] William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 1998. ISBN: 978-0-471-19713-3.

[BTW15] M. Beetz, M. Tenorth, and J. Winkler. "Open-EASE – A Knowledge Processing Service for Robots and Robotics/AI Researchers". In: *IEEE International Conference on Robotics and Automation (ICRA), Seattle, Washington, USA*. 2015.

[CER] CERN. *Storage | CERN*. URL: *https://home.cern/science/computing/storage* (visited on 01/12/2019).

[CFG16] Jürgen Cito, Vincenzo Ferme, and Harald C. Gall. "Using Docker Containers to Improve Reproducibility in Software and Web Engineering Research". In: *Web Engineering*. Ed. by Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 609–612. ISBN: 978-3-319-38791-8.

[DVC] DVC. *Data Science Version Control System*. URL: *https://dvc.org/* (visited on 01/12/2019).

[FF14] Benedikt Fecher and Sascha Friesike. "Open Science: One Term, Five Schools of Thought". In: *Opening Science: The Evolving Guide on How the Internet is Changing Research, Collaboration and Scholarly Publishing*. Ed. by Sönke Bartling and Sascha Friesike. Cham: Springer International Publishing, 2014, pp. 17–47. ISBN: 978-3-319-00026-8. DOI: *10.1007/978-3-319-00026-8_2*. URL: *https://doi.org/10.1007/978-3-319-00026-8_2* (visited on 01/12/2019).

[GD] J.-L. Gailly and P. Deutsch. *ZLIB Compressed Data Format Specification version 3.3*. URL: *https://tools.ietf.org/html/rfc1950* (visited on 01/12/2019).

[GH] Anthony Goldbloom and Ben Hamner. *Kaggle: Your Home for Data Science*. URL: *https://www.kaggle.com/* (visited on 01/12/2019).

[Gin91] Paul Ginsparg. *The physics e-print arxiv*. 1991. URL: *www.arxiv.org*.

[Gro+10] Robert L. Grossman, Yunhong Gu, Joe Mambretti, Michal Sabala, Alex Szalay, and Kevin White. "An Overview of the Open Science Data Cloud". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. HPDC '10. New York, NY, USA: ACM, 2010, pp. 377–384. ISBN: 978-1-60558-942-8. DOI: *10.1145/1851476.1851533*. URL: *http://doi.acm.org/10.1145/1851476.1851533* (visited on 01/12/2019).

[Gro+12] Robert L. Grossman, Matthew Greenway, Allison P. Heath, Ray Powell, Rafael D. Suarez, Walt Wells, Kevin White, Malcolm Atkinson, Iraklis Klampanos, Heidi L. Alvarez, Christine Harvey, and Joe J. Mambretti. "The Design of a Community Science Cloud: The Open Science Data Cloud Perspective". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 2012 SC Companion: High-Performance Computing, Networking, Storage and Analysis (SCC). Salt Lake City, UT: IEEE, Nov. 2012, pp. 1051–1057. ISBN: 978-1-4673-6218-4 978-0-7695-4956-9. DOI: *10.1109/SC.Companion.2012.127*. URL: *http://ieeexplore.ieee.org/document/6495909/* (visited on 01/12/2019).

[Hai+18] Andrei Haidu, Daniel Beßler, Asil Kaan Bozcuoglu, and Michael Beetz. "KNOWROB-SIM a Game Engine-enabled Knowledge Processing for Cognition-enabled Robot Control". In: *International Conference on Intelligent Robots and Systems (IROS)*. 2018.

[Har14] Tim Harford. "Big data: A big mistake?" In: *Significance* 11.5 (2014), pp. 14–19. ISSN: 1740-9713. DOI: *10.1111/j.1740-9713.2014.00778.x*. URL: *https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.1740-9713.2014.00778.x* (visited on 01/12/2019).

[Het16] Simon Hettrick. "Research Software Sustainability: Report on a Knowledge Exchange Workshop". In: *Copyright, Fair Use, Scholarly Communication, etc.* (Feb. 1, 2016). URL: *http://digitalcommons.unl.edu/scholcom/6*.

[Isa+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 59–72. ISBN: 978-1-59593-636-3. DOI: *10.1145/1272996.1273005*. URL: *http://doi.acm.org/10.1145/1272996.1273005* (visited on 01/12/2019).

[KH] James G. Kim and Michael Hausenblas. *5-star Open Data*. URL: *https://5stardata.info/en/* (visited on 01/12/2019).

[Lan16] Alex Lancaster. *Open Science and its Discontents | Ronin Institute*. June 28, 2016. URL: *http://ronininstitute.org/open-science-and-its-discontents/1383/* (visited on 01/12/2019).

[Leh80] M. M. Lehman. "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE* 68.9 (Sept. 1980), pp. 1060–1076. ISSN: 0018-9219. DOI: *10.1109/PROC.1980.11805*.

[McK84] James R. McKee. "Maintenance As a Function of Design". In: *Proceedings of the July 9-12, 1984, National Computer Conference and Exposition*. AFIPS '84. New York, NY, USA: ACM, 1984, pp. 187–193. ISBN: 978-0-88283-043-8. DOI: *10.1145/1499310.1499334*. URL: *http://doi.acm.org/10.1145/1499310.1499334* (visited on 01/12/2019).

[Men07] Paul B. Menage. "Adding generic process containers to the linux kernel". In: *Proceedings of the Linux Symposium*. Vol. 2. Citeseer, 2007, pp. 45–57.

[Mer14] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583. URL: *http://dl.acm.org/citation.cfm?id=2600239.2600241* (visited on 01/12/2019).

[Mül97] Bernd Müller. *Reengineering: eine Einführung*. Stuttgart: Teubner, 1997. ISBN: 978-3-519-02942-7. URL: *http://katalog.suub.uni-bremen.de/DB=1/LNG=DU/CMD?ACT=SRCHA&IKT=8000&TRM=31196285**.

[Nie11] Michael Nielsen. *Reinventing Discovery: The New Era of Networked Science*. Princeton, UNITED STATES: Princeton University Press, 2011. ISBN: 978-1-4008-3945-2. URL: *http://ebookcentral.proquest.com/lib/suub-shib/detail.action?docID=773462* (visited on 01/12/2019).

[OC] OpenAIRE and CERN. *Zenodo - Research. Shared.* URL: *http://about.zenodo.org/* (visited on 01/12/2019).

[PBZ19] Avery Pennarun, Rob Browning, and Zoran Zaric. *bup*. Version 0.29.2. original-date: 2012-09-03T22:51:40Z. Jan. 21, 2019. URL: *https://github.com/bup/bup/* (visited on 01/12/2019).

[Pon+15] Nancy Pontika, Petr Knoth, Matteo Cancellieri, and Samuel Pearce. "Fostering Open Science to Research Using a Taxonomy and an eLearning Portal". In: *Proceedings of the 15th International Conference on Knowledge Technologies and Data-driven Business*. i-KNOW '15. New York, NY, USA: ACM, 2015, 11:1–11:8. ISBN: 978-1-4503-3721-2. DOI: *10.1145/2809563.2809571*. URL: *http://doi.acm.org/10.1145/2809563.2809571* (visited on 01/12/2019).

[PTF08] Gordon Paynter, Len Trigg, and Eibe Frank. *ARFF (stable version) - Weka Wiki*. Nov. 2008. URL: *https://waikato.github.io/weka-wiki/arff_stable/* (visited on 01/12/2019).

[Rij+13] Jan N. van Rijn, Bernd Bischl, Luis Torgo, Bo Gao, Venkatesh Umaashankar, Simon Fischer, Patrick Winter, Bernd Wiswedel, Michael R. Berthold, and Joaquin Vanschoren. "OpenML: A Collaborative Science Platform". In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Železný. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 645–649. ISBN: 978-3-642-40994-3.

[Sch14] Jonathan W. Schooler. "Metascience could rescue the 'replication crisis'". In: *Nature News* 515.7525 (Nov. 6, 2014), p. 9. DOI: *10.1038/515009a*. URL: *http://www.nature.com/news/metascience-could-rescue-the-replication-crisis-1.16275* (visited on 01/12/2019).

[SS13] S. Sagiroglu and D. Sinanc. "Big data: A review". In: *2013 International Conference on Collaboration Technologies and Systems (CTS)*. 2013 International Conference on

Collaboration Technologies and Systems (CTS). May 2013, pp. 42–47. DOI: *10.1109/CTS.2013.6567202*.

[Tec17] Akamai Technologies. *Vergleich der durchschnittlichen Verbindungsgeschwindigkeit der Internetanschlüsse in Deutschland, Österreich und der Schweiz vom 3. Quartal 2007 bis zum 1. Quartal 2017*. Statista. May 2017. URL: *https://de.statista.com/statistik/daten/studie/416684/umfrage/durchschnittliche-internetgeschwindigkeit-in-dach/* (visited on 01/12/2019).

[Ten+15] Moritz Tenorth, Jan Winkler, Daniel Beßler, and Michael Beetz. "Open-EASE: A Cloud-Based Knowledge Service for Autonomous Learning". In: *KI - Künstliche Intelligenz* 29.4 (Nov. 1, 2015), pp. 407–411. ISSN: 1610-1987. DOI: *10.1007/s13218-015-0364-1*. URL: *https://doi.org/10.1007/s13218-015-0364-1* (visited on 01/12/2019).

[TJG98] Richard Taylor, Rittwik Jana, and Mark Grigg. *Checksum Testing of Remote Synchronisation Tool*. DSTO-TR-0627. DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION CANBERRA (AUSTRALIA), DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION CANBERRA (AUSTRALIA), Mar. 1998. URL: *http://www.dtic.mil/docs/citations/ADA355894* (visited on 01/12/2019).

[TPH] Linus Torvalds, Shawn O. Pearce, and Junio Hamano. *git*. Version 2.7.4. URL: *https://mirrors.edge.kernel.org/pub/software/scm/git/git-2.7.4.tar.gz*.

[Wes93] Richard West. *Reverse engineering: An overview*. HM Stationery Office, 1993.

[Wie+12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. "SWI-Prolog". In: *Theory and Practice of Logic Programming* 12.1 (Jan. 2012), pp. 67–96. ISSN: 1475-3081, 1471-0684. DOI: *10.1017/S1471068411000494*. URL: *https://www.cambridge.org/core/journals/theory-and-practice-of-logic-programming/article/swiprolog/1A18020C8CA2A2EE389BE6A714D6A148* (visited on 01/12/2019).

[Wil16] Chris Williams. *How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript*. Mar. 23, 2016. URL: *https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/* (visited on 01/12/2019).

[Yaz+18] Fereshta Yazdani, Gayane Kazhoyan, Asil Kaan Bozcuoglu, Andrei Haidu, Ferenc Balint-Benczedi, Daniel Beßler, Mihai Pomarlan, and Michael Beetz. "Cognition-enabled Framework for Mixed Human-Robot Rescue Team". In: *International Conference on Intelligent Robots and Systems (IROS)*. 2018.

A.4 List of Abbreviations

API Application Programming Interface, S. 13, 18, 19, 28, 36, 41, 42

CSS Cascading Style Sheets, S. 24, 32, 34

DOI Digital Object Identifier, S. 8
dvc Data Science Version Control, S. 45, 47

FTP File Transfer Protocol, S. 20, 22, 26

GUI Graphical User Interface, S. 16, 17, 24, 31–36, 53

IAI Institute for Artificial Intelligence, S. 12, 13, 15, 18, 20, 26

PyPI Python Package Index, S. 24, 43

ROS Robot Operating System, S. 16–21, 24, 26–28, 32–34
RPC Remote Procedure Call, S. 19

TLS Transport Layer Security, S. 19

URL Uniform Resource Locator, S. 33, 34

vcs version control system, S. 27

XML eXtensible Markup Language, S. 60

A.5 Glossary

APT
Short for Advanced Packaging Tool, a package distribution system. It is used in Debian GNU/Linux operating systems (and derivates) to distribute binary packages, such as software and updates. APT uses dpkg for the actual package management.
S. 20–22, 25, 27, 38, 43, 44, 51

dpkg
Short for Debian Package Manager. It is a software package manager used in the GNU/Linux distribution Debian and its derivates. It defines its own package format, files of this format have the file-ending `.deb`
S. 59

git A de-central data repository initially created by Linus Torvalds for the development of the Linux kernel [TPH]
S. 20, 22, 25–27, 31, 34, 40, 45, 49

Gradle
Gradle is a Java build management tool. The description of the software to assemble is done in a domain specific language based on the Groovy language. Gradle is used in popular frameworks and development kits, like the mobile operating system Android by Google.
S. 24

Java
An object oriented programming language. It features automated, garbage collection based memory management and platform portability through the use of byte code, which is executed on a Java Runtime Environment available for many common operating systems and processor instruction sets.
S. 20, 21, 59, 60

JSON
Short for Java Script Object Notation, a format allowing the textual representation of strings, numbers, lists and maps in an unstructured way.
S. 19, 35, 43, 45, 60

Maven
Maven is a widely-used automated Java build and dependency management tool developed by the Apache Software Foundation. It assembles Java software based on an eXtensible Markup Language (XML) description of the software. Maven defines a repository structure for hosting and resolving dependencies of Java software, which has been adopted by other build management tools for Java, as well.
S. 20, 22, 24, 25, 38, 43, 51

Node.js
Platform for running JavaScript code as standalone software. It uses the V8 runtime from the web-browser Google Chrome and is mainly used for server-side development.
S. 20, 21, 24, 60

NoSQL
A database offering the representation and querying of data in a non-relational way. It includes the capability to process unstructured data, such as JSON
S. 16

npm
Package manager for Node.js
S. 20, 21, 24, 25, 27, 34, 38, 43, 51

pip
Package manager for Python
S. 20, 21, 24, 25, 27, 38, 43, 51

Prolog

A logic programming language. Programs written in Prolog comprise of rules and facts, which are declaration of predicates that evaluate to true when the given condition is also true. Prolog runs within an interpreter, which takes Prolog statements and searches the declared rules for variable configurations which let the statement evaluate to true (backtracking).

S. 12, 16–19, 28, 32

Python

A dynamically typed programming language interpreted at runtime. It is notable for its relatively simple syntax using indentation for structuring code blocks, and thus being very popular among software development beginners and widely used for fast prototyping and advanced scripting.

S. 9, 18–21, 24, 32, 34, 60

svn

Short for subversion, a centralized version control system.

S. 20, 22, 27

A.6 Content of the disc

- `thesis.pdf` - This document
- `openease.zip` - A copy of the openEASE main repository from *https://github.com/daniel86/docker*, last updated on 2nd Aug 2018, containing both the analyzed source code of openEASE and the amendments and improvements described in this document.
- `openease-dependencies.zip` - A copy of the dependencies of openEASE, excluding the large datasets, as described in 3.1 on page 22, downloaded on 25th Jan 2019
- `multirepo.zip` - A copy of the main repository of the MultiRepo software suite developed during the creation of this thesis.
- `multirepo-bin.zip` - A compiled and runnable version of the MultiRepo software with all required dependencies included.