Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Development of a Build System for Cross-Platform Open-Source Projects

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Alexander Leutgöb, BSc
Matrikelnummer 0625881

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter
Mitwirkung: Univ.Ass. Dipl.-Ing. Dipl.-Ing. Dr.techn. Christoph Redl

Wien, 14. Oktober 2016

Alexander Leutgöb Thomas Eiter

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Development of a Build System for Cross-Platform Open-Source Projects

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Alexander Leutgöb, BSc
Registration Number 0625881

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter
Assistance: Univ.Ass. Dipl.-Ing. Dipl.-Ing. Dr.techn. Christoph Redl

Vienna, 14th October, 2016

Alexander Leutgöb   Thomas Eiter

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Alexander Leutgöb, BSc
Roßauer Gasse 4/4, A - 1090 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 14. Oktober 2016

Alexander Leutgöb

Kurzfassung

Software-Projekte entwickeln sich ständig durch aufeinander folgende Versionen weiter. Mit jedem neuen Release wird neue Funktionalität hinzugefügt oder bestehende angepasst und weiter verbessert. Während neue Versionen von kommerzieller Software in der Regel als Binärdateien verteilt werden und so für ihre Zielgruppe bereit zum Einsatz sind, verfolgen Open-Source Software Projekte in der Praxis meist einen anderen Ansatz. Diese Projekte werden in der Regel von Entwickler-Communities betreut, und mit Hilfe von agilen Methoden und Tools zur Online-Zusammenarbeit fortlaufend entwickelt. Aufgrund dieser Transparenz im Entwicklungsprozess unterscheidet sich auch der Weg der Veröffentlichung neuer Versionen. Neue Versionen werden oft nur in einem Versionskontrollsystem ausgezeichnet und dann als Quellcode-Archiv zum Download zur Verfügung gestellt. Obwohl dieses Archiv einfach zugänglich gemacht wird, vernachlässigen Entwicklungs-Teams die explizite Erstellung einer ausführbaren Binär-Version für Endnutzer. Vielmehr wird von jenen Nutzern erwartet, die Software selbst zu erstellen.

Genau dieser Schritt stellt jedoch nicht selten eine große Hürde für die Zielgruppe dar. Nicht alle potentiellen Endanwender einer Software sind aufgrund mangelndes Knowhows in der Lage diese selbst in ein ausführbares Dateiformat umzuwandeln oder oft auch nicht bereit Zeit darin zu investieren. Dieses Problem ist vor allem im Umfeld von Open-Source Software evident, wo Projekte oftmals für Endanwender bestimmt sind, welche nicht aus dem Bereich der Softwareentwicklung stammen. Ein Beispiel dafür stellt Software für Wissensrepräsentation dar, welche vor allem im Bereich *Business Informatics* zum Abbilden von Ontologien oder zur Extraktion von Information verwendet wird. Diese wird vorwiegend von akademischen Forschungsgruppen eingesetzt, welche zu einem gewissen Grad die vertiefenden Kenntnisse zum Erstellen der Software vermissen und daher für genau diese Zielgruppe nicht einsetzbar macht.

Um dieses Problem zu lösen, werden wir ein generisches Framework für ein Build-System entwerfen, welches verwendet werden kann, um den Erstellungs- und Veröffentlichungs-Prozess eines Software-Projekts zu automatisieren und somit Entwickler bei der Bereitstellung von aktuellen Binär-Paketen für Endnutzer zu unterstützen. Besonderes Augenmerk wir dabei darauf gelegt, das Framework möglichst allgemein zu entwerfen, um es so für eine breites Spektrum von Software-Projekten unterschiedlicher Größe einsetzbar zu machen. Das Framework unterstützt dabei drei Zielsysteme: Ubuntu Linux, Mac OS X

und Windows. Ziel des Frameworks ist es dabei, Binär-Dateien von C ++ Projekten zu erstellen, welche *GNU Autotools* benutzen.

Als letzter Schritt dieser Arbeit wird das Framework gegen ein Projekt aus der realen Welt validiert. Das Forschungsgebiet der logischen Programmierung gibt dabei ein gutes Beispiel für eine Open-Source Software-Projekt, für welches die Verteilung in Binärform aktuell eine Hürde nach dem oben beschriebenen Problem darstellt. Die dafür verwendete Software namens DLVHEX, ein Reasoner für HEX-Programme, wartet aufgrund externer Bibliotheksabhängigkeiten mit einem hohen Grad an Komplexität auf. Aufgrund dieser Komplexität folgt die Software aktuell einem seltenen und unregelmäßige Release-Zyklus. Dies macht neue Funktionen für Endanwender über einen längeren Zeitraum hinweg nicht zugänglich, da die Software vor allem von Menschen ohne tiefere Kenntnisse von Softwareentwicklung verwendet wird. Schlussendlich wird das Framework dazu verwendet werden, um eine neue Version der betrachteten Software zu erstellen.

Abstract

Software projects evolve through successive releases. With every release, new functionality gets added or existing functionality is adapted and improved. While new releases of commercial software are usually distributed in binary form, and are thus ready for usage for their target group, open-source software projects face a different approach. Open-source projects are typically maintained by developer communities who use agile methods and online collaboration tools. However, due to this transparency in the development process, the way of releasing new versions differs compared to closed-source projects. New releases are often only tagged within a version control system and then provided as source code archives. While these snapshots of the development efforts are accessible for everyone, project members often skip building a binary for distribution and end users of the software are therefore expected to build the software themselves.

Software maintainers however do not see that building software is a barrier for ordinary users. Not all potential users are able and willing to build software from source. Often, users might lack the necessary knowledge or do not want to spend time on this issue. This problem of missing binary builds is especially true for open-source software, targeting people that are not originating from the field of software engineering, like software created for knowledge representation and reasoning. Such software is mainly used by academic research groups in the field of *Business Informatics* for describing ontologies or information extraction. People of that target group often miss the in-depth knowledge for building different available solving tools from source, which makes the usage of such tools more difficult or not applicable at all.

To address this problem, we will design a generic framework for a build system that can be used to largely automate the release process of a software project and thus can support developers to provide up-to-date packages for end-users. The framework will be designed in a generic way, so that it is applicable for a wide range of software projects of different size and will support three target systems: Ubuntu Linux, Mac OS X and Windows. The framework will further be designed to build C++ projects that make use of *GNU Autotools*.

As a last step, the framework will be validated against a real-world project. The field of logical programming will be used to derive a good example for an open-source software that lacks of binary distribution, according to the problem described above. The used software DLVHEX, a reasoner for HEX programs, comes up with a high level of complexity

due to external library dependencies. Because of that complexity, the DLVHEX software currently does not follow a frequent and periodical release cycle, which makes new features for end-users inaccessible for a longer time, as the software is mostly used by people without deeper knowledge of the software build process. The framework will finally be used to create a new release of the used software.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Description 1
1.3 Aim of the Work 2
1.4 Approach 2
1.5 Structure of the Thesis 3

2 Existing Approaches **5**
2.1 Introduction to the Software Release Process 6
2.2 Build Tools 13
2.3 Build Concepts 15
2.4 Summary 18

3 Development of the Framework **21**
3.1 Decision Dimensions 22
3.2 Build System Approaches 27
3.3 Linkage of Dimensions and Approaches 30
3.4 Framework Design and Implementation 34
3.5 Summary 44

4 Case Study **47**
4.1 Project Selection 47
4.2 Goals and Expected Results 51
4.3 Framework Implementation 52
4.4 Issues and Limitations 58
4.5 Future Work 59
4.6 Summary 60

5 Conclusion and Outlook **63**
5.1 Summary . . . 63
5.2 Future Work . . . 64

A Framework Scripts **67**
A.1 Installation Stage . . . 67
A.2 Build Stage . . . 70
A.3 Test Stage . . . 72
A.4 Deployment Stage . . . 73

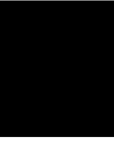
List of Figures **75**

List of Tables **75**

Acronyms **77**

Bibliography **79**

CHAPTER 1



Introduction

1.1 Motivation

A software project evolves through successive releases. Over time, new functionality gets added or existing functionality is modified and improved. While new releases of commercial software are usually distributed in binary form, and are thus ready for usage for their target group, open-source software projects face a different situation. Open-source projects are typically maintained by developer communities who use agile methods; online collaboration tools for code hosting, reviewing and testing make it easy to accelerate the development. However, due to this openness in the development process, also the way of releasing new versions differs from closed-source software. As a snapshot of the current development efforts is accessible for everyone, new releases are often only tagged within a version control system and then provided as source code archives. Project members often skip pre-built binary distribution, as most of them see their role in maintain code, not releasing packages. End users of the software are therefore expected to build the software themselves.

1.2 Problem Description

Not all potential end-users of a software project are able and willing to build software from source. Some users do not want to spend time on this issue. Most of them however often do lack the necessary knowledge for that. Dependencies to other software libraries and additional software tools, which are not part of the project and therefore must be installed separately, can transform the build process in a time consuming and error-prone task. If software is released for multiple platforms and architectures, dependencies have to be met for a multitude of possible build variants. Such constellations of dependencies and platforms increase the difficulty for users even more. To circumvent issues, projects

often make use of build tools. Such build tools however still come with shortcomings, as most of them are designed for a specific task within the build process.

Therefore, build attempts from end-users often fail. On the other hand, developers are experts who often do not see that barriers for ordinary users and therefore do not provide binaries for the target group. This creates a high but unmet demand for binary builds.

The absence of pre-built binaries often results in less usage of software, which then lacks of distribution and awareness in the market. Subsequently, projects suffer from missing support within the target user community and in less contributions or donations for developing the project further.

The problem of missing binary builds is especially true for open-source software, targeting people that are not originating from the field of software engineering, like software created for knowledge representation and reasoning. Such software is mainly used by academic research groups, whose people often miss the in-depth knowledge for building different available solving tools from source and therefore making the usage of such tools more difficult or not applicable at all. A concrete target group that demands pre-built binary packages is represented by people from the field of Business Informatics, where knowledge reasoning is used for representing ontologies and for information extraction.

Open-source projects therefore often suffer of low accessibility and visibility due to absent binary packages for end-users. Providing such packages is often neglected as creating them is a time-consuming task that cannot easily be automated.

1.3 Aim of the Work

To address this problem, this thesis first identifies and elaborates different existing build system approaches that can make the distribution of cross-platform software binaries easier. This requires the breakdown of the different stages and steps within a software release process and to identify the most complex and time-consuming steps.

After that, we look into existing build tools and different concepts of creating a new build. This allows to summarize the difficulties and shortcomings of current approaches that originate for software projects supporting different target platforms.

As a main objective, the thesis finally aims to design a generic framework for a build system that can be used to largely automate the release process of software projects and thus supports developers to provide up-to-date packages.

1.4 Approach

This thesis consists of a theoretical part and an empirical part.

The theoretical part covers the examination of different existing build tools and concepts and identifies the different steps involved in building and distributing cross-platform open-

source software. Subsequently different state-of-the-art concepts for creating software builds are examined.

We also introduce different build system approaches and identify orthogonal dimensions for selecting one of the approaches. Finally, those dimensions and approaches are linked, serving as foundation for the empirical part.

The empirical part selects one of the introduced build system approaches from the theoretical part and then designs a generic framework for creating a new software build on top of that. A concrete case study finally tests the framework against an existing scientific open-source project.

1.5 Structure of the Thesis

The thesis is structured as outlined in the following paragraphs.

Chapter 2 gives a theoretical introduction into a usual software release process. It covers all the steps required to build a new version of a given software project and distribute it to the target groups that uses the software in their work-flows. The chapter then introduces existing build tools that can be used to build the binary outputs of different software projects. Finally, we describe different build concepts that can be applied in order to use the introduced tools for building a new version release.

Chapter 3 covers the practical part of this thesis. Given the theoretical reflections of the previous chapter, this chapter defines different decision dimensions that can be used to evaluate a concrete software project in terms of complexity, release cycles, the target group or target platforms. The decision dimensions are then linked to introduced build system approaches and thus serve as a basis for selecting an appropriate approach for every given software project. As a result of these considerations the chapter designs a build system framework on top of a concrete build system approach.

The following Chapter 4 uses the previously designed framework to implement a build system for the concrete software project DLVHEX. The chapter describes the considerations of the selection of the case study subject and defines the goals and outcome of the implementation. It then compares the implemented build system with the initial goals and describes potentials issues and restrictions coming from the framework designed in Chapter 3. It also lists possible improvements and additions when used in a production environment.

Chapter 5 finally summarizes the obtained results and gives a short outlook for related work.

CHAPTER 2

Existing Approaches

This chapter gives an overview about a complete software release cycle. It describes state-of-the-art tools and approaches for building and releasing cross-platform software. It finally covers different techniques how these approaches are designed and implemented.

A software system evolves through successive releases [GJR99]. With every release the system is modified in a way that new functionality gets added or existing functionality is modified and improved. A single release identifies a well-defined implementation of the system.

In the most basic setup a software project is made up of a set of programming instructions, collected in source files. These instruction sets are converted into an executable binary that is used as software. Typically a project release however does not only cover changes to that source code files. Beside that collection of changes, a software release also implies changes to related artifacts such as documentation, associated configuration files or software build scripts related to a specific project. A software release is tagged with a unique version, the wordings are used interchangeably in the following sections.

In production-grade software projects building a binary or library is only the first step towards a release. A defined software version is defined by a set of build products that are often to referred as *build artifacts*. A build artifact is a product or associated piece of information that is attached to a specific software release.

The following listing gives an overview of commonly created build artifacts:

- Binaries or libraries
- Documentation
- Release notes and change logs

- License information and acknowledgments
- Usage examples

Depending on the type and the size of a software project it may contain only a binary, multiple types of artifacts or all of them.

2.1 Introduction to the Software Release Process

Different types of build artifacts require different steps during a software release. Creating and releasing a specific version of a software system therefore takes the following actions, which are described in more detail in the following subsections:

1. Source Code Management: Identifying source code and track its changes.
2. Creating Binaries: Converting source code into an executable format.
3. Testing: Check if the implemented system fulfills the desired goals.
4. Assembly: Collecting binaries and other build artifacts.
5. Distribution: Release the software and related artifacts.

2.1.1 Source Code Management

The essential routines of a software system are described by its source code. In its simplest form source code is organized on a local machine within a source directory. However, as open-source software is mostly maintained by a group of people an appropriate solution for organizing the source code for collaborating on the same project needs to be found. These solutions are summarized under the term *Version Control System (VCS)* [Nag05]. A VCS is a software tool designed to help software projects track and control changes to source code over time [Roc75].

Grouped changes to source code are identified by labels composed of numbers or letters (often referred to as *revision* or *commit*). Due to that, it is possible to manage different versions of content over time and read historical data. VCS also incorporate user accounts that make it traceable who changed which module of a software at any point in time.

A release of a software project references a defined state of the source code. Such states are then often frozen and marked with labels called *tags*.

Version control systems exist since the very beginning of software development. One of the first system dates back to 1972, when *Bell Laboratories* released a software named Source Code Control System (SCCS) [DNMV+13].

Over time, the way files and projects are handled changed as much as existing tools, work-flows and collaboration patterns. Today modern VCS are mostly organized around

distributed approaches. Further, with the advent of cloud services a lot of public offerings for VCS allow to collaborate in distributed groups and teams. The largest code hosting provider alone (*GitHub, Inc.*) reports over 11 million people[1] collaborating on more than 27 million online repositories [GVSZ14].

Today VCS can be categorized in different dimensions, such as:

- Single user vs. multi user systems
- Centralized vs. distributed systems
- Different branching models

Due to the tagging nature of software releases in a VCS system it is a common pattern to not only store source code files in the system but also documentation, configuration files or other associated entities.

2.1.2 Creating Binaries

Compiling an executable of a software system describes the process of converting one or more source code files into an executable file format used on a specific target machine configuration to run the actual software.

Executables are platform-dependent, meaning that a program compiled for a specific system architecture or operating system may not be executable on another platform. This is due to the fact that compiled binaries include low-level instructions that directly address the underlying platform and may be unknown to others.

There is usually a differentiation between the system on which a binary is compiled on (the *build* system), and the system on which the binary should be executed on (the *host* or *target* system) [MG03]. If the build and target are the same, the compilation is called *native compilation*. If the systems are different, the term *cross-compiling* is used.

Build Preparations

Before the actual compilation routines take place, a set of preparation steps need to be done:

1. Preparing the build system
2. Resolving and installing dependencies
3. Retrieving the software files to build

[1] github.com/about/press. GitHub. Retrieved 2015-09-29.

Building a software requires a valid build system to be set up. A build system often needs to fulfill a defined requirements list regarding hardware capabilities and operating system features such as a minimum amount of main memory. It further needs installed compiler suites (Section 2.1.2) and configured compiler tool-chains for the target system.

Most software systems are not self-contained but depend on a set of other software components or libraries. Creating a binary also means that these dependencies need to be resolved before the actual compilation step and that those dependencies can be found by the compiler tools. Dependencies can either be compiled from source or installed manually or with the help of package managers (Section 2.1.5).

As soon as all dependencies are resolved, the build system needs to get the latest valid source files of a software project. In most cases, source files are retrieved from a VCS system (Section 2.1.1).

After all those steps the software can finally be compiled as described in the following section.

Input and Output

The main goal of compilation is transforming source code into an executable file format, representing the software system, as shown in Figure 2.1.

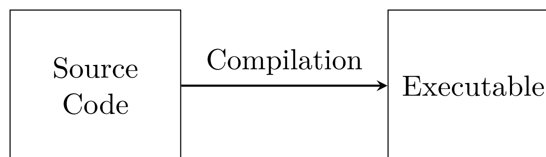


Figure 2.1: Compilation Process

Input files for compilation are source code files written in a high-level programming language such as *C++* or other languages. In contrast to compiled languages there are also interpreted languages that do not need compilation but are executed step-by-step by a tool called *interpreter*.

The output of the compilation step is either an executable binary built for a specific target platform or a software library that can be used by other executables on supported platforms.

Compilation

From a technical viewpoint, *compiling* only refers to the step of transforming source code into binary code, known as *object code*. However, the steps from source code to an executable binary also include *preprocessing* and *linking*. Figure 2.2 shows that each step produces intermediate products that are the input for the following compilation step [HGM03].

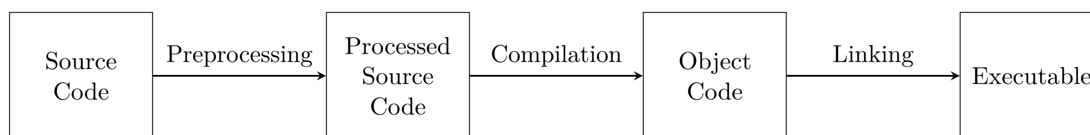


Figure 2.2: Detailed Compilation Process

A *preprocessor* manipulates the input source code on a lexical level. This includes the replacement of so-called preprocessor macros with source code. This includes tasks as removing comment blocks and replacing them with spaces, stripping blocks from *#ifdef* statements that are not in the current scope or expanding *include* statements and replacing *define* statements with actual code.

The preprocessed source files are forwarded to a tool called *compiler* that transforms human-readable source code files, written in high-level programming languages, into machine-readable object code or machine code. This machine code represents instructions that are directly executable on the processor unit of the intended target machine. If the compiled code can be executed on a system with a processor architecture or operating system different from the one on which the compiler runs, the compiler is known as a *cross-compiler*.

In order to use object code, it must be linked into an object file, a library file or an executable. This final step in the compilation pipeline is done by a *linker*.

In certain setups compiling source code into executables requires additional steps such as source code generation or bootstrapping. These steps are not further described as part of this section.

Existing Tools

Software building requires a set of tools, which usually highly depend on the target platform for which the software is built, preprocessors, compilers, assemblers, linkers and other build tools come bundled into *compiler suites*. To maintain flexibility, individual tools can however still be replaced case-by-case.

The most popular compilers are *GCC*[2] and *LLVM/Clang*[3] for Unix platforms and *MSVC/Visual Studio*[4] for Microsoft platforms. All in common is that compilers come in suites with supported preprocessors and linkers and that these suites support several programming languages and run on a variety of hardware architectures and operating systems, either by build to target compilation or cross-compiling.

[2] https://gcc.gnu.org. GCC, the GNU Compiler Collection.
[3] http://llvm.org. The LLVM Compiler Infrastructure.
[4] https://visualstudio.com. Visual Studio

2.1.3 Testing

Software Testing is an important task to determine the quality grade of a software and to check if the implemented system fulfills the desired goals and solution. It is therefore a significant part of the entire development process.

The formal definition of testing varies in different publications, depending on the purpose, process and level of testing [Luo01]. The goal of software testing is to affirm the quality of software systems by systematically exercising the software in defined and controlled circumstances.

The area of software testing involves a lot of different steps and techniques, both from a technical viewpoint and a non-technical. They include specification, design and implementation, maintenance and process and management issues. For our purpose it is sufficient to look at different test types and methods.

Testing software creates value in every stage of a software development process, in evaluating the initial system and for subsequent releases when a system grows over time.

The following subsection describe different levels of testing. Each level can make use of the testing methods described in the consequent subsection.

Testing Levels

Different stages in the software development life cycle require different test methods. Each test on each layer of development follows different goals and objectives [Luo01].

On an abstract view tests can be categorized in four different levels:

1. **Unit Testing** involves testing basic units of software and compares single lines of source code or source code sets against well-defined expected results. Unit testing is therefore done at the lowest level of software development. The quality of unit testing can be measured with test coverage indicators.

2. **Integration Testing** is done as soon as two or more tested units are combined into a more complex component. It often requires well-defined interfaces for every single unit that is the base for performing tests.

3. **System Testing** subsumes test procedures where testing is done on an end-to-end base. With system tests, an implemented system is tested against its functional requirements and checks if it fulfills the desired goals. System testing also involves testing non-functional requirements such as reliability, security or performance indicators in real-world environments.

4. **Acceptance Testing** is the very last and highest level test stage and involves testing the entire system by target user groups. The goal of acceptance tests is to get confidence that the implemented software is working rather than finding errors.

Testing Methods

Orthogonally to levels, tests can further be categorized in two main techniques, depending on the stage of software development:

- **Static Analysis Tests** are performed without actual execution of source code and examined on a source level. Methods of static analysis are used to evaluate software quality indicators without the actual execution of code. Approaches include manual code inspection, program analysis, symbolic analysis or model checking [SKW08].
- **Dynamic Analysis Tests** are performed through actual execution of source code. This is done with a well-defined input set of test data or actual real-world circumstances depending on the level of software testing. Approaches include synthesis of inputs, the use of structurally dictated testing procedures, and the automation of testing environment generation.

Documentation Tests

As stated in the beginning of this chapter, software releases are compiled of a list of different build artifacts beside the core software binary or library. One example is the existence of documentation for a specific software release.

Testing a documentation involves checking the correctness, completeness and integrity of all related documents, e.g. coverage of new features or functions, spelling and phrasing or correct version references in listed changes and related files.

Due to a common trend to online software distribution, associated documentation is also most commonly made available in an Internet compatible file format like Hyper Text Markup Language (HTML). One advantage of such file formats is the possibility to link to referenced documents that also requires additional checks for broken hyperlinks or missing documents.

Test Automation

Depending on the type of test and level within the test process tests are either automated executed from within a build system or need manual procedures according to test protocols. A more detailed description of test automation is given in Section 2.2.

2.1.4 Assembly

After compiling binaries, documenting changes and other build-related release steps, the resulting artifacts are usually collected at a centralized data storage. In such a store, the artifacts are either put within a specific directory hierarchy or bundled in an immutable archive file for subsequent access to build data.

This data storage represents the last stage of the build process before a new version of a software project is finally distributed to end-users.

2.1.5 Distribution

One of the most dynamic parts of a software release process is its distribution. This step was subject of fundamental changes with the last decades.

In early days, commercial software was mainly distributed with retail stores or shipment of physical data storage mediums like *floppy disks* or *CDs* and *DVDs*. This differed for open source projects. Only bigger and proprietary supported projects like the Linux distribution *Ubuntu Linux* were able to ship storage mediums. Most software projects only had the possibility to be distributed peer-to-peer or with FTP servers on a much smaller scale.

However, with the success and adoption of the Internet software projects gained a new distribution channel. Especially for open-source projects, Internet distribution is the most important approach as of today, therefore the following paragraphs focus on online distribution only.

Methods of online distribution include:

- Manual Downloads
- VCS Platforms
- Package Managers
- Custom Installers

The most basic method of software distribution is the download of binaries from an HTTP or FTP server (other protocols and services are possible but not mentioned due to their order of popularity). Associated documentation is often stored on a web storage in an HTML format or as part of a *Wiki* system. In addition, documentation may be bundled to the binary download as part of a download archive file format.

A more unified way of providing build artifacts is the application of source code management platforms like *SourceForge*[5] or *GitHub*[6]. These platforms provide dedicated places where tagged releases can be uploaded and made available for download. That has the advantage that binaries, documentation, test reports and other build artifacts are collected and provided at one easily to find place.

More popular software projects with periodical release cycles adopt another software distribution approach named *package managers*. Package managers are very popular on Unix-based system, especially Linux distributions often ship with system-wide package managers as main source of software installations.

Package managers are used to search, install, upgrade and remove software components or packages on target machines [ADTZ11]. A package manager most often involves a

[5] http://sourceforge.net. SourceForge.
[6] http://github.com. GitHub.

dependency solver for resolving software dependencies and different versions of software releases during installation. Beside software component binaries package managers also optionally install related artifacts like documentation and configuration files.

Package managers usually install well-tested pre-compiled binaries of software releases, a few others like Mac OS X based *Homebrew*[7] compile binaries from source on the target machine itself.

Most popular examples for package managers are *Advanced Packaging Tool* (Advanced Packaging Tool (APT)) on *Debian*-based Linux distributions, *RPM Package Manager (RPM)* on *Red Hat*-based distributions or already mentioned *Homebrew* on Mac OS X.

Another possibility is the usage of custom installers that copy files or packages of files to a defined target destination. Installers can be simple binaries dedicated for installation of a single software project or can also incorporate some kind of package control. This is especially true for software that might be extend-able with a plugin system. Examples for package-based installers are the *Qt Installer Framework*[8] or *Oomph*[9] used by the *Eclipse* platform.

2.2 Build Tools

Section 2.1.2 already gives a short introduction into the main step of a software project release: Creating a software build, which is the process of converting one or more source code files into an executable file format. This section describes the practical aspect of that step and lists some available tools that support binary compilation.

Depending on the project size, manually executing compiler commands for source files may be an appropriate way and can be semi-automated by custom batch scripts that call tools like the compiler and the linker. For bigger projects, maintenance of such custom scripts may get tedious and more error-prone as input files or dependencies change over time.

2.2.1 Build Utilities

This led to the creation of utility tools that help automating software builds based on simple and repeatable tasks. Build utilities manage dependencies and executing commands in a specific, repeatable order. They also optimize the build process by only calling commands for input files or dependencies of input files that actually changed since the last execution. Most build utilities also support different sets of rules that allow to either build a binary, clean out a project from intermediate files or copy build artifacts to a desired installation location.

[7] http://brew.sh. Homebrew.
[8] http://doc.qt.io/qtinstallerframework. Qt Installer Framework.
[9] http://projects.eclipse.org/projects/tools.oomph. Oomph.

Due to its distribution on Unix-derived operating systems, the most popular build utility is *GNU Make*[10].

In order to use Make or another build utility, a custom configuration file, often called *Makefile*, is required. Makefiles define input source, dependencies and other rules, subsumed in command blocks called *rules*. Such a rule can include commands for a compiler, a linker and copy commands for installing a software build.

2.2.2 Differences Between Target Platforms

Thus far we referenced a software build as an abstract step independent from its target environment. In conjunction to source code, executables are however not independent of the platform on which they are run. Depending on the architecture of the underlying system, every target requires a dedicated executable derived from a configuration of different compilers, options, specific source files to compile and link, and different supporting tool sets. This implies that a single compiled binary is not necessarily executable on all target platforms but that a binary must be compiled for every specific platform.

A computer platform can be defined on to two different abstraction levels. First, a software is compiled for a specific hardware architecture. Each architecture supports a minimal set of Central Processing Unit (CPU) commands that mark the division between the actual hardware and a software, called the *instruction set* of an architecture. A compiler creates machine code that complies to a specific instruction set architecture. The execution of that machine code can therefore break on other architectures than the compiled one [SN05]. Another major difference between hardware architectures is the order of multi-byte data represented in computer memory, the so-called *endianness* of a system [BM05]. Second, an operation system provides runtime libraries that allow the reuse of code and provide abstraction layers between low-level functionality and the software. As these libraries differ from system to system, software has to be compiled for every intended software platform.

As stated, tools like *GNU Make* support the build process by automatically resolving dependencies and calling compiler commands. This however does not solve the issue of different target systems, as Makefiles are still bound to a target platform and depend on a specific set of compiler tools. Therefore, ensuring that software can be built and run identically on many systems often creates a big overhead in the development process and during the creation of a new release.

As a software can often be viewed a collection of different features or packages. Depending on the target platform features may be available or not. To disable specific features software maintainers may have to edit header files prior to building a new release. That task requires in-depth knowledge of both, the software project to be built and the underlying platform.

[10] https://gnu.org/software/make/. GNU Make.

As a consequence, source code often incorporates preprocessor-specific macros or *#ifdef* statements that result in complex code structures and nested code targeting different platforms, making code maintenance even more difficult [Zad02].

2.2.3 Supporting Different Platforms

Because of these issues with different target platforms, a set of tools, often referred to as *build tools* or *build suites*, are available. Such tools help to automate different steps in the build process. These steps include the configuration of compiler tools depending on the desired target platform and to automatically create executable binaries from a single set of source code.

Build tools solve the target complexity by providing predefined tests that can dynamically detect which features are supported on the tested system. Depending on the test results, they can then enable or disable feature packages, ensuring that a build completes successfully on every supported platform.

After several test steps these tools can also auto-generate Makefiles incorporating platform-dependent compiler flags and configuration values and finally start the build process itself.

As build tools run identically regardless of the target system and version it is possible to maintain software projects that are both portable and easy to maintain.

Due to its availability on Unix platforms the most popular build tool is the suite of *GNU Autotools*[11]. Other popular options are *CMake*[12], *qmake*[13], *SCons*[14] or *Waf*[15].

2.3 Build Concepts

Section 2.1.2 introduced the differentiation between the build system and the target system and that software is compiled for a specific target platform, depending on processor architectures and operating systems [MG03]. As compiler suites can often build for a multitude of architectures different build concepts, depending on the build and target system, can be defined:

- Native Compilation
- Cross Compilation
- Hardware Virtualization

[11] https://www.gnu.org/software/automake/. GNU Automake.
[12] https://cmake.org. CMake.
[13] http://doc.qt.io/qt-5/qmake-manual.html. qmake Manual.
[14] http://scons.org. SCons.
[15] https://waf.io. Waf.

2.3.1 Native Compilation

If the compilation environment of the build system matches the target system, the compilation process itself happens in a *native* environment. Dependencies are most often installed and used system-wide and compilation tool-chains do not require additional translation steps between the build and target platform.

Compiling software directly on the target system brings some advantages, including:

- It provides the most possible target platform compatibility as binaries are created with the actual dependencies within the same environment on which the software will be executed on.
- Compiling software on the native platform with a native compiler usually creates more efficient executables regarding memory consumption and runtime performance, as native compilers are optimized for a single target architecture, whereas cross-compilers are optimized for portability.

Native compilation also can bring up some drawbacks, such as:

- Every additional platform for which a software project should be built requires an additional native build system. This makes hardware setups more complex and resource-intensive.
- The automation of the build process may get more complicated due to different build system control mechanisms.

2.3.2 Cross Compilation

In comparison to native compilation, here the build and target system may differ, either by different hardware architectures (e.g. building a binary for an *ARM* architecture on an *x86* computer system) or on a software level by different operating systems (e.g. different Linux distributions or building a Mac OS X library on a Linux installation).

As cross-compiling binaries separates the build environment from the target environment, it brings in some advantages, such as:

- Cross-compiling software for a multitude of target platforms only requires one physical build machine, saving costs and complexity in a hardware setup.
- A lot of target systems, especially embedded devices and smartphones provide only a limited set of resources. For that use-cases cross-compiling software on build systems with better performance save development time and ease the use of build tools.

Bringing a specific programming language compiler to a new system, often includes cross-compiling the language support from another architecture on which the compiler already exists. Under certain circumstances this process is called *bootstrapping* [App94].

There are also some downsides of cross-compiling software:

Setting up a build environment for cross-compilation is more complex than native compilation [MG03]. In order to build functional programs, many tools and files besides the compiler are required. Build suites like GNU Compiler Collection (GCC) need equivalent target platform tools for the compiler, assembler, linker, binary tools, header files and library dependencies.

Due to the complexity in software projects cross-compiling may sometimes not be the appropriate approach, especially if platform-depended tools for source code generation or code parsing are needed.

Creating software projects that can be cross-compiled require additional setup effort and maintenance, even though tools like *GNU Autotools* may be used.

Popular compilers like *GCC* or *LLVM/Clang* already include cross-compilation support, tool-chains for popular target operating systems can be installed with package managers. Development environments like *MinGW*[16] allow cross-compiling Windows applications from Unix-based build systems and the other way round.

2.3.3 Hardware Virtualization

Section 2.3.1 references a possible downside of native compilation: every target platform requires a separate native build system, making hardware setups more complex. A special setup for native compilation is the compilation in virtualized hardware configurations that can eliminate that argument.

Virtualization is the mechanism to abstract hardware and system resources from a given host operating system [YHB+11]. From the abstraction layer, often referred to as *Hypervisor*, one or more virtualized operating systems, called a Virtual Machine (VM), can be started in parallel. The VM systems do not communicate with the underlying hardware directly, instead they use an emulated hardware layer provided by the hyper-visor.

This abstraction comes with a couple of advantages, such as:

As already stated, hardware virtualization can combine the advantages of both, native compilation and cross compilation. It allows to reduce hardware complexity but keeps the concept of native tool-chains without cross-compilation requirements.

[16] http://mingw.org. Minimalist GNU for Windows.

Because virtualization drives a multitude of different operating systems on a single host machine, hardware economies of scale can reduce the costs of hardware setups (both physically and personnel).

Beside operating cost, virtualization also adds benefits regarding security, reliability and availability of systems. As VMs can be seen like other user-level applications isolated from the actual hardware, it is possible to start, stop and modify or reinstall those systems without affecting any other VM instances or the underlying system itself on the host machine [AM11].

The following arguments should be taken into account when deciding for virtualization for software builds:

As hardware for guest operating systems in VM instances is only emulated there might be performance drawbacks compared to native compilation on the host system itself. The performance may depend on the implementation of the hyper-visor (software or hardware virtualization), the number of running instances and the load on each instance.

Virtualization can be implemented on a software or hardware level. For high-performance hardware virtualization implementations, hardware costs may be higher than for usual setups.

A common argument is that it is possible to save licensing costs when running virtualized operating systems. Even if it is possible to save costs for hardware licensing[17], costs for operating system licenses or the build software that is executed may apply like in a pure native compilation setup. Additional costs may result from virtualization layer licensing.

Operating system licensing terms may restrict or forbid to run a system on a virtualized environment. As an example, Mac OS X licensing terms from Apple forbid to install the operating system on other than Apple-based hardware and it is only allowed to use a virtualized system for development purposes or personal usage[18].

2.4 Summary

This chapter covered the theoretical knowledge that is required for the further research into existing build systems and methodologies. We learned that a software release process

[17] A common model is to license hardware or a host operating system depending on the number of processor cores. To save costs, it is possible to use a multitude of logical units on a small set of physical cores with hardware virtualization techniques.

[18] http://images.apple.com/legal/sla/docs/OSX1011.pdf. Software License Agreement for OS X El Capitan.

does not only consist of building a specific binary but also covers steps such as source code management, testing and distribution.

Source code management helps maintainers to collaborate on the same source code base, which acts as input for every new release. The main step of building a software transforms that source code into an executable file format, including steps for preprocessing, compilation and linking. With testing it is possible to determine the quality grade of a software and to check if the implemented system fulfills the desired goals. This testing can be done on different levels, from unit testing to acceptance testing. Finally, the distribution step covers the release of software through physical data storage mediums or modern ways involving distribution over the Internet with manual downloads or through package managers.

These individual steps already show that a software build is a complex process that gets challenging when supporting different platforms. With the help of state-of-the-art build tools, it is possible to semi-automate certain tasks during compilation. An example for such build tool is *GNU Make* that uses a custom configuration file to determine the compilation steps needed for a certain software project. Even though such build tools support certain steps of the build process, there is still improvement for multi-platform projects.

The introduced build concepts give an overview for later discussed build system approaches that try to overcome those challenges. With the help of native compilation, cross-compilation or hardware virtualization it is possible to create binary builds for different target platforms, where the host platform and the target platform to not necessarily have to match.

CHAPTER 3

Development of the Framework

The success of a software project relies upon stable and in-time releases. In most cases a release is the only user-visible and relevant result and thus is shaping the public image of a software. Especially in the context of open source projects, the quality and frequency of releases is important to increase the user adoption rate and to maintain an active developer community contributing to the project [WP09].

Because of that, the design and decision for an appropriate build system tailored to specific needs is crucial for a successful software project.

Even though build systems are usually designed to support a high rate of automation, a prior research study has shown that build system maintenance may add an additional overhead of about 12% to 35% to software projects [Epp02].

This chapter therefore describes different orthogonal dimensions in the decision making process and assigns different weights for each dimension. Afterwards we create a link between these decision dimensions and described build approaches.

This allows to derive an appropriate build system approach, depending on the characteristics of several decision dimensions of a software project. The given summary is a conclusion of theoretical considerations and deduced from practical learnings from literature, where quoted.

The final result of this chapter is a framework for implementing a build system based on one of the introduced build system approaches. The framework is structured according to the software release process introduced in Chapter 2. Thus, it is designed to create the binaries, run tests and to collect and distribute binaries. Not included is the action of source code management, as a concrete framework implementation is often part of the project source code and therefore kept in synchronization within the main code repository. This framework is later on used in Chapter 2.2 to set up a build system in form of a case study.

3.1 Decision Dimensions

Depending on technical and non-technical traits and features of a specific project, different build system approaches may be taken into consideration.

The decisioning can be supported by answering the questions from the following categories:

Complexity:

- Does the software project use external libraries?
- How long does a manual build of the project take?
- Are there any external dependencies related to a release?
- Which build tasks take the longest time and which tasks may be automated?

Release Cycles:

- How often are new versions released?
- Which event or circumstance triggers a new release?

Target Group:

- For whom is the software creating value?
- Who uses the software and how often?
- Are there any time or date restrictions for releases?

Target Platforms:

- Which platforms are supported by the software project?
- On how many and which platforms is the software actually in use?
- Are there any additional platforms that should be supported in the future?

Organization:

- Who is in charge of building and distributing new releases?
- Are there any resources available for configuring and maintaining an automated build system?

We discuss these dimensions in more detail below.

3.1.1 Complexity

The complexity of software projects is observed on different perspectives in common bibliography. Traditionally, the complexity is measured either by measurements for the software product itself or by taking characteristics of the software development process into account [LB06]. Derived metrics include characteristics, such as the size, complexity, reliability or quality.

This already implies that software complexity as monolithic key performance indicator cannot be defined by just looking at the product. It also implies looking into several areas of software project management and organizational concepts, including metrics from areas of studies such as time management, cost management, human resources or project management tool support.

This section however focuses on main metrics from a technical viewpoint of complexity of the software itself. We restrict the discussion to structural complexity such as the design and structure of the software itself. The conceptual complexity, that refers to the difficulty in understanding the system and its requirements or the source code itself, and the computational complexity during the build process not looked into detail here [Fit09].

Size

A simple but effective characteristic for the complexity of software is the *size* of the project. In most cases, the size is directly related with complexity of the software.

There are different sizes that can be defined and used for measuring complexity:

- Number of source or input files
- Lines of code or other metrics like classes or functions
- Number of build results or output files

A commonly used software sizing technique is counting the number of Lines Of Code (LoC) of the source code [Par92]. LoC represent the count of the executable lines of code without comments or empty lines. This metric gives a rough estimation, it does however not take architecture or code quality into account. Also, the number of lines of code highly depends on the underlying programming language and incorporated software libraries. A software implemented in a high-level programming language can have less LoC but may be way more complex regarding architecture.

Because of that, another common unit of measurement is a *function point* [Par92], that describes the amount of business logic functionality, without taking the language-specific characteristics into account.

Dependencies

There is another flaw in using the LoC measurement for the complexity of software projects. A large but well-designed and structured software with calculated cost and time requirements can be less complex than a project with less LoC that has a highly integrated and proprietary design and restricted time and money budgets.

It therefore is reasonable to have a look into another characteristic of software projects: the amount of type of *dependencies*. A project can have several dependencies:

- Source code dependencies
- Build tools dependencies
- Process dependencies

Source Code Dependencies

Source code dependencies summarize external libraries or operating system functions that a software links against. Both dependency types can either add or decrease the level of complexity.

Functionality of operating system libraries or external libraries, often called *standard libraries*, usually abstract low-level functionality to more reusable, standardized modules and therefore decrease the complexity of a software. External libraries can however also be used to solve a very specific problem on top of standard functionalities and therefore increase the level of complexity.

The grade of complexity added by source code libraries also varies depending on the provision. Pre-compiled, ready to use code libraries for all supported architectures are easier to use than dependencies that are only available as source code. Latter therefore have to be compiled themselves, adding another layer of complexity in the build process.

Build Tools Dependencies

Build tools, as described in Section 2.2, often rely on other software and tools to create an executable binary from a software project. Every step in the software release process Section 2.1 can add additional dependencies on external tools. As an example, source code retrieved from a VCS may need a tool to automatically pull the latest changes in a project and add them as input to the build tool.

Additional complexity can be added depending on whether the tools are called from the build tool suite itself or as part of the overall process before an actual build is triggered.

Process Dependencies

Complexity can also be increased by the software release process itself. These added dependencies are not related to the technical build steps itself but to the organizational processes.

As an example, test results from automatically performed quality testing steps need to be accepted by a responsible person in the project team. This example can be generically used for every step within the release process.

For the design of the framework of this thesis, process dependencies are not further considered.

3.1.2 Release Cycles

Designing and executing a build system implies that a software is released more than once in the lifetime of the project. Usually open source projects follow well-defined release cycles for distributing new versions. Triggers for new releases are mostly time-driven or feature-driven.

Triggers for Release Cycles

Time-based releases distribute new versions in a fixed timetable. As an example, the open-source project *GitLab*[1] defines a regular release cycle for new versions on every month on the 22$^{\text{nd}}$. Time-based releases usually define not only the specific release date but a set of dates for a release:

1. **Feature freeze** defines a specific point in time after that no new features are added to the release.

2. **Alpha release**, **beta release** and **release candidate** dates are used to build special pre-releases targeting testing and bug fixing.

3. **Final release** defines the actual, publicly available software release.

Feature-based release cycles are not bound to specific release dates but release new software versions as soon as all planned features are implemented and ready for deployment. Feature-based cycles incorporate a much higher risk of delayed releases in comparison to time-based ones. They however can improve the quality of specific releases and allow a better management of planned features. An example is the used case study DLVHEX in Chapter 4 of this thesis, for which new releases are created after a specific set of new features got implemented.

As a special form of release management a hybrid model between time- and feature-based releases is also possible. As an example, the VCS *SVN* used a hybrid-model where developers would wait around six months to determine which features were completed and use those to define the next release [WP09].

[1] http://doc.gitlab.com/ce/release/README.html. GitLab Documentation.

Advantages of Fast Release Cycles

The resulting release cycles for different software projects can vary from cycles of more than a year down to a daily release. Traditional software projects like operating systems usually came up with release cycles of many years in the past. Other software projects do have an automated build system in place that provides daily builds of the software (often called *daily build* or *nightly build*), regardless of the current feature implementation state [KDZA12].

These fast release cycles bring a couple of advantages, such as:

- Identified failures in a software (*bugs*) can be fixed faster in more often distributed releases. End-users therefore do not have to wait for a long time until they can use the software without found errors.
- Releases developed in fast release cycles are usually adopted faster by users, implying that outdated versions with potential security flaws are replaced with up-to-date versions more often.
- End-users do not have to wait long times for an updated version of the used software, which is especially relevant if software uses external dependencies for which new versions are available too.
- Faster and more often releases offer better marketing opportunities for software organizations and probably attract new users due to evident activity in a project.

3.1.3 Target Group

Every software project is designed to solve problems in a specific domain by different target groups with varying areas of expertise. Therefore, depending on the target group a software has to fulfill different requirements. Software that is distributed as source code can most probably be built to an executable binary by software engineers without further documentation. The same software however might need additional resources and guidance if it should be used by a completely different target group. Even better, such software is then already distributed in binary form, often requiring a build system for new releases.

Beside the categorization of target groups by areas of expertise, the estimated number of users, i.e. the size of the target group, is another metric for deciding for a build system. Software that is only used by a couple of people, in most cases the authors of the project themselves, does not necessarily require a dedicated build suite, software projects that are however used by hundreds of people can however take advantage of such a system.

3.1.4 Target Platforms

Sections 2.2.2 and 2.2.3 already describe that software projects can be built for different target platforms. The number of supported platforms can also affect the choice of an

appropriate build system. Supporting only one platform can lead to a light-weight build system, running on a single machine. However, when a projects needs to support a multitude of platforms, a more complex build setup may be required to support the software release.

3.1.5 Organization

Most open-source software projects start small from an initial idea. In that case, the development team is often only built up of a single person or a small group of developers.

As projects grow, the project organization is most likely structured in several sub-teams with different responsibilities. Larger software projects then consist of several core development teams, teams for Quality Assurance (QA) or dedicated release management teams.

3.2 Build System Approaches

Depending on several factors of a software project that are grouped in decision dimensions in Section 3.1, automating the build creation steps can bring several advantages to the release process, such as:

- Releases may be built and distributed in a faster way, as no manual or blocking steps from the human factor are involved.
- Releases may be of better quality due to automated testing and pre-defined build step order and execution. Automated builds also prevent the risk of skipped or forgotten steps that might occur from a manual build.
- Development resources are freed up for the actual project development and not allocated for building and releasing the software.

Given these advantages, it is recommended for each software project to look into build automation. The automation grade of such systems can be categorized in different levels, described as *build system approaches*:

1. **Manual Creation**: Create binaries as a manual action for every release.
2. **Build Scripts**: Design and execute scripts to semi-automate specific steps within the binary creation process.
3. **Continuous Integration**: Deployment of continuous integration services to fully automate builds run by triggers or in periodical intervals.

The stated approaches are described in more detail in the following sections.

3.2.1 Manual Creation

Many software projects make use of build tools as *GNU Make* referenced in Section 2.2 to configure and create builds. If those projects do not link to external dependencies and do not bundle additional build artifacts that need to be created with tools independent from the build suite, executing the build suite commands one by one may be sufficient for creating a new software release version.

In the most basic example, building a software with *GNU Make* incorporates two or three commands:

1. *configure* to resolve dependencies and prepare a build instruction file, called *Makefile*.
2. *make* that uses the *Makefile* as input and processes all defined build steps.
3. *make install* that optionally installs the built software to a predefined location on the build system or packages the build for distribution on other systems of the same architecture.

For cross-platform software projects those commands need to be executed on every supported target platform, unless approaches like cross-compilation are implemented (see Section 2.3.2).

3.2.2 Build Scripts

In many projects building a software requires additional steps, such as compiling or installing dependencies, creating build artifacts like documentation or copying the final build results for distribution. These steps are mainly identical actions for every release and can sum up to a couple of thousand single commands for every build. It is therefore common practice to automate them with the help of custom build scripts.

Those scripts are implemented in different languages for different platforms. Most build scripts are a set of UNIX terminal commands that are collected in *shell* scripts. Another approach is to use a scripting language like *Ruby*, *Python* or other that allow to use a script independent from the underlying platform. Build scripts are then executable on every platform for which a script interpreter exists for the specific language (as an example, Python scripts can be executed on all major operating systems like Linux, Mac OS X or Windows).

Most of such scripts have in common that they are designed and written individually on a per-project basis, therefore the re-usage potential between projects tends to be very small. That is however not necessarily a disadvantage: the approach allows to quickly set up build systems that fulfill the requirements, especially useful for small to medium-sized projects.

Build scripts usually can take a list of parameters, so that specific builds can be configured without the adaption of a script itself. Parameters can then define a build version number,

the used version of a specific dependency or used features of a project that should be incorporated into a release.

Depending on the used scripting language it is possible to structure a script into smaller pieces (like functions) and split it up into multiple files. Such an approach makes maintenance easier and allows to scale a build script parallel to the linked software project.

3.2.3 Continuous Integration

The first two approaches have in common that both build actions are usually triggered manually at a well-defined point of time. This might be suitable for small to medium-sized projects and teams. However, as projects grow, the requirements for a build system also change. As an example, certain source code changes or updated dependencies can break a build that might lead to a delayed or failed software release. Testing the software with unit tests and integration tests and building intermediate builds in-between large releases can reduce the risk of failed releases. For that an agile build system approach has been broadly adopted by open-source communities, often referred as Continuous Integration (CI) [DMG07].

CI is the process of building a software with every change committed to the source code of a project. Changed parts are immediately tested to ensure that the code base remains stable [ERP14]. The term *continuous* refers to the fact that testing and building a software is done automatically as soon as a change happened, instead of a time-based or manually trigger as seen with manual compilation or build scripts.

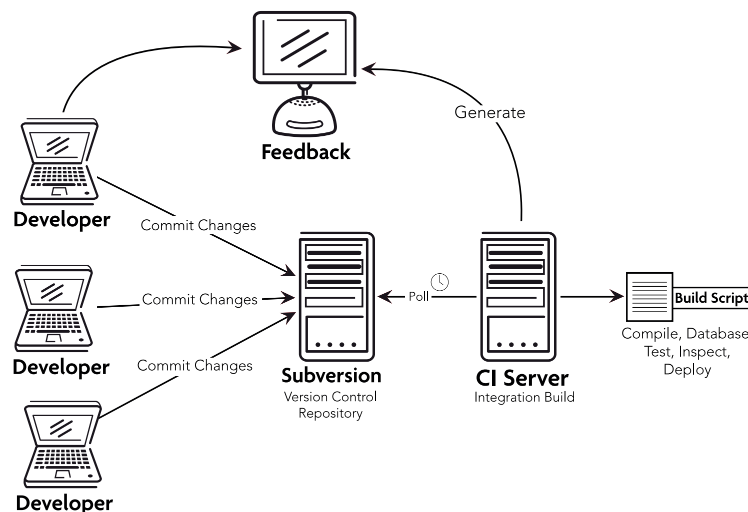


Figure 3.1: Continuous Integration Overview

As illustrated in Figure 3.1, a CI system (*CI Server*) is usually connected to a source code control system and polls periodically for changes or gets notified with a trigger from

the SCCS itself about new changes [DMG07]. As soon as the CI system starts a new build from a trigger an automated build script is called and the results from of the build feedback are reported back.

Systems for continuous integration in open-source software projects are often designed in a *master-slave* architecture. For that approach a central server (the *master*) schedules and coordinates builds between one or more connected clients (the *slaves*) that execute the actual builds. That approach allows to run multiple build concurrently across different platforms. The results are then collected by the master and provided within a centralized place [BW12].

From a practical perspective, continuous integration systems are mainly used for creating nightly builds in fast release cycles, as they allow to fully automate building and testing for a small subset of changes pushed to such a service. The following section covers that in more detail.

3.3 Linkage of Dimensions and Approaches

In theory every software project can make use of automated build systems and CI services, as they provide a bunch of advantages as mentioned in the section above. However, when looking at the big picture, setting up an automated build system may not always be the most efficient solution for every single project.

The elementary questions in the decision process are:

- How long does setting up and configuring an automated build system take?
- Compared to a manual build, does the time needed for setting up an automated system pay off compared to manual builds?

As there are a lot of influencing factors, these questions cannot be answered empirically in a generally valid and adequate way. Thus the following assumptions are based on theoretical considerations and from a qualitative view on open-source software projects, where applicable.

3.3.1 Dimensions to Approaches

The linkage is divided based on the introduced decision dimensions and gives an overview of resulting approaches.

Complexity

It stands to reason that an increasing project complexity suggest the introduction of builds scripts or a CI system.

The increase in source code often only has a negative impact on the build time. Building and linking dependencies is a tedious process that often comes with lots of new issues and errors along the way. Thus, the amount of dependencies often is a bigger driver for a semi- or fully automated build process than the overall size of source code lines. The exact numbers where to switch from a manual build to build scripts or a CI system highly depend on the project constraints and need to be evaluated individually.

As an example, the free Computer-Aided Design (CAD) software *FreeCAD* lists a couple of third-party software dependencies, that add additional complexity when building the software.[2] To make installation of stable and daily builds easier, the project team set up a daily build repository that automatically generates binary packages with the Launchpad platform.[3]

Complex process dependencies can have a contrary effect. As an example, a CI system may be unsuitable if building a software involves a lot of manual steps that cannot be automated.

Release Cycles

Some open source projects do release only a few versions per year. Such projects have to assess if introducing and maintaining build automation can save a reasonable amount of time compared to a manual release.

On the other hand, larger projects on a monthly release plan or projects that also provide nightly builds greatly benefit from an automated build system with CI systems.

As an example, the 3D modeling software *Blender* follows a rather short release cycle of 8 to 12 weeks, where the release itself is done within one or two days.[4] Additionally to stable release versions, Blender provides nightly builds on a daily base. Because of that, the Blender project team decided to make use of a CI system, called *Buildbot* that performs builds automatically and provides them as download.[5]

As with complexity, there are no specific numbers of releases per year that define the selected approach. However, as a fuzzy rule of thumb, automation should be taken into consideration as soon as maintaining a build system takes less resources than doing manual builds.

Target Group

Open source projects targeting engineers or in an academic context are often distributed as source code, as end users often bring in the technical know how to build and use the

[2]http://www.freecadweb.org/wiki/index.php?title=CompileOnUnix. FreeCAD Getting the dependencies.
[3]https://launchpad.net/ freecad-maintainers/+archive/ubuntu/freecad-daily. FreeCAD Daily Builds.
[4]https://wiki.blender.org/index.php/Dev:Doc/Process/Release_Cycle. Blender Release Cycle.
[5]https://builder.blender.org/download/. Blender Builder.

software on their own. Software in that environment does not necessarily need build automation and can stick to the distribution model if no issues are known.

On the other hand, software that targets non-engineers can greatly benefit from build automation, as it is most commonly distributed in pre-built binary form instead of source code packages. Also, if there is a large group of people using the software and those are depending on stable and timely releases, build automation may be the way to go. It provides less room for errors and makes timing releases easier.

As an example, the cross-platform remote desktop software *TightVNC*[6] targets end-users connecting to remote PCs. As most people within the target group are usually not familiar with building such software on their own, the project provides pre-built binaries on the project website[7] in addition to the source code repository.[8]

Target Platforms

Software may be built for a single target platform, like a specific Linux distribution, it is also possible to release software for a multitude of platforms, to reach a large target group. Offering a software for more platforms not only makes the implementation more complex but also adds a lot of overhead during the build process. It there might make sense to set up an automated build system if more than one platform build type is offered.

As an example, the open-source flight simulator *FlightGear* provides binary packages for Linux, Mac OS X and Windows. After several issues with creating manual builds, the project team set up a *Jenkins* CI server[9], that creates a complete release after every commit and handles version number handling across all supported platforms.[10]

It should however be kept in mind that builds for different platforms also multiply the effort of setting up a build system witch each additionally supported platform. Concepts likes cross-compiling can be used to minimize the overhead of different build systems, as discussed in Section 2.3.2. To give an example, the referenced project *FlightGear* is currently working on cross-compilation support to allow Linux contributors to build Windows versions without the need for additional hardware.[11]

Organization

Especially in the starting phase of a project only a few people are working on it. Team organizations are not existent or only defined informally and loosely. In that situation almost all involved people are working on the implementation and therefore are able to

[6] http://tightvnc.com. TightVNC Software.
[7] http://tightvnc.com/download.php. Download TightVNC.
[8] https://sourceforge.net/p/vnc-tight/code/HEAD/tree/. TightVNC Code.
[9] http://wiki.flightgear.org/FlightGear_build_server. FlightGear build server.
[10] http://wiki.flightgear.org/Release_plan/Lessons_learned. FlightGear Release plan/Lessons learned.
[11] http://wiki.flightgear.org/Building_FlightGear_-_Cross_Compiling. Building FlightGear - Cross Compiling.

build the software locally on their own. An automated build system would only add overhead, bringing in no advantage for the organization.

As a projects grows, a software team is often made up of different departments for development, testing, quality assurance and marketing. These departments usually stand in relation on both dimensions, vertically (implementation to testing) and horizontally (different implementation teams for different features). In that context, build automation and continuous integration can save a lot of resources, as otherwise every team would have to build things on their own.

As an example, the Enterprise Resource Planning (ERP) software *Adempiere*[12] is developed to support several business management tasks, such as product planning, manufacturing controlling or inventory management. Beside build level tests performed with test scripts, testing functionality of a new release is done by a team of volunteers with business management background.[13] This team is provided with automated builds from a release branch and can therefore test the software without the need of waiting for manual builds created by the development team.

3.3.2 Decision Weighting

Until now, we treated the different decision dimensions and their connections equally, so that no one can have precedence over another one or a decision dimension can overrule the result of the other dimension. When evaluating different dimensions in practice, one often finds out that those dimensions can overlap and their individual result can influence each other.

There is however no proposal given which dimension should be looked into the first and in which order all other dimensions can be evaluated. To make the process more streamlined it is possible to weight the dimensions.

Given the example that for most projects the complexity plays an important role, it can be weighted higher than others like the release cycles or the organization. As another example, the quantity of target platforms can be a less important decision dimension if the development team works on all of the supported platforms and is also able to create those specific builds.

Based on that, it may be appropriate to order the dimensions by their importance on a per-project base.

3.3.3 Summary

The given open-source project examples in combination with the theoretical considerations and reflections serve as a basis how decision dimensions and build system approaches can be linked. Table 3.1 summarizes the key findings, illustrated in form of a matrix.

[12]https://sourceforge.net/projects/adempiere/. ADempiere ERP Business Suite.
[13]http://wiki.adempiere.net/Software_Development_Procedure#Testing_ADempiere. Testing ADempiere.

Dimension		Manual	Script	CI
Complexity	Low	X		
	Medium		X	
	High		X	X
Release Cycles	Low	X		
	Infrequent		X	
	Often			X
Target Group	Tech	X	X	
	Non-Tech			X
	Small	X		
	Medium		X	X
	Large		X	X
Target Platforms	Single	X		
	Multiple		X	X
Organization	Loose	X	X	
	Structured			X

Table 3.1: Decision Dimensions and Build System Approaches Overview

3.3.4 Application of Approaches in Practice

The previous section outlined possible relations between different dimensions of a project and a resulting build approach. This model is mainly designed from a theoretical viewpoint. However, in reality, a software project may change its build approach over time.

It is common practice to start with a simple build scripts that automates a few build commands into a single call. Over time, a build script may be extended to fulfill more tasks or target more than the initial target platform. This point in time can be seen as a transition from the approach of manual creation to the incorporation of build scripts.

If those scripts are later on executed from a dedicated build hardware and called from different time or event triggers, the approach transitions from a set of small build scripts to fully automated builds with continuous integration systems.

3.4 Framework Design and Implementation

Section 3.2 introduced three different approaches for building up a build system for a given software project. In this section we first design a framework on a conceptual level and then discuss a generic implementation, which can later on be used for specific software projects. The framework implements a system with *build scripts*. Besides a basic level of automation for most small to mid-sized software projects, build scripts furthermore can easily be adapted for usage in a CI system approach, as also stated in Section 3.3.4.

The main goal of the given framework is to provide basic features for building a software from the technical side. Thus, other build artifacts such as documentation or test reports are intentionally out of scope and not processed. The introduced architecture however allows to extend the framework to also incorporate the creation of related artifacts, such as documentation, in future work. Such tasks then complete the software release cycle introduced in Chapter 2.

3.4.1 Architecture

One of the main goals of the designed framework is to provide a generic solution that provides a high level of reusability and flexibility and is able to create builds for a variety of software projects. To follow this goal, the framework is designed around different static building blocks that interact in a specific command flow within the lifecycle of the execution, as described in the following two sections.

Static Building Blocks

Build scripts fulfill a couple of actions in the lifecycle of a build. As an example, a good approach for a build script is to check for installed dependencies, build the actual software, validate it against defined test cases and then deploy the successful build to a specified location.

To adapt that best-practice and make the designed framework usable in a more modular way, the execution steps are actually split up into different sub-modules, assigned to four main stages in the build lifecycle. These building blocks are outlined in Figure 3.2.

Figure 3.2: Four main stages in the build lifecycle

The modular approach allows to run steps from certain stages independently from others, to restart a certain stage after an error or skip a stage at all (e.g. when dependencies are already installed from an operating system snapshot image in a virtual machine).

Each stage of the framework defines a default behavior that fulfills the requirements for each individual stage. In practice, projects however differ in several aspects of the build process. To take such differences into account, the framework contains two conceptual components that control or adapt the build process, defined as *configuration variables* and *execution blocks*, respectively.

The *configuration variables* component defines various project-dependent options that control the execution of the framework implementation, such as log output visibility or required build dependencies. On the other hand, the *execution blocks* serve as hooks, which allow executing custom code at specific points in the build process. This creates

the possibility of injecting commands that may strongly depend on the current project and thus cannot easily be represented by variables. Examples include patching specific source files before building or downloading, extracting and preparing additional resources from the Web. For such use cases the concept of *execution blocks* can be applied. An execution block implements an algorithm in a procedural way. Instead of providing a set of configuration variables, commands can directly be added to an execution block, which is then called from the main execution process of the framework.

Figure 3.3 summarizes all introduced static components of the framework.

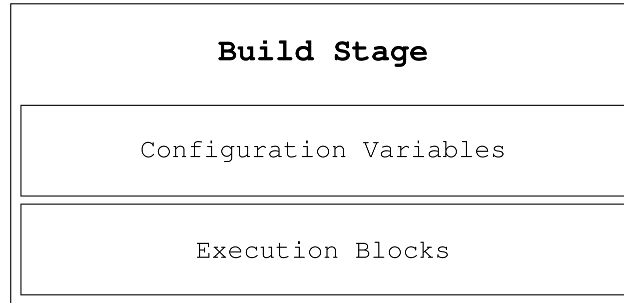


Figure 3.3: Framework Components for each Stage

Lifecycle and Command Flow

When executing the framework, the default implementation reads in the configuration variables to control the build steps and calls the execution blocks for specific subroutines, as illustrated in Figure 3.4.

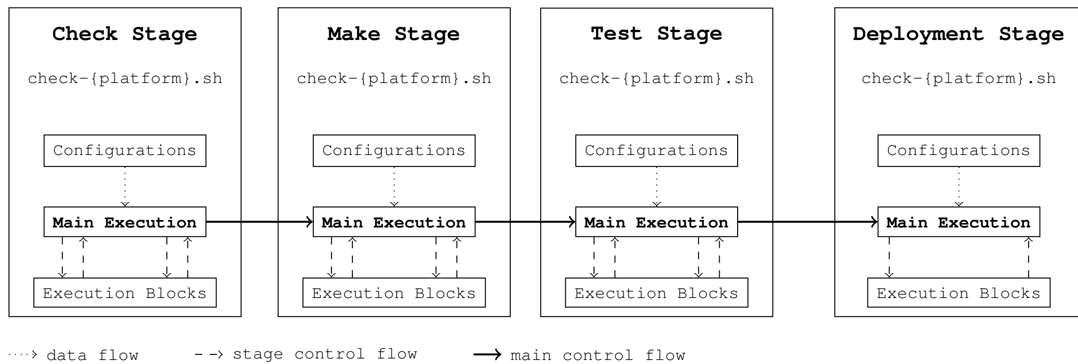


Figure 3.4: Stage Execution Flow

Informally, instantiating the framework for a specific project therefore means adapting the build by setting a set of configuration variables and altering the implementation by adding custom build commands to a set of execution blocks. Depending on the stage, a concrete implementation might have none, one or several configuration variables and execution blocks. The stage definitions themselves and the main execution flow

are independent from the specific implementation and therefore an integral part of the designed framework command flow.

3.4.2 Implementation

The individual stages of the framework are implemented in a modular way. As a result, it is possible to execute single step of a specific build stage more than once and without any side effects due to an unclean build environment. In practice it is therefore possible to re-do a certain step if it failed or skip certain other steps, such as building dependencies from source, for a subsequent build.

The implemented build system framework is designed as a collection of *shell scripts*. These scripts consist of individual calls to native system functions and programs. As these calls are tight to the native underlying platform, scripts are not platform-independent. They are however implemented to only use shell-equivalent commands and can therefore be run on every environment such as shell, bash and others on POSIX-like UNIX operating systems like a variety of Linux distributions, Mac OS X or OpenBSD.

The framework supports three target build systems: Ubuntu Linux, Mac OS X and Windows. Ubuntu Linux and Mac OS X are natively compiled. Both platforms can execute UNIX commands within a command line shell interpreter, which allows great re-usage of program routines or implemented scripts. For Windows builds the approach of cross-compiling was chosen. This allows to also use the same build tools as for the Linux target and does not require additional software licenses or maintaining a third build environment.

The framework is designed to build C++ projects that make use of *GNU Autotools*, as introduced in Section 2.2.3. If a certain project makes use of other build tools it is however possible to adapt the scripts, without changing the main idea and concepts of the framework itself.

As the framework is a starting point for a certain software project, the scripts of the framework need to be copied over to a specific project and adapted to the project-specific build steps. It is advisable to save the scripts in the main repository of the project to have them in sync according to the current state of the project.

In the context of the framework implementation, each build stage illustrated Figure 3.2 corresponds to a single shell script. The scripts are postfixed for every supported platform, e.g. the build sub-script for platform Linux is named *build-linux.sh*, the script for the deployment stage for platform Mac OS X is named *deploy-osx.sh*, and so on. So in total, the framework defines a maximum of 12 scripts, as listed in Table 3.2.

All individual scripts have in common that they look for and process errors. If an individual call within a script fails, a descriptive error message is printed and the script exits with a return value not equal to zero. This allows to chain the scripts of the different stages in a generic main script that exits on appropriate errors. Moreover, an optional configuration activates verbose build output for further debugging into the scripts.

	Linux	Mac OS X	Windows
Check	check-linux.sh	check-osx.sh	check-windows.sh
Make	make-linux.sh	make-osx.sh	make-windows.sh
Test	test-linux.sh	test-osx.sh	test-windows.sh
Deployment	deploy-linux.sh	deploy-osx.sh	deploy-windows.sh

Table 3.2: Build Scripts

Before looking into specific implementation details, we recap the main two building blocks, *configuration variables* and *execution blocks*, with the help of some examples.

As already stated, configuration variables define project-specific options that control the execution. Options are defined hard-coded within the file or passed as command line argument when executing the framework, depending on the actual use case of the option. The following listing demonstrates build dependencies, given in hard-coded manner within a script file:

```
1 required_dependencies ="sed git buildessential autoconf"
```

On the other hand, execution blocks serve as hooks to execute custom code at specific points in the build process. As an example, a common approach is to use the *stream editor (sed)* utility to search for a given string and replace it with another one. The following listing adapts a project build file to use a cross-compiler instead of the platform-default compiler:

```
1 sed -i 's/g++/i686-w64-mingw32-g++/' Makefile
```

Another application is downloading and using additional resources from the Web. The following example demonstrates an execution block that downloads a collection of patches from the Web, extracts the archive to a temporary directory on the build machine and applies the patches to a project file using the *Git patch* command:

```
1 wget https://example.org/patchset.tar.gz
2 tar xzf patchset.tar.gz
3 pushd patchset
4 find . -type f -iname "*.patch" -exec patch ../Makefile {} \;
5 popd
```

It is now time to have a closer look into the different *configuration variables* and *execution blocks* of each individual build stage.

Check Stage (*check-{platform}.sh*)

Section 3.1.1 describes that builds may have two types of dependencies: dependencies of the software itself and dependencies of the build environment. The check stage steps ensure that both dependency types are met before triggering the actual build.

Usually, dependencies are installed as pre-built binaries from package managers. The framework uses *apt-get* on Linux hosts (therefore for Linux and Windows builds) and *homebrew* on Mac OS X hosts.

For dependencies that do not exist in binary packages, the framework additionally defines another intermediate stage, called *make-dependencies.sh*. This stage can be used to compile custom dependencies from source or install dependencies from other package sources, such as a Linux *.deb* installation file.

The script defines three **configuration variables**:

install-dependencies Controls whether the script should automatically install the missing dependencies from the platform-specific package manager, which is *apt-get* on Linux or *homebrew* on Mac OS X, respectively.

If the configuration is set to *true*, the list is passed to the package-manager install command, which again translates to *apt-get install <dependencies>* or *brew install <dependencies>*, respectively.

If the configuration is set to *false*, the script just exits with a hint how to install the missing dependencies but does not try to install the dependencies on its own.

This configuration is useful if used on a dedicated build machine setup for automated builds but might be skipped if run on a personal PC where the package management is done manually by the end-user.

required_dependencies The list of dependencies, as named within the platform-specific package manager, which are required in order to build the software project.

A minimal dependency list for building a C++ project can look like the following:

```
1 required_dependencies ="sed git buildessential autoconf"
```

This configuration checks the platform-specific package manager if every single package is already installed (i.e. if the executable can be found within the provided *PATH* environment). If *install-dependencies* is set, the script then tries to install missing packages automatically, otherwise the missing dependencies are printed out with additional hints for manual installation.

forced_dependencies This list of dependencies behaves similar to the list of *required dependencies* but forces the installation of a dependency from the platform-specific

package manager, even if it was already installed and found within the *PATH* environment variable.

This is useful for dependencies that might already be pre-installed on a system but need to be upgraded to the latest version or if the package manager provides an alternative package with another feature set of a specific tool or dependency.

As with *required dependencies*, the installation is only done if the *install-dependencies* configuration is set.

The script defines one **execution block**:

make_dependencies{} This step defines custom build steps for building dependencies from source.

Often, dependencies are not available via the used package managers or a build requires a newer version of a dependency that is currently not available via package manager. Another use-case for building a dependency from source is that package manager libraries are only provided as shared libraries but dependencies should be linked in statically. Also, as-designed cross-compiling software for Windows usually requires cross-building dependencies from source too.

The following example downloads, builds and installs the *libcurl library*[14] for Linux targets:

```
wget http://curl.haxx.se/download/curl-7.46.0.tar.gz
tar xzf curl-7.46.0.tar.gz
pushd curl-7.46.0
./configure
make
make install
popd
```

If there are multiple dependencies that need to be installed from source it is recommended to use another external script, which encapsulates all commands and keeps a clean structure of the check stage. The following example calls an additional script for building all required Windows build dependencies:

```
./make-windows-dependencies.sh
```

After executing this script, the system is pre-configured with all dependencies for starting a build of the specific software project.

The listing of the entire script can be found as Appendix A.1.

[14] https://curl.haxx.se. curl.

Make Stage (*make-{platform}.sh*

The script of that build stage performs all steps required for building the actual software. As the framework is designed to be usable for projects that use *GNU Autotools*, a default build command chain of *configure* and *make* calls is suggested.

The filename of the script is based on *GNU Autotools*, where the *make* command processes the actual build steps defined in a build file.

The script defines two **configuration variables**:

verbose If this configuration is set, the script outputs the entire log output from build tools to *stdout*. If omitted, the build output is hidden and only some informative log statements are printed to *stdout*.

Omitting the build output can be useful to observe the process of an individual build. If build errors occur, it is recommended to set the option, which allows a more detailed look into the build steps themselves.

configure_parameters A usual build process of the framework involves executing the commands *configure* and *make*. If this configuration is skipped, a default build, as defined by the project, is created. Often a build should however be customized depending on a debug or a distribution build. Thus, providing options to the *configure_parameters* configuration passes those flags directly to the *configure* command.

As an example, one can define the following option in order to build a library with position-independent code, providing the *-fPIC* compiler flag:

```
1 configure_parameters="CXXFLAGS=-fPIC"
```

The script defines two **execution blocks**:

setenv{} Usually, a software build requires some environment variables to be set up correctly. Such operations are collected in the *setenv* execution block.

The following example sets the *PYTHON_BIN* environment variable before executing the build steps, which usually controls the selected Python version within *GNU Autotools* configuration:

```
1 export PYTHON_BIN=python2.7
```

patch{} Additionally, the make stage is the optimal place to put in custom commands to patch certain source files or build configurations. For that purpose it is possible to define patch operations in the *patch{}* execution block. The example shown in the listing below shows how to comment out a configuration line in a *Makefile* file, using the *sed* command.

```
sed -i "" "s/liba_la_LIBADD/#liba_la_LIBADD/" Makefile.am
```

It should be mentioned that patch tools used within this execution block, must also be installed within the check stage.

After executing this script, the project binary is built and ready for the test and distribution step.

The listing of the entire script can be found as Appendix A.2.

Test Stage (*test-{platform}.sh*)

Projects that provide tests to check the specific implementation for functional correctness can use this stage to perform additional testing.

As the tests operate on the built binary directly, the test stage is implemented right after the make stage. This is in contrast to unit tests, discussed in Section 2.1.3, as these are performed during the build or in a separate build.

Usually, tests are performed with a call to *make check* or *make test* with *GNU Autotools*. If needed, different tools or testing suites can also be executed within this stage.

If the project does not incorporate any testing steps, this stage can be left blank or skipped entirely.

The script defines no **configuration variables**.

The script defines one **execution block**:

runtests{} This step defines custom test commands that should be executed after a successful binary build from the build stage script.

Usually, tests are defined as custom make target and can be run by providing *make check* or *make test*:

```
make check
```

It is however also possible to run other test tools, installed from the check stage, or call another script that defines additional tests, as the following example demonstrates:

```
./run-linux/tests.sh
```

As mentioned, if the software projects does not provide any tests, the *runtests{}* execution block can be left empty and the framework proceeds with the subsequent stage.

The listing of the entire script can be found in Appendix A.3.

Deployment Stage (*deploy-{platform}.sh*

After the successful build stage the software binaries are ready for distribution. The deployment stage collects the built binaries and can copy or upload them to one or multiple target destinations.

For the most basic implementation, the framework copies over the given filename to the given location. It is however also possible to define custom steps for deployment, such as copy or upload steps with tools like *cp* or *rsync* to mounted remote volumes or via FTP, SCP, SMB and NFS. Other possibilities are uploading via Hypertext Transfer Protocol (HTTP) APIs to external services or pushing the builds to a VCS.

The script defines two **configuration variables**:

out_file This configuration defines the relative path to the binary being built from the previous make stage.

An example could look like the following:

```
out_file="src/thebinary"
```

The variable can be used in the described *copysteps{}* execution block below. If it is not used, like in the case of an automated deployment from a project-specific script, the configuration can be skipped.

destination_path Defines the path where the built binary should be copied to. This variable can also be used in the described *copysteps{}* execution block or alternatively left blank.

An example to copy the binary to a mounted network directory could look like the following:

```
destination_path="/mnt/shares/builds/theproject/"
```

The script defines one **execution block**:

copysteps{} This step defines custom copy or deployment steps for the binary given in the *out_file* configuration. During implementation of that execution block, it should be made sure that the specific copy command binary was previously installed with the check stage script.

The following example copies the binary to a local directory, using the defined configuration variables *out_file* and *destination_path*:

```
1 cp $out_file $destination_path
```

Another example uses the *Secure Copy (SCP)* tool to copy the binary to a remote computer, again using our defined configuration variables *out_file* and *destination_path*:

```
1 scp $out_file user@example.com:/home/bobbuilder/theproject/
```

After executing this script, the built binaries are distributed to the desired destination and are ready for usage.

The listing of the entire script can be found as Appendix A.4.

3.5 Summary

Before being able to implement a concrete build system framework we first had to define the two terms of *decision dimensions* and *build system approaches.*

Decision dimensions are orthogonal considerations that help categorizing a reviewed project in respect of its characteristics. The first dimension looked at the complexity of a software, in terms of the overall size and dependencies. For that we showed common metrics for defining a size, such as lines of codes, and analyzed dependencies regarding the source code, build tools and process dependencies. Another decision dimension covers the usual release cycle of a software. We defined different release stages like beta releases up to final releases and introduced release triggers with a date-driven or a feature-driven approach. We also brought up advantages of fast release cycles, including software quality through bug fixing, the user adoption rate and marketing opportunities for a better market visibility in general. The decision dimension of the target group showed that software can address different target groups with different knowledge levels of building and using a software project. The organization dimension splits up projects regarding the maintenance team size and organization of engineering, testing and release management

teams. Finally, we also showed that projects may differ in the type and quantity of supported target platforms, as single-platform projects often lead to light-weight build systems, whereas multi-platform projects often require a more complex build setup.

As a next step we identified different build system approaches that are used in practice. Ordered by the level of automation from none to fully, these approaches cover the manual creation of binaries, semi-automated build scripts and continuous integration services. Manual creation means that developers have to execute a set of commands, often noted down as checklist, to build a specific software. The build script approach allows to semi-automate lots of that manual steps from the first approach. Those scripts are implemented in different scripting languages and on different platforms. Build scripts targeting UNIX systems are mainly implemented as shell scripts, whereas scripts that should be run on different target platforms make use of scripting languages like *Ruby*, *Python* and others. These scripts are often designed and implemented on a per-project basis, meaning that the level of re-usage tends to be very low. The more advanced approach of continuous integration make use of automated build triggers, that differentiates it from the first two approaches. With CI, the process of building a software is triggered with every change committed to the source code. Testing and building a software is then done automatically as soon as this change happened, instead of a time-based or manually trigger as seen with manual compilation or build scripts.

Looking at the bigger picture, we saw that not every project might be suitable for a complex continuous integration service. Therefore, we linked the introduced decision dimension characteristics to concrete build system approaches, which allows to choose an appropriate build system for every given project. Projects with low to medium complexity make use of manual building or built scripts, whereas more complex projects make use of continuous integration services. This categorization is also true for release cycles and target platforms, the more often a release happens and the more platforms are supported, the higher is the demand for automated builds and setting up such systems pays off. It is also shown that projects targeting a user group without knowledge of building software on their own tend to more build automation than software that is mainly used by software engineers. Finally, a well-structured organization in a large team also requires build automation to provide test builds and timely releases to prevent delays in a release process. This linkage of decision dimensions and build system approaches is a theoretical model, in reality a software project may change its build approach over time.

Using the approach of build scripts, we finally were able to design a generic framework that is applicable for a wide range of small to mid-sized software projects. The build system framework was designed as a collection of shell scripts, consisting of individual calls to native system functions of the underlying platform. The framework supports three target build systems: Ubuntu Linux, Mac OS X and Windows. Binaries for Linux and Mac OS X are compiled natively, Windows binaries use the concept of cross-compiling on a Linux host. The framework is designed to build C++ projects that make use of *GNU Autotools*.

For a good level of modularization and re-usage the scripts are defined in four different

stages, which are executed in subsequent order. The *check stage* looks for required build dependencies and can compile library dependencies from source or install them from a supported package manager. The *make stage* performs the main build steps for creating the platform-specific binary from source code. If this stage succeeded the following *test stage* optionally runs tests on the built binaries, before the last defined stage, the *deployment stage*, is used for distributing the new release to end-users.

All of those stages use a set of optional parameters as input that allow further customization for every single build. Due to the modular design the framework can be used for a broad range of software-projects.

CHAPTER 4

Case Study

As a practical proof of concept, this chapter uses an open-source project to evaluate the framework developed in Chapter 3. This also demonstrates how it can be used to implement a build system for similar projects.

Before we start the implementation with the concrete implementation, the chapter first describes the decisioning for the project and the chosen research area it is coming from.

4.1 Project Selection

Most software is written for target groups that are not directly related to software engineering, such as software for logic programming.

Logic programming is a *declarative* programming paradigm. In contrast to *imperative* programming, the user describes the problem and its solutions, but not how the problem is solved using concrete solving algorithms [Llo95].

Nowadays logic programming is used to solve problems from fields like decision support, combinatorial problems, artificial intelligence or Semantic Web. A popular approach for such applications is *Answer Set Programming (ASP)* [Lif08]. ASP is an outgrowth of *default logic* [Rei80] and *stable model semantics* [GL88], oriented towards difficult search problems. ASP defines a program as a set of rules that state constraints in a given problem space. The solution to that program is one defined answer set of predicates for every possible problem world manifestation that fulfills the program, i.e. defines what must be *true* so that the program is satisfied.

HEX extends the ASP paradigm with so-called higher-order and external atoms [EFI^{+}16]. Via external atoms it is possible to include bidirectional communication between a program and external computation sources, which would be infeasible to do with ASP programs themselves [EIST05]. Thus, as examples, external atoms allow to communicate with description logic reasoners, database systems or Web resources.

HEX programs are built on mutually disjoint sets $\mathcal{C}$ as *constants*, $\mathcal{X}$ as *external predicates*, and $\mathcal{G}$ as *variables*. Elements from $\mathcal{X}$ are usually written with first letter in upper case, elements from $\mathcal{C}$ with first letter in lower case and elements from $\mathcal{G}$ are prefixed with &. Constant names serve as either individual names or as predicate names. $\mathcal{C}$ may be infinite [EIST06]. Elements from $\mathcal{C} \cup \mathcal{X}$ are called terms.

A *higher-order atom* (or atom) is a tuple $(Y_0, Y_1, ..., Y_n)$, where $Y_0, ..., Y_n$ are terms with Y_0 as predicate name and $n \geq 0$ as its arity. It is therefore also possible to use the familiar notation $Y_0(Y_1, ..., Y_n)$. The atom is *ordinary*, if $Y_0 \in \mathcal{C}$. For example, $(x, rdf : type, c)$ and $node(X)$ are ordinary atoms, while $D(a, b)$ is a higher-order atom.

An *external atom* is of the form

$$\&g[Y_1, ..., Y_n](X_1, ..., X_m), \tag{4.1}$$

where $Y_1, ..., Y_n$ and $X_1, ..., X_m$ are two lists of terms, called *input list* and *output list*, and $\&g$ is an external predicate name.

HEX programs are sets of rules of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, ..., \beta_n, not\beta_n + 1, ..., not\beta_m, \tag{4.2}$$

where $m, k \geq 0, \alpha_1, ..., \alpha_k$ are higher-order atoms, and $\beta_1, ..., \beta_m$ are either higher-order atoms or external atoms. The operator *not* is *negation as failure* (or *default negation*).

The semantics of HEX-programs can be derived by generalizing the answer-set semantics [EIST06].

The following listing shows a HEX-program example for solving the three-coloring problem [BP04], where the external atom *edge* is used:[1]

```
1 % use external edge relation to identify the nodes
2 node(X) :- &edge[](X,Y).
3 node(Y) :- &edge[](X,Y).
4
5 % guessing part
6 colored(X,r) v colored(X,g) v colored(X,b) :- node(X).
7
8 % checking part
9 :- &edge[](X,Y), colored(X,C), colored(Y,C).
```

Concrete implementations of ASP or HEX program solvers are provided on the market. Solvers for ASP programs include commercially driven software like DLV[2] or solvers

[1] http://www.kr.tuwien.ac.at/research/systems/dlvhex/demo.php. DLVHEX Online Demo.
[2] http://www.dlvsystem.com/dlv/. DLV.

developed in scientific research groups and other communities such as Potassco[3]. A solver for HEX-programs is provided with the DLVHEX[4] project.

Whereas Potassco does provide pre-built binary packages for a new release once in a while, projects are mainly distributed as source-code archives. This implies that end-users have to build the software on their own. As logic programming is not necessarily a sub-domain of software engineering, end-users might not be fully familiar with setting up build environments and compiling software on their own.

Projects within the field of logic programming seem therefore to be a suitable type of projects to choose for this case study of a build system composed of custom build scripts. In particular, we choose DLVHEX due to the following characteristics, divided into the decision dimensions from Chapter 3.

4.1.1 Complexity

As many software projects, DLVHEX uses *GNU Autotools* for configuring and building the project and is therefore suitable for usage with the designed framework.

The software allows to use different solver backends, such as DLV, *clasp* or *Gringo*. The configure step decides which backends are used depending on the availability during build time.

The DLVHEX project itself is built up of two parts, a core module for the most common functionality, and a set of plugins, which allow to extend the functionality of the core module dynamically during runtime. Plugins can be used to connect other software libraries, additional solver backends or even remote services from third parties.

The core module has dependencies such as the *Boost C++ library collection*[5], used for graph algorithms, threading and as Python interface. It also links against *libcurl library*[6] to execute programs loaded from HTTP sources.

In addition to that, the core module provides an interface to use *Python* to extend the feature set without the need of creating a C++ plugin. This python integration therefore also requires the core to link against *Python 2.7* libraries.[7]

We see that there are a lot of source code dependencies in place, which add additional complexity for building for a new release.

4.1.2 Release Cycle

DLVHEX is undergoing a lot of development activity due to different scientific studies and projects. Due to the high software complexity, new releases are however only distributed

[3] http://potassco.sourceforge.net. Potassco, the Potsdam Answer Set Solving Collection.
[4] http://www.kr.tuwien.ac.at/research/systems/dlvhex/. DLVHEX.
[5] http://www.boost.org. Boost C++ library collection.
[6] https://curl.haxx.se. curl.
[7] https://www.python.org. Python.

occasionally. At the time of beginning the work on this thesis, the latest released version 2.4.0 dates back to September of 2014, the release before this dates back to December of 2013.

During work on this thesis version 2.5.0 was released in April of 2016 as result of the implemented build system in this case study.[8]

Given the high development activity and the spare releases, a build system can improve the release cycles and thus may improve adoption of the project itself.

4.1.3 Target Group

As mentioned in the beginning of the chapter, software of the logical programming area mostly targets people without deeper knowledge of software building. Also, the target group often does not want to deal with the build process itself but wants to utilize the binary software.

Like many other projects derived from scientific research, dlvhex is also used in academical courses for teaching purposes. Due to the lack of pre-built binaries, the current distribution approach is to provide pre-configured virtual machine images to students. With binaries built from an automated build system, the initial onboarding for students may get a lot easier and time can be spent with using the software and not with building it.

4.1.4 Target Platforms

dlvhex is a cross-platform software project, designed and implemented to run on Linux, Mac OS X and Windows machines. On Linux and Mac OS X, *GNU Autotools* are used to configure and build the project. A custom Visual Studio project available in the source code repository additionally allows to build the software for Windows systems. This additional project however needs to be maintained separately from the *GNU Autotools* build template files, which adds additional complexity.

With the cross-compilation approach of the designed framework, Windows builds may get subject of automated builds from a Linux host machine.

4.1.5 Organization

The core team is built up from a group of scientific employees of Vienna University of Technology and research groups from other universities, located at different geographical regions in Europe.

Unlike users of dlvhex, most members of the development team have a strong knowledge in the field of logical programming and software engineering in general.

[8] https://github.com/hexhex/core/releases. dlvhex releases.

4.1.6 State-of-the-Art Releases

As stated in Section 4.1.2, the last release before introducing a concrete implementation of the designed framework dates back to September, 2014. In a discussion with the project team, it was confirmed that the current manual binary releases are hard to maintain due to the high complexity of the software. As there are no external release triggers, a new version got released in the form of a source-code archive whenever the project team incorporated a specific new feature or fixed a series of bugs.

The release process consists of the following manual steps:

1. Decide upon a specific code base for a release within the project team.
2. Tag the code base with a new Git release tag in the code repository.
3. Create a compressed source-code archive and upload it to the website.
4. Update the website to link to the latest source-code release.

Binary releases are mostly skipped due to the high complexity in building and using the packages, as the project uses a lot of external dependencies that may be linked during runtime.

With the implementation of a build system derived from the work of this thesis, version 2.5.0 was finally released in binary form in April of 2016.

4.2 Goals and Expected Results

A build system for DLVHEX should fulfill the following requirements:

- The build system should use the script framework designed in Section 3.4. Due to the less frequent release cycle builds will be triggered manually by the project team but the implementation of the scripts should consider automated builds for the future, so dependencies and environment variables need to be set up for subsequently clean builds.
- As proposed in the framework, Windows builds should be done with cross-compilation and *GNU Autotools*, removing the requirement of custom Visual Studio project maintenance.
- The distributed binaries should depend on as few dynamic libraries as possible, to make installation and usage easier than it is as of now. This requires the build system to look for static libraries or build them from source when possible.
- Due to semi-automated builds and thus more releases over time, end users should be motivated to use the latest releases and the general usage of the software should be increased due to higher visibility with pre-built binaries for all major platforms.

4.3 Framework Implementation

The designed framework already defines different stages and provides a stub implementation. Thus the implementation describes the project-specific build commands for every script file listed in Table 3.2.

4.3.1 Check Stage

The check stage is responsible for checking code and tool build dependencies. Optionally these dependencies can be installed when providing the corresponding command line parameter to the script.

The check stage defines a list of package dependencies for creating the builds.

Linux

The script defines the following configuration variable:

```
1 required_dependencies="sed git build-essential autoconf autotools-dev
      libtool wget scons bison re2c libboost-all-dev python-dev libpython-all
      -dev libcurl4-openssl-dev libbz2-dev"
```

scripts/dlvhex/check–linux.sh

Beside the essential build tool packages the script installs binary build packages for the libcurl, libbz2, Python and Boost dependencies.

Mac OS X

The script defines the following configuration variable:

```
1 required_dependencies="git autoconf automake libtool pkg-config wget scons
      re2c python boost-python"
```

scripts/dlvhex/check–osx.sh

As Linux, Mac OS X installs the build tools from *GNU Autotools* and binary packages for Python and Boost.

The installation of *bison* parser generator tool requires special attention. As Mac OS X ships with an outdated version of bison that does not meet the minimum version requirement of the DLVHEX project, the tool is defined as *forced dependency*:

```
1 forced_dependencies="bison"
```

scripts/dlvhex/check–osx.sh

Windows

The script defines the following configuration variable:

```
required_dependencies="sed git build-essential autoconf libtool wget pkg-
    config scons bison re2c p7zip-full python-dev libxml2-dev libxslt1-dev
    ccze mingw32-x-binutils mingw32-x-gcc mingw32-x-runtime mingw32-x-zlib
    mingw32-x-openssl"
```

scripts/dlvhex/check–windows.sh

As Windows uses cross-compilation, the script installs both, Linux-native build tools and a tool-chain for the MinGW cross-compiler suite.

The script also defines an execution block for building the dependencies for *libcurl*, *bzip*, *Python* and *Boost* from source. To better separate the different dependency types, the steps for building dependencies from source are defined in an additional script called *make-windows-dependencies.sh* and executed from the *make_dependencies* execution block:

```
  # Source additional script for installing dependencies from
  # sources
  ./make-windows-dependencies.sh
```

scripts/dlvhex/check–windows.sh

The libraries for *libcurl bzip* and *Boost* can be compiled with default cross-compiling commands. Unfortunately *Python* installation for Windows on Linux for usage with cross-compilation is a more complex task, as there are no pre-built packages for GCC available:

```
# Install Python
wget https://sourceforge.net/projects/mingw-w64/files/Toolchains%20
    targetting%20Win32/Personal%20Builds/mingw-builds/4.9.2/threads-posix/
    dwarf/i686-4.9.2-release-posix-dwarf-rt_v3-rev1.7z
7z x i686-4.9.2-release-posix-dwarf-rt_v3-rev1.7z -opython mingw32/opt/
    include/python2.7 mingw32/opt/bin/python* -r
sudo rsync -a python/mingw32/opt/include/python2.7 $MINGW_DIR/include
sudo rsync -a python/mingw32/opt/bin/ $MINGW_DIR/bin
sudo chmod -R 755 $MINGW_DIR/include
sudo chmod -R 755 $MINGW_DIR/bin

# Use MinGW python config vars for auto tools
hash -p $MINGW_DIR/bin/python-config.sh python-config

# Install Python mingw dll.a file for local python installation
# (the one from mingw toolchain requires other dependencies)
wget https://bitbucket.org/carlkl/mingw-w64-for-python/downloads/libpython
    -cp27-none-win32.7z
7z x libpython-cp27-none-win32.7z
sudo mv libs/libpython27.dll.a $MINGW_DIR/lib/libpython2.7.a
```

scripts/dlvhex/make–windows–dependencies.sh

The script defines the following steps for installing Python libraries:

1. Download the *MinGW* tool-chain from the official project website.
2. Unzip Python-only files into the installed tool-chain directory from the package manager.
3. Construct the terminal session to use the tool-chain-included *python-config* command instead of the system-wide configuration binary. This is required in order that *GNU Autotools* detect the correct Python configuration during configuring the cross-compilation build.
4. Download and copy a custom *libpython27.dll.a* definition library file, as the tool-chain-included file is built for Microsoft Visual Studio compilers and not GCC.

4.3.2 Make Stage

The make stage is responsible for building the actual binary builds.

The make stage defines a list of configure parameters and allows to set up custom environment variables and optionally patch make files right before building the binary.

Linux

The script defines the following configuration variable in order to build a static binary of the project:

```
1 configure_parameters="CXXFLAGS=-fPIC LOCAL_PLUGIN_DIR=plugins --enable-
    python --enable-shared=no --enable-static-boost"
```

scripts/dlvhex/make–linux.sh

GNU Autotools pick up the Python version according to the *PYTHON_BIN* variable, which gets set to version 2.7 within the script:

```
1   export PYTHON_BIN=python2.7
```

scripts/dlvhex/make–linux.sh

There are no patches needed, thus the *patch* execution block is empty.

Mac OS X

The script defines the following configuration variable in order to build a static binary of the project:

```
1 configure_parameters="LOCAL_PLUGIN_DIR=plugins --enable-python --enable-
    shared=no --enable-static-boost"
```

scripts/dlvhex/make–osx.sh

GNU Autotools pick up the Python version according to the *PYTHON_BIN* variable, which gets set to version 2.7 within the script. The script also explicitly links the *bison* version from the package manager into the path environment, otherwise the system version would be used for configuring the project, which fails due to an outdated version.

```
1   export PYTHON_BIN=python2.7
2
3   # Use bison from homebrew, not linked automatically
4   brew link bison --force
```

scripts/dlvhex/make–osx.sh

The used clang/llvm compiler suite requires a patch for the included *Makefile* generator file, which is done in the *patch* execution block:

```
1   sed -i "" "s/libdlvhex2_base_la_LIBADD/#libdlvhex2_base_la_LIBADD/" src/
     Makefile.am
```

scripts/dlvhex/make–osx.sh

Windows

The script defines the following configuration variable in order to build a static binary of the project and linked the custom built dependencies:

```
configure_parameters="LDFLAGS=\"-static -static-libgcc -static-libstdc++ -L$MINGW_DIR/lib\" LOCAL_PLUGIN_DIR=plugins --enable-python --enable-static --disable-shared --enable-static-boost --with-boost=$MINGW_DIR --with-libcurl=$MINGW_DIR --host=$HOST_PREFIX CFLAGS=\"-static -DBOOST_PYTHON_STATIC_LIB -DCURL_STATICLIB\" CXXFLAGS=\"-static -DBOOST_PYTHON_STATIC_LIB -DCURL_STATICLIB\" CPPFLAGS=\"-static -DBOOST_PYTHON_STATIC_LIB -DCURL_STATICLIB\""
```

scripts/dlvhex/make–windows.sh

Setting up the cross-compilation environment requires a couple of steps:

```
MINGW_DIR=/opt/mingw32
HOST_PREFIX=i686-w64-mingw32

# Set python version
export PYTHON_BIN=python2.7
export PYTHON_INCLUDE_DIR=$MINGW_DIR/include/python2.7
export PYTHON_LIB=python2.7

# CC environment
export CC=$HOST_PREFIX-gcc
export CXX=$HOST_PREFIX-g++
export CPP=$HOST_PREFIX-cpp
export RANLIB=$HOST_PREFIX-ranlib
export PATH="$MINGW_DIR/bin:$PATH"

# Environment helper
cat >$HOME/mingw << 'EOF'
#!/bin/sh
HOST_PREFIX=i686-w64-mingw32
MINGW_DIR=/opt/mingw32
export CC=$HOST_PREFIX-gcc
export CXX=$HOST_PREFIX-g++
export CPP=$HOST_PREFIX-cpp
export RANLIB=$HOST_PREFIX-ranlib
export PATH="$MINGW_DIR/bin:$PATH"
exec "$@"
EOF
chmod u+x $HOME/mingw

# Use MinGW python config vars for auto tools
hash -p $MINGW_DIR/bin/python-config.sh python-config
```

scripts/dlvhex/make–windows.sh

The *setenv* execution block sets up the Python version, exports some environment

variables for cross-compilation and also creates a helper script for reusing the cross-compilation environment in script sub-calls and processes. Finally, the step also makes sure that the correct *python-config* utility is used.

It is also required to define a patch for using the cross-compiler binaries for the custom *Scons* build system used for building the Gringo grounder included in the core module:

```
1  sed -i "1s|^|62c62\n< env['CXX']            = 'g++'\n---\n> env['CXX']
              = '$HOST_PREFIX-g++'\n68c68\n< env['LIBPATH']        = []\n
   ---\n> env['LIBPATH']        = ['$MINGW_DIR/lib']\n|" buildclaspgringo/
   SConstruct.patch
```

scripts/dlvhex/make–windows.sh

4.3.3 Test Stage

The test stage is used to run function tests on the built binary.

As the scripts are implemented to build static versions of the binaries, the included test cases are also built into a static library and thus cannot be used for running the tests with the *make check* command. As a consequence of this, the test stage scripts are skipped for that project and not executed during a build as of now.

4.3.4 Deployment Stage

The deployment stage handles the distribution of the built binaries.

As deployment can be done in several ways, the framework only defines an execution block that can be implemented specifically for every built project.

For the implementation of the DLVHEX project all platforms define an identical implementation. The binary is copied to a sub-directory called *build*. The following listing shows the *copysteps* from the Linux script as an example:

```
1  mkdir -p "build"
2  cp $OUT_FILE $DESTINATION_PATH
```

scripts/dlvhex/deploy–linux.sh

The copy steps make use of the defined configuration variables

```
1  OUT_FILE="src/dlvhex2"
```

scripts/dlvhex/deploy–linux.sh

and

```
1 DESTINATION_PATH="./"
```

scripts/dlvhex/deploy–linux.sh

4.4 Issues and Limitations

During the implementation of the build scripts for DLVHEX, a couple of issues and errors were experienced, which are summarized here:

4.4.1 Python Versions

The project is implemented to build with different Python major version, Python 2.7 and Python 3. Due to that, *GNU Autotools* define some custom configuration steps that try to automatically detect the Python version. Because these steps look for the system-wide *python-config* tool, cross-compilation for Windows on Linux usually breaks, as the configuration variables are then set up for Linux builds, and not Windows builds. In order to overcome that issue, the Windows cross-compilation scripts override the path environment to use a custom shell script that defines the configuration for Windows builds, which is currently hard-coded to Python version 2.7.

Moreover, the setup of Python development libraries requires some advanced steps to succeed with configuration and make steps.

4.4.2 Scons Build System

A direct dependency of the core module (*Gringo* solver) is built with an alternative build system called *Scons*.

This makes cross-compilation more complex, as the custom build files need to be patched and configuration variables for *GNU Autotools* need to be duplicated for the additional build system.

4.4.3 Build System Modifications

The Windows build requires the build scripts to create additional files on the file system outside of the build directory. It is therefore not recommended to use the script on a personal working machine, unless the user is aware of the performed changes.

The recommended way of creating Windows binaries with the implemented solution is to use virtual machine images for Windows builds.

4.4.4 Windows Plugin Interface

The cross-compilation on Windows builds static libraries per default. The plugin system on Windows with MinGW requires correctly defined export statements from the symbol table, otherwise it is not possible to build *dll* shared libraries at all.

4.5 Future Work

The implemented system performs binary builds for Linux, Mac OS X and Windows systems, based on the framework from Chapter 3. The system can be improved as follows.

4.5.1 Cross-Compilation for Mac OS X

The framework already makes use of the concept of cross-compilation for building Windows binary files. This greatly reduces the complexity and maintenance effort of build system hardware. On Mac OS X builds are still performed on Macintosh-based computers.

As there are no platform-specific dependencies for building DLVHEX, it is possible to develop a cross-compilation solution for Mac OS X too and therefore further simplify the build system setup. With that solution in place, the designed framework, as discussed in Section 3.4, can be adapted to have a generic solution for Mac OS X cross-compilation in place.

4.5.2 Tests

Due to static builds of both the core binary and the libraries, tests defined for the project are currently not executed, as the test plugin library cannot be linked dynamically. For future work it is possible to only build the test libraries as dynamic libraries and run the defined tests in the test stage of the framework.

4.5.3 Plugin Builds

The solution currently creates builds of the core module of DLVHEX. Through plugins it is possible to extend the functionality without a new core module build.

As all available plugins share the very same tool-chain and dependencies as the core module, it is possible to extend the build system for plugin builds too.

4.5.4 Automated Distribution

In the current solution the built binaries are made available in a top-level directory of the source code repository (a directory called *build*). After a build, the team has to deploy the binaries for each platform to the website manually.

This can be automated when adapting the deploy step. With tools such as *ftp* or *scp* it is possible to upload the binaries to the web server of the project or a cloud storage like *Amazon S3* and make them publicly available without any further manual steps to do.

4.5.5 Continuous Integration Service

The implemented solution provides different build scripts for creating a DLVHEX build. Builds however still need to be triggered manually by a maintainer of the project.

As all scripts are implemented in an atomic and modular way they can however also be used in conjunction with a CI service, such as *Travis CI*[9] or others.

CI services can be configured to automatically create a new build upon the push of a new VCS branch or tag and can automatically distribute binaries with the help of the deployment stage of the scripts. This allows a complete automation of new version releases.

4.6 Summary

The field of logic programming gave a good example for an open-source software that lacks of binary distribution according to the initial problem description.

The used software DLVHEX, a reasoner for HEX programs, comes up with a high level of complexity due to external library dependencies. Libraries like Boost or Python add additional overhead for configuring builds, given the fact that the software is released for three platforms, Linux, Mac OS X and Windows, in parallel. Because of that complexity, the DLVHEX software did not follow a frequent and periodical release cycle. This made new features for end-users inaccessible for a longer time, as the software is mostly used by people without deeper knowledge of the software build process.

We therefore defined the goal to implement a build system from the framework designed in Chapter 3. As proposed, the implemented build system creates binary builds for platforms Linux and Mac OS X on target platforms and uses cross-compilation on Linux to create a binary build for Windows, which eliminates a redundant Visual Studio project and its maintenance efforts. During the implementation, we also gave the goal of as less as needed dynamic library dependencies special attention, the resulting binaries therefore only link against system-installed Python libraries.

The implementation of the check and make stage of the framework comes up with some smaller issues due to pre-installed but outdated build tool versions on Mac OS X. As assumed in the design phase of the case study, cross-compiling Windows software with a Python dependency brought up multiple issues during the configuration steps, which were all resolved in a custom check stage. The usage of the Scons build tool for the linked in *Gringo* solver also needed some patching in order to work with cross-compilation. Finally, some small patches for linking in static library dependencies were applied to all of the platform builds.

The test stage is mainly skipped and therefore not implemented in that case study, as running the tests require a software build with dynamic library dependencies, which were not intended for that build system. After a successful build the deployment stage finally copies the built binaries into the root of the project working directory, this step can be adapted to the needs of the specific project.

[9] https://travis-ci.org. Travis Continuous Integration.

The implemented system performs binary builds of the core module for Linux, Mac OS X and Windows systems. As future work it is also possible to build DLVHEX plugin components with that framework. To reduce the build hardware setup complexity, it is also possible to look into cross-compilation for Mac OS X from Linux systems. Finally, the implementation of the test stage and the improvement of the deployment stage for automated distribution can further automate the build process.

CHAPTER 5

Conclusion and Outlook

5.1 Summary

Today, open-source projects are maintained by online developer communities who make active use of collaboration tools, allowing a highly agile development process. Due to this openness, the way of releasing new versions of a software differs compared to commercial, closed-source projects. Software is mainly available as a source code archive of a specific development snapshot. Maintainers of such software projects however often do not provide pre-built binary packages, as they see their role in maintaining code and developing the community and not in building binaries for end-users. This is especially problematic for projects targeting people that are not originating from the field of software engineering and projects emerging from scientific research, for which potential end-users are not able to build the software from source. Although packages are highly demanded, providing them is often neglected as the creation is a time-consuming task that cannot easily be automated. As a result, in many cases, such software suffers of low accessibility and visibility.

To address this problem, we first covered the theoretical aspects that are required for the further research into existing build systems and methodologies. We identified the steps involved in a software release cycle, which showed that a new build is a complex process that gets challenging when supporting different platforms and hardware architectures. The introduction of different build concepts then gave an overview for the discussed build system approaches that try to overcome those challenges.

Based on the described challenges that come up during a build and the analysis of available build tools, we designed a framework for a build system, which can be used to largely automate the release process of software projects and thus supports developers to provide up-to-date packages for end-users. The aim of the given framework is to provide basic features for building a software from the technical side. The main design goal was

to provide a generic solution that incorporates a high level of reusability and flexibility and that is able to create builds for a variety of software projects.

The proposed solution is using the approach of build scripts. Keeping the goal of a generic solution in mind, the framework is applicable for a wide range of software projects of different size. The build system framework is organized as a collection of shell scripts, consisting of individual calls to native system functions of the underlying platform. The framework supports three target build systems: Ubuntu Linux, Mac OS X and Windows, whereas binaries for Linux and Mac OS X are compiled natively and Windows binaries use the concept of cross-compiling on a Linux host. The framework is intended to build C++ projects that make use of *GNU Autotools*.

For a good level of modularization and re-usage the scripts are defined in four different stages, which are executed in subsequent order. The *check stage* looks for required build dependencies and can compile library dependencies from source or install them from a supported package manager. The *make stage* performs the main build steps for creating the platform-specific binary from source code. If this stage succeeded the following *test stage* optionally runs tests on the built binaries, before the last defined stage, the *deployment stage*, is used for distributing the new release to end-users.

Finally, the field of logical programming gave a good example for an open-source software that lacks of binary distribution according to the initial problem description. The used software DLVHEX, a reasoner for HEX programs, comes up with a high level of complexity due to external library dependencies. Libraries like Boost or Python add additional overhead for configuring builds, given the fact that the software is released for three platforms, Linux, Mac OS X and Windows, in parallel. Because of that complexity, the DLVHEX software did not follow a frequent and periodical release cycle. This made new features for end-users inaccessible for a longer time, as the software is mostly used by people without deeper knowledge of the software build process.

We therefore defined the goal to implement a build system from the framework designed in Chapter 3. During the implementation, we also gave the goal of as less as needed dynamic library dependencies special attention, the resulting binaries therefore only link against system-installed Python libraries.

With the implemented framework it was finally possible to release version 2.5.0 as a first proof of the versatility of the scripts.

5.2 Future Work

To reduce the build hardware setup complexity, future work can look into cross-compilation for Mac OS X from Linux systems. Furthermore, the implementation of the test stage and the deployment stage can further be improved to allow more automation in the distribution process.

From a design point of view, the modular structure of the build scripts allows to test the feasibility of using them within a continuous integration environment. This leads to a

system design that can be used to fully automate the software release cycle for a wide range of software projects.

APPENDIX A

Framework Scripts

The following listings show the complete scripts as defined in Chapter 3.

A.1 Installation Stage

```
1  #!/usr/bin/env bash
2  set -e
3
4  ################################################################
5  # check-platform.sh
6  #
7  # Author: John Doe <john.doe@example.com>
8  #
9  # Run this script to check if all dependencies are met and
10 # optionally install them via apt-get on Linux or homebrew
11 # on Mac OS X.
12 #
13 # Parameters:
14 #
15 # --install-dependencies
16 #
17 #   If set the script automatically tries to install the
18 #   missing dependencies with the platform-specific package
19 #   manager (apt-get on Linux or homebrew on Mac OS X).
20 #
21 #   If the parameter is omitted, the script exits with a hint
22 #   how to install the missing dependencies.
23 #
24 #   Please note that the script may need root permissions and
25 #   will prompt you for that.
26 #
27 ################################################################
```

```
28
29
30 ###################
31 # Configuration vars
32 ###################
33
34 # A list of dependencies, as named within the platform-specific
35 # package manager, that are required in order to build the
36 # software project.
37 #
38 # E.g.:
39 # required_dependencies="sed git build-essential autoconf"
40 #
41 required_dependencies=""
42
43 # A list of dependencies, as named within the platform-specific
44 # package manager, that are required in order to build the
45 # software project and should be installed again, even if
46 # found in the path configuration of the system.
47 #
48 # E.g.:
49 # forced_dependencies="bison"
50 #
51 forced_dependencies=""
52
53
54 ###################
55 # Execution steps
56 ###################
57
58 # This step defines custom build steps for building
59 # dependencies from source that are not available via package
60 # managers.
61 function make_dependencies {
62   # Example 1: Download and build curl
63   # wget http://curl.haxx.se/download/curl-7.46.0.tar.gz
64   # tar xzf curl-7.46.0.tar.gz
65   # pushd curl-7.46.0
66   # ./configure
67   # make
68   # make install
69   # popd
70
71   # Example 2: Call additional script
72   # ./make-windows-dependencies.sh
73 }
74
75
76 ###################
77 # Internal
78 ###################
79
80 # Used to temporarily save missing dependencies
```

```
81  missing_dependencies=""
82
83  # Check parameters
84  while [ -n "$1" ]; do
85    case $1 in
86      --install-dependencies )     install_deps=1
87                                   ;;
88    esac
89    shift
90  done
91
92  # Add forced dependencies
93  for dep in `echo $forced_dependencies`; do
94    missing_dependencies="$missing_dependencies $dep"
95  done
96
97  # Platform switch
98  if [ "$(uname)" == "Darwin" ]; then
99
100   # Check for dependencies on Mac OS X platform
101   for dep in `echo $required_dependencies`; do
102     if ! which $dep &> /dev/null; then
103       missing_dependencies="$missing_dependencies $dep"
104     fi
105   done
106
107   if [ -n "$missing_dependencies" ]; then
108     # Check if we should auto-install dependencies
109     if [ "$install_deps" -eq 1 ]; then
110       echo "Installing dependencies..."
111       brew update
112       brew install$missing_dependencies
113
114       echo "Building dependencies from source..."
115       make_dependencies
116     else
117       echo "Error: Missing build dependencies, use:"
118       echo "brew install$missing_dependencies"
119       exit 1
120     fi
121   else
122     echo "Dependencies up to date"
123   fi
124
125 else
126
127   # Check for dependencies on Linux platforms
128   for dep in `echo $required_dependencies`; do
129     if ! dpkg -s $dep &> /dev/null; then
130       missing_dependencies="$missing_dependencies $dep"
131     fi
132   done
133
```

```
  if [ -n "$missing_dependencies" ]; then
    # Check if we should auto-install dependencies
    if [ "$install_deps" -eq 1 ]; then
      # Uses sudo command, may result in a user prompt
      echo "Installing dependencies..."
      sudo apt-get update -qq
      sudo apt-get -y install$missing_dependencies

      echo "Building dependencies from source..."
      make_dependencies
    else
      echo "Error: Missing build dependencies, use:"
      echo "apt-get update && apt-get install$missing_dependencies"
      exit 1
    fi
  else
    echo "Dependencies up to date"
  fi

fi
```

scripts/check–platform.sh

A.2 Build Stage

```
#!/usr/bin/env bash
set -e

###############################################################
# make-platform.sh
#
# Author: John Doe <john.doe@example.com>
#
# Run this script to build the project binary. Make sure to
# call check-platform.sh to check for missing dependencies
# beforehand.
#
# Parameters:
#
# --verbose
#
#   If set the script outputs all build output from build
#   tools to stdout.
#
#   If omitted, the build output is hidden and only some
#   informative log statements are printed to stdout.
#
###############################################################


####################
```

```
27 # Configuration vars
28 ###################
29
30 # Additional parameters that should be passed to the configure
31 # step of GNU Autotools.
32 #
33 # E.g.:
34 # configure_parameters="CXXFLAGS=-fPIC"
35 #
36 configure_parameters=""
37
38
39 ###################
40 # Execution steps
41 ###################
42
43 # Set custom environment variables and functions here
44 function setenv {
45   # e.g. set python version to 2.7
46   # export PYTHON_BIN=python2.7
47 }
48
49 # Add patches for make files or other related steps
50 function patch {
51   # e.g. patching Makefile.am:
52   # sed -i "" "s/liba_la_LIBADD/#liba_la_LIBADD/" Makefile.am
53 }
54
55
56 ###################
57 # Internal
58 ###################
59
60 # Configuration
61 OUTPUT_IO=/dev/null
62
63 # Check parameters
64 while [ "$1" != "" ]; do
65   case $1 in
66     --verbose )      OUTPUT_IO=/dev/stdout
67                      # set -v
68                      ;;
69   esac
70   shift
71 done
72
73 # 1. Set environment
74 setenv
75
76 # 2. Create build files
77 if [ -f bootstrap.sh ]; then
78   ./bootstrap.sh &> $OUTPUT_IO
79 fi
```

```
80
81 # 3. Optionally patch input files
82 patch
83
84 # 4. Configure and build
85 ./configure $configure_parameters &> $OUTPUT_IO
86 make &> $OUTPUT_IO
```

scripts/make–platform.sh

A.3 Test Stage

```
1  #!/usr/bin/env bash
2  set -e
3
4  ################################################################
5  # test-platform.sh
6  #
7  # Author: John Doe <john.doe@example.com>
8  #
9  # Run this script to test the built binaries from
10 # build-platform.sh scripts with tests defined in a custom
11 # test make target.
12 #
13 # Parameters:
14 #
15 # none
16 #
17 ################################################################
18
19
20 ####################
21 # Execution steps
22 ####################
23
24 # This step defines the custom test command after a successful
25 # binary build from the build-platform.sh script.
26 function runtests {
27   # Example 1:
28   # make check
29   # Example 2:
30   # make test
31 }
32
33
34 ####################
35 # Internal
36 ####################
37
38 runtests
```

scripts/test–platform.sh

A.4 Deployment Stage

```
#!/usr/bin/env bash
set -e

###############################################################
# deploy-platform.sh
#
# Author: John Doe <john.doe@example.com>
#
# Run this script to deploy the built binaries from
# build-platform.sh scripts.
#
# Parameters:
#
# none
#
###############################################################


####################
# Configuration vars
####################

# Defines the relative path to the binary being built from the
# previous make-platform.sh script.
#
# E.g.:
# out_file="src/thebinary"
#
out_file=""

# Defines the path where the binary should be copied to. This
# variable can be used in a custom copy step defined in the
# copysteps execution step below.
#
# E.g.:
# destination_path="/mnt/shares/builds/theproject/"
#
destination_path=""


####################
# Execution steps
####################

# This step defines custom copy or deployment steps for the
```

```
46 # binary given in the out_file variable. Make sure that the
47 # specific copy command binary is installed with the
48 # check-platform.sh script.
49 function copysteps {
50   # Example 1: local copy step
51   # cp $out_file $destination_path
52   # Example 2: Remote copy over SCP
53   # scp $out_file user@example.com:/home/bobbuilder/theproject/
54 }
55
56
57 ###################
58 # Internal
59 ###################
60
61 copysteps
```

scripts/deploy–platform.sh

List of Figures

2.1 Compilation Process 8
2.2 Detailed Compilation Process 9

3.1 Continuous Integration Overview 29
3.2 Four main stages in the build lifecycle 35
3.3 Framework Components for each Stage 36
3.4 Stage Execution Flow 36

List of Tables

3.1 Decision Dimensions and Build System Approaches Overview 34
3.2 Build Scripts 38

Acronyms

APT Advanced Packaging Tool. 13

ASP Answer Set Programming. 47, 48

CAD Computer-Aided Design. 31

CI Continuous Integration. 29–32, 60

CPU Central Processing Unit. 14

ERP Enterprise Resource Planning. 33

GCC GNU Compiler Collection. 17

HTML Hyper Text Markup Language. 11, 12

HTTP Hypertext Transfer Protocol. 43, 49

LoC Lines Of Code. 23, 24

QA Quality Assurance. 27

RPM RPM Package Manager. 13

SCCS Source Code Control System. 6, 30

SCP Secure Copy. 44

sed stream editor. 38

VCS Version Control System. 6, 7, 24, 25, 43, 60

VM Virtual Machine. 17, 18

Bibliography

[ADTZ11] Pietro Abate, Roberto DiCosmo, Ralf Treinen, and Stefano Zacchiroli. Mpm: A modular package manager. In *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, pages 179–188. ACM, 2011.

[AM11] Ishtiaq Ali and Natarajan Meghanathan. Virtual machines and networks-installation, performance study, advantages and virtualization options. 2011.

[App94] Andrew W Appel. Axiomatic bootstrapping: A guide for compiler hackers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1699–1718, 1994. Retrieved on 2016-01-24 from `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.8683&rep=rep1&type=pdf`.

[BM05] Bertrand Blanc and Bob Maaraoui. Endianness or where is byte 0. *White Paper*, 2005. Retrieved on 2016-07-08 from `http://3bc.bertrand-blanc.com/endianness05.pdf`.

[BP04] Stefan Boettcher and Allon G Percus. Extremal optimization at the phase transition of the three-coloring problem. *Physical Review E*, 69(6):066703, 2004.

[BW12] Amy Brown and Greg Wilson. *The Architecture of Open Source Applications: Structure, Scale, and a Few More Fearless Hacks*, volume 2. Kristian Hermansen, 2012.

[DMG07] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk.* Pearson Education, 2007.

[DNMV+13] Tom De Nies, Sara Magliacane, Ruben Verborgh, Sam Coppens, Paul T Groth, Erik Mannens, and Rik Van de Walle. Git2prov: Exposing version control system content as w3c prov. In *International Semantic Web Conference (Posters & Demos)*, pages 125–128, 2013.

[EFI+16] Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, Christoph Redl, and Peter Schüller. A model building framework for answer set programming with external computations. *Theory and Practice of Logic Programming*, 16(04):418–464, 2016. Retrieved on 2016-07-06 from `http://arxiv.org/pdf/1507.01451.pdf`.

[EIST05] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *In Proceedings of the 19th International Joint Conference on Artificial Intelligence*, volume 5, pages 90–96. Citeseer, 2005.

[EIST06] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. dlvhex: A system for integrating multiple semantics in an answer-set programming framework. *Proceedings 20th Workshop on Logic Programming and Constraint Systems*, 6:206–210, 2006.

[Epp02] GKT Epperly. Software in the doe: The hidden overhead of the build. *Lawrence Livermore National Laboratory, CA, USA, Technical Report*, (UCRL-ID-147343), 2002. Retrieved on 2016-03-20 from `http://grosskurth.ca/bib/2002/kumfert.pdf`.

[ERP14] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–245. ACM, 2014.

[Fit09] Panos Fitsilis. Measuring the complexity of software projects. In *2009 World congress on computer science and information engineering*, pages 644–648. IEEE, 2009.

[GJR99] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference*, pages 99–108, 1999.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Symposium on Logic Programming*, volume 88, pages 1070–1080, 1988.

[GVSZ14] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 384–387. ACM, 2014.

[HGM03] Richard C Holt, Michael W Godfrey, and Andrew J Malton. The build/-comprehend pipelines. In *Proceedings of the second ASERC Workshop on Software Architecture*, 2003.

[KDZA12] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. Do faster releases improve software quality? an empirical case study of mozilla firefox. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference*, pages 179–188. IEEE, 2012.

[LB06] Linda M Laird and M Carol Brennan. *Software measurement and estimation: a practical approach*, volume 2. John Wiley & Sons, 2006.

[Lif08] Vladimir Lifschitz. What is answer set programming? In *Association for the Advancement of Artificial Intelligence*, volume 8, pages 1594–1597, 2008.

[Llo95] John W Lloyd. Declarative programming in escher. Technical report, Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, 1995.

[Luo01] Lu Luo. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, 15232(1-19):19, 2001.

[MG03] Luke Mewburn and Matthew Green. build. sh: Cross-building netbsd. In *BSDCon Conference*, pages 47–56, 2003.

[Nag05] William Nagel. *Subversion Version Control: Using the Subversion Version Control System in Development Projects*. Prentice Hall PTR, 2005.

[Par92] Robert E Park. Software size measurement: A framework for counting source statements. Technical report, Defense Technical Information Center, 1992.

[Rei80] Raymond Reiter. A logic for default reasoning. *Artificial intelligence*, 13(1):81–132, 1980. Retrieved on 2016-07-02 from `http://www.umiacs.umd.edu/~horty/courses/readings/reiter-default-1980.pdf`.

[Roc75] Marc J Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, (4):364–370, 1975. Retrieved on 2016-09-21 from `http://www-public.tem-tsp.eu/~gibson/Teaching/Teaching-ReadingMaterial/Rochkind75.pdf`.

[SKW08] Vijay D Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, 27(7):1165–1178, 2008. Retrieved on 2016-02-28 from `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.473.7220&rep=rep1&type=pdf`.

[SN05] James E Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005. Retrieved on 2016-07-08 from `http://www.ittc.ku.edu/~kulkarni/teaching/archieve/EECS800-Spring-2008/smith_nair.pdf`.

[WP09] H.K. Wright and D.E. Perry. Subversion 1.5: A case study in open source release mismanagement. In *Emerging Trends in Free/Libre/Open Source Software Research and Development, 2009. FLOSS '09. ICSE Workshop on*, pages 13–18, May 2009.

[YHB+11] Andrew J Younge, Robert Henschel, James T Brown, Gregor Von Laszewski, Judy Qiu, and Geoffrey C Fox. Analysis of virtualization technologies for high performance computing environments. In *Cloud Computing (CLOUD), 2011 IEEE International Conference*, pages 9–16. IEEE, 2011.

[Zad02] Erez Zadok. Overhauling amd for the'00s: A case study of gnu autotools. In *USENIX Annual Technical Conference, FREENIX Track*, pages 287–297, 2002.