A comprehensive approach for software dependency resolution

by

Hanyu Zhang
B.Sc., Zhejiang University, 2002

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Hanyu Zhang, 2011
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

A comprehensive approach for software dependency resolution

by

Hanyu Zhang
B.Sc., Zhejiang University, 2002

Supervisory Committee

Dr. Daniel German, Supervisor Main, Supervisor
(Department of Computer Science)

Dr. Micaela Serra , Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Daniel German, Supervisor Main, Supervisor
(Department of Computer Science)

Dr. Micaela Serra , Departmental Member
(Department of Computer Science)

ABSTRACT

Software reuse is prevalent in software development. It is not uncommon that one software product may depend on numerous libraries/products in order to build, install, or run. Software reuse is difficult due to the complex interdependency relationships between software packages. In this work, we presented four approaches to retrieve such dependency information, each technique focuses on retrieving software dependency from a specific source, including source code, build scripts, binary files, and Debian spec. The presented techniques were realized by a prototype tool, DEx, which is applied to a large collection of Debian projects in a comprehensive evaluation. Through the comprehensive analysis, we evaluate the presented techniques, and compare them from various aspects.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables vii

List of Figures viii

Acknowledgements x

Dedication xi

1 Introduction 1

1.1 Outline 2

2 Background 4

2.1 Free and open source software 4

2.1.1 FOSS: Advantages and Disadvantages 6

2.2 Software reuse 7

2.2.1 What is software reuse 8

2.2.2 Benefits of software reuse 10

2.2.3 Problems of software reuse 10

2.3 Software dependency 12

2.3.1 What is a dependency 13

2.4 Extraction of dependencies 15

2.4.1 Dependency resolution from software documentation 16

2.4.2 Dependency resolution by analysing source code 16

2.4.3 Dependency resolution by mining software repository 19

2.4.4 Dependency resolution by processing binary files 20

2.5 Build system 22

2.5.1 Popular build systems 22

3 Dependency resolution: a comprehensive way 25

3.1 Research environment 25

3.2 DEx: a prototype tool for comprehensive dependency resolution . . . 27

3.2.1 The architectural overview of DEx 27

3.3 Dependency resolution: a build system approach 28

3.3.1 Retrieving dependencies from Autoconf 29

3.3.2 Retrieving dependencies from CMake 37

3.4 Dependency resolution: a source code approach 42

3.4.1 General process 44

3.4.2 Identify include files in C/C++ 45

3.4.3 Create the candidate artifact repository 48

3.4.4 Matching algorithm 51

3.5 Dependency resolution: a spec approach 54

3.6 Dependency resolution: a binary program approach 59

3.6.1 General process 60

3.7 Resolving duplicate dependencies 61

4 Evaluation 64

4.1 Study I: verifying correctness of DEx 64

4.1.1 Sampling of subjects 66

4.1.2 Results of Evaluation 66

4.2 Study II: evaluating dependency resolution techniques 67

4.2.1 Data preparation for analysis 68

4.2.2 Study II-I: Which technique retrieves more dependencies? . . . 71

4.2.3 Study II-II: Which technique is more accurate? 74

4.2.4 Study II-III: How do dependencies retrieved with different techniques compare with each other? 76

4.2.5 Study II-IV: Why are dependencies retrieved with different techniques not the same? 78

4.3 Discussion 80

4.3.1 Feature summary of dependency resolution techniques 80

4.3.2 Threats to validity 84

5 Conclusions **92**

5.0.3 Contributions 94

5.0.4 Future Work 94

Bibliography **95**

List of Tables

Table 3.1 Autoconf macros for dependency validation 34
Table 3.2 Database table: db_table_dep_rep 52
Table 3.3 Database table: db_table_deb_src_bin_pkg_match 55

Table 4.1 Sampled projects of study I 67
Table 4.2 Evaluation of DEx 67
Table 4.3 Database table: db_table_experiment_data 71
Table 4.4 Experiment data of project flac 72
Table 4.5 Statistical summary of coverage rate of the four techniques of all sampled projects in study II 73
Table 4.6 Statistical summary of accuracy of source code (clear mode), build, binary techniques of all sampled projects in study II . . . 75
Table 4.7 Correlation coefficient of any two dependency resolution techniques 78
Table 4.8 Correlation code with dependency resolution pattern encoded . 89
Table 4.9 Top 10 uniquely retrievable dependencies by technique 90
Table 4.10 Feature comparison of dependency resolution techniques 91

List of Figures

Figure 2.1 *Dependent libraries of hugin 0.7.0 retrieved by Dependency Checker.* 21

Figure 3.1 *A comprehensive approach to retrieve dependencies from multiple sources.* . . . 26
Figure 3.2 *High level architecture of DEx.* . . . 29
Figure 3.3 *An Autoconf build script excerpt shows the usage of dependency checking macros.* . . . 30
Figure 3.4 *General process to retrieve dependency from Autoconf build scripts* 31
Figure 3.5 *An Autoconf macro spans over one line.* . . . 33
Figure 3.6 *A CMakeLists.txt excerpt shows the usage of dependency checking macros.* . . . 38
Figure 3.7 *General process of dependency retrieving from build scripts of project using CMake.* . . . 39
Figure 3.8 *General process of dependency retrieving from source code of a project.* 43
Figure 3.9 *General process to identify include files in C/C++.* . . . 47
Figure 3.10 *General process to construct candidate artifact repository.* . . . 51
Figure 3.11 *Flow chart of include file - candidate artifact matching algorithm.* . . 53
Figure 3.12 *A sample Debian package control file* . . . 56
Figure 3.13 *A sample Debian source spec* . . . 58
Figure 3.14 *General process to retrieve dependencies from Debian document.* . . . 59
Figure 3.15 *General process to retrieve dependencies from binary files.* . . . 61
Figure 3.16 *General process to resolve name conflicts when merging dependencies retrieved with different techniques* . . . 62

Figure 4.1 *Process to perform study I* . . . 65
Figure 4.2 *Process to prepare* $D^{Sourcecode}$, D^{Build}, $D^{Binary+}$, *and* D^{Spec} *of subjects in study II* . . . 70
Figure 4.3 *Process to update database that stores experiment data for study II* 87

Figure 4.4 *Boxplot of coverage rate of the four techniques of all sampled projects in study II* 88
Figure 4.5 *Scatter plot of coverage rate of the four techniques of all sampled projects in study II* 88
Figure 4.6 *Boxplot of accuracy of source code (clear mode), build, binary techniques of all sampled projects in study II* 90

ACKNOWLEDGEMENTS

I would like to thank:

My supervisor, Dr. Daniel German, for his guidance, support, and extreme patience.

Dr. Micaela Serra, for her valuable advice and the happiness she brought to me.

My wife, Du Juan, who always gives me the strongest and unconditional support.

DEDICATION

To my wife and my parents.

Chapter 1

Introduction

Software is rarely built from scratch. Software reuse is the incorporation of previously developed software elements into a system under development [2]. Traditionally, these programs and libraries are provided in binary form so that reuse can be implemented by inheritance, calling libraries, SaaS (Software as a Service), etc. With the emergence of Free and Open Source Software (FOSS), software reuse has gained more and more popularity across the entire IT industry in the past decade. Open source software developers reuse code just like their counterparts in enterprises because developers usually operate under limited resources in terms of time and skills, and they always try to migrate development costs, integrate functionality quickly, and write preferred code whenever possible [19]. However, one of the main problems with component-based software reuse is the growing complexity of interdependencies among software packages, where the interdependencies of a package are the set of packages that are required to build and execute that package, but are not distributed with the original application [17]. The interdependencies of a package must be successfully installed in the environment before the package can be built, installed and executed. Understanding the attributes of interdependencies is very important for software reuse. For example, build-time interdependency such as a compiler or build system will not be required once the application is built from source code form to binary form; specific features of an application might rely on the availability of an optional interdependency.

In addition, an accurate and complete list of interdependencies is crucial for software reuse from legal perspective. Because interdependencies may rely on another set of packages, reusing one package usually means introducing a whole series of dependent packages with different licenses. Commercial software usually has a strict license

that prohibits or restricts its reuse. Nevertheless, even open source software licenses may have various restrictions. There exists over 60 widely used open source licenses as approved and listed by the Open Source Initiative [38]. Mistakenly integrating a package with an incompatible license may bring legal issues. Understanding interdependencies and their licenses thus becomes a real need to avoid potential lawsuit.

Unfortunately, dependency information is not always available. It is not uncommon for people to experience an installation error or runtime error due to a missing dependency but have nowhere to find what dependencies are missing. A user may expect dependency information recorded in the documentation that distributed along with the reused package. However, neither the availability nor accuracy of such documentation are guaranteed. In this thesis, we presented four distinct techniques for automatic dependency resolution within the Debian ecosystem. Each technique aims to resolve software dependency from a different source. The source code technique retrieves dependency by analyzing source code; the build technique focuses on extracting dependencies by parsing build scripts; the binary technique retrieves dependencies by resolving dynamically linked libraries of binary files; the spec technique retrieves dependencies from Debian spec.

The objective of our research is to offer various techniques to retrieve software dependencies so that when one technique is not available, a user may still have other options to retrieve dependency information. As the first study that comparatively analyzes various dependency resolution techniques, our research shows that using different techniques together to retrieve dependencies not only results in a more complete result but also helps identify falsely reported dependency through cross-validation.

1.1 Outline

The rest of this paper is organized as follows.

Chapter 1 Briefly introduce the importance of dependency resolution for software reuse and presented our solutions to retrieve dependency information.

Chapter 2 Discuss the background research related to the study of dependency resolution, in particular the prior work and knowledge that help build our dependency resolution techniques.

Chapter 3 Present the methodology of four dependency resolution techniques.

Chapter 4 Conduct a comprehensive evaluation in order to comparing dependency resolution techniques from a variety of aspects. Threats to validity and limitations of presented dependency resolution techniques are also discussed in this chapter.

Chapter 5 Provides a conclusion of the thesis, summarizes our contributions, and identifies future works in this research area.

Chapter 2

Background

Because of the availability of source code, less restrictions on copyright, and the great popularity among academic and commercial users, FOSS (Free and Open Source Software) becomes a hotspot and a major subject in academic research. Our research resides in the context of FOSS, software reuse, dependencies, and license issues. It is thus essential to have some knowledge about these concepts. This chapter begins with an overall introduction to FOSS.

2.1 Free and open source software

Software could be roughly classified into two categories, proprietary software and free and open source software. FOSS, which is officially recognized as the acronym for Free and Open Source Software by the United Nations Educational, Scientific and Cultural Organization (UNESCO), has underwent rapid development during the past decade. It is said that FOSS not only has a big impact on labour economics [26] but also significantly influences software development [45].

According to [26], there are three stages in the development of open source software. In the first era, which is from early 1960s to early 1980s, the sharing of source code by programmers from various organizations was prevalent. In the beginning, there is no such concept of software product. As IBM sold its computers with libre software that came with source code which can be freely shared, modified, and improved, all software is basically free and open source. Starting from late 1960s, as the concept of commercial and proprietary software began to emerge, independent software companies started to sell copyright software products for profits. The devel-

opment of FOSS progressed at a relatively slow pace until a breakthrough with Unix, which was created by the employees of AT&T's Bell laboratories in the 1970s. Unix was offered to academic institutions only for the price of media. This led to a great popularity of Unix in the academic circle and resulted in the creation of BSD (Berkeley Standard Distribution) by University of California Berkeley. Unix and Unix-like systems such as BSD start a major movement in open source software and have a significant impact on computer industry. Variants and decedents of Unix are still the backbone of some of today's most popular operating systems such as Apple Inc.'s Mac OS X and IBM's AIX.

The second era started in early 1980s and lasted for about a decade. One of the most significant events in this period was the foundation of Free Software Foundation (FSF) by Richard Stallman in 1985. One major innovation introduced by FSF was the formal licensing procedure (which is generally known as GNU General Public License or GPL) that aimed to preclude the assertion of copyright concerning cooperatively developed software [26]. GPL enforces any program that adopts, uses, changes, or even enhances GPL licensed components to be licensed under the same terms. This innovation made a great contribution to the booming of FOSS.

The third and current era started in the early 1990s. The development of open source software was greatly accelerated by Internet. Since FOSS was developed more collaboratively and shared more widely, the popularity, quantity and quality of FOSS was significantly improved. Thanks to the internet, the open source projects that were originally isolated could be integrated into more comprehensive software product or even complete environment. For instance, some of the most exciting events in this era were the creation of Linux by Linus Torvalds and the implementation of the BSD family (including FreeBSD, NetBSD, and OpenBSD). Another innovation was the diversification of licensing approaches. Software license refers to the kind of permission granted by a copyright or patent holder to use his or her intellectual property in a particular way [41]. The FSF's GPL licensing approach, though is essential to the development of FOSS, suffers from some serious problems [14]. To overcome these disadvantages, various FOSS licenses were invented to address issues in different situations.

These licenses generally fall into three categories [10]: (1) Academic licenses such as the BSD license and the MIT license, which only require licensees to give credit to the original authors. Software packages with these licenses are basically free to use without any limitation. (2) Keep-open licenses such as GNU LGPL (Lesser General

Public License) and Mozilla Public License. Modifications to software under these licenses are required to be open source too; however, larger works incorporating such works can remain proprietary. (3) GNU GPL (General Public License) like licenses, which require application that incorporates any software under this license be forced to open source too. This category of license has the strictest requirements to open source, and might not be appropriate for most commercial software. In fact, there exist over 60 widely used open source licenses that are approved and listed by the Open Source Initiative as of 2009 [38]. A study of FOSS licenses conducted by Capiluppi [4] shows that the most commonly used FOSS license is GPL (77%) , followed by LGPL (6%) and BSD (5%). Today, open source projects from various vendors such as Mozilla, Apache, and Eclipse are intensively adopted by industry for both free and commercial use.

2.1.1 FOSS: Advantages and Disadvantages

Compared to proprietary software, FOSS has many advantages in terms of cost, security, ease of development and reusability. The most obvious advantage of FOSS is its low cost. Because consumers don't need to pay for a license, FOSS seems very attractive in the beginning. The total cost, however, may not be trivial after training fees and maintenance fees are considered. Because FOSS usually precludes its responsibility to software defects and any loss caused by these defects, its users (particularly enterprise users) sometimes need to pay for service and maintenance. But for most users of FOSS, they usually don't need to purchase service, and this makes FOSS' unbeatable advantage in end user market. In the end, FOSS provides the possibility to adopt sophisticated technologies which improve software development process and quality at a low cost. Commercial software companies offer a complete product line (for example the requirements engineering tool such as IBM Rational RequestPro, the project management tool such as Microsoft Office Project, the integrated development environment such as IBM Rational Application Developer, the version control and defect management systems such as IBM Rational ClearCase and ClearQuest) to support the full software development process. It is no doubt these commercial software products have significantly increased the productivity and quality of software development, but they are usually very expensive. Nowadays, FOSS offers alternatives to almost every commercial product for software development. There is Linux versus Windows in operating system, PostgreSQL versus Microsoft SQL Server in database, SubVersion

versus IBM ClearCase in version management. Such open source projects increase the accessibility of complicated software products that support important software engineering processes. Because of them, even small organizations and startup companies can adopt the latest technologies to assure their product quality at a very low cost.

The most obvious benefit of FOSS to software developers is the opportunity to base a design on existing software elements [45]. As claimed by the FSF that 'free software is software that gives you the user the freedom to share, study and modify it' [25], software reuse has been one of the core philosophies of FOSS. FOSS and software reuse are mutually promoted. Due to the benefits brought by FOSS, software reuse is prevalent in both enterprises and open source community. For example, [19] found that open source software developers routinely and widely reuse software components. Another study analyzed 25 projects at NASA and discovered that 32% of the modules within those projects were reused from prior projects [42]. Generally speaking, the availability of source code significantly improves the reusability of FOSS. It also brings a side benefit: the publicly accessible code repositories (including Google code search, Koders, Merobase, etc.) containing millions of lines of code from fine quality software product that are available for investigating for learn and research purposes.

On the other hand, FOSS is not perfect and still suffers from some limitations and drawbacks. For instance, FOSS is not supported by hardware manufacturers as well as proprietary software; many commercial software companies are hostile to FOSS which leads to FOSS' compatibility problems; the source code of many FOSS products are not well documented which makes them hard to understand; there is still a long way to improve usability for FOSS [37]. Nevertheless, it is safe to say that FOSS has completely changed the way to develop and use software products and has developed to be the backbone of software industry.

2.2 Software reuse

As stated earlier, the most obvious benefit brought by FOSS is the increased possibility of software reuse. Software reuse enables developers to use past achievements, and it improves software productivity and quality [42]. Thanks to the abundance and availability of FOSS, the popularity of software reuse has dramatically increased across the entire IT industry in the past decade. Not only does open source community, but also enterprises widely reuse open source libraries, applications, or components in

their products. For example, most IBM flagship software products such as Lotus Notes, Rational Software Architect, and WebSphere Business Modeler are now based on the Eclipse platform, a heavily used open source project. As FOSS has shown its value to bring profits to commercial enterprises, more and more software companies (many of which used to fight against FOSS) start to embrace FOSS and adopt FOSS [26]. It should be noted that software reuse is also prevalent in closed source software. Our research focuses on software reuse in FOSS because of availability of its source code and less restrictions on copyright.

2.2.1 What is software reuse

Software reuse can take place at many levels, from low level reuse such as a few lines of code, a bunch of functions, some classes or interface, to high level reuse such as integration of a component or even an entire application. Numerous software artifacts such as requirements, architecture, design, documentation, source code, or test cases may be reused [13]. In general, there exists three broad forms of code reuse, including algorithms and methods, lines of code, and components [19].

Reuse of algorithms and methods

Simply put, an algorithm is a set of instructions to perform a specific task or make computations for a final state based on an initial state. One algorithm may spread over several methods and one method may also contains more than one algorithms. Reuse of algorithms falls into the realm of knowledge reuse. Opening any book about data structure and algorithm analysis, though the implementation may differ based on programming language or the code developer, the principal idea of the algorithms remains. Developers may acquire the knowledge of new algorithm from technical books, computer journals, documents of software products, publicly accessible source code repositories, or even online discussion forums. In Haefliger and Krogh's study [19], nearly all of the developers they interviewed mentioned the reuse of algorithms and methods in their open source software development. Reuse of algorithms and methods is frequent and popular, but it is not well credited. It is usually not possible to completely identify those algorithms and method reuse that are not accompanied with source code reuse.

Reuse of source code

There is no easier way to realize software reuse than simply copying and pasting. The internet based code repositories such as Google Code Search and SourceForge.net provides ample source code for FOSS developers to learn and reuse. Copying a section of projects from such code repositories is the most direct means to realize software reuse. This approach, however, may lead to serious issues (such as plagiarism) regarding intellectual property if used inappropriately.

Component based software reuse

The most widely adopted and efficient means to reuse software is the component based approach. Sofware components are binary, independently deployable building blocks, that can be composed by third parties [47]. Programs offering specific functions are usually encapsulated into a component for reuse by other developers. These components provide APIs and come with user manual or descriptions.

Reuse of components can be realized by linking or connecting an application to the components; executing the component from an application; or simply integrating the components into the project. An application can link to a component either at compile time (in which case called static linking) or runtime (in which case called dynamic linking). Third party components are usually considered as a black box by developers. Application developers directly interact with the APIs provided by the components, and don't care too much about the components' internal structures as long as the components offer satisfactory result and performance. Sometimes, when the reused components do not totally meet the users' expectations, developers may adjust or reprogram part of the components. In this situation, the components become white box to developers.

A software application usually adopts more than one components to provide its comprehensive functionality. Such behavior could create complex connections between components and the host application, hence it is necessary to explore the relationships between components. Capretz and Li in [5] describes several relationships, including has-a, is-a, uses-a, and is-part-of. Corresponding to these relationships, four pre-defined schemes (compose, inherit, use, and context) are proposed to describe the relationship network of an application.

2.2.2 Benefits of software reuse

The purpose of reuse is to improve software quality and productivity [29]. The benefits brought by software reuse are obvious and direct. First, reused software components usually have good quality. Open source software components enjoy the intrinsic benefits of FOSS (such as cost, security, scalability, etc.) Unpopular reused components (such as a low ranking project in SourceForge.net) may have doubtful quality, but well-known software projects (which are heavily reused across open source community and enterprises) are well designed, developed by enthusiastic and skillful developers, and tested by thousands to millions of global users. Since defects are found and fixed rapidly in FOSS [40], it is not surprising that these mature components enjoy good quality.

Second, reusable software provides comprehensive functionalities. There is a large collection of reusable libraries and components in the open source community. There is no doubt that developers may implement required functionalities on their own, but in many cases, such effort is more than what they could afford. Besides, they could always modify FOSS reusable elements.

Third, as FOSS promotes software reuse, the high volume use of reusable FOSS elements may also improve their own quality by collaboratively fixing defects and contributing new features. As claimed in [45], this factor often mitigates the risk of orphaned components or incompatible evolution paths that are associated with the reuse of proprietary components.

Moreover, component-based software reuse is one way to reduce the problems of legacy system maintenance [50], which has been traditionally estimated at 50-75% of total software life-cycle costs [28].

No matter what, reusable software not only saves developers' time and effort but also saves the money of the organization that develops the application. Since software reuse offers many benefits, even enterprises support and offer their own technologies (e.g, Microsoft's COM+, IBM's Component Broker, etc.) to promote software reuse.

2.2.3 Problems of software reuse

Like a double edge sword, software reuse may also cause problems. Take the abundance of FOSS as an example. The incredibly huge amount of open source libraries supply software developers endless resource for software reuse; however, the quantity, quality, and organization of reusable libraries also make them problematic to use or

even to find. In addition, since a consumer may search for a reusable element with a keyword different from the terminology used by the creators of reusable software elements, the discordant terminologies used by software development professionals and open source software developers may bring problems when identifying such components.

Another shortcoming of software reuse is particularly significant for FOSS. The voluntary nature of FOSS determines that FOSS may not be planned and designed as well as close source or proprietary software. Such shortcoming would result in more complex code and less modular architecture for FOSS, which increases the complexity of the applications that reuse FOSS elements, limits the extensibility and reusability of such applications, and increases the cost of maintenance as well.

Furthermore, software reuse may increase the chance of the so-called orphan component. An integrated reusable component that is not updated and maintained in time may isolate the evolution path of such component from its original one and create an orphan component. Such case is particularly likely to happen when software reuse is in the form of code reuse (though this problem might also occur when reusing binary components). Developers may directly integrate the source code of reusable software elements into their own applications. In such situation, any changes, fixes, or upgrades made by the original author of the reused elements will not be reflected in the applications. Such divergent evolution paths slow the propagation of new functionality and security fixes, and not only increase the risks of reusing source code, but also waste valuable development resources (in which case the application developers fix a bug that is already fixed in the newer version of the reused software element, but not included in the version of software code that the application uses).

Incompatible licenses may be another undesirable consequence of software reuse. It is not unusual that an application may comprise several modules with different licenses. Without properly planning, there is quite a chance for an application to adopt reusable components with partially or completely incompatible licenses. Even the way to bind components may affect the compatibility of licenses.

Dependency analysis may be a constructive and effective means to solve many software reuse related problems. Our research focuses on the problem of dependency identification. The next section makes a comprehensive review of dependency related knowledge.

2.3 Software dependency

Applications are hardly isolated: they are subject to complex dependencies that have to be satisfied for the whole system to work [18]. Dependency is everywhere. Software artifacts, from as small as snippet of source code, some header files, a plug-in, a shared object in Linux, a DLL in Windows, to as large as a package level component, a database or even an entire application may all create dependencies. A modern software product is usually composed of modules. Maintaining reused modules could be a complex work. Modules may have various versions for different operating systems or different hardware architectures. Considering together with the fact that reused modules may have exchangeable components, software dependency may become extremely complex. If not handled well, problematic dependencies could make software hard to install, uninstall, use and maintain.

Giving the importance and complexity of dependency management, one of the most important issues of software reusing is to understand the dependency relationship between a software product and its reused components [50]. Software dependency creates coupling between software modules. Software reuse issues such as license incompatibility may become particularly dangerous when such coupling is unknown. Having a better understanding of the dependency relationships between an application and its components may help solve or relieve the following problems:

1. Help developers understand the software architecture, hence overcome the increased architectural complexity brought by software reuse.

2. Potential law suits may be avoided if problematic modules with incompatible licenses can be located before intaken. This problem is a serious subject matter in software industry, therefore, considerable effort has been done both in industry and academic circles on the topic of license analysis. License analysis is the precondition for legally reuse of software components, and dependency analysis is the first and very fundamental step of license analysis.

3. Orphan components may be identified and managed.

4. May be used as a basis for program optimization, debugging, and testing [46].

As such, it is essential that software dependency relationships be identified and solved to secure effective software reuse and ultimately, better software.

2.3.1 What is a dependency

A FOSS application is an integration of artifacts that can be used to compile, build, and execute that application [17]. The application may be further divided into multiple components, which are capable of being installed independently of the rest. Based on such theory, we may generally classify dependency into two categories: inner-dependency and inter-dependency.

We use the terminology inner-dependency to refer to the coupling within the same application. For example, a .cpp file may depend on a .h file; a method in class X may call a method in class Y that is in the same application of X. We assume all the resources (such as libraries, classes, interfaces, methods and even variables) relevant to inner-dependency be within and distributed along with the original application. In other words, the author of the application should create all the inner-dependency related resources, which should not include third party software elements that are not solely created for and originally distributed with the application.

On the other hand, German in [17] defines inter-dependencies of a package as the set of packages that are required to build and execute the package, but are not distributed with the original application. Compared to inner-dependency, inter-dependency describes the external dependency relationships that concern with packages that are not originally distributed with and independent of the application.

An inner-dependency describes the internal structure of the application while an inter-dependency faces the problems introduced by software reuse. Our research is about software reuse of FOSS, therefore we only focus on inter-dependency. Unless explicitly mentioned, the word 'dependency' in the rest of paper refers to inter-dependency.

Classification of dependencies

Comprehensive research has been done in [17] on the topic of dependency. German classifies dependencies based upon the following criteria:

1. **Type**: dependencies can be explicit or abstract. An explicit dependency states the name and possibly the version of package that satisfies the dependency. Abstract, on the other hand, states a set of packages whose element will have the same functionality, although their implementations may be completely different. Any package in the set will satisfy the dependency. Take the Java Runtime Environment (JRE) as an example. Many software products depend

on JRE to run. Sun provides the original JRE while Apache foundation also produces Apache Harmony that is compatible with Sun JRE. In this example, Sun JRE and Apache Harmony can satisfy the abstract dependency of Java runtime environment.

2. **Importance**: dependencies can be either considered required or optional. Required packages are those critical dependencies that must be satisfied. Absence of required packages may lead to failure of the application. Optional dependencies are those packages that are not critical to an application, but without them some features may be missing.

3. **Stage**: depending on the phases where dependencies may be needed, dependencies may be categorized into build time dependencies (such as a compiler), installation dependencies (such as tools to modify configuration files), test time dependencies, and runtime dependencies such as dynamic libraries.

4. **Usage method**: based on the usage method, dependencies could be standalone programs (such as a compiler), middleware-based components (such as COM+ or CORBA), plug-ins (such as an Eclipse plug-in), and linkable libraries (including both static linkable libraries and dynamic linkable libraries).

Li in [27] categorize dependencies into data dependency, control dependency, user interface dependency, time dependency, state dependency, cause and effect dependency, input/output dependency, and context dependency.

1. **Data dependency**: represents data defined in one component but used in another one.

2. **Control dependency**: implicit dependencies produced by control integration.

3. **User interface dependency**: interactions between components triggered by events in user interface.

4. **Time dependency**: interactions between components must be conducted in a specific order.

5. **State dependency**: the state of one component decides the behavior of another.

6. **Cause and effect dependency**: behavior of one component implies the behavior of another.

7. **Input/output dependency**: a component produces/requires information from/to another.

8. **Context dependency**: a component must be running under a specific context.

Properties of dependency

A dependency indicates the dependent relationship between one package and another. There are several properties that describe a dependency [17]:

1. **Version**: an application may depend on an exact or a minimum version of a specific package.

2. **Source**: the software application that the dependent package is created from.

3. **License**: the license of dependent package must be compatible with the application's own license.

4. **Cost**: costs to use the dependent package. Such cost includes monetary cost to purchase a license of the dependent package, footprint cost such as the memory and disk space the package would consume, and maintenance cost such as the effort and difficulty to build, install, and use the package.

5. **Its own dependencies**: the dependent package may further depend on other packages to build, install, or run. It is thus necessary to know the dependencies of a dependent package.

2.4 Extraction of dependencies

The main goal of our research is to find a comprehensive and effective dependency resolution solution, which may benefit software reuse in numerous ways. The core step of dependency analysis is to extract dependency information from software elements so that we may analyze the relationships between these elements. The dependency information can be retrieved from multiple sources such as source code, build scripts, binary code, documentation. This section makes a comprehensive review of current research on dependency extraction and resolution.

2.4.1 Dependency resolution from software documentation

The most convenient way to learn the interdependencies of a package is to check its documentation that is released with it. Such documentation could be a readme file, a user manual, a release note , a web page or in any form that can be interpreted by its readers. The advantage of documentation is that it is the most direct and convenient source where dependency information can be found. However, dependencies listed in the documentation could be incomplete, invalid, or contain more dependencies than truly needed. Moreover, because software documents can be written in any style or language, a universal solution to automate the process of retrieving dependencies from software documentation hardly exists. However, some automatic dependency resolution technique is available in a specific context.

Automatic dependency resolution in Debian environment

FOSS distributions such as Debian is made of several thousands of packages that have complex dependency relationships among them [24]. A package, which usually corresponds to a library or an application, is a bundle that contains a component, all the data needed to its correct functioning and some metadata which describe its attributes and its requirements with respect to the environment in which it will be deployed [33]. In Debian, there could be either source projects or binary packages. Source code is included in a source project to build one or more installable binary packages. For each package, a set of binary distributions may also be created, each of which corresponds to a specific architecture (such as i386 or amd64).

German in [17] and [18] proposed an automatic method to draw a complete inter-dependency graph of packages in Debian by analyzing their documentation. [17] uses two passes to parse the source project descriptions and binary package descriptions, collecting the dependent packages listed in the field Depends, Pre-Depends, Build-Requires, Recommends, Suggests, and Provides to retrieve the list of dependences for a specific package. The researcher than uses the retrieved dependency information to draw an inter-dependency graph (IDG) of that package.

2.4.2 Dependency resolution by analysing source code

The source code approach may be divided on the basis of programming languages. The most frequently used programming languages in FOSS development are C, C++,

Java, and Perl [50]. As such, most researchers choose to analyze programs written in these languages for the purpose of dependency resolution. Many reverse engineering tools (including both open source and commercial) can be used to retrieve dependencies by analysing source code. The majority of these tools, however, focus on statically analyzing inner-dependency relationship of the target program to gain a better understanding of its architecture. By parsing the source code, such tools may generate a call graph for the methods or describing how each source file depends on another. How well the target project is related to an external project or library, i.e. the inter-dependency, is not investigated as heavily as inner-dependency.

Dependency analysis of C and C++ programs

In C and C++, the source files must include all the header files that the program depends. Moura in [8] analyses dependencies for large complex programs written in C and C++. Their method parses the source file, looking for #include clauses to find dependency relationship between source files and header files.

[16] presents a theory that identifies candidate modularization by analyzing source files (*c, *.cpp) and header files (*h). To achieve this goal, the paper introduces a tool - srcdep, which analyzes the source code to draw the interdependency graph between sources and headers. The header-source interdependency graphs are then processed based on various graph theories (including bridge detection, edge-cut detection, cut vertex detection, cut set detection) to identify software modules. Their method applies the graph theory that modularizes the software at file level for better understanding of software structure and dependencies. It also presents the classification of dependencies between source and header file relationship (i.e. header-header, source-source, source-header), and the theory to simplify the relationship graphs for better understanding of software dependencies.

Dependency analysis of Java programs

Ossher in [39] presents a method that can automatically resolve inter-dependencies for `FOSS` written in the Java programming language by cross-referencing the project's missing type information with a reliable code repository of candidate artifacts. There are three steps in this approach:

1. **Creation of artifacts repository**: the first step of this approach is the establishment of a reliable artifact repository that is comprehensive enough to

include possible dependencies for the majority of FOSS and indexed in a way such that components can be retrieved on the basis of their type information. The author chose Apache Maven 2 Central Repository [1], which is a comprehensive and reliable collection of Java artifacts that are frequently used. To focus on inter-dependency resolution, the author customizes the Maven repository by extracting and indexing all the provided types of artifacts in Maven repository. To accomplish this, the author uses a stripped down version of Sourcerer [2] feature extractor to extract entities that can only be used externally, including interfaces, classes, annotations, enums, fields, and packages.

2. **Identification of missing types**: similar to #include clause in C and C++, the import clause is used in Java to indicate dependency. To identify the missing dependencies, the method parses Java source code, searches for the use of import clause, and uses the Sourcerer feature extractor to report missing types to establish the collection of missing types for a Java project. One problem, however, is the difficulty to reliably determine the Fully Qualified Name (FQN). Such problem is brought by Java's ambiguity function when using import clause (e.g., using import x.y.* instead of using x.y.z). Fortunately, since the method does not focus on reconstructing type names for partial program, it is safe to simply ignore all the unsolved import statements and missing FQN.

3. **Resolution algorithm**: in order to match the identified missing types against the artifacts in the repository, the author developed a simple greedy algorithm that repeatedly picks the artifact that provides the largest amount of missing types, discounting missing types that already provided by previously included artifacts, and iterates until either no missing types or artifacts remain.

It is important to mention that besides Apache Maven repository, there are many publicly accessible code repositories, including Google Code Search [3], Merobase [4], Koders [5], and findJAR [6]. Tools such as Rascal [36], Prospector [34], ParseWeb [49], Strathcona [20], and Code Conjurer [21] have been developed to assist software reuse

[1] Maven 2 central repository, http://repo1.maven.org/maven2/
[2] An infrastructure for large-scale indexing and analysis of open source software [30]
[3] Google code search, http://www.google.com/codesearch
[4] Merobase, http://www.merobase.com
[5] Koders, http://www.koders.com
[6] findJAR, http://www.findjar.com

by collaborating with popular code repositories. Researchers may use similar technique as presented in [39] and the code repositories (with assistance of the tools mentioned above) to perform dependency resolution for programs written in other programming languages.

Support of other programming languages

Tuunanen in [50] developed a tool C ASLA, which uses GCC to retrieve dependencies. ASAL supports any programing language as long as it can be compiled by GCC. Because GCC only produces information on source code dependencies, ASLA uses modified versions of linker (ld) and archive builder (ar) to collect information about the actual binary level dependencies. This approach is very practical and effective in generating interdependency information. The author claims that ASLA is the only known license analysis tool that provides full information on build process outputs and their dependencies. ALSA, however, is not able to identify runtime dependencies, neither is it able to identify plug-in sort of dependencies. Another drawback is that the packages need to be compiled, which could be time consuming.

2.4.3 Dependency resolution by mining software repository

The MSR (Mining Software Repository)[7] approach, i.e mining the evolution of software is useful and effective in discovering dependencies of large software applications with rich history repositories. By examining different versions of a program and witnessing the co-changes of code, the MSR approach may detect hidden dependencies that are usually not possible to be found by other means. The applicability and precision of the MSR approach, however, are limited by the availability and quality of software repositories.

Gall in [15] presents an approach that reveals logical dependencies within the evolution of particular entities by analyzing classes of CVS release history data. The major contribution of this paper is the QCR[8] methodology, particularly the relation analysis technique, which can reveal hidden interdependencies that may not be found by regular static or dynamic source code analysis. The QCR methodology investigates historical development of classes regarding time when classes are added or updated,

[7]Please refer to [23] for a comprehensive literature review that provides a wide spectrum of software repository mining related research.

[8]QCR, **Q**uantitative analysis, **C**hange Sequence analysis, and **R**elation Analysis

and the attributes related with changes of classes. The found common changes, aka logical coupling, reveal the hierarchical organization of classes and uncover hidden dependencies. Three techniques, including quantitative analysis, change sequence analysis, and relation analysis, are used to implement the QCR methodology. With relation analysis, which compares classes and reveals class dependencies, the evolution of classes is compared to find those classes that were most frequently changed together. Authorship and date, which can be extracted from version control system, are used for the comparison.

Ying in [51] also describes an approach which can determine change patterns (that files were changed together frequently in the past) and reveal valuable dependencies that may not be apparent from other existing analyses by mining software repositories. The approach is composed of three stages. Stage 1: Data preprocessing, which extracts information from Software Configuration Management (SCM) system; stage 2: association rule mining; stage 3: query, which applies the data mining algorithm to the preprocessed data results in a collection of change patterns. An evaluation on Eclipse and Mozilla by this approach shows that it can reveal valuable dependencies that may not be apparent from other existing analyses.

2.4.4 Dependency resolution by processing binary files

The most common way to distribute software is using binary files. Binary files are almost the only way to distribute closed and proprietary software products. Most `FOSS` products are also available in the form of binary file for the convenience of end users. As such, a method that may extract dependency information from binary files could be extremely useful. The way to process binary files varies upon operating systems and platforms.

Dependency networks can be constructed on Windows computers using memory profiling tools, and determining interactions based on shared .DLL (Dynamic Library Link) files and Active-X controls [24].

Dependency analysis tools are also available for Linux. The Linux Foundation created the Dependency Checker Tool [3], which is able to discover the linkages occurring between a given binary and any given library. The goal of this tool is to flag problematic code combinations based on predefined linkage and license policies. Dependency Checker Tool can detect both dynamic and static linkages. We can use either the CLI interface or GUI interface of Dependency Checker to re-

```
/home/.../hugin_0.7.0..._i386/usr/bin/hugin_stitch_project:
libhuginbase.so.0.0
libpano13.so.0
libtiff.so.4
libwx_baseu-2.6.so.0
...
WARNING: Could not check for static dependencies
/home/.../hugin_0.7.0..._i386/usr/bin/hugin:
libhuginjhead.so.0.0
libhuginbase.so.0.0
libboost_thread-gcc42-mt-1_34_1.so.1.34.1
libpano13.so.0
libhuginvigraimpex.so.0.0
libtiff.so.4
libwx_baseu-2.6.so.0
...
libdl.so.2
WARNING: Could not check for static dependencies
/home/.../hugin_0.7.0..._i386/usr/bin/nona_gui:
libhuginbase.so.0.0
libpano13.so.0
libtiff.so.4
libwx_baseu-2.6.so.0
...
WARNING: Could not check for static dependencies
```

Figure 2.1: *Dependent libraries of hugin 0.7.0 retrieved by Dependency Checker.*

trieve dependencies for a binary file or a directory of binary files. For example, hugin_0.7.0 svn3191+beta5-2_i386.deb (which can be found at the i386 distribution of Debian 5.0.7) can be extracted into a directory. By applying command *./readelf.py /home/hanyuz/Desktop/hugin_0.7.0 svn3191+beta5-2_i386/*, Dependency Checker is able to generate the list of dynamic linked libraries that hugin 0.7.0 depends on (see figure 2.1).

2.5 Build system

The sources where dependency information may be retrieved from are not limited to the ones listed above. Configuration scripts of build systems are a promising source from which valuable dependency relationships can be observed. Build systems play a vital role in the creation and production of software products. A build system processes a set of source code and resource files based on a series of configuration scripts to generate executable programs in the creation and production phases of software development. By specifying platform-specific parameters and build dependencies, build systems are able to provide a universal yet user friendly interface for selecting desired features and environment as well as efficiently invoke construction tools such as compilers in appropriate order [1]. Considering the difficulty to compile and handle software programs, build systems are quite effective in leveraging the variety and complexity of programming languages and promoting the portability of source code packages. As such, build systems form the backbone of modern software development [1].

2.5.1 Popular build systems

There are numerous build automation software products in market[9]. In this section, we make a brief introduction to some of the most popular build systems used in `FOSS` development.

GNU build system

The GNU build system has been widely adopted for build automation of both `FOSS` and proprietary software products. The major components of the GNU build system include Autoconf[10], Automake[11] and Libtool[12]. Programs such as GNU make, pkg-config, and GCC[13] are often used together to support build aumotation.

Automake is mainly used for the generation of portable makefiles. As an improvement of GNU make, which lacks the support of portability, dependency tracking, recursive builds in sub-directories and many useful features, Automake allows a project

[9] Please refer to http://en.wikipedia.org/wiki/List_of_build_automation_software for a comprehensive list of these products
[10] Autoconf, http://www.gnu.org/software/autoconf/
[11] Automake, http://www.gnu.org/software/automake/
[12] Libtool, http://www.gnu.org/software/libtool/
[13] GNU Compiler Collection, http://gcc.gnu.org/

to specify build needs in a file called Makefile.am which is way more powerful than a plain old make file yet keeps a simple and user friendly syntax. Unlike repetitively inserting redundant targets, the makefile.in files generated by Automake supports not only all standard targets but also valuable features such as dependency tracking. The makefile.in file will be then used with Autoconf.

Autoconf, written in shell scripts and GNU M4 macros, is a tool for producing portable shell scripts that automatically configure source code that is compatible with various unix like operating systems. The shell scripts (i.e. the Autoconf configure scripts) are responsible for performing essential tasks such as logging and dependency checking. The created Autoconf configure scripts is based upon a file called configure.ac/configure.in, which contains Autoconf macros that test and validate system features required by the target software package. configure.ac and configure.in are the same, except configure.in is promoted by previous version of Autoconf whereas configure.ac is now the preferred file name. There are many standard Autoconf macros available to use. The user is always able to perform customized tests based on Autoconf template macros. In such case, aclocal.m4 and acsite.m4 files may also be created. Although can be used independently, Autoconf and Automake are often used together.

It is not uncommon that software developers may prefer to contribute their work as a shared library to benefit other programs. One important feature of a shared library is that one copy of a shared library should be used by multiple linked programs at the same time but can still be updated independently of any linked programs. It is however not an easy job to make shared libraries portable. Libtool frees the burden of creating portable shared libraries by hiding portability issues behind a clean, portable, and consistent interface. Like Autoconf, Libtool is also frequently used along with Automake.

CMake

CMake[14] is another popular open source build utility focusing on building C/C++ projects in a platform independent manner. Similar to configure.ac/configure.in in Autoconf, CMake has a configuration script CMakeLists.txt that functions as the reference to perform environment parameter checking and application (e.g., executable, library) construction. By processing one or more CMakeLists.txt that reside in direc-

[14] CMake, http://www.cmake.org/

tories and subdirectories of a project, it is easy to generate standard build files even if the directory hierarchy is complex. One feature that distinguishes CMake from other build system is that CMake is designed to work in conjunction with native build environments and has the ability to leverage the differences of these build environments (no matter it is a Visual Studio project on Windows or makefiles on Unix-like systems). Such feature makes CMake quite powerful to support development teams (in particular small organizations [35]) creating portable software products. Considering the other useful features (e.g., static/dynamic library creation, source code compilation, wrappers generation), the popularity of CMake has grown steadily. One famous example could be KDE[15] switched its build system from Autoconf to CMake in 2006.

Ant

Apache Ant[16] is a platform independent and highly extensible open source build system written in Java and best suited for Java projects. Like CMake, Apache Ant solves an intrinsic flaw of make like build systems - portability on operating system. Make like build systems use shell based commands to describe targets, parameters, dependencies and other artifacts that are participate in the build process. Such design releases the power of shell scripts but restricts its application to a specific type of operating system, usually Unix-like systems. Apache Ant, on the other hand, describe configuration parameters in a pure XML file - build.xml based on which a series of tasks get executed. Rejecting the creating platform specific shell scripts approach, Apache Ant provides a rich collection of built-in functionality that have universally identical behaviours across platforms.

Other build systems

There are many open source and proprietary build automation systems available in the market. Apache Maven is a project management tool that can manage the build process of a project based on its Project Object Model (POM). Other widely used systems include Hudson, MSBuild, AnthilllPro, etc. Each build system has advantages but may also suffer from several limitations. As such, different build systems are sometimes used together to achieve the best result.

[15] KDE, http://www.kde.org/
[16] Apache Ant, http://ant.apache.org/

Chapter 3

Dependency resolution: a comprehensive way

In this chapter, we present four dependency resolution techniques: the source code technique, the build technique, the binary technique, and the spec technique. The binary and spec techniques are developed based on the research presented in [17] and [3], respectively. Although similar research has been done for other programming languages (e.g. Java), the presented source code technique is the first study to retrieve inter-dependency information from C/C++ source code. The build technique is the first study to discover dependencies from software build scripts. Each technique retrieves dependency information by analyzing a specific source and can be used alone for dependency resolution. However, because of limitations of each technique, dependencies retrieved with a single technique may be incomplete and inaccurate. The objective of our research is to develop a comprehensive approach for dependency resolution that can take advantage of different techniques and balance their shortcomings. As illustrated in figure 3.1, the main principle is to apply various techniques to the same target project so that multiple sets of interdependency data can be retrieved (each set corresponds to a particular technique). The retrieved sets will be cross validated and merged to achieve the super set of dependencies.

3.1 Research environment

Our research intends to resolve dependency problems specifically for free and open source software. The ultimate goal of dependency resolution is a universal solu-

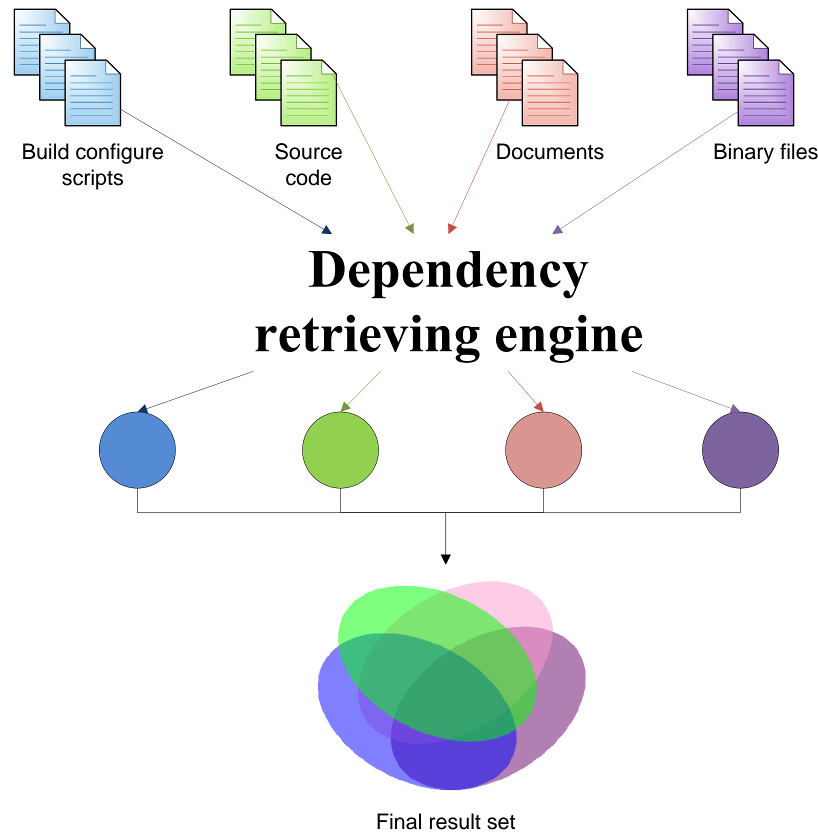

Figure 3.1: *A comprehensive approach to retrieve dependencies from multiple sources.*

tion that is workable for any FOSS (or even commercial) product, however, lack of standardization, variety of sources, and limitation of each dependency resolution approach, make such universal solution hardly exists. For example, software documentation can be written in any format or style, making it impossible to write a universal parser to recognize documentation of every software product. Therefore, it is practical to restrict our research to a certain environment. Some research focuses on a specific product or project. For example, projects such as Mozilla Bugzilla, Firefox, Apache Web Server, Eclipse, and various Linux distributions have been intensively researched. Our research focuses on inter-dependency, which explores dependent relationship between different projects in a software eco-system. A software eco-system is a collection of software projects which are developed and co-evolve in the same environment [32]. As far as our research concerns, such eco-system is Debian, which is a canonical representative of FOSS. Debian is a free distribution of the Linux operating system with more than 29000 packages [9]. Debian is better suited for our

research because of its popularity in open source community and it avoids biases and contains unique information only available in an integrated environment [44]. Since its creation in 1993, Debian has evolved to be the most popular and complete Linux distribution with over 25,000 packages maintained by a large community of individuals all around the world. With licensing and interdependency information available for most Debian packages, Debian has become a valuable source for the research of software evolution and intellectual properties. Debian's history and comprehensiveness make it attractive to study as it represents a large universe of `FOSS` projects [44]. Some researchers also favor SourceForge.net because it is the worlds largest open source software development website with more than 260,000 registered projects [43]. However, results derived from SourceForge.net can be biased as it is intended specifically as an incubator for small projects [44].

It should be noted that our research focuses on Debian; the accomplishments, however, may still benefit the research of dependency resolution in a wider spectrum. This chapter introduces in detail our techniques to extract interdependency information by analyzing build scripts, program source code, user documentation, and executable binary programs.

3.2 DEx: a prototype tool for comprehensive dependency resolution

The proposed comprehensive dependency resolution approach is realized with a prototype tool `DEx` (**D**dependency **Ex**tractor). The Perl programming language is selected for the development of `DEx` due to its powerful text processing capability, easy to use and support for regular expression processing.

3.2.1 The architectural overview of DEx

`DEx` is designed as a comprehensive tool that can retrieve interdependency information from different sources. A user may use one or more dependency resolution techniques supported by `DEx` to retrieve dependencies, which will be reported in a text file as well as inserted into a PostgreSQL database. Dependency records stored in the database will be processed to retrieve the comprehensive list of dependencies that includes the results produced by each technique. Figure 3.2 illustrates the high level architecture of `DEx`. The system is composed of several modules:

- An individual module for the implementation of each dependency resolution technique. DepAnalyzerProgLang.pm, DepAnalyzerBuild.pm, DepAnalyzerBin ary.pm, and DepAnalyzerDox.pm implement the source code technique, the build technique, the binary technique, and the spec technique, respectively.

- DepAnalyzerComprehensive.pm processes and merges the dependency records stored in the database to produce the most comprehensive result.

- Utility.pm: a utility module that provides methods (such as string_truncate, compare_package_name, compare_version, etc.) commonly used by other modules.

- DatabaseManager.pm: a database management module that centralizes database management functionalities such as connection management.

- DEx.pl: a central management module that accepts user input and triggers corresponding modules for dependency resolution.

- dex.conf: Configuration information (such as database authentication information, file path of generated reports and logs, etc.) is stored in dex.conf, which can be accessed by each modules through methods in the utility module

3.3 Dependency resolution: a build system approach

We propose a method to extract interdependency information of a software project by processing its build scripts. As stated earlier, there exist three major `FOSS` build systems, including CMake, GNU Automake/Autoconf, and Ant, which have been widely used to build applications from source code to executables. To use these cross platform build systems, the project has to have one or more build scripts (configure.ac or configure.in for Autoconf, CMakeLists.txt for CMake, and build.xml or ivy.xml for Ant), within which dependent packages are specified (including package name, minimum required version, and whether they are optional or required). By processing the build scripts, build systems check the existence of the dependencies on the target system and make sure the system is ready to build the executables. Unlike software documentation, build scripts usually have a sound structure and a well-defined

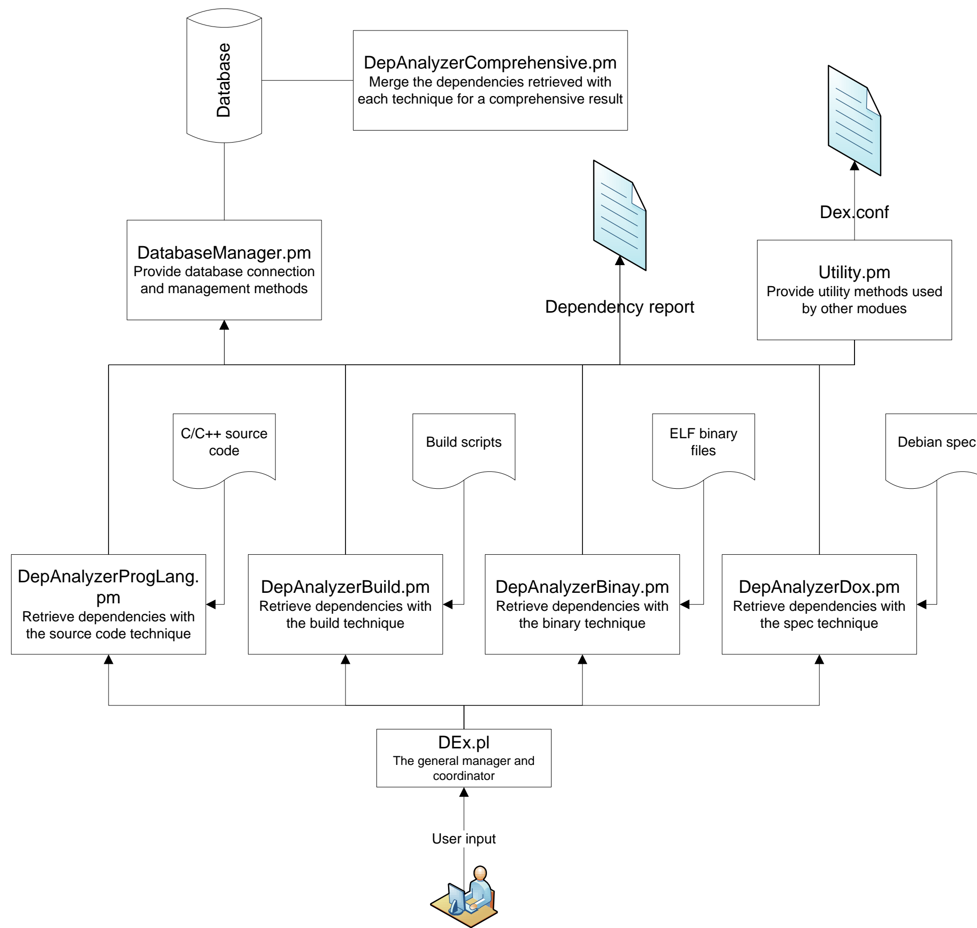

Figure 3.2: *High level architecture of DEx.*

grammar. Because these build systems usually support a variety of programming languages, build scripts are very reliable and robust source to retrieve dependency information beyond the boundary of a specific programming language or platform.

3.3.1 Retrieving dependencies from Autoconf

The GNU build system that comprises of various tools including Automake, Autoconf, and Libtool is a mainstream build system in open source community. The general process to use GNU build system is: *(1) writing Makefile.am; (2) preparing configure.in/configure.ac; (3) Automake scans configure.in/configure.ac and uses the Makefile.cm to generate Makefile.in; 4) Alcohol may also be used to make local copies of all Autoconf macros; (5) using Autoconf to generate configure.* Among the com-

```
...
m4_define(pygtk_required_major_version, 2)
m4_define(pygtk_required_minor_version, 10)
m4_define(pygtk_required_micro_version, 3)
m4_define(pygtk_required_version, pygtk_required_major_version
    .pygtk_required_minor_version
    .pygtk_required_micro_version)
m4_define(libgnomevfs_required_version, 2.14.0)
...
PKG_CHECK_MODULES(PYGTK, pygtk-2.0 >=
    pygtk_required_version)
AC_SUBST(PYGTK_CFLAGS)
AC_PATH_PROG(PYGTK_CODEGEN, pygtk-codegen-2.0, no)
...
PKG_CHECK_MODULES(GNOMEVFS, [gnome-vfs-2.0 >=
    libgnomevfs_required_version],
build_gnomevfs=true,
build_gnomevfs=false)
...
```

Figure 3.3: *An Autoconf build script excerpt shows the usage of dependency checking macros.*

prising tools, Autoconf is particularly useful for validating dependency requirements. Since Autoconf and Automake are almost always used together, the dependency relationship of a project that uses GNU build system can be resolved by analysing and extracting dependency information embedded in build scripts of Autoconf.

With Autoconf, dependencies can be tested by calling various Autoconf macros from within configure.ac/configure.in. Therefore, the primary source to extract dependencies from Autoconf build scripts is the Autoconf library and program testing macros found in configure.in/configure.ac. Figure 3.3 is an excerpt of an Autoconf configure.ac of project gnome-python 2.22.0 found in Debian 5.0.3.

Autoconf provides various M4 macros for different purposes. Among these practical macros, some are specifically designed or have the ability to validate interdependency, i.e. checking the existence of a required external software package/program on the target system. Figure 3.3 demonstrates the usage of macro PKG_CHECK_ MODULES and AC_PATH_PROG. These macros make sure library pygtk-2.0 (version no less than 2.10.3), pygtk-codegen-2.0, and gnome-vfs-2.0 (version no less than 2.14.0)

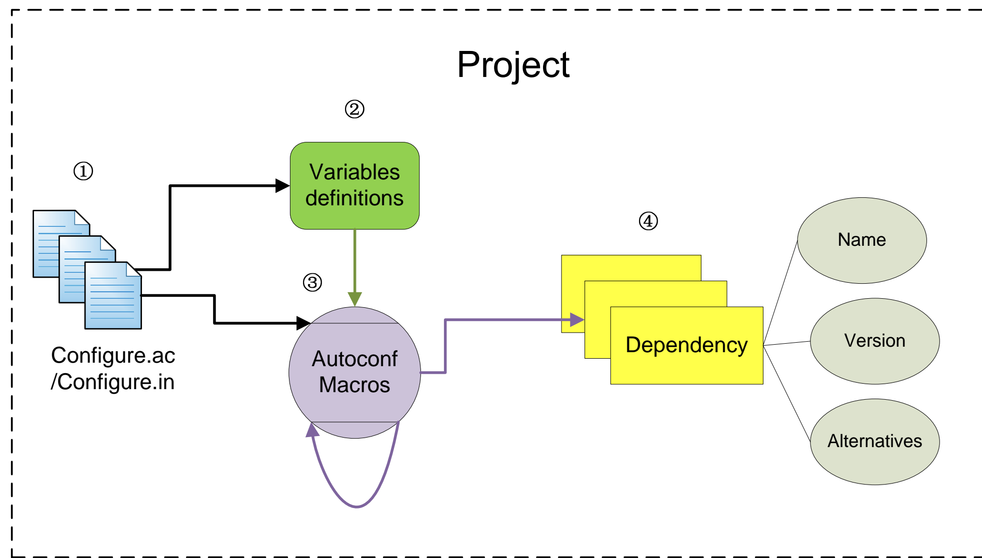

Figure 3.4: *General process to retrieve dependency from Autoconf build scripts*

are in the target system before the project gnome-python 2.22.0 can be successfully built.

The general process

The general process to extract dependency from Autoconf build scripts of a project is illustrated in figure 3.4. The process can be broken down into four steps:

1. **Collecting configure scripts**: each project to be examined may have more than one configure.ac/configure.in scripts located under the root directory and/or its sub-directories. It is essential to locate every configure scripts to make sure all of the possible dependencies are identified.

2. **Identifying variable definitions**: parameters of the macros (such as the version of the required library or the name of the required library) are sometimes declared as individual variables. For instance, the script in figure 3.3 shows that minimum version of the required library gnome-vfs-2.0 is defined in m4_define(libgnomevfs_required_version, 2.14.0) rather than PKG_CHECK_MODULES(GNOMEVFS, [gnome-vfs-2.0 >= libgnomevfs_required_version]. For each build script, all the variable definitions should be collected as a pair of variable name and value (preferably stored in a hash as key and value, respectively).

Each build script file has its own namespace. The value of variables should replace the variable name during dependency analysis.

3. **Processing macros**: the core part of this approach is the ability to recognize and parse various Autoconf dependency testing macros. By parsing the use of certain macros such as AC_CHECK_LIB, we could extract the dependency information (including the package name, version, type, importance, etc.). It should be noted that the use of one macro could span over one line. It is also possible that one macro could be embedded in another. Figure 3.5 is an excerpt of an Autoconf configure.ac of project gnubiff 2.2.10 found in Debian 5.0.3. As can be seen, shell scripts and other macros are used within PKG_CHECK_MODULES, which spans 11 lines. Since such case is not uncommon, it is important that the proposed approach has the ability to handle multi-line macros and process embedded macros when necessary. To achieve this goal, a recursive approach that would concatenate all lines and recursively process all of the embedded macros is applied.

4. **Extracting the dependency**: for any project, after the macros in all of its configure.ac/configure.in scripts are processed, a collection of its dependent software packages/programs could be obtained. Not only the name of a dependency, depending on the type of the processed macros, other information such as version, type of the dependency (required, optional), alternatives (any other packages/programs with the same functionality) may also be obtained.

Processing Autoconf macros

The effectiveness of the Autoconf approach hinges on whether Autoconf macros can be recognized and parsed. Table 3.1 lists the Autoconf macros that are typically used to test dependent software packages or programs.

As listed in table 3.1, Autoconf testing macros may be categorized into the package testing macros and generic program testing macros. Package testing macros focus on testing static dependencies at library or package level; program testing macros check the presence of generic programs in the user's PATH (unless another search path is explicitly specified). Our approach analyzes Autoconf macros as described below within configure.ac/configure.in to retrieve dependencies.

```
...
PKG_CHECK_MODULES(GNOME_PANEL_DEP,libpanelapplet-2.0,
[
    AC_SUBST(GNOME_PANEL_DEP_CFLAGS)
    AC_SUBST(GNOME_PANEL_DEP_LIBS)
    AH_TEMPLATE([USE_GNOME])
    AC_DEFINE(USE_GNOME)
    AC_PATH_PROG(GCONFTOOL, gconftool-2, no)
    if test "x$GCONFTOOL" = "xno"; then
        AC_MSG_ERROR([gconftool-2 executable not found in your
        path - should be installed with GConf])
    fi
    ], OPT_USEGNOME="no")
...
```

Figure 3.5: *An Autoconf macro spans over one line.*

AC_CHECK_LIB: AC_CHECK_LIB is used to check the existence of certain C/ C++/ Fortran library files by calling a function (declared by *function*). As shown below, *library* defines the library (dependency) to be tested; *other-libraries* lists alternative libraries in case *library* is not found. While parsing AC_CHECK_LIB, *library* becomes a required dependency, while *other-libraries* are abstract dependencies that are alternatives of *library*.

AC_CHECK_LIB (*library, function, [action-if-found], [action-if-not-found], [other-libraries]*)

AC_SEARCH_LIBS: AC_SEARCH_LIBS validates a dependent library by checking the availability of a specific function declared by *function*. Autoconf will try to search the function with no libraries at first. If the function is not found, libraries declared by *search-libs* would be tested one by one until the function can be successfully linked. Libraries declared by *other-libraries* will be called if they can help resolve the symbols that are not resolved by *search-libs*. As far as dependency analysis concerns, the libraries declared by *search-libs* are abstract dependencies. More precisely, the first library found in *search-libs* is considered as a required dependency,

Table 3.1: Autoconf macros for dependency validation

Package testing macros	**Generic program testing macros**
AC_CHECK_LIB	AC_CHECK_PROG
AC_SEARCH_LIBS	AC_CHECK_PROGS
PKG_CHECK_MODULES	AC_PATH_PROG
AX_CHECK_XXX	AC_PATH_PROGS
	AC_PATH_PROGS_FEATURE_CHECK
	AC_CHECK_TARGET_TOOL
	AC_CHECK_TARGET_TOOLS
	AC_CHECK_TOOL
	AC_CHECK_TOOLS
	AC_PATH_TARGET_TOOL
	AC_PATH_TOOL

the rest (if there is any) would be regarded as its alternatives. Libraries declared by *other-libraries* are not parsed for two reasons: 1) these libraries only help solve symbols, thus are not necessary to be considered as a dependency; 2) usage of parameter *other-libraries* is extremely rare in the real world (among all occurrences of AC_SEARCH_LIBS found in configure.ac/configure.in of projects in Debian main archive, less than 0.15% make use of parameter *search-libs*).

AC_SEARCH_LIBS (*function, search-libs, [action-if-found], [action-if-not-found], [other-libraries]*)

PKG_CHECK_MODULES: PKG_CHECK_MODULES is a very powerful Autoconf macro to validate dependencies. PKG_CHECK_MODULES is provided by Autoconf in order to utilize Pkg-config, which is an open source project that is adopted by most modern libraries to insert include and linker information during compilation. Pkg-config is platform independent and programming languge agnostic. Although the intention of Pkg-config is not for dependency validation, the libraries uncovered during the use of PKG_CHECK_MODULES do reveal the software libraries that are required by the host project. The syntax of PKG_CHECK_MODULES is shown below. *libraries* contains the name and version of the required libraries separated by blank space. *Prefix* names this validation. All the libraries listed in parameter *libraries* would be considered as required dependencies.

PKG_CHECK_MODULES (*prefix, libraries*)

AX_CHECK_XXX: Sometimes the scripts to test a certain library may be too complicated for a standard Autoconf macro. In such case, developers may opt to write an individual script file to handle a specific situation. AX_CHECK_XXX is a useful Autoconf macro that is used to test a dependent library whose test logic is written in an individual file. XXX stands for the name of the dependency that is to be tested. Corresponds to such dependency, an ax_check_XXX.m4 file, which implements the code to test the dependency, is provided. For example, when AX_CHECK_GL is found in configure.ac/configure.in, a file ax_check_gl.m4 should be found in the same project directory (probably under different sub directory). Autoconf expands each AX_CHECK_XXX in the configure file with the content defined in its corresponding m4 file. The naming of XXX could be anything, but it is a de facto standard to name it after the name of the testing dependency. We trust the developers' professional behavior, and treats XXX as the name of the required dependency.

Single program testing macros: AC_CHECK_PROG, AC_CHECK_TARGET_TOOL, AC_CHECK_TOOL, AC_PATH_PROG, AC_PATH_TARGET_TOOL, and AC_PATH_TOOL are single program testing macros. These macros share the same general purpose - testing whether the program specified by *prog-to-check-for* exists in *path*. Their behaviours, however, may be different. For example, if *prog-to-check-for* is found, AC_PATH_PROG assigns the the absolute name of *prog-to-check-for* to *variable*, whereas AC_CHECK_PROG sets *variable* the value stored in *value-if-found*. The syntax of single program testing macros is listed below. Please refer to [12] for a complete explanation of each macro. With our approach, when single program testing macros are parsed, *prog-to-check-for* becomes a required dependency.

- AC_CHECK_PROG (*variable, prog-to-check-for, value-if-found, [value-if-not-found], [path = '$PATH'], [reject]*)
- AC_CHECK_TARGET_TOOL (*variable, prog-to-check-for, [value-if-not-found], [path = '$PATH']*)
- AC_CHECK_TOOL (*variable, prog-to-check-for, [value-if-not-found], [path = '$PATH']*)
- AC_PATH_PROG (*variable, prog-to-check-for, [value-if-not-found], [path = '$PATH']*)
- AC_PATH_TARGET_TOOL (*variable, prog-to-check-for, [value-if-not-found], [path = '$PATH']*)
- AC_PATH_TOOL (*variable, prog-to-check-for, [value-if-not-found], [path = '$PATH']*)

Multiple programs testing macros: AC_CHECK_PROGS, AC_CHECK_ TARGET_TOOLS, AC_CHECK_TOOLS, AC_PATH_PROGS, AC_PATH_PROGS_ FEATURE_CHECK are multiple programs testing macros. Like single program testing macros, this category of macros also test whether a specific program exists in *path*. The major difference is that multiple programs, listed in *progs-to-check-for* and separated by an empty space, are abstract dependencies. That means they are replaceable with each other, and anyone would satisfy the dependency. When these macros are parsed, the list of programs would be retrieved from *progs-to-check-for*, among which, the first program becomes a required dependency, while the rest (if there is any) become the alternatives of the required dependency. The syntax of multiple programs testing macros is listed below. The usage of each macro is different, please refer to [12] for a detail explanation.

- AC_CHECK_PROGS (*variable, progs-to-check-for, [value-if-not-found], [path = '$PATH']*)
- AC_CHECK_TARGET_TOOLS (*variable, progs-to-check-for, [value-if-not-found], [path = '$PATH']*)
- AC_CHECK_TOOLS (*variable, progs-to-check-for, [value-if-not-found], [path = '$PATH']*)
- AC_PATH_PROGS (*variable, progs-to-check-for, [value-if-not-found], [path = '$PATH']*)
- AC_PATH_PROGS_FEATURE_CHECK (*variable, progs-to-check-for, feature-test, [action-if-not-found], [path = '$PATH']*)

3.3.2 Retrieving dependencies from CMake

CMake is a very powerful and popular build system. Many famous `FOSS` projects such as OpenGC[1] adopt CMake as their build system. In fact, CMake has been replacing make and GNU Autoconf/Automake in recent years. Our proposed method supports retrieving dependency information from CMake build scripts. Like configure.ac/configure.in of Autoconf, CMake has its own build script named CMakeLists.txt. By parsing the library and program testing macros found in CMakeLists.txt, a list of dependencies required by the host project could be obtained. Figure 3.6 is an excerpt of a CMakeLists.txt of the project kdelibs 4.1.0 found in Debian 5.0.3. The excerpt shows the use of macro FIND_PACKAGE and CHECK_LIBRARY_EXISTS and indicates that KDE4Internal, Carbon, ZLib, Strigi, NSL, and Socket are software libraries that kdelibs 4.1.0 may depend on.

The general process

As illustrated in figure 3.7, the process to extract dependency from CMake build scripts for a project is very similar to Autoconf. One project may have multiple CMakeLists.txt files located in its root and sub-directories to build the entire project.

[1] OpenGC, http://opengc.sourceforge.net/

```
...
find_package(KDE4Internal REQUIRED)
include(KDE4Defaults)
include (MacroLibrary)
if (APPLE)
find_package(Carbon REQUIRED)
endif (APPLE)
...
find_package(ZLIB REQUIRED)
find_package(Strigi REQUIRED)
...
check_library_exists(nsl gethostbyname "" HAVE_NSL_LIBRARY)
check_library_exists(socket connect "" HAVE_SOCKET_LIBRARY)
...
```

Figure 3.6: *A CMakeLists.txt excerpt shows the usage of dependency checking macros.*

Each CMakeLists.txt will be parsed in order to extract dependency information from its library testing macros. The superset that contains the dependencies retrieved from every CMakeLists.txt is the result set.

Processing CMake macros

Among hundreds of CMake macros, the macro FIND_PACKAGE, CHECK_LIBRARY_EXISTS, and PKG_CHECK_MODULES are most frequently used for testing the existence of external libraries. According to our study that counts the usage of library testing macros in CMakeLists.txt of 99 Debian projects that use CMake as their build system, around 70% of dependency checkings are performed via the PKG_CHECK_MODULES macro, and around 15% are performed via the FIND_PACKAGE macro. By parsing these macros, we are able to extract dependency information from CMakeLists.txt.

In order to make use of CMake library and program testing macros, it is reasonable to understand how CMake manages to test external libraries. A host project is a project being built. The use of any external software package creates an inter-dependency relationship between the host project and the external package. To utilize the external package, the host project must be able to locate the package, namely knowing the file path of libraries and headers of the package. CMake facilitates such

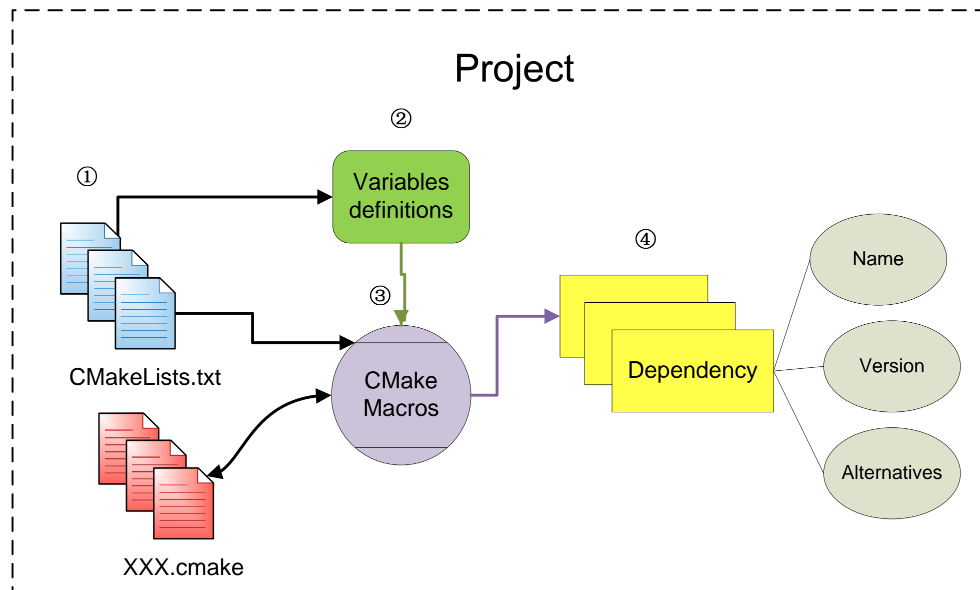

Figure 3.7: *General process of dependency retrieving from build scripts of project using CMake.*

process by presenting the concept of modules, each of which has sole responsibility to find one external library.

Each module should be provided in the form of a Find<package _name>.cmake file, where package _name stands for the name of library to be checked. For example, FindOpenSSL.cmake is responsible for locating the OpenSSL encryption library. CMake macros such as FIND_LIBRARY(), FIND_PATH(), FIND_PROGRAM(), and FIND_FILE() may be used in Find<package_name>.cmake to perform the actual work to seek and locate the external library. By default, CMake searches for Find <package>.cmake in the file path defined in the CMake variable CMAKE_MODULE _PATH (usually /usr/local/share/CMake/Modules/ on Unix like systems). If the package is found, CMake module will define the following return variables that can be accessed in CMakeLists.txt, including:

- XXX_FOUND: specifies whether library XXX is found in system
- XXX_INCLUDE_DIR/XXX_INCLUDES: specifies the file path of the directory that library XXX is to be found
- XXX_LIBRARY/XXX_LIBRARIES: specifies the libraries that are required to make use of library XXX

Any external library to be checked should have a corresponding Find<package _name>.cmake and such file should be shipped with the host project. Certain libraries (such as OpenGL or Qt) are particularly popular. Since so many projects depend on these libraries, standardizing Find<package _namegreater.cmake for these libraries would not only improve the quality of build scripts but also save the effort and time of CMake build script developers. As such, CMake presents a collection of standard CMake modules[2]. CMake standard modules include utility modules, test modules, check modules, and some modules that are specifically designed to find external libraries. To use a CMake standard module, a user does not need to provide a Find<package _name>.cmake as such file is shipped with CMake. For instance, there is no need to write your own FindOpenGL.cmake to find the OpenGL library in system.

FIND_PACKAGE: FIND_PACKAGE is a frequently used CMake macro to check external libraries by making use of CMake modules. FIND_PACKAGE has two modes, the module mode and the config mode. By default, FIND_PACKAGE will operate in module mode, trying to find and load external package *package_name* by calling the corresponding Find<package _name>.cmake module. By processing the Find<package_name>.cmake file, FIND_PACKAGE can perform the tasks such as locating the package, generating messages, and validating the version of the package. If Find<package_name>.cmake is not found, FIND_PACKAGE will then operate in the so called config mode. In the config mode, CMake attempts to find a file called <package_name>Config.cmake (file name is case insensitive). <package_name>Config.cmake should reside in the file path specified by the CMake variable <package_name>_CONFIG defined in CMakeLists.txt. Package version may be checked by processing the corresponding version file <package_name>Version.cmake located in the same directory of <package_name>Config.cmake.

The syntax of FIND_PACKAGE is described below. Parameter *package-name* indicates the name of the package to find. Parameter *version* in a format of *major[.minor[.patch[.tweak]]]* specifies the version that the found package must be compatible with. Leaving the version information empty usually indicates there is no version constraint and therefore any version should be acceptable. Parameter *EXACT* requires an exact match of version. However, if the macro is invocated inside another find module, the version and EXACT arguments from the outer level will be

[2]CMake standard modules, http://www.cmake.org/cmake/help/cmake-2-8-docs.html

propagated to the inner level. Parameter *REQUIRED* indicates the package to be found is a required dependency, in which case CMake will simply cease processing until the required package is available. If *REQUIRED* is not specified, the package to be checked is an optional dependency. A list of package specific components may be listed after the *COMPONENTS* parameter. Our approach uses the information extracted from the *package-name*, *version*, and *REQUIRED* to establish a dependency relationship for the target project.

```
FIND_PACKAGE(<package-name>[VERSION] [EXACT]
[QUIET] [REQUIRED—COMPONENTS] [COMPONENTS...]]
[NO_POLICY_SCOPE])
```

CHECK_LIBRARY_EXISTS: CHECK_LIBRARY_EXISTS checks if the function specified by parameter *FUNCTION* is available in *LIBRARY*, which can be found under *LOCATION*. Parameter *VARIABLE* sets the variable that stores the result. The value of *LIBRARY* indicates an inter-dependency of the host project.

```
CHECK_LIBRARY_EXISTS (LIBRARY FUNCTION LOCATION
VARIABLE)
```

PKG_CHECK_MODULES and PKG_SEARCH_MODULES: As mentioned earlier, Pkg-config is an open source project that enables libraries to insert include and linker information during compilation. Pkg-config works beyond the scope of a specific platform or programming language. Similar to PKG_CHECK_MODULES of Autoconf, CMake macros PKG_CHECK_MODULES and PKG_SEARCH_MODULES call the CMake module findPkgConfig, which utilizes Pkg-config to find external packages in CMake the environment.

- PKG_CHECK_MODULES(<PREFIX>[REQUIRED] [QUIET] <MODULE>[<MODULE>]*)
- PKG_SEARCH_MODULE(<PREFIX>[REQUIRED] [QUIET] <MODULE>[<MODULE>]*)

The syntax of CMake version of PKG_CHECK_MODULES is similar to the Autoconf version. When using PKG_CHECK_MODULES, parameter *PREFIX* simply names the check. The dependent libraries, separated by blank space, are declared in the *MODULE* parameter. Not only library name but its version could be specified as well. For instance, PKG_CHECK_MODULE(XML libxml2 libxml>=2) will check the existence of libxml2 and libxml while making sure the version of libxml is no less than 2. Like FIND_PACKAGE, *REQUIRED* indicates a 'required' dependency. Libraries without a REQUIRED tag are optional libraries. Parameter *QUIET* switches the command to silent mode where no message will be prompted.

PKG_SEARCH_MODULES is very similar to PKG_CHECK_MODULES. Unlike PKG_CHECK_MODULES that checks all the libraries specified in *MODULE* parameter, PKG_SEARCH_MODULES stops checking once a working library is found. For single library checking, PKG_SEARCH_MODULES and PKG_CHECK_MODULES are the same.

The information extracted from the *MODULE* and *REQUIRED* parameters is used to establish the dependency relationship between the host project and external libraries. More precisely, the libraries specified by *MODULE* of PKG_CHECK _MODULES are retrieved dependencies. In the case of PKG_SEARCH_MODULES, among the libraries specified by *MODULE*, the first one becomes the retrieved dependency, while the rest are considered as its alternatives.

3.4 Dependency resolution: a source code approach

The most distinguishable characteristic of FOSS is the availability of its source code. Source code is the ultimate source to explore different levels of dependency (e.g., method level dependency, class level dependency) and different types of dependency (e.g. inner-dependency and inter-dependency). Depending on the programming lan-

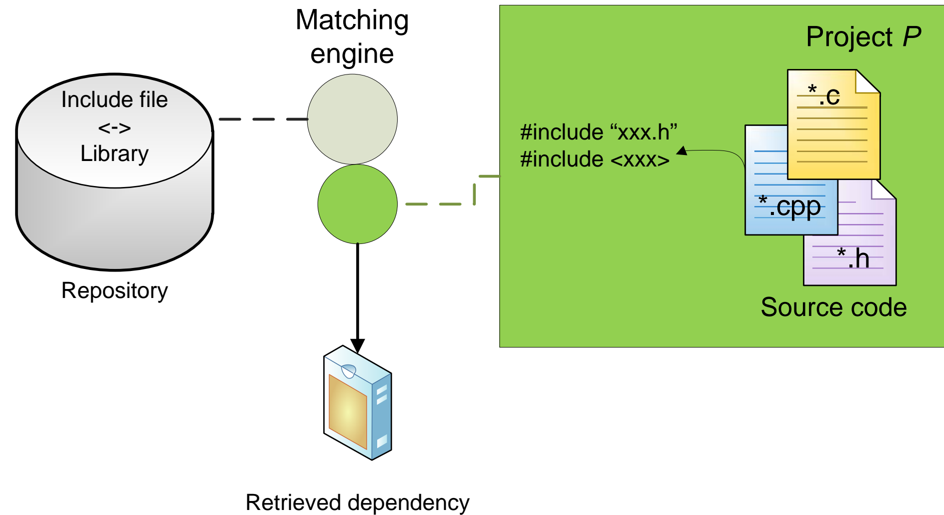

Figure 3.8: *General process of dependency retrieving from source code of a project.*

guage, the technique to parse and process the source code varies. Chapter 2 shows that numerous research has been done on extracting software dependency by parsing source code. Most research focuses on investigating inner-dependency. By analyzing the use of global variable and function calls, a call graph, which may reveal the encapsulation of code and architecture of a project, could be constructed.

It has been found that as much as 80% of Debian projects are developed with the C or C++ programming languages, and around 3% projects use Java[18]. Research ([39], [48]) has been conducted to explore inter-dependencies of projects written in Java. Nevertheless, no solid research about inter-dependency resolution with C and C++ has been done. The main reason could be (1) C is not an object oriented programming language; (2) C and C++ use the include directive (e.g., #include<stdout.h>) to indicate a file dependency. The include directive is ambiguous in locating files and thus not as effective as Java, which uses import directive with a fully qualified name (e.g., ca.uvic.hanyuz.Example). Our study aims to fill the gap of inter-dependency research with a focus on the C/C++ programming language. Although the research is restricted in the area of Debian, we believe its outcome is still useful in a wider spectrum of FOSS.

3.4.1 General process

Unlike inner-dependency extractor that works by processing project source code, inter-dependency extractor relies upon matching include files with a candidate repository. The general process of dependency resolution with source code is illustrated in figure 3.8. The process can be broken down into three components:

1. Identify include files by parsing source code. An include file refers to a dependent artifact that is indicated in the project's source code, namely a header file indicated by an include directive in C/C++ source code. Similar concept in other programming languages are a script/library indicated by the require or use keyword in Perl, a file or a package indicated by the import keyword in Java.

2. Prepare the candidate artifact repository. An artifact is any identifiable include files. For instance, *stdout* is an artifact if #include <stdout> is found when parsing a C/C++ file. Simply knowing an include files is not enough for practical use. It is inappropriate to report to the user that the project being examined depends on xyz.h. Instead, the real dependent library, which xyz.h belongs to, should be reported. In order to locate the external dependent libraries, we create a candidate artifact repository for each programming language. The artifact-library relationship is stored in the candidate repository. Each relationship records an artifact and its belonging library. For example, stdout.h belongs to C/C++ standard library. The repository should be large enough to include as much artifact-library relationship as possible. A high quality candidate repository associated with a given eco-system is the main source of information for extracting information about inter-dependencies [31].

3. Develop a mechanism that retrieves external dependent libraries by searching the candidate repository with the identified include file on the basis of artifact-library relationship. Given a satisfactory matching rate with high precision, a list of inter-dependencies can be retrieved on the basis of source code.

Depending on the programming language, artifact identification method and candidate repository may vary. The general philosophy of source code dependency resolution technique, however, should follow the process described above. For instance,

Ossher in [39] introduces a Java inter-dependency extraction method with the similar concept. In the following sections, our methodology of C/C++ inter-dependency resolution within the Debian environment is discussed in detail.

3.4.2 Identify include files in C/C++

Unlike a build script or a spec within which the name and possibly the version of a dependent library would be clearly stated, dependency retrieved with the source code approach is at file level. To promote code reuse, reusable artifacts such as global variables or functions are usually defined in a header file that is accessible by other source files. A dependency between a source file X and a header file Y is established when file X makes use of a code artifact defined in file Y. Such code artifact could be a variable, a constant, a data type, a macro, a function, or even a class. When the header file and the source file belong to the same project (even if they are placed in different directories), an inner-dependency is established. If the header file is a part of a third party library that is not distributed with the source file, an inter-dependency is established. The content of a C/C++ header file (*.h) is usually declarations or definitions of forwardable artifacts, while a source file (*.c, *.cpp, *.cc) can contain anything as long as it is recognizable by the compiler. Therefore, interdependency relationship can be established between a header file and a source file, or even between two header files.

Under no circumstance should an artifact be used before it is defined. If a source file needs to use an artifact defined in a header file, #include directive must be used in the source file so that the preprocessor could load the contents of the header file and treat the contents as if they are defined at the point where the #include directive appears. A header-header or header-source dependency is the main form of source code reuse in C/C++. For example, if project A depends on library L, we should be able to find a header file of L in an #include directive declared in A's header or source file. By examining contents of #include directives of project A, we can retrieve a list of header files that project A depends on. It should be noted that there is no restriction on the type of file to be included by the #include directive, however, developers usually include header files only. For this reason, in our research, we assume all the included files are header (*.h) files.

The syntax of #include directive has two forms: the quoted form (e.g., #include "filePath") and the angle-bracket form (e.g., #include <filePath>). filePath is the

file name of the included header file optionally proceeded by the file path (either relative or absolute) of directory where the header file could be found. In the case that filePath is an absolute and unambiguous path of the included file, the preprocessor would only look for the included file in the specified path regardless of the syntax form. For instance, #include </usr/ucbinclude/sys/signal.h> found in last.c of project ACCT 6.3.5 would instruct the preprocessor to look for signal.h only under the directory /usr/ucbinclude/sys/. Similarly, #include "/usr/include/sys/types.h" found in shmsys.c of project ACM 4.7 would instructor the preprocessor to locate types.h only under the directory /usr/include/sys/. The behavior of the two syntax forms may be different when filePath is only a file name or an relative file path.

- **The angle-bracket form**: it is a convention to use this form to include system headers. As such, this form usually implies an inter-dependency relationship between the project and a system library. This form instructs the preprocessor to look up included file under the directory specified by the -I option when compiling or the INCLUDE environment variable. In Debian, system headers are always placed under the directory /usr/include/. For instance, #include <stdio.h> found in reverbst.cpp of the Debian project alsaplayer 0.99.26 would instruct the preprocessor to look for stdio.h under /usr/include/, #include <sys/ioctl.h> would instruct the preprocessor to search for ioctl.h under /usr/include/sys/.

- **The quoted form**: this form is usually used to include application specific headers by convention. More specifically, this form instructs the preprocessor to search for the included file along the same path where the file (say X) that contains the #include directive locates. If fails, the preprocessor would then look up directories within which any file is found to include file X. If the include file is still not found, the quoted form would behave in the same way of the angle-bracket form. For example, #include "tcl.h" found in tclMacEnv.c of project dejagnu 1.3 would instruct the preprocessor to look for tcl.h under projectLocation/tcl/mac/ where tclMacEnv.c locates. As can be seen, this form may indicate both inner-dependency between header and source files of the same project or an inter-dependency.

Simply put, dependent header files can be retrieved by parsing #include directives of C/C++ header and source files. Figure 3.9 illustrates the general process to parse C/C++ source code to retrieve dependent header files.

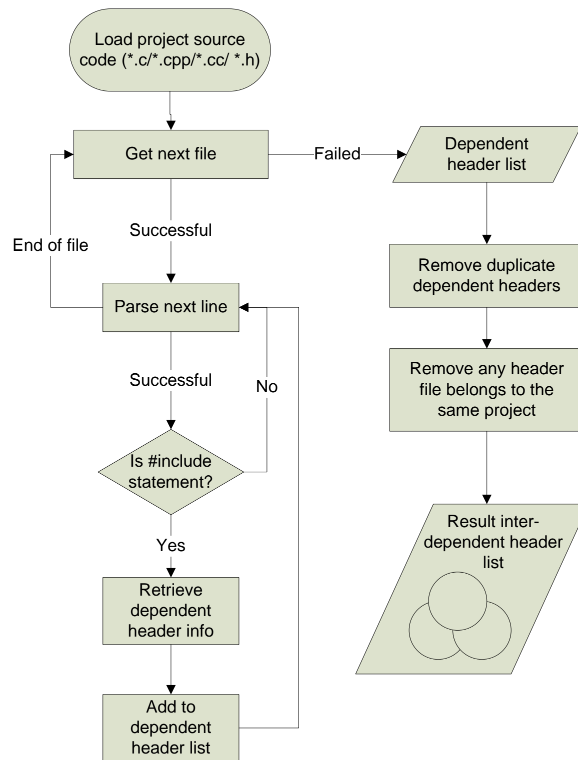

Figure 3.9: *General process to identify include files in C/C++.*

1. For any project, to retrieve its external dependent headers, all of its *.h and *.c/*.cpp/*.cc files are loaded. That includes not only the files under the project's root directory, but files under any sub-directories as well.

2. Loaded files are placed in a queue and parsed in order. Each file is parsed line by line, from the beginning to the end of file. Once an #include statement is encountered, the information of the included header will be retrieved. The information includes the name of the header and its file path (if there is any) and the syntax form (angle-bracket or quoted) of the #include directive. It is noticeable that it is a convention to put all the #include statements in the beginning of the file; however, there is no rule that forbids people to put the #include statement anywhere else in the file as long as the file can be compiled. To achieve an unbiased result, we choose to scan the entire file, even though it is more time consuming.

3. After every #include statement of each file is processed, a complete list of header files should be obtained. The list is processed to remove any item belongs to the

same project so that inner-dependency can be neglected. Given that a header file could be referenced multiple times by different files in the same project, a procedure that eliminates duplicate entities is also applied. Two headers are considered identical only if they have the same file name. After the eliminating procedure, the final list of inter-dependent headers of the target project should be obtained. This list should only contain distinct header files that are not part of the project but depended by the project. The list is ready to be cross referenced with the candidate repository.

It should be noted that the general process for Java, Perl or even other programming languages is the same. In case of Perl, requireuse statement is processed while in Java, import statements are processed to retrieve the dependent files.

3.4.3 Create the candidate artifact repository

Identification of include files results in a list of header files that are depended by the target project. But it is more meaningful to learn the libraries where those header files are defined. After all, it is meaningless to show the end-user a bunch of *.h files, which cannot be independently installed, or compiled, or executed, or even distributed. Candidate artifact repository functions as the bridge between header files and their belonging libraries. By looking up the repository with the file path of a header, the information of its belonging project can be retrieved.

The quality of the candidate artifact repository determines the effectiveness of the source code dependency extractor. Therefore, it is vital that the repository is as comprehensive as possible to ensure most open source projects are included. Given the size of open source community, it is simply impossible to collect all of the open source projects. Our research aims to solve dependency problems in the Debian environment, it becomes reasonable and necessary to ensure the effectiveness of our method as long as our repository collects the information of most libraries or packages that a Debian project would depend on. Although there are numerous open source code repositories available (e.g., Apache Maven, Google code search, Koders, Merobase, etc.), utilizing them may have some restrictions or difficulties.

- They are not designed for the Debian environment. Hence, they may miss some projects that are frequently used by Debian projects.

- Precesion is compromised. There is no doubt that the volume of these reposito-

ries could be extremely large and comprehensive, covering a significant amount of open source projects written in different programming languages and run on different platforms. However, a large collection could easily leads to a higher risk of name collision. Unlike Java that uses a Fully Qualified Name (FQN) to indicate dependency, C/C++ simply relies on the file name and optionally the absolute or even relative path of a header file to indicate dependency. With a giant repository of no specific platform preference, it is not unusual that more than one projects may have a header file with the same name. For example, a hit of Google Code Search with any project that contains the header tcl.h would return 2053 results. It is obviously ridiculous to present the end user 2053 projects that the target project potentially depends on (even though the genuine dependency is almost certain to be within the presented projects).

- Many of such public code repositories do not have free and open API that can be programmably used by client software for automatic analysis.

As a result, we believe a repository designed specifically to the Debian platform would lead to higher precision and success rate for our research.

Source of the repository

As mentioned earlier, a dependent header could belong to a system library or a regular project. Debian is such a comprehensive and mature environment. Inter-dependencies of a Debian project are usually either Debian system libraries or projects that already included in Debian distributions. Therefore, a repository that contains 'header - belonging library' relationship of all the Debian system libraries and Debian projects should be fair enough to provide dependency resolution for the very most of Debian projects. Also because the repository itself originates from Debian, the precision of the matching result should be acceptable.

- **Finding 'header - belonging library' relationship of system headers**: System header files such as stdio.h are heavily depended by a large amount of Debian projects. In Debian, system headers files are placed under directory */usr/include*. These header files belong to core system libraries such as libc6-dev, linux-libc-dev, x11-apps, etc. With a full installation of Debian 6.0.0, 1233 system header files were found. In order to find out which library a system header file belongs to, we utilize Debian's dpkg utility, a lower level application

of Debian package management system. Being a powerful tool, dpkg is equipped with different options to fulfill various tasks such as install, remove, or update a package. We utilize -S option of dpkg to find the belonging package of any qualified file. The syntax of the command is *dpkg -S <pattern>* where pattern could be the absolute path of a header file. For example, the command *dpkg -S /usr/include/video/sisfb.h* returns *linux-libc-dev: /usr/include/video/sisfb.h*, which indicates the header file sisfb.h found under directory /usr/include/video/ belongs to system library linux-libc-dev. By running dpkg against all of Debian system header files, a candidate artifact repository of system headers is created.

- **Finding 'header - belonging library' relationship of Debian packages**: Unlike system headers that are usually placed under */usr/include*, Debian applications can be installed anywhere. In order to establish the list of 'header - belonging library' relationship for all of Debian packages, we scan the Debian source code, and create the relationship for every header file that is found. For example, bn16.h and bnprint.h are found in the source code archive of project bnlib 1.1. This infers that any project that have #include "bn16.h" or #include "bnprint.h" found in their source code could depend on bnlib 1.1. The source code of Debian projects, distributed as many as five DVD, is available at Debian's official website[3] and its mirror sites. The projects are categorized into three archive areas: main, contrib, and non-free. The main archive area contains those packages comply with DFSG (Debian Free Software Guidelines) and do not depend on packages outside of main. Packages in the contrib archive area still comply with DFSG, but may depend on proprietary packages. Packages in the non-free archive area do not comply with DFSG. Since the official Debian distribution only contains projects in the main archive area, we scan source code of projects found in the main archive area.

Creation of repository

The 'header - belonging library' relationship has to be stored in a database for cross reference. Figure 3.10 illustrates the generate process to establish the candidate artifact repository of the C/C++ programming language. For any Debian system header file and header file of every Debian package, its 'header - belonging library' relationship is recorded in a database table. The table is designed as described in

[3]Debian source, http://www.debian.org/CD/http-ftp/#stable

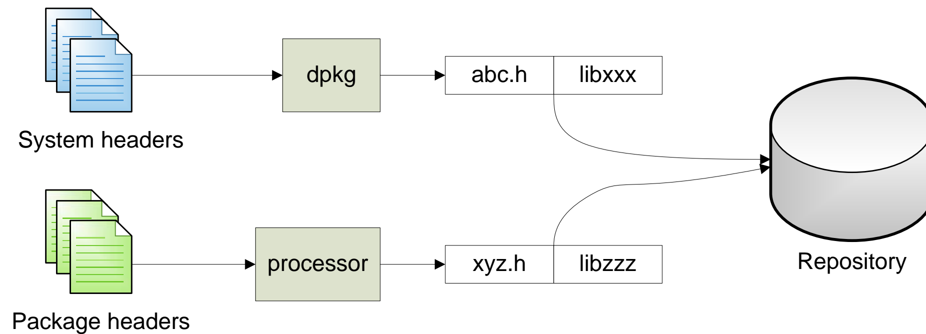

Figure 3.10: *General process to construct candidate artifact repository.*

table 3.2. Each header file contributes a record to the table. If a header file does not have a matching library, it will not be recorded in the database. Given that C and C++ are case sensitive languages, the data stored in the database is case sensitive, too.

3.4.4 Matching algorithm

To retrieve the list of libraries/packages that a certain project depends on, we need to match the identified include files against the artifact candidate repository. A matching algorithm is developed to accomplish this task with satisfactory precision. Figure 3.11 illustrates the workflow of the algorithm. Identification of include files results in a list of headers with their header-information, including header's name, file path of the header file (if applicable), and the way the header is declared (i.e. angle-bracket include or quoted include). The headers in the dependent headers list are processed one at a time until every header has been processed. Information of each header will be used to match against the artifact candidate repository. If the file path of the header is an absolute path, it will be used as the search key first. The absolute path will be matching with the path field of every row in the db_table_dep_rep table. If a match is not found, the file name of the header will be used to look up the artifact candidate repository, matching each record with its dep_identifer field. Depending on the type of the header, if it is declared by angle-bracket include directive, it will be matched against record whose origin field is 1 (i.e. system header); if it is declared by quoted include directive, it will be matched against record whose origin field is 2 (i.e. header of regular Debian project). If no match is found, the next header file in the list is processed. Upon a match is found, if one and only one match is found, the

Table 3.2: Database table: db_table_dep_rep

COLUMN	TYPE	Comment
dep_identifer (PK)	text	File name of the header file.
package_name (PK)	text	Name of its belonging package.
package_version (PK)	text	Version of its belonging package. Empty when not available.
package_path (PK)	text	Path of the header file: (1) absolute path when the file is system header. (2) file path relative to project root directory otherwise. For example, if cards.h is found under the lib directory of package ace 1.1, the value would be lib.
prog_lang (PK)	integer	Programming language of the header file. 1 means C/C++; 2 means Perl; 3 means Java.
origin (PK)	integer	Type of the header file: 1 means system header; 2 means a header file defined in a regular Debian project.
deb_ver (PK)	text	Version of Debian where the file is found. Current Debian version in the repository is 6.0.0.

package_name and package_version of the matching record are added to the result list. If more than one match is found, conflict must be resolved. For example, there are multiple matching projects (mesa 7.7.1, virtualbox-ose 3.2.10, seti, and vnc 4.1.1) of header 3dnow.h. Sometimes only one of them could be the genuine dependency, but it is also possible that several matching projects may offer the same functionality, and therefore form an abstract dependency. Source code approach features two modes:

- **Full mode**: In full mode, the source code approach reports any matching projects as dependency. In case more than one matching projects are found, they will be reported as one abstract dependency. For example, "about.h" matches 65 projects in the candidate artifact repository. So if #include "about.h" is found in the source code of project P, `DEx` may report an abstract dependency of 65 projects. The 65 matching projects are considered as one retrieved dependency, but exchangeable to each other.

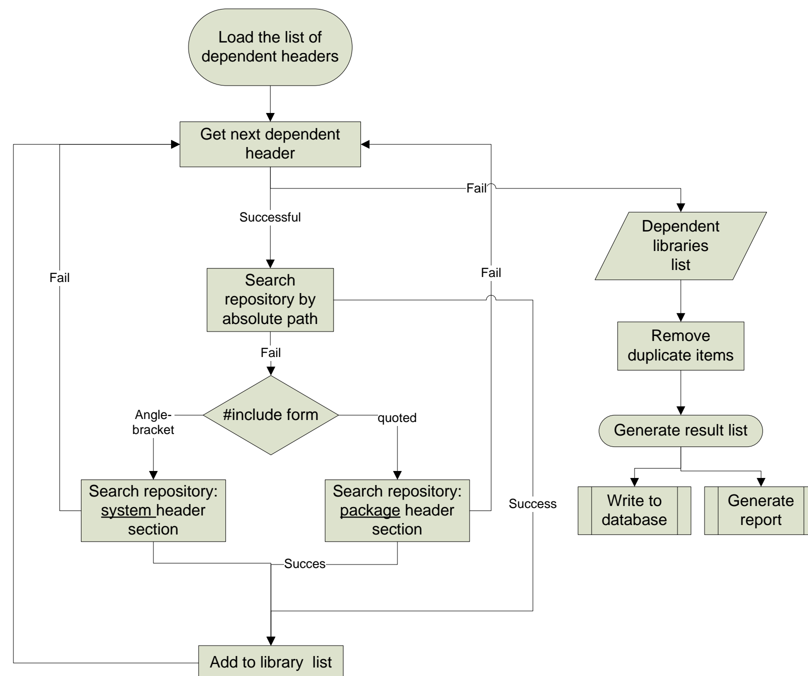

Figure 3.11: *Flow chart of include file - candidate artifact matching algorithm.*

- **Clear mode**: In clear mode, only inambiguous dependencies that have a one-to-one relationship between include file and matching project in the repostory are retrieved. For example, ”xorsa_opengl.h” has only one matched project orsa 0.7.0, therefore if #include ”xorsa_opengl.h” is found in the source code of project X, orsa 0.7.0 will become a retrieved dependency in clear mode. Sometimes, multiple projects matched to the same include file may be the same project but only with name variations or version differences. In such case, a dependency should still be reported. We apply a fairly strict algorithm to address this issue: we compare package_name of matching projects with each other with a name equivalence algorithm. We consider two package_names are identical when they are equal (case insensitive) after noisy characters (such as + - . beta dev lib digit) are removed. For example, libc6-dev and libc6 will be considered the same because they are eqal after the noise data dev is removed; if all matching projects have the same package_name, the conflict is resolved; otherwise, we perform a regular expression matching between file path of the header file and package_path of each matching project. For example, if header file 3dnow.h in project P is declared with #include ”mesa/x86”, we would

know project P depends on mesa 7.7.1 since only mesa 7.7.1 has a matching package_path ("src/mesa/x86") among the four matching projects (mesa 7.7.1, virtualbox-ose 3.2.10, seti, and vnc 4.1.1). If the conflict is resolved, the matching project is reported as a retrieved dependency; otherwise the header file will be marked unsolvable.

After all of the include files (i.e. dependent headers) have been processed, a duplicate check is conducted against the result dependent project list to remove duplicate items. In the end, a list of packages that the sampling project depends on should be retrieved.

3.5 Dependency resolution: a spec approach

As explained in chapter 2, Debian provides source code of each project as well as compiled packages for different platforms (e.g., i386, ia64, mips, etc.). Just like many software products have a read-me file, both source project and binary packages in Debian come with their spec. Debian source spec and package control file (package spec) are Debian documents from which inter-dependencies can be retrieved.

Our approach to retrieve inter-dependencies from Debian documents borrows some idea from the method explained in [17] and [18]. Dependencies retrieved from source spec are usually build time dependencies, namely dependent packages to compile and build the target project into binary package; dependencies retrieved from control file of a binary package are usually installation time dependencies or runtime dependencies. The spec approach features three modes: source spec mode, package spec mode, and full mode. In source spec mode, only dependencies retrieved from the source spec are reported; in package spec mode, only dependencies retrieved from the package spec are reported; in full mode, for any Debian binary package, a complete list of dependencies would include dependencies retrieved from its control file as well as dependencies retrieved from source spec of the source project that the package is compiled from. For instance, retrieved dependencies of Debian package *libacovea 5.1.1* include dependencies retrieved from its control file and source spec of its source project *acovea*. This example also reveals the fact that one source project can be compiled to numerous packages with different package names. In order to retrieve the most comprehensive list of dependencies of a Debian package, we should locate its source project and its source spec. Fortunately, in the source spec of a Debian source project, there is a field

Binary, which lists the name of binary packages compiled from the source project. For example, figure 3.13 shows that binary package of source project obconf 2.0.3 is also obconf 2.0.3. By analyzing source spec of Debian source projects, we are able to create a database table db_table_deb_src_bin_pkg_match, which stores the project-package mappings of Debian projects. The design of db_table_deb_src_bin_pkg_match is described in table 3.3.

Table 3.3: Database table: db_table_deb_src_bin_pkg_match

COLUMN	**TYPE**	**COMMENT**
pkg_name_bin (PK)	text	Name of a Debian binary package.
pkg_name_src	text	Name of its corresponding source project.
pkg_ver (PK)	text	Version of the Debian package.
deb_ver (PK)	text	Version of Debian where the file is found. Current Debian version in the repository is 6.0.0.

Retrieving dependencies from a control file

Each Debian binary package has a control file. A control file contains the control fields (such as package name, package version, description, dependencies, etc) for the package. Figure 3.12 shows an excerpt of the control file of obconf 2.0.3 binary package in Debian 5.0.6.

In a Debian package control file, the following fields are used to declare dependency relationship between packages[4]:

- **Depends**: lists required dependencies that must be already in the system or at least installed at the same time before the package can be correctly configured. The listed dependencies can be either explicit or abstract. In the case of abstract dependencies, multiple packages will be concatenated by the operator |.
- **Recommends**: lists strongly recommended (although not required) optional dependencies.

[4] Please refer to [22] for a complete description about declaring relationship between packages in Debian.

Package: obconf
Priority: optional
Section: x11
Installed-Size: 496
Maintainer: Openbox Maintainers <pkg-openbox@modprobe.de>
Architecture: i386
Version: 2.0.3-3
Depends: libatk1.0-0 (>= 1.20.0), libc6 (>= 2.7-1), libcairo2 (>= 1.2.4), libfontconfig1 (>= 2.4.0), libfreetype6 (>= 2.3.5), libglade2-0 (>= 1:2.6.1), libglib2.0-0 (>= 2.16.0), libgtk2.0-0 (>= 2.12.0), libice6 (>= 1:1.0.0), libobparser21, libobrender21, libpango1.0-0 (>= 1.20.2), libsm6, libstartup-notification0 (>= 0.8-1), libx11-6, libxft2 (>>2.1.1), libxml2 (>= 2.6.27), and zlib1g (>= 1:1.1.4)
Recommends: openbox
Filename: pool/main/o/obconf/obconf_2.0.3-3_i386.deb
Size: 93698
MD5sum: 400383e7f8feb6906c4a2a3e0f2ede66
SHA1: a810b6ac50922f5f486d0d4fe7d7ff8d3049c562
...

Figure 3.12: *A sample Debian package control file*

- **Suggests**: similar to Recommends field, but dependent packages listed in this field enhance the functionality of the origin package. In other words, the importance of the dependent packages listed in this field is less than those listed in Recommends field.

- **Pre-Depends**: similar to the Depends field, but dependent packages listed in this field must be installed before the installation of the package.

- **Provides**: lists virtual packages whose effect is "as if the package that provide a particular virtual package name has been listed by name everywhere the virtual package name appears." [22]. This field works in the opposite direction of concrete dependency field such as Depends. It declares dependencies for other packages.

- **Enhances**: similar to Suggests, but is used to declare that "a package can enhance the functionality of another package." [22]

For a Debian binary package, Depends and Pre-Depends fields found in its control file are regarded as required dependencies; the packages found in the Recommends, Suggests, and Enhances fields of its control file are considered as optional dependencies. Take obconf 2.0.3 in figure3.12 as an example, obconf 2.0.3 for i386 platform optionally depends on openbox and mandatorily depends on libatk1.0-0 (>= 1.20.0), libc6 (>= 2.7-1), libcairo2 (>= 1.2.4), libfontconfig1 (>= 2.4.0), libfreetype6 (>= 2.3.5), libglade2-0 (>= 1:2.6.1), libglib2.0-0 (>= 2.16.0), libgtk2.0-0 (>= 2.12.0), libice6 (>= 1:1.0.0), libobparser21, libobrender21, libpango1.0-0 (>= 1.20.2), libsm6, libstartup-notification0 (>= 0.8-1), libx11-6, libxft2 (>>2.1.1), libxml2 (>= 2.6.27), zlib1g (>= 1:1.1.4).

Retrieving dependencies from Debian source spec

A Debian binary package is compiled from its source project. The Debian team started to maintain a Sources.gz file since Debian 2.0. This file, a.k.a the Debian source spec, is the documentation of all the source projects in Debian. For each source project, its project name, version, binary packages built from it, priority, section, name and email of its maintainer, build dependents, and other information are included. Figure 3.13 is an excerpt of the source spec of project obconf 2.0.3 found in Debian 5.0.3. In Debian source spec, dependency related control fields are Build-Depends and Build-Depends-Indep. By declaring required packages in the field Build-Depends and Build-Depends-Indep, a project may require certain packages to be available when the it is being built. The difference between Build-Depends and Build-Depends-Indep is that packages required by the clean target should be declared in the Build-Depends field instead of Build-Depends-Indep field. For any Debian source project, the packages found in the Build-Depends and Build-Depends-Indep fields of its source spec are considered as required dependencies. Take obconf 2.0.3 in figure3.13 as an example, debhelper (>= 5), autotools-dev, gettext, openbox-dev, libstartup-notification0-dev, libgtk2.0-dev, libglade2-dev, perl, libsm-dev, and dpatch are its required dependencies.

Merging the result

As illustrated in figure 3.14, the general process to retrieve the most comprehensive list of dependencies of a Debian package is:

1. Retrieve dependencies by analyzing its control file.

Package: obconf
Binary: obconf
Version: 2.0.3-3
Priority: optional
Section: x11
Maintainer: Openbox Maintainers <pkg-openbox@modprobe.de>
Build-Depends: debhelper (>= 5), autotools-dev, gettext, openbox-dev, libstartup-notification0-dev, libgtk2.0-dev, libglade2-dev, perl, libsm-dev, dpatch
Architecture: any
Standards-Version: 3.7.3
Format: 1.0
Directory: pool/main/o/obconf
...

Figure 3.13: *A sample Debian source spec*

2. Look up db_table_deb_src_bin_pkg_match to locate the name of its source project.

3. Retrieve dependencies by analyzing source spec of its source project.

4. Merge the two lists of dependencies retrieved from control file and source spec.

5. Remove any duplicate items from the list.

Take obconf 2.0.3 as an example. By analyzing control file and source spec, we get the dependencies of the binary package and its source project, respectively. The result dependencies after the merge are: debhelper (>= 5), autotools-dev, gettext, openbox-dev, libstartup-notification0-dev, libgtk2.0-dev, libglade2-dev, perl, libsm-dev, dpatch, libatk1.0-0 (>= 1.20.0), libc6 (>= 2.7-1), libcairo2 (>= 1.2.4), libfontconfig1 (>= 2.4.0), libfreetype6 (>= 2.3.5), libglade2-0 (>= 1:2.6.1), libglib2.0-0 (>= 2.16.0), libgtk2.0-0 (>= 2.12.0), libice6 (>= 1:1.0.0), libobparser21, libobrender21, libpango1.0-0 (>= 1.20.2), libsm6, libstartup-notification0 (>= 0.8-1), libx11-6, libxft2 (>>2.1.1), libxml2 (>= 2.6.27), and zlib1g (>= 1:1.1.4).

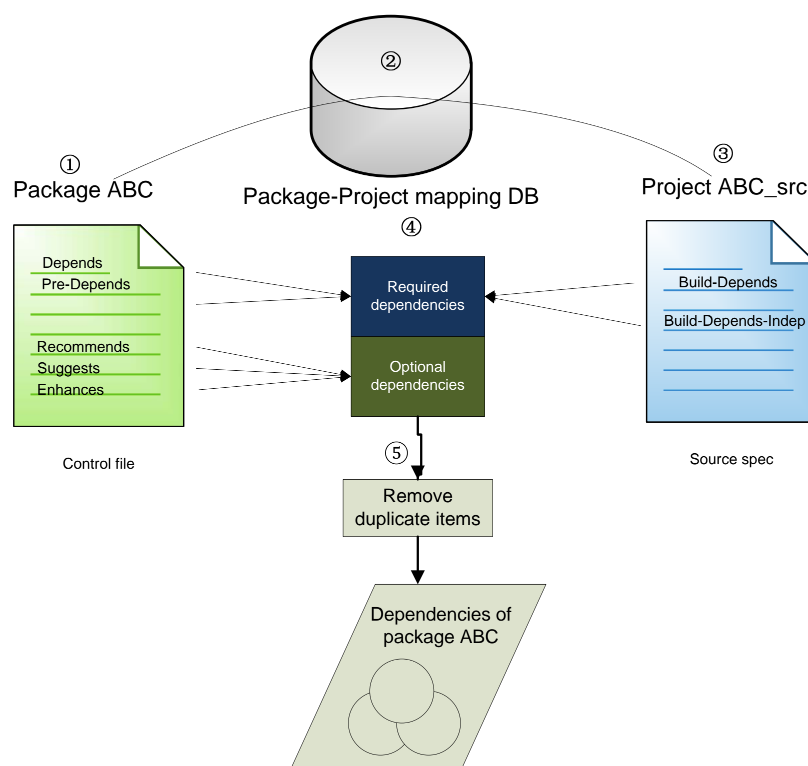

Figure 3.14: *General process to retrieve dependencies from Debian document.*

3.6 Dependency resolution: a binary program approach

Binary files, essentially ELF files (e.g., *.o *.so files), are effective source for dependency resolution. Our approach for binary files' dependency resolution references the methodology of Dependency Checker Tool [3] created by the Linux Foundation. As introduced in chapter 2, Dependency Checker is able to discover linked libraries by scanning a binary file or a directory of binary files. An acceptable binary file for Dependency Checker should be in ELF (Executable and Linkable Format) format, which is the standard binary file format for executables, shared objects, object code, and core dumps on Unix and Unix like systems.

There are two linking methods in Unix/Linux: static linking and dynamic linking.

- **Static linking**: linker copies dependent libraries directly into the program that is being built. Statically linked libraries become part of the executable.

- **Dynamic linking**: unlike static linking that integrates actual library files into

the executable, dynamic linking only places the library name in the executable. Actual linking to the depended libraries does not happen until execution time.

Dependency checker supports the detection of both static and dynamic linked libraries. Simply put, Dependency checker reads output of Unix/Linux standard utility *readelf* to populate the list of dependent libraries. The software takes ELF files and finds dynamically linked libraries using the command below.

- *readelf -a file |grep NEEDED |awk '{print $5}'*

In order to detect static linkages, the software uses the equivalent of the command below to gather a list of interfaces linked to each ELF file from any source. The software then read a list of interfaces that have debugging information. By comparing the two lists, the software generates a table of symbols, which will be consulted to determine the libraries that involved.

- *readelf -s file |grep FUNC |awk '{ print $8 }'*

Dependency checker is effective in discovering dynamically linked libraries but less confident in discovering statically linked libraries [3]. The main reason is *readelf* populates name of dynamically linked libraries but only symbol names in case of statically linked libraries. Unfortunately, normal build process does not embed any information about symbols linked from static libraries, therefore the process to discover statically linked libraries is attempting to derive the data using hints in the application that are not directly relevant to the linking process [3]. Given the inaccuracy and limitations of dependency checker in discovering static linkages, our approach only supports retrieving dynamically linked libraries based on readelf utility in Unix/Linux.

3.6.1 General process

The general process to retrieve dependencies by examining binary files is illustrated in figure 3.15. The dependency extractor accepts a directory of files. Any non-ELF files in the directory are ignored. Each binary file is processed with readelf utility to populate its dynamically linked libraries, which are added to the a list structure that stores retrieved dependencies. For each shared library in the list, we only keep the library name (for instance, libhuginbase.so.0.0 would become libhuginbase). Since different binary file may depend on the same dynamic library, a duplicate check on the

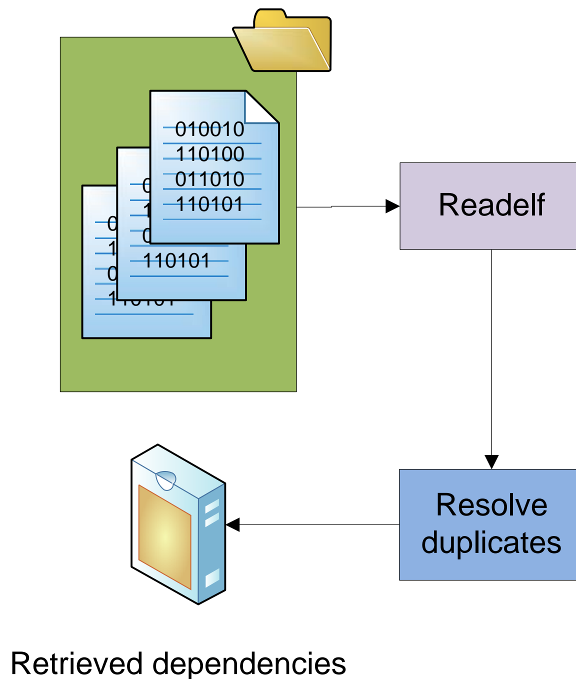

Figure 3.15: *General process to retrieve dependencies from binary files.*

library's name is conducted before the library is added. The result list of dependencies should contain distinct name of shared libraries that the binary files depend on.

Our approach is particularly easy to use for Debian packages[5]. A Debian package is a standard Unix ar archive with deb as the file extension. Binary files belong to that package should be packed into the archive. Our approach accepts a Debian package (or a list of packages), extracting it to a directory, and retrieving dynamically linked libraries of ELF files in that directory.

3.7 Resolving duplicate dependencies

Our research retrieves the most comprehensive list of dependencies by merging the results produced by different dependency resolution techniques. The major problem that arises in the merging process is the possibility of duplicate dependencies. Let's call the set of dependencies retrieved with different techniques the preliminary sets and name the set of merged dependencies as the ultimate set. As illustrated in

[5]Debian packages for various platforms can be found at Debian's binary distribution, http://www.debian.org/CD/http-ftp/#stable

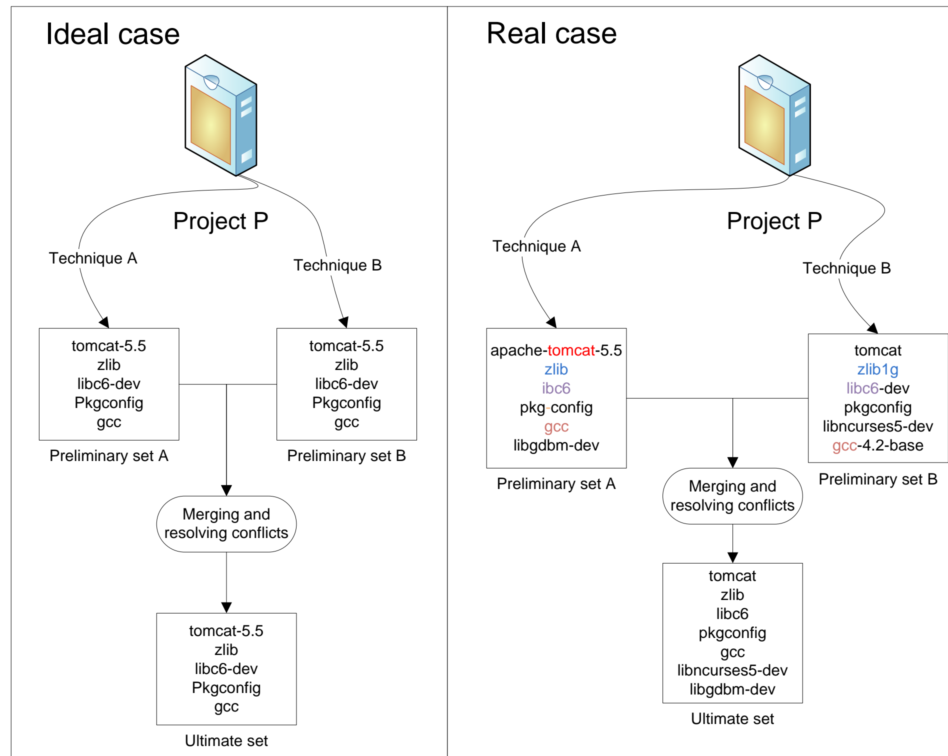

Figure 3.16: *General process to resolve name conflicts when merging dependencies retrieved with different techniques*

figure 3.16, in the ideal world, preliminary sets should be exactly the same as the ultimate set. That is being said different approaches to extract the dependencies should have the same result. However, due to the imperfection of each approach, there are always differences (and sometimes huge differences) between preliminary sets. To make sure the ultimate set only contains distinct items, it is important to remove any duplicate items when merging preliminary sets. Unfortunately, resolving name conflicts in this situation is more complicated than a simple case-insensitive comparison of strings. That is because even the same dependency retrieved with different techniques may have different names. For example, dependency *anon-proxy* may be retrieved with the build technique as *anon-proxy*, but retrieved as *anon-proxy-00.05.38+20080710* with the source code technique. Reporting both anon-proxy and anon-proxy-00.05.38+20080710 in the ultimate set brings redundant information. We call data such as *00.05.38+20080710* the noisy characters. Another case could be Apache Tomcat, which can be retrieved as apache-tomcat with the source code

approach but tomcat5.5 with the spec approach. In this case, *apache* is not noisy data, but *apache-tomcat* and *tomcat5.5* are actually the same project. To overcome this problem, we adopt a name equivalence algorithm that first erases noisy characters, include [+] [-] [.] [**beta**] [ ] [**any digit from 0 to 9**] [**lib**] [**dev**] [*] [?] [**blank space**]. After the noisy characters are removed, a case insensitive non-exact matching comparison based on regular expression is performed. For example, *tomcat5.5* and *apache-tomcat* will be considered the same because tomcat is a sub-string of apache-tomcat after the noise character (i.e. 5.5) is removed.

Based on a duplicate check with the above name conflict resolving algorithm, we are able to retrieve the most comprehensive set of dependencies with only distinct items of a project.

Chapter 4

Evaluation

As explained in chapter 3, our comprehensive dependency resolution approach comprises of four different techniques. The presented techniques to extract inter-dependency by analyzing build scripts, source code, spec, and binary files are realized by a prototype tool `DEx`. To evaluate the effectiveness of `DEx` and the four dependency resolution techniques, we conducted a comparative study, aiming to learn:

- **RQ1**: Is `DEx` able to find the dependencies that are expected to be found (i.e. is the code implementation of `DEx` correct)?
- **RQ2**: Which technique is better, in regards of quality and quantity of retrieved dependencies?

Our evaluation is composed of two studies, aiming to answer research question 1 and 2, respectively.

- **Study I**: manually verifying the dependencies found by `DEx` on a collection of sampling projects.
- **Study II**: programmatically cross-validating dependencies retrieved by each technique on all the Debian projects whose dependencies can be retrieved by all four dependency resolution techniques.

4.1 Study I: verifying correctness of DEx

Study I aims to learn if the techniques described in chapter 3 are accurately and correctly implemented by `DEx`. The steps to perform study I are illustrated as psudo

```
FOREACH technique IN source code technique, spec technique, build
technique, and binary technique
    FOREACH project IN the sampling projects
        1. Retrieve dependencies with DEx, obtaining result set D
        2. Manually retrieve dependencies, obtaining result set M
        3. Compare D and M by calculating
            - TP: the number of true positives in D
            - FP: the number of false positives in D
            - FN: the number of false negatives that
              are in M but not in D
    END FOREACH
END FOREACH
```

Figure 4.1: *Process to perform study I*

code in figure 4.1. For each individually evaluated technique, we carefully compare `DEx` retrieved dependencies with manually discovered dependencies on a collection of sampling projects. Manual dependency extraction uses the same theory as `DEx`. Take the build technique as an example. For any given project, `DEx` should retrieve all possible dependencies from build scripts by examining use of program/library testing macros. As use of macros in CMake and Autoconf could be quite versatile, the investigator should manually read through the build scripts of the sampling project, locate testing macros supported by `DEx` and record the dependencies. The investigator then manually compare the `DEx` discovered dependencies with the ones he/she discovered by examining the build scripts. Evaluation of each sampling project should result in the number of true positive, false positive, and false negative of `DEx` retrieved dependencies. Ideally, the number of false positive and false negative should be 100% for any sampling projects, indicating that all of `DEx` discovered dependencies can be manually verified.

We measure the correctness of `DEx` by using information retrieval metrics. We use the following formula to calculate precision, recall, and accuracy of the study. Precision measures the percent of the dependencies retrieved by `DEx` that are correct; recall measures the percentage of manually retrieved dependencies that were also found by `DEx`; and accuracy measures percentage of all dependencies (i.e retrieved manually or with `DEx`) that were correctly retrieved by `DEx`.

- $$Precision = \frac{TP}{TP + FP} \tag{4.1}$$

- $$Recall = \frac{TP}{TP + FN} \tag{4.2}$$

- $$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.3}$$

TP, FP, FN, TN stands for True Positive, False Positive, False Negative, and True Negative respectively. Calculation is based on total amount of dependencies. For example, while evaluating the build technique, the total number of `DEx` discovered dependencies of the 68 sampled projects are 500, among which 495 dependencies are true positives, 5 are false positives, 7 are false negatives, and 0 are true negatives.

4.1.1 Sampling of subjects

Because our dependency retrieving solution is within the Debian ecosystem, the evaluation was performed on projects in Debian 6.0.0. Each of the four techniques implemented by `DEx` is evaluated individually. Table 4.1 shows the sample size (i.e number of selected sampled projects) and the population size (i.e the number of all projects the technique can be applied to retrieve dependencies) for each technique. For instance, `DEx` can be used to retrieve dependencies with the binary technique for 12230 Debian packages in Debian 6.0.0 main archive. Given it is impractical to do a manual analysis for all the eligible projects, only a portion are selected for the evaluation. We applies the Cochran formula [6] to determine the sample size. Although the population size varies upon different techniques, a sample size of 68 provides 90% confidence level and an error rate (confidence interval) of 10% for all evaluated technique. Only projects whose dependencies can be retrieved by all four techniques are selected as a sample project. To assure fairness and correctness, all of the sampled projects are selected on a random basis.

4.1.2 Results of Evaluation

The result of study I is summarized in table 4.2. All techniques other than the build technique achieve 100% precision, recall, and accuracy, indicating their design is completely and accurately implemented by `DEx`. This result is reasonable: the spec

Table 4.1: Sampled projects of study I

Technique	**Sample size**	**Population size**
Source code	68	7309
Build	68	3009
Binary	68	12230
Spec	68	14515

technique works by parsing highly standardized Debian spec files; the source code technique only processes well-formed and simple #include directives in C/C++ source code (no complex code structure is parsed); the binary technique is built upon highly mature readelf utility. The false positives and false negatives found with the build technique are introduced by complex and embedded testing macro statements found in the processed build scripts. Nevertheless, the build technique still achieves nearly 100% precision, recall, and accuracy. As a result, it is safe to say that the theory of the four dependency resolution techniques introduced in chapter 3 are correctly implemented. This gives us confidence that the rest of the study is sound.

Table 4.2: Evaluation of DEx

Method	**TP**	**FP**	**FN**	**TN**	**Precision**	**Recall**	**Accuracy**
Source code	415	0	0	0	100.00%	100.00%	100.00%
Build	495	5	7	0	99.00%	98.61%	98.62%
Spec	1213	0	0	0	100.00%	100.00%	100.00%
Binary	701	0	0	0	100.00%	100.00%	100.00%

4.2 Study II: evaluating dependency resolution techniques

The high precision, recall, and accuracy found in study I suggests a successful implementation of `DEx`. However, study I does not evaluate the validity of the theory based on which `DEx` is implemented. The result of study I does not indicate the dependencies found by `DEx` (or found manually based on the same theory) also have high precision, recall, and accuracy in genuinity; nor does it unveil any flaws or limitations of the theory. For example, 0 false positive and false negative of source code technique found in study I only indicate that all `DEx` retrieved dependencies can be manually

verified. However, because an include header may match more than one project in the candidate artifact repository, it is very likely that only few of manually retrieved dependencies are actually genuine. In other words, manually retrieved dependencies of a project are the expected result of `DEx`, but there is no guarantee that they are the genuine dependencies truly required by that project. In fact, while conducting study I, we found that the dependencies retrieved by different techniques are hardly identical for the same project. To evaluate the four dependency resolution techniques, we conducted study II, a programmatical cross-validation of dependencies retrieved by different techniques, aiming to learn which technique is better in regards of quality and quantity of retrieved dependencies.

When retrieving dependencies of a project with the four techniques, we could get four result sets. For any project, we define $D^{Sourcecode}$ the set of dependencies retrieved with the source code technique, D^{Build} the set of dependencies retrieved with the build technique, D^{Binary} the set of dependencies retrieved with the binary technique, and D^{Spec} the set of dependencies retrieved with the Debian spec technique. Evaluation of the four techniques is achieved by comparing $D^{Sourcecode}$, D^{Build}, D^{Binary}, and D^{Spec}. The subjects of study II are $D^{Sourcecode}$, D^{Build}, D^{Binary}, and D^{Spec} of all Debian projects whose dependencies can be retrieved with all of the four techniques. Essentially, we would like to know what dependencies can be retrieved by which technique(s) for the sampled projects. Based on this information, we can then evaluate the four techniques.

4.2.1 Data preparation for analysis

We choose all the subjects from Debian 6.0.0 main archive[1]. As described before, in Debian there are source projects and binary packages. A source project contains the source code, build script, and source spec, therefore we can use the source code technique, build technique, and spec technique to retrieve dependencies of a source project. A binary package is compiled from a source project and contains binary files and a control file (i.e package spec), therefore we can use the binary technique and spec technique to extract dependencies of a binary package. A source project may be compiled into more than one binary packages. For instance, binary packages hugin, hugin-tools, and hugin-data are all compiled from source project

[1] Source projects are chosen from main archive of Debian 6.0.0 source; binary packages are chosen from main archive of Debian 6.0.0 i386

hugin. This may cause an issue when attempting to do a cross-comparison of dependencies retrieved by different techniques. For instance, binary technique and spec technique can be used to retrieve dependencies of binary package hugin-tools, but hugin-tools can not work with the source code technique and build technique as there is no source project named hugin-tools. Fortunately, since any binary package must correspond to one source project, we can address this issue by merging the dependencies of all binary packages that are compiled from the same source project. Since a binary package and its corresponding source project could have the same name, we use $D^{Binary+}$ to denote this superset of dependencies. For example, we merge D^{Binary} of binary packages hugin, hugin-tools, and hugin-data, hence $D^{Binary+}_{hugin(the\ source\ project)} = D^{Binary}_{hugin\text{-}tools} \cup D^{Binary}_{hugin\text{-}data} \cup D^{Binary}_{hugin(the\ binary\ package)}$. Since $D^{Binary+}_{hugin}$ now contains dependencies of all binary packages compiled from source project hugin, we can compare $D^{Binary+}_{hugin}$ with D^{Spec}_{hugin}, D^{Build}_{hugin}, and $D^{Sourcecode}_{hugin}$. The steps to prepare $D^{Sourcecode}$, D^{Build}, D^{Binary}, and D^{Spec} of subjects are illustrated as psudo code in figure 4.2.

In order to compare the four techniques, the subjects of study II should be $D^{Sourcecode}$, D^{Build}, D^{Binary}, and D^{Spec} of Debian projects that can be processed by all four techniques. For example, if a project is developed with Java (not supported by `DEx` yet), it cannot be selected since $D^{Sourcecode}$ would not be available for that project. In Debian 6.0.0 main archive, there are 22891 binary packages that are compiled from 16011 source projects. A Debian project is selected as a sample if (1) its dependencies can be retrieved with the source code, build, and spec techniques; (2) dependencies of the binary package compiled from that project can be retrieved with the binary technique. Among the 16011 source projects, only 2313 source projects and their corresponding binary packages can be processed with all four techniques. The 2313 Debian projects form the sampled projects of study II. To facilitate our evaluation, we created a database table, which keeps $D^{Sourcecode}$, D^{Build}, D^{Binary}, and D^{Spec} of all subjects. For each sampled project, we collect the attributes as described in table 4.3. A record in the database table denotes if dependency D of project P can be retrieved with the four techniques. The value of a $\text{Column}_{technique}$ is 1 when D can be retrieved with that technique or 0 otherwise. $D^{Sourcecode}$ is split into two sets: $D^{Sourcecode}_{Full}$ and $D^{Sourcecode}_{Clear}$, denoted by $\text{Column}_{source_full}$ and $\text{Column}_{source_clear}$, respectively. As described in chapter 3, the source code technique may function in full mode or clear mode. For any project, $D^{Sourcecode}_{Full}$ includes the dependencies retrieved with the source code technique in full mode; $D^{Sourcecode}_{Clear}$ includes the dependencies

```
FOREACH project IN Debian 6.0.0 main archive source projects
    APPLY source code technique with DEx on all C/C++
     source code of project, get D^{Sourcecode}_{project}
    APPLY build technique with DEx on all build scripts
     of project, get D^{Build}_{project}
    APPLY spec technique with DEx on source spec of project,
     get D^{Source-Spec}_{project}

    FOREACH package IN binary packages that are compiled from project
        APPLY binary technique with DEx on all binary files
         of package, get D^{Binary}_{package}
        ADD D^{Binary}_{package} to D^{Binary+}_{project}
        APPLY spec technique with DEx on control file (package spec)
         of package, get D^{Package-Spec}_{package}
        ADD D^{Package-Spec}_{package} to D^{Package-Spec}_{project}
    END FOREACH

    REMOVE duplicate items in D^{Binary+}_{project}, get the final D^{Binary+}_{project}
    MERGE D^{Source-Spec}_{project} and D^{Package-Spec}_{project},
     LET D^{Spec}_{project} = D^{Source-Spec}_{project} + D^{Package-Spec}_{project}
    REMOVE duplicate items in D^{Spec}_{project}, get the final D^{Spec}_{project}

    D^{Sourcecode}, D^{Build}, D^{Binary+}, and D^{Spec} of project are prepared
END FOREACH
```

Figure 4.2: *Process to prepare $D^{Sourcecode}$, D^{Build}, $D^{Binary+}$, and D^{Spec} of subjects in study II*

retrieved with the source code technique in clear mode.

We follow the procedure explained as psudo code in figure 4.3 to create the dependency data, which contains the 57916 dependencies retrieved by the four methods of all the 2313 sampled projects. As an example, table 4.4 shows data corresponds to project flac. From the table, we can tell libogg can be retrieved by all four methods, but debhelper is only retrieved with the spec technique. With all the data being prepared, we can do cross-validation and answer the following research questions:

1. Which technique retrieves more dependencies?

2. Which technique is more accurate?

Table 4.3: Database table: db_table_experiment_data

ATTRIBUTE	COMMENT
source_full	Denote if dep_name can be retrieved with source code (full mode) technique.
source_clear	Denotes if dep_name can be retrieved with source code (clear mode) technique.
build	Denote if dep_name can be retrieved with build script technique.
binary	Denote if dep_name can be retrieved with binary file technique.
spec_source	Denote if dep_name can be retrieved with spec technique by processing source spec.
spec_package	Denote if dep_name can be retrieved with spec technique by processing package spec.
spec	Calculated by spec_source | spec_package. Denote if dep_name can be retrieved with spec technique by processing either source spec of package spec

3. Is there any correlation between different techniques?

4. Why do dependencies retrieved by different techniques vary?

5. Can the dependencies retrieved by different techniques validate each other?

4.2.2 Study II-I: Which technique retrieves more dependencies?

It is not uncommon that for the same project, the number of dependencies retrieved with the four techniques is different. We would like to know for a random project, which technique retrieves more dependencies. The best technique should retrieve most (if not all) of the dependencies that a project has. To answer this research question, we use a concept of coverage rate to evaluate the number of dependencies retrieved by a technique. As defined in the formula below, we define the **C**overage **R**ate ($CR_{project}^{Technique}$) of a project per technique as a percentile number calculated by dividing the number of dependencies retrieved with the technique by the total number of distinct dependencies retrieved with all of the four techniques. A higher $CR_{project}^{Technique}$ means compared to other techniques, more dependencies can be retrieved with the

Table 4.4: Experiment data of project flac

dep_name	**project_ name**	**source_ full**	**source_ clear**	**build**	**binary**	**spec_ source**	**spec_ package**	**spec**
libstdc++	flac	0	0	0	1	0	1	1
libc6	flac	1	1	0	1	0	1	1
gcc	flac	1	0	0	0	0	1	1
debhelper	flac	0	0	0	0	1	0	1
libm	flac	0	0	0	1	0	0	0
autotools	flac	0	0	0	0	1	0	1
autoconf	flac	0	0	1	0	0	0	0
doxygen	flac	1	0	1	0	1	0	1
gas	flac	0	0	1	0	0	0	0
as	flac	0	0	1	0	0	0	0
libid3	flac	0	0	0	0	1	0	1
dpatch	flac	0	0	0	0	1	0	1
libogg	flac	1	0	0	1	1	1	1
nasm	flac	0	0	1	0	0	0	0
docbook-to-man	flac	0	0	1	0	1	0	1

specified technique for the same project. In the flac project example, 15 dependencies can be retrieved with the four techniques, among which 5 can be clearly retrieved with the source code technique, 6 with the build technique, 4 with binary technique, and 10 with the spec technique. Therefore, the coverage rate of the four techniques of flac is 33.33%(5/15), 40%(6/15), 26.67%(4/15), 66.67%(10/15), respectively.

$$D_{project}^{Total} = D_{project}^{Sourcecode_clear} \cup D_{project}^{Build} \cup D_{project}^{Binary} \cup D_{project}^{Spec} \tag{4.4}$$

$$CR_{project}^{Technique} = \frac{D_{project}^{Technique}}{D_{project}^{Total}} \times 100\% \tag{4.5}$$

By calculating coverage rate of the four techniques of all sampled projects, we may learn which technique generally retrieves more dependencies from a statistical perspective. While evaluating the source code technique, we analyzed both $D^{Sourcecode_full}$ and $D^{Sourcecode_clear}$. $CR^{Sourcecode_full}$ and $CR^{Sourcecode_clear}$ denote how comprehensive the source code technique would be in its full capacity and in clear mode. We calculate coverage rate of build technique, binary technique, spec technique

Table 4.5: Statistical summary of coverage rate of the four techniques of all sampled projects in study II

Technique	**Min**	**Q1**	**Median (Q2)**	**Mean**	**Q3**	**Max**	**Standard Deviation**
Source code (full)	0%	7.692%	16%	19.390%	27.273%	90%	14.748
Source code (clear)	0%	3.846%	6.897%	8.612%	11.111%	48.485%	6.368
Build	1.418%	18.750%	29.545%	31.758%	42.718%	89.744%	16.869
Binary	1.887%	31.25%	46.667%	46.638%	62.5%	90.78%	19.537
Spec	11.11%	56.41%	66.67%	65.94%	75.68%	100%	15.254

of a project based on $D^{Build}_{project}$, $D^{Binary}_{project}$, and $D^{Spec}_{project}$, respectively.

Figure 4.4 is the box percentile plot diagram that depicts the coverage rate of the four techniques of all sampled projects. As an improvement of regular box plot diagram, box percentile plot diagram uses width to encode information about the distribution of the data over the entire range of data values [11]. Coverage rate data of each technique is presented as one box percentile plot in figure 4.4. The width of a plot depicts distribution of data. Five-number summaries (smallest observation (Min), lower quartile (Q1), median (Q2), upper quartile (Q3), and largest observation (Max)) of the data set are depicted by the lowest point, lower line, middle line, upper line, and highest point in the plot. Statistical summary of each data set is presented in table 4.5. By comparing Q1, mean, median, and Q3 of each data set, we can easily tell $CR^{Spec} > CR^{Binary} > CR^{Build} > CR^{C/C{+}{+}_full} > CR^{C/C{+}{+}_clear}$. That being said, for a random project, spec technique generally retrieves most dependencies while source code (clear mode) retrieves the least. The high standard deviation indicates that the coverage rate of a technique for different projects may vary significantly. For instance, the build technique may find as much as 89.744% dependencies of a project, but it may also find as low as 1.418% dependencies of another project. In fact, all four techniques present high standard deviation in coverage rate, which indicates that using any technique alone does not guarantee a constant high coverage rate for a random project. Despite the uncertainty exposed by high standard deviation, we may still learn from figure 4.4 that coverage rates of most projects are distributed between Q1 and Q3, which means for a randomly selected project:

- Source code (clear mode) technique are most likely to retrieve 3.846% - 11.111% of all dependencies.
- Source code (full mode) technique are most likely to retrieve 7.692% to 27.273% of all dependencies.
- Build technique are most likely to retrieve 18.75% - 42.718% of all dependencies.
- Binary technique are most likely to retrieve 31.25% - 62.5% of all dependencies.
- Spec technique are most likely to retrieve 56.41% - 75.68% of all dependencies.

The finding may also be confirmed through the scatter plot in figure4.5. We assign each of the 2313 sampled projects a unique ID (ranging from 1 to 2313), which forms the X-axis. The Y-axis denotes the coverage rate (ranging from 0% to 100%). Each technique is represented with a unique glyph. For example, the green triangle located at coordinate (99, 80) indicates build technique can retrieve 80% of all dependencies of project 99 (project name: tf).

Summary

- Ranked by the number of dependencies may be retrieved for the same project with different techniques, $Technique^{Spec} > Technique^{Binary} > Technique^{Build} > Technique^{Sourcecode(full)} > Technique^{Sourcecode_clear}$
- Coverage rate of a technique for different projects may vary significantly, therefore no technique may guarantee a constantly high/low coverage rate.

4.2.3 Study II-II: Which technique is more accurate?

In study II-I, we learn which technique generally retrieves more dependencies, i.e the performance of the four techniques from the quantity perspective. We assume the total dependency set of a project ($D^{Total}_{project}$) is the superset unioned by dependencies retrieved with each technique. However, dependencies retrieved with any technique may contain false data, which would contaminate the accuracy of $D^{Total}_{project}$. Inaccurate $D^{Total}_{project}$ would then contaminate $CR^{Technique}_{project}$ of all techniques. From the quality perspective, it is important to learn which technique is more accurate. Essentially, we would like to know if N dependencies can be retrieved for a project with a technique, how many of them are actually genuine and how many are false? It is important

Table 4.6: Statistical summary of accuracy of source code (clear mode), build, binary techniques of all sampled projects in study II

Technique	**Min**	**Q1**	**Median (Q2)**	**Mean**	**Q3**	**Max**	**Standard Deviation**
Source code (clear)	0%	100%	100%	86.78%	100%	100%	26.917
Build	0%	25%	50%	45.09%	66.67%	100%	28.846
Binary	0%	50%	65.22%	66.24%	80%	100%	21.246

to have the set of genuine dependencies ($D^{Genuine}_{project}$) so that the accuracy of each technique can be evaluated by comparing $D^{Technique}_{project}$ and $D^{Genuine}_{project}$. Unfortunately, $D^{Genuine}_{project}$ is unknown. The best replacement of $D^{Genuine}_{project}$ would be $D^{Spec}_{project}$. The spec technique retrieves dependencies by analyzing source spec and package spec, which are maintained by official Debian maintainers and heavily relied by Debian package management tools (e.g., apt-get, dselect, aptitude). Therefore, it is safe to claim that $D^{Spec}_{project}$ is the best (though not the perfect) replacement of $D^{Genuine}_{project}$.

$$Accuracy^{Technique}_{project} = \frac{D^{Technique}_{project} \cap D^{Spec}_{project}}{D^{Technique}_{project}} \tag{4.6}$$

We use formula 4.6 to calculate the accuracy of a technique. We define the accuracy of a technique per project is the number of dependencies retrieved by that technique intersect dependencies retrieved with spec technique divided by the number of dependencies retrieved with that technique. A higher accuracy indicates a higher proportion of retrieved dependencies can also be found in Debian spec. Similar to study II-I, we calculate $Accuracy^{Technique}_{project}$ of the source code (clear mode), build, and binary technique for all of the 2313 sampled projects. We did not analyze source code (full mode) technique as it is already known that dependencies retrieved with this technique may contain many false positives. We compare accuracy of different techniques with a statistical analysis to the result.

According to figure 4.6 and table 4.6, the accuracy of source code (clear mode) method is surprisingly high, indicating that dependencies retrieved with the source code (clear mode) method can usually be found in Debian spec. This finding significantly contrast the poor coverage rate of the source code technique. In fact, for

more than 75% of the sampled projects, dependencies retrieved with the source code (clear mode) technique have 100% accuracy. In contrast, the build technique has a relatively lower accuracy, but $Accuracy_{project}^{Build}$ of half sampled projects is still more than 50%. The binary technique, although not as accurate as the source code (clear mode) technique, still achieves at least 50% accuracy for great majority ($> 75\%$) of sampled projects.

In this study, we use $D_{project}^{Spec}$ to measure the accuracy of other techniques, but what if Debian spec is wrong? It is possible that there might be incorrect or outdated information in Debian spec, but given that all of the dependencies listed in Debian spec files have been manually verified by official Debian maintainers and been tested by millions of Debian users when they manage packages via Debian package managers, the high accuracy of Debian spec should not be doubted.

Summary

- When using Debian spec as the standard, and calculate the accuracy of a technique by comparing dependencies retrieved with that technique with Debian spec, dependency resolution techniques ranked by accuracy are:
  $Technique^{Sourcecode(clearmode)} > Technique^{Binary} > Technique^{Build}$

4.2.4 Study II-III: How do dependencies retrieved with different techniques compare with each other?

When comparing dependency resolution techniques, it is interesting to learn how are these techniques correlated. In another word, if a dependency can be retrieved with technique X, can it also be retrieved by technique Y? Or if a dependency cannot be retrieved with technique X, can it also not be retrieved with technique Y? Knowing the correlationship of techniques can help user choose a replacement when one technique is not available to use. We would like to know the correlation between source code (full mode), source code (clear mode), build, binary, and spec techniques.

We research the correlation of dependency resolution techniques by analyzing how the 57916 dependences in db_table_experiment_data are retrieved. Such information is revealed by 1 or 0, which indicates a dependency can or cannot be retrieved by a technique. Because the data is not linear, it is cannot measured by the Pearson's correlation. The key factor to determine a correlation between dependency resolution techniques is to find out a pattern of co-occurrence. Given that only 1 or 0 are used to

denote information in db_table_experiment_data, such pattern can be analyzed with binary coding. We define a 5 digit binary code *Correlation-code*. For any dependency, bits 1 to 5 of *Correlation-code* represents whether this dependency can be retrieved with source code (full mode), source code (clear mode), build, binary, and spec technique, respectively. In the flac example, *Correlation-code* of dependency doxgen would be 10101, implying doxgen can be retrieved with source code (full mode), build, and spec technique. In theory, there are 2^5 (i.e 32) possible patterns, which are explained in table 4.8. Each pattern, decoded by a correlation code, represents a combination of how a dependency can be retrieved with different techniques. Each pattern can be uniquely identified by its ID, which is the decimal value of the correlation code. $Column^{Count}$ and $Column^{Usage}$ lists the total number and percentage of dependencies among the 57916 dependencies in db_table_experiment_data. $Column^{Srcfull}$, $Column^{Srcclear}$, $Column^{Build}$, $Column^{Binary}$, and $Column^{Spec}$ indicates the dependency can be retrieved with the corresponding technique. For example, the second row in table 4.8 implies 16866 or 26.613% of all the sampled dependencies can only be retrieved by the spec technique.

We define correlation coefficient of two techniques ($CE_{X,Y}$) as the sum of the value of $Column^{Usage}$ of all correlation-codes whose two digis representing the two techniques are the same. For example, $CE_{Build,Binary} = \sum_{ID=1,6,7,17,22,23,24,25,30,31} Column^{Usage}$, namely the sum of value in $Column^{Usage}$ where the third and fourth digit in $Column^{Correlation-code}$ are the same (either both 1, or both 0). $CE_{X,Y}$ indicates the probability a dependency may be retrieved with technique X when it can be retrieved with technique Y or the probability a dependency cannot be retrieved with technique X when it cannot be retrieved with technique Y. For any two techniques X and Y, $CE_{X,Y} = CE_{Y,X}$. $CE_{X,Y}$ of all the dependency resolution techniques are presented in table 4.7.

From table 4.7, we can learn that source code (full mode) correlated with source code (clear mode) in a very high degree (86.387% correlation coefficient), which was expected. Also considering the low accuracy of source code (full mode) technique, and extremely high accuracy of source code (clear mode) technique, we may conclude that source code technique clear mode is as comprehensive as full mode without sacrificing accuracy. Most of the abstract dependencies retrieved with source code (full mode) technique are actually false positives. We may also find build technique and build technique have a relatively high correlation coefficient with source code (clear mode) technique, indicating there is 65.554% chance that retrieving a dependency

Table 4.7: Correlation coefficient of any two dependency resolution techniques

	Source (full)	**Source (clear)**	**Build**	**Binary**	**Spec**
Source (full)	100				
Source (clear)	86.387	100			
Build	60.899	65.554	100		
Binary	48.382	51.221	40.851	100	
Spec	44.982	39.105	32.077	45.844	100

with build technique or source code (clear mode) technique would have the same result. We also notice that the spec technique has relatively low correlation coefficient (less than 50%) with any other technique, indicating that there is more than 50% chance that retrieving a dependency with spec technique and another technique would have different result.

Summary

- Source code (clear mode) technique is as comprehensive as full mode without sacrificing accuracy, therefore source code (clear mode) can replace source code (full mode) in most cases.
- Correlation coefficient between different techniques is usually low. That being said, for the same project, its dependencies retrieved with different techniques usually have less than 50% similarity.
- The spec technique has relatively low correlation coefficient with any other techniques.

4.2.5 Study II-IV: Why are dependencies retrieved with different techniques not the same?

From the previous studies, it is surprising to learn that dependencies retrieved with different techniques are not always the same. In fact, from table 4.7, we can learn that in 53.788% cases, a dependency can only be retrieved with one technique. Ideally, a user should always retrieve the same result regardless of what technique is being used.

The unexpected low correlation coefficient drives us to conduct this study, aiming to learn why the results vary so significantly.

Table 4.9 lists the top 10 dependencies that are uniquely retrieved by build, binary, and spec technique. The result of source code technique is not included because the low sample size (amount of any dependency that are uniquely retrieved with source code technique are no more then 2). Although each technique has over 1000 uniquely retrieved dependencies, the top 10 most popular dependencies account for 29.4%, 67.8%, 36.4% of all dependencies uniquely retrieved by build, binary, spec technique respectively. Therefore, by analyzing the top 10 list, we could have a general idea of why different techniques generate different results. From table 4.9, we find that:

- Most time, the build technique uniquely retrieved dependencies are either Debian built-in programs (socket, ar, rm, etc.) or standard programs (autotools, pkgconfig) for build process. Many of these programs are most basic tools or commands in Linux/Unix, it is not surprising that they are not retrieved by other techniques. For example, Debian maintainer may not bother to add rm in the dependency list as rm really is available in every Linux/Unix system. It is quite normal that build process related dependencies are not retrieved with the binary technique, because binary technique only retrieves dynamic linked libraries. These build related dependencies, however, should be retrieved with the spec technique, otherwise there might be an error in Debian spec.

- The binary technique retrieves the dynamically linked libraries that an ELF binary file depends on. The retrieved results are the name of shared object files (*.so files). The results retrieved with other techniques, on the other hand, are the name of dependent packages. The name of *.so file usually instructs a binary package (e.g, libc6.2-2.so.3 retrieved with the binary technique indicates a dependency on package libc6). However, a *.so file does not have to be named the same as its belonging package. Therefore, it is not uncommon that dependencies retrieved with the binary technique may have a unique name.

- Most of spec technique uniquely retrieved dependencies are either Debian build related programs (autotools, automake, cdbs, pkgconfig, dpatch, etc) or package management programs (dephelper, dpkg, etc.) They are infrastructural utilities to build and manage Debian packages, therefore they can only be recorded in Debian spec.

Summary

- Different results produced by dependency resolution techniques do not necessarily imply inaccuracy of the techniques. The theory behind each technique determines that some dependencies may only be retrieved with a specific technique. Therefore, a most complete list of dependency of a project should include dependencies retrieved with all possible approaches. That being said a comprehensive approach is more effective when compared with each composing technique used alone.

4.3 Discussion

4.3.1 Feature summary of dependency resolution techniques

The four techniques presented in chapter 3 can be compared from 12 general aspects as summarized in table 4.10.

Source

The most obvious difference between these techniques are the source from which dependency information is extracted. The build technique relies on build scripts of various build systems; the source code technique analyzes source code and matching against an artifact candidate repository to obtain the list of dependent libraries; the spec technique parses Debian spec of a specific project to retrieve the dependency information described in it; the binary technique processes ELF binary files to obtain a list of dynamically linked libraries. The various sources of techniques offer users a broader choices to perform dependency resolution. Therefore, when one technique is not available to use, the user may always opt for another one.

Platform independent

One important factor that affects the utilization of a technique is its ability to function across platforms. The build and source code are purely based on text parsing and matching, therefore they can be used on any operating system. The spec technique only works on Debian. Essentially as a user document approach, a spec-like technique must be developed for any non-Debian platforms. The binary technique, however, retrieves dependencies by analyzing ELF files, which are standard binary file format

of Unix and Unix-like systems only. Different platform such as Microsoft Windows, have their own standard binary file format. Therefore, a binary dependency resolution technique must be developed for each platform.

Programming language independent

The build and spec techniques work regardless of what programming language is used to develop the target project. In contrast, the source code technique is intrinsically dependent on programming language. Since an ELF files are usually compiled from source code written in C/C++ programming language, the binary technique only works on projects written in C/C++.

Require repository

The source code technique requires an artifact candidate repository for cross referencing header file with a collection of known libraries. The other techniques, however, do not require any repository to function.

Comprehensiveness of result

We evaluate the comprehensiveness of a technique by its coverage rate. According to study II-I, $CR^{Spec} > CR^{Binary} > CR^{Build} > CR^{Sourcecode}$. Build techniques retrieve dependencies required during build or installation. The binary technique retrieves dynamically linked libraries, which are runtime dependencies of a project. The source code technique reveals compilation time dependencies. The spec technique, on the other hand, may retrieve all types of dependencies.

Retrieve version of dependency

Knowing the name of a dependency is sometimes not enough. A project sometimes has a specific version requirement of dependency. The library testing macros in build scripts can check not only the name but also the version of a dependency, therefore, the build technique can retrieve dependency version quite precisely. The spec technique also reveals version of dependencies as long as version information is recorded in Debian spec. The source code technique may retrieve version information, depending on whether such information exists in artifact candidate repository, however, if the repository has multiple records that matching the same dependency but with different

versions, the source code technique may report incorrect or ambiguous version information. In this situation, analyzing the source code of the matching projects and measuring objects of interests (e.g, calls to external libraries, methods signatures, dynamic libraries) in various ways (e.g, count-based, set-based, sequence-based, or relationship-based) may be helpful in identifying the correct version [7]. The binary technique does not reveal version, but version may be found as part of dependency name.

Support abstract dependencies

Abstract dependencies are a set of packages which have the same functionality and can be exchanged with each other. The build and spec techniques can retrieve a dependency as well as all its alternatives. The dependency itself and its alternatives form an abstract dependency. The source code technique may also retrieve abstract dependencies. In the artifact candidate repository, projects contain the same header file form an abstract dependency. This approach may find some abstract dependencies. Nevertheless, considering two different projects containing the same header file do not also indicate the projects have the same functionality, the accuracy and effectiveness of source code technique in supporting abstract dependencies are questionable. The binary technique does not support abstract dependencies.

Support dependency type

The build and spec techniques are able to tell whether a retrieved dependency is a required or optional. The other techniques have no such ability.

Platform specific dependencies

Sometimes, a dependency may be required on a specific platform (such as i386, amd64, ia64, etc.). Because binary and spec techniques process platform specific packages, they are able to retrieve such platform specific dependencies. Such platform specific dependencies may also be verified in built scripts, however, `DEx` does not construct the abstract syntax tree of a build file, and therefore cannot determine if a retrieved dependency is platform specific. The source code technique cannot determine whether a retrieved dependency is platform specific either. The binary amd spec techniques, however, are able to report dependencies specific to the target platform.

Easy to use

The build, spec and binary techniques do not rely on external repositories. They are standalone programs that are quite easy to use. On the other hand, the convenience of the spec technique is greatly affected due to the requirement of external repository. Such repository usually requires a large space to store and external programs such as database. However, this shortcoming can be overcome by offering access to the repository via web services.

Convenience for automatic analysis

Build scripts, ELF files, and source code are highly standardized. They either have well-defined structure or standard format. This makes automatic processing of such files quite convenient. The writing style of a document could be quite diverse; thus, it could be very difficult to design a universal but automatic technique to extract dependency from them.

Special advantages of each technique

Each technique has its own unique advantages. For example, the build system technique is more effective in retrieving build time and installation time dependencies; the source code technique may reveal inner-dependencies; the spec technique retrieves most comprehensive results; the binary technique generates more precise results. For example, the binary technique may retrieve a dependency ”gdk-x11”, but the other techniques may retrieve more general dependency such as ”gdk”.

Generalizability

Generalizability of a technique denotes how convenient to use it outside the ecosystem of this research, i.e the Debian system. The build technique can be applied to any project within any environment (Windows, Linux, Unix, etc.) as long as the project uses the GNU Autotools build system or CMake. The binary technique can be used in any Unix and Unix-like environment. Because the artifact candidate repository used in our research only contains Debian projects, the source code technique implemented by `DEx` might be less effective when used outside the Debian environment, but may still discover many common dependencies. By replacing the current repository with a larger repository that indexes more projects, the source technique may easily work

beyond the boundary of Debian. The spec technique is strictly related to Debian spec, therefore it cannot be used outside Debian environment at all. The generalizability of the spec technique is particularly low.

4.3.2 Threats to validity

Identified limitations during evaluation studies include concerns over general process to retrieve dependencies with each technique as well as validity of the evaluation methodology. The effectiveness and validity of each dependency resolution technique presented in chapter 3 may be affected by a variety of factors. The findings from study I and study II may also be compromised by limitations of evaluation methodology.

Threats to validity of source code dependency resolution technique

The effectiveness of the source code technique heavily relies on the completeness of artifact candidate repository. The repository in our research only indexes the header-project mappings of Debian projects in the Debian main archive. Compared to code repositories such as Google code search or Merobase, the size of the repository used in our research is relatively small. The imperfection of the repository might be a reason of why the source code technique generally retrieve less dependencies than other techniques. There might be several reasons that cause an include file not having a matching project in the repository. For example, our repository only includes system header files found under /usr/include directory of a standard Debian installation, therefore the indexed system headers might not be complete; when indexing the header files of projects in the Debian main archive, some header files are not indexed because they are compressed into a tar ball, which is within another tar ball (we only extract the outer level tar ball); some include files declared with the #include directive are C/C++ source files (*.c, *.cpp), but our system only processes header (*.h) files. The solution is to upgrade the current repository to a better one with more header files and source files indexed.

Another threat may be the general philosophy of the source code technique, namely if file X is part of project ABC, then any project, say DEF, that declares dependency (by using import keyword in Java, include in C/C++, require or use in Perl, etc) of file X is deemed to be dependent on project ABC. However, just because file X is part of project ABC does not necessarily imply ABC has the functionality that is required by project DEF. It is possible that the file X just coincidentally has

the same file name the DEF requires. Or file X may be cloned to project ABC just to avoid dependency declaration. However, the high accuracy of source code technique found in study II imply that the impact of these threats is not significant.

Threats to validity of build dependency resolution technique

The build technique retrieves dependencies by parsing build scripts of a project, therefore, the quality of build scripts greatly affect the results. For example, CMakeLists.txt of project cmake-2.8.2 found under subdirectory Tests/FindPackageTest contains package testing macros such as FIND_PACKAGE(NotAPackage QUIET), FIND_PACKAGE(VersionTestA 1), FIND_PACKAGE(VersionTestB 1.2), FIND_PACKAGE(VersionTestC 1.3), FIND_PACKAGE(VersionTestD 1.2.3.4). In this case, the build technique may claim NotAPackage, VersionTestA, VersionTestB, VersionTestC, and VersionTestD are dependencies of cmake-2.8.2. Clearly, they are not inter-dependencies of cmake-2.8.2. In fact, by observing the directory where CMakeLists.txt is placed, we can tell this build script is for testing purpose only. However, when retrieving dependency with build technique in an automatic fashion, such false positive results are difficult to catch.

In addition, as new library/program testing macros may be supported in by CMake and Autoconf, the current build technique may not find dependencies which are tested by non-supported testing macros. This problem can be solved by constantly updating supported macros in `DEx`.

Threats to validity of binary dependency resolution technique

As the binary dependency purely relies on readelf tool in Unix and Unix-like systems, any defect of readelf may impact the dependencies retrieved with the binary technique. In addition, the binary technique cannot resolve any dependency of non-ELF files.

Threats to validity of spec dependency resolution technique

The effectiveness of the spec technique is wholly determined by the quality of Debian spec. Inaccurate information in spec file will leads to inaccurate results of spec technique. For example, with spec technique, `DEx` may reports libid3 a dependency of flac. That's because libid3 is listed in flac's source spec. However, an examination of flac documentation (which states: Removed support for ID3 tags) discovers that libid3 is no longer required by flac. In this case, the source spec is outdated and includes

packages that are no longer needed. Since a dependent module may have a different license, falsely including unnecessary dependency may also negatively impact the accuracy of license analysis.

Threats to validity of evaluation

Generally speaking, studies I and II evaluate the effectiveness of four dependency resolution techniques. However, the validity of evaluation result may be affected by several threats:

- Sample size in study I only provides a confidence level of 90% and confidence interval of 10%. More defects may be found if sample size is larger. In addition, because all `DEx` retrieved dependencies are manually verified, human errors may also affect the result of study I.
- Evaluation of accuracy of dependency resolution techniques in study II is based on the assumption that the Debian spec should have the most comprehensive and accurate information. However, there might be mistakes in Debian spec. Study II-IV also shows that a dependency not recorded in Debian spec is not always a false positive.
- In study II-IV, we only analyzed the top 10 uniquely retrieved dependencies of each technique. The conclusion of study II-IV may not apply to dependencies that are not analyzed.

FOREACH *project* IN sampled projects

MERGE $D^{Sourcecode_clear}_{project}$, $D^{Build}_{project}$, $D^{Binary+}_{project}$, and $D^{Spec}_{project}$.
DEFINE $D^{Comprehensive}_{project} = D^{Sourcecode_clear}_{project} \cup D^{Build}_{project} \cup D^{Binary+}_{project} \cup D^{Spec}_{project}$
REMOVE duplicate items in $D^{Comprehensive}_{project}$

FOREACH *dependency* IN $D^{Comprehensive}_{project}$
IF (*dependency* IN $D^{Sourcecode_Full}_{project}$)
UPDATE database SET source_full = 1
WHERE dep_name = *dependency* AND project_name = *project*
END IF
IF (*dependency* IN $D^{Sourcecode_clear}_{project}$)
UPDATE database SET source_clear = 1
WHERE dep_name = *dependency* AND project_name = *project*
END IF
IF (*dependency* IN $D^{Build}_{project}$)
UPDATE database SET build = 1
WHERE dep_name = *dependency* AND project_name = *project*
END IF
IF (*dependency* IN $D^{Binary+}_{project}$)
UPDATE database SET binary = 1
WHERE dep_name = *dependency* AND project_name = *project*
END IF
IF (*dependency* IN $D^{Source-Spec}_{project}$)
UPDATE database SET spec_source = 1
WHERE dep_name = *dependency* AND project_name = *project*
END IF
IF (*dependency* IN $D^{Package-Spec}_{project}$)
UPDATE database SET spec_package = 1
WHERE dep_name = *dependency* AND project_name = *project*
END IF
IF (*dependency* IN $D^{Spec}_{project}$)
UPDATE database SET spec = 1
WHERE dep_name = *dependency* AND project_name = *project*
END IF
END FOREACH

END FOREACH

Figure 4.3: *Process to update database that stores experiment data for study II*

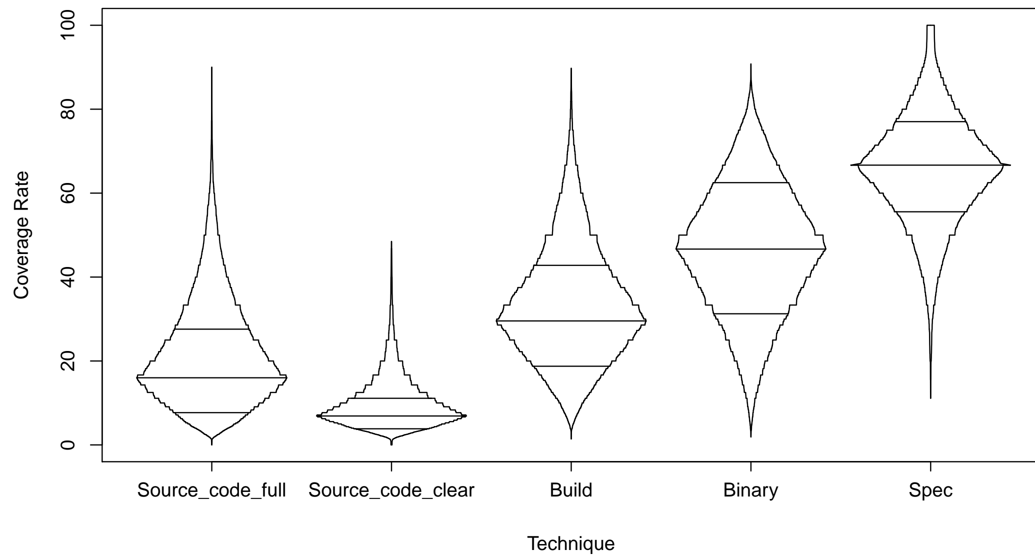

Figure 4.4: *Boxplot of coverage rate of the four techniques of all sampled projects in study II*

Figure 4.5: *Scatter plot of coverage rate of the four techniques of all sampled projects in study II*

Table 4.8: Correlation code with dependency resolution pattern encoded

ID	Correlation code	Count	Usage	Src full	Src clear	Build	Binary	Spec
0	00000	0	0%					
1	00001	13480	23.275%					•
2	00010	9693	16.736%				•	
3	00011	9365	16.170%				•	•
4	00100	6807	11.753%			•		
5	00101	2204	3.806%			•		•
6	00110	1391	2.402%			•	•	
7	00111	3379	5.834%			•	•	•
8	01000	20	0.035%		•			
9	01001	2	0.003%		•			•
10	01010	0	0%		•		•	
11	01011	31	0.054%		•		•	•
12	01100	0	0%		•	•		
13	01101	0	0%		•	•		•
14	01110	0	0%		•	•	•	
15	01111	0	0%		•	•	•	•
16	10000	0	0%	•				
17	10001	3106	5.363%	•				•
18	10010	819	1.414%	•			•	
19	10011	1365	2.357%	•			•	•
20	10100	1228	2.120%	•		•		
21	10101	399	0.689%	•		•		•
22	10110	150	0.259%	•		•	•	
23	10111	764	1.319%	•		•	•	•
24	11000	1172	2.024%	•	•			
25	11001	94	0.162%	•	•			•
26	11010	2	0.003%	•	•		•	
27	11011	2307	3.983%	•	•		•	•
28	11100	12	0.021%	•	•	•		
29	11101	25	0.043%	•	•	•		•
30	11110	0	0%	•	•	•	•	
31	11111	101	0.174%	•	•	•	•	•

Table 4.9: Top 10 uniquely retrievable dependencies by technique

Build	**Binary**	**Spec**
autotools	libm	debhelper
socket	pthread	cdbs
gconftool-2	libgobject	quilt
libnsl	gdk-x11	pkgconfig
pkgconfig	librt	automake
libperl	gdk-pixbuf	dpkg
ar	gmodule	dpatch
rm	pangocairo	intltool
glib-genmarshal	libdl	libgtk-2.0
sed	libgio-2.0	libtool

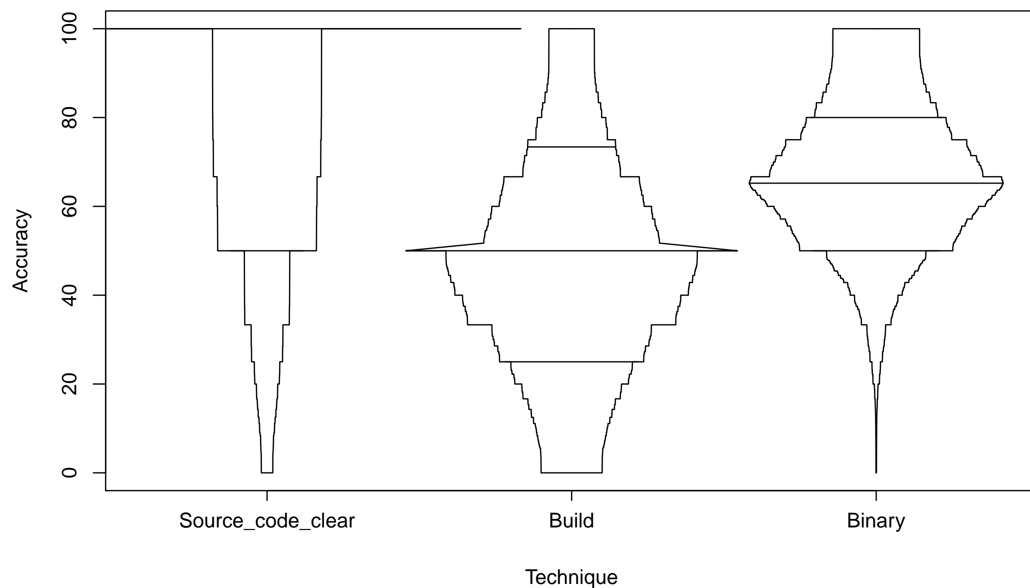

Figure 4.6: *Boxplot of accuracy of source code (clear mode), build, binary techniques of all sampled projects in study II*

Table 4.10: Feature comparison of dependency resolution techniques

	Build	**Source code**	**Spec**	**Binary**
Source	Build scripts	Source code	Debian spec	ELF files
Platform independent	Yes	Yes	No	No
Programming language independent	Yes	No	Yes	No
Require repository	No	Yes	No	No
Comprehensiveness of result	Medium	Low	High	Medium
Retrieve version of dependency	Yes	Yes*	Yes	No
Support abstract dependencies	Yes	Yes	Yes	No
Support dependency type	Yes	No	Yes	No
Platform specific dependencies	No*	No	Yes	Yes
Easy to use	High	Low	High	High
Convenience for automatic analysis	High	High	High*	High
Special advantages	Build and installation time dependency	Inner dependency	Most comprehensive result	Find dynamically linked libraries
Generalizability	High	Medium	Low	High

Chapter 5

Conclusions

This work focused on discovering dependency information of software packages with a comprehensive approach. Dependencies of a software package are those software packages or libraries that are required to build, install, or execute the package but are not originally distributed with it. Obtaining the dependency information is crucial for successful software reuse from both technique and legal perspective. The motivation of our work is to automate the dependency resolution process with various techniques. Dependencies retrieved with different techniques may not only be merged to produce the most comprehensive result, but also used to cross-validate with each other to detect inaccurate output.

We presented four dependency resolution techniques: the source code technique, the build technique, the binary technique, and the spec technique. The source code technique retrieves dependency by analyzing the C/C++ source code of a project, discovering the required include files (i.e. a file declared with an #include directive), and matching those files against an artifact candidate repository (where artifact-library relations are stored) to locate dependent packages. The source code technique features two modes: a full mode and a clear mode. In the clear mode, only a project that uniquely match a include file is retrieved as a dependency. In the full mode, multiple projects that match a include file are also reported as an abstract dependency. The build technique parses build scripts (CMakeLists.txt or configure.ac/configure.in) and discover dependencies by processing library/program testing macros found in the build scripts. The binary technique discovers dynamically linked libraries by applying readelf utility to binary files with ELF file format. The spec technique retrieves dependencies of a Debian source project by extracting dependency information recorded in Debian source spec. The spec technique also retrieves dependencies of a Debian

binary package by extracting dependency information recorded in its control file as well as the source spec of the source project that the package is compiled from.

Among the presented techniques, the binary and spec techniques are designed based upon prior research; the source code and build techniques, designed by us, are the first studies to retrieve inter-dependencies from C/C++ source code and software build scripts, respectively. All four dependency resolution techniques are realized by a prototype tool `DEx`, which is developed in the Perl programming language. An evaluation shows `DEx` correctly implemented the design of the dependency resolution techniques.

A comprehensive evaluation of the four techniques conducted on 2313 Debian projects shows that dependencies retrieved with different techniques are not always the same. The techniques, ranked by the number of retrieved dependencies, are the spec, the binary, the build, and the source code technique. Performance of each technique, in regards of the quantity of retrieved dependencies, varies significantly from project to project, therefore using any technique alone does not guarantee the most comprehensive list of dependencies is constantly retrieved. Although the source code technique reports more abstract dependencies in full mode, our study shows that in most cases (86.378%), performing in clear mode and full mode have the same result. When using the dependencies retrieved with the sepc technique as the standard, we found that on average, most of dependencies retrieved with the source code technique can be verified by spec, while only 45.6% and 66.24% of dependencies found by build and binary technique can be confirmed by Debian spec. A further evaluation shows dependencies retrieved with different techniques usually have less than 50% similarity.

Our analysis shows that different results produced by dependency resolution techniques do not necessarily imply inaccuracy of the techniques. Each technique may discover some genuine dependencies that cannot be retrieved by another technique. The build technique is more suitable for finding build time and installation time dependencies; the binary technique is best in discovering dynamically linked libraries; the source code technique should discover static dependencies but the performance is heavily affected by the quality of artifact candidate repository; the spec technique usually contains the most comprehensive result. However, the dependencies retrieved with each technique may not be 100% complete or 100% accurate, therefore, using multiple techniques together not only discovers the most comprehensive result but also helps identify inaccurate findings.

5.0.3 Contributions

Following contributions have been made with our research:

- Four techniques to solve dependency resolution problems in software reuse.
- `DEx`, a prototype tool that allows users to retrieve dependencies with the four techniques.
- Design and creation of an artifact candidate repository which can be used for dependency resolution or the research of software evolution in Debian.
- A comprehensive review and evaluation of different dependency resolution techniques. Features, advantages, and limitations of each technique are described, compared, and analyzed.

5.0.4 Future Work

Future research work involves improving the dependency resolution techniques presented in this thesis. We pointed out in chapter 4 that the low coverage rate problem suffered by the source code technique can be approved by indexing more projects in the artifact candidate repository. The source code technique may also be improved by adding the support of programming languages such as Java, Perl, etc. The build technique can be improved by supporting more build systems. It should also be updated when new library testing macros are supported in CMake and Autoconf. In addition, a graphic user interface of `DEx` may also be implemented for better user interactions.

Bibliography

[1] B. Adams. Co-evolution of source code and the build system. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 461 –464, 2009.

[2] R.D. Banker, R.J. Kauffman, and D. Zweig. Repository evaluation of software reuse. *Software Engineering, IEEE Transactions on*, 19(4):379 –389, April 1993.

[3] Stew Benedict and Jeff Licquia. Dependency checker tool - overview and discussion.

[4] A. Capiluppi, P. Lago, and M. Morisio. Characteristics of open source software projects. In *7th European Conference on Software Maintenance and Reengineering, CSMR 2003, March 26, 2003 - March 28, 2003*, pages 317–330, Benevento, Italy, 2003. IEEE Computer Society.

[5] L.F. Capretz, M.A.M. Capretz, and Dahai Li. Component-based software development. volume 3, pages 1834 –1837 vol.3, 2001.

[6] William G. Cochran. *Sampling Techniques, 3rd Edition*. John Wiley, 1977.

[7] Julius Davies, Daniel M. Germn, Michael W. Godfrey, and Abram Hindle. Software bertillonage: finding the provenance of an entity. In Arie van Deursen, Tao Xie, and Thomas Zimmermann, editors, *MSR*, pages 183–192. IEEE, 2011.

[8] Alessandro P.S. De Moura, Ying-Cheng Lai, and Adilson E. Motter. Signatures of small-world and scale-free properties in large computer programs. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 68(1 2):171021 – 171024, 2003. Information flow;.

[9] debian.org. Debian.

[10] Arnoud Engelfriet. Choosing an open source license. *IEEE Software*, 27(1):48–49, 2010. Compilation and indexing terms, Copyright 2009 Elsevier Inc.; M1: Compendex.

[11] Warren W. Esty and Jeff Banfield. The box-percentile plot. *Journal of Statistical Software*, 8(17):1–14, 10 2003.

[12] Free Software Foundation. Gnu autoconf manual - generic program and file checks, 2010.

[13] William Frakes and Carol Terry. Software reuse: metrics and models. *ACM Comput. Surv.*, 28(2):415–435, 1996.

[14] freebsd.org. Gpl advantages and disadvantages.

[15] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. *Software Evolution, 2003.Proceedings.Sixth International Workshop on Principles of*, pages 13–23, 2003.

[16] G. C. Gannod and B. D. Gannod. An investigation into the connectivity properties of source-header dependency graphs. In *8th Working Conference on Reverse Engineering (WCRE 2001), October 2, 2001 - October 5*, pages 115–124, Stuttgart, Germany, 2001 2001. Dept. of Computer Science and Eng., Arizona State University, Box 875406, Tempe, AZ 85287-5406, United States, Institute of Electrical and Electronics Engineers Computer Society. Compilation and indexing terms, Copyright 2009 Elsevier Inc.; M1: Compendex; T3: Reverse Engineering - Working Conference Proceedings; undefined.

[17] Daniel M. German, Jesus M. Gonzalez-Barahona, and Gregorio Robles. A model to understand the building and running inter-dependencies of software. pages 140 – 149, Vancouver, BC, Canada, 2007. Inter-dependencies;New applications;Software applications;.

[18] Jesus Gonzalez-Barahona. Macro-level software evolution: A case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009. ID: EVII; ID: Compendex (Ei Village 2).

[19] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. Code reuse in open source software. *Management Science*, 54(1):180 – 193, 2008. Code reuse;Knowledge reuse;Open source software;.

[20] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952 – 970, 2006. Example recommendation;Heuristic search;Strathcona;Structural context matching;.

[21] Oliver Hummel, Werner Janjic, and Colin Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45 – 52, 2008. Component-based development;Eclipse plug-in;Engines;Java;Libraries;Open source software;Presses;Programming;Reuse recommendation;Software;Software reuse;Software search engines;Test-driven search;.

[22] Christian Schwarz Ian Jackson. *Debian Policy Manual*. Debian.org, 2010.

[23] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution*, 19(2):77 – 131, 2007. Mining software repositories (MSR);Multi version analysis;Software evolution;.

[24] Nathan Labelle and Eugene Wallingford. Inter-package dependency networks in open-source software, 2004.

[25] Matt Lee. What is free software and why is it so important for society?

[26] Josh Lerner and Jean Tirole. Some simple economics of open source. *Journal of Industrial Economics*, 50(2):197 – 234, 2002.

[27] Bixin Li. Managing dependencies in component-based systems based on matrix model. In *Proc. Of Net.Object.Days 2003*, pages 22–25, 2003.

[28] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.

[29] W.C. Lim. Effects of reuse on quality, productivity, and economics. *Software, IEEE*, 11(5):23 –30, sep. 1994.

[30] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: Mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300 – 336, 2009. Author-topic probabilistic modeling;Code retrieval;Code search;Mining software;Program understanding;Software analysis;.

[31] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 309–312, New York, NY, USA, 2010. ACM.

[32] Mircea F. Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, 2009.

[33] Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 199–208, Washington, DC, USA, 2006. IEEE Computer Society.

[34] David Mandelin, Lin Xu, Rastislav Bodik, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. *ACM SIGPLAN Notices*, 40(6):48 – 61, 2005. Program synthesis;Programmer;Queries;Reuse;.

[35] K. Martin and B. Hoffman. An open source approach to developing software in a small organization. *Software, IEEE*, 24(1):46 –53, 2007.

[36] Frank Mccarey, Mel O. Cinneide, and Nicholas Kushmerick. Rascal: A recommender agent for agile reuse. volume 24, pages 253 – 276, 2005. Agile processes;Agile reuse;Collaborative filtering;Content based filtering;Recommender agent;Software reuse;.

[37] David M. Nichols and Michael B. Twidale. The usability of open source software, 2003.

[38] Opensource.org. Approved open source licenses.

[39] J. Ossher, S. Bajracharya, and C. Lopes. Automated dependency resolution for open source software. pages 130 –140, may. 2010.

[40] J.W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *Software Engineering, IEEE Transactions on*, 30(4):246 – 256, apr. 2004.

[41] Lawrence Rosen. *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[42] R.W. Selby. Enabling reuse-based software development of large-scale systems. *Software Engineering, IEEE Transactions on*, 31(6):495 – 510, jun. 2005.

[43] SourceForge.net. About sourceforge.net.

[44] Sebastian Spaeth, Matthias Stuermer, Stefan Haefliger, and Georg von Krogh. Sampling in open source software development: The case for using the debian gnu/linux distribution. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 166a –166a, 2007.

[45] D. Spinellis and C. Szyperski. How is open source affecting software development? *Software, IEEE*, 21(1):28 – 33, jan. 2004.

[46] Judith A. Stafford and Alexander L. Wolf. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431 – 451, 2001. Anomaly checking;Architecture description languages (ADL);Information capture;Program testing;.

[47] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming (ACM Press)*. Addison-Wesley Professional, December 1997.

[48] Jean Tessier. Dependency finder.

[49] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213, New York, NY, USA, 2007. ACM.

[50] Timo Tuunanen, Jussi Koskinen, and Tommi Karkkainen. Automated software license analysis. *Automated Software Engineering*, 16(3-4):455–490, 2009. Compilation and indexing terms, Copyright 2009 Elsevier Inc.

[51] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *Software Engineering, IEEE Transactions on*, 30; 30(9):574–586, 2004. ID: 1.