UNIVERSITY OF CALIFORNIA,
IRVINE

Automatic Builds of Large Software Repositories

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Informatics

by

Rohan Achar

Thesis Committee:
Professor Cristina Videira Lopes, Chair
Professor André van der Hoek
Associate Professor Ian G. Harris

2015

TABLE OF CONTENTS

Page

LIST OF FIGURES **iii**

LIST OF TABLES **iv**

ACKNOWLEDGMENTS **v**

ABSTRACT OF THE THESIS **vi**

1 Introduction **1**
1.1 Motivations and Objectives 1
1.2 Related Work 3

2 Repository and Tools used **6**
2.1 A Repository of Open Source Software - Sourcerer 6
2.2 Using *Apache Ant* to Compile Software Projects 9
2.3 Other Tools and Repositories 10

3 Building the Projects **12**

4 Improving Build Success **16**
4.1 Resolving Dependencies using Sourcerer and Maven 16
4.2 Resolving Encoding Errors 20
4.3 Searching for Missing Dependencies 21
4.4 A new strategy for Solving Dependencies 26

5 Discussion and Conclusion **29**
5.1 Final Analysis of Results 29
5.2 Generality of the Process 31
5.3 Conclusion 33

Bibliography **35**

LIST OF FIGURES

Page

2.1 Relation between the tables *Entities*, *Files*, *Jars* and *Projects* in the Sourcerer database. 8

3.1 The steps taken to get maximum number of successfully compiling projects . 12

5.1 The success and failure of each step that we took. 29

LIST OF TABLES

Page

3.1 Build Errors - First Try 15

4.1 Build Errors - Second Try 19
4.2 Build Errors - Third Try 24
4.3 Build Errors - Fourth Try 28

ACKNOWLEDGMENTS

I would like to express my deepest gratitude and thank my advisor Professor Cristina Videira Lopes for the knowledge, guidance, and opportunity she has given me throughout this project. Her timely advise helped me overcome the various road blocks encountered on the way.

I would also like to deeply thank Pedro Martins for all the help he has given me in the writing of this thesis.

This work was partially supported by grant No. 1018374 from the National Science Foundation and by a grant from the DARPA MUSE program.

I would also like to thank my parents and my friends. Their support has made me the person I am today.

ABSTRACT OF THE THESIS

Automatic Builds of Large Software Repositories

By

Rohan Achar

Master of Science in Informatics

University of California, Irvine, 2015

Professor Cristina Videira Lopes, Chair

A large number of open source projects are hosted on the Internet by popular repository sites like GitHub, SourceForge, BitBucket, etc. These repositories are becoming more popular and growing in size. There are many research projects that mine these software repositories for valuable information.

Compiling the projects found in these repositories can help filter out the good, and usable projects. It gives us a guarantee that the source code is syntactically correct, and that all the dependencies of the project are either self contained or accessible on the Internet. Projects can be maintained and organized in different ways depending on the developer culture and practice. Unfortunately, very often repositories fail to capture the environmental assumptions made by the developers such as build tools, versions, presence of external dependencies, etc. This heterogeneous nature of the projects makes the successful compilation of large numbers of projects a challenging task as one solution cannot be applied to all. It is impractical to manually correct the compilation of every project. We designed several heuristics to maximize the number of projects compiling successfully in a repository.

Sourcerer is an infrastructure for large-scale collection and analysis of open-source code. It crawls open-source Java projects from various sources on the Internet and builds an aggregated repository, and database. We used the information found in the database to

automatically compile more than 55,000 *Java* projects in the Sourcerer repository. We propose several general, language independent heuristics to tackle the most common errors. Using these heuristics, we were capable of building 33.18% of the projects in the repository, successfully building more than 18,000 *Java* projects.

Chapter 1

Introduction

1.1 Motivations and Objectives

The popularity of open-source repository sites like GitHub, SourceForge and BitBucket provide access to huge amounts of source code, but the heterogeneous, unconstrained nature of these repositories creates difficulties regarding the analysis and usage of their contents. Often the code in these repositories is dated, the solution is only partially written, and the quality of the code is questionable. Some repositories were abandoned in the middle of their life-cycle, or sometimes were left unfinished. Some contain good quality code written by long-time professionals, other times they were created by an amateur with not much experience. Both for a programmer that wants to reuse code, or for a researcher that wants to understand code quality and evolution, taking the gigabytes of existing heterogeneous code and obtaining information from it is a time-consuming and complex task.

There are various successful projects that focus on the analysis of huge corpus of source code, such as [1,2,8,14]. These provide either techniques or pre-processed repositories such as databases (or a combination of both), from where it is easier to obtain information regarding

code quality, existing patterns in the source or complexity metrics, just to name a few. These greatly simplify the task of analyzing and reusing source code in its heterogeneous, disperse form as it is available (and constantly growing) online.

However, the existing projects to analyze big corpus of source code often rely on static code analysis, and discard compilation as an important step for obtaining information. Compilation, despite being a critical step in software production, also provides valuable information about a software project. It guarantees everything in that project - not only a few classes or modules - is syntactically correct; it gives an indication of the age of the code, by knowing with which compiler/dependencies versions it works; it guarantees that all the dependencies of that project are either self contained or accessible on the internet. This information is very hard to guarantee with a static analysis, but through successful compilation these crucial characteristics are ensured.

Another important advantage of successfully compiling projects is that we generate a new repository of artifacts composed by the results of the compilation. Being it *Java bytecode* in *Java*, *Common Intermediate Language* for *.NET* applications or binary executables produced by *gcc*, successfully obtaining corpus of these final formats opens a new window of analysis, explored by projects such as [4,7,9,17,24]. The analysis done in this paper can also augment several projects like the one by Martinez et al. [16] for automating program repair, Williams et al. [22] to improve bug detection or Thummalapenta et al. [20] for improving the quality of source code searches.

Compiling projects found in a large repository is therefore a good start to filter good, usable projects and further analyze them, but this task is hard because it requires extremely detailed information about the project. This includes, for example, the exact list of dependencies and their exact location, the precise order with which the different components should be compiled and their interdependency, or the specific chain of processes that should be used. Projects are usually complemented with information that drives the compilation, such as

ant files for *Java*, a *Makefile* for *C/C++*, or *Microsoft Build Engine* to be used with *.NET* applications, but due to the heterogeneous nature of online repositories, the presence of this information is far from being guaranteed.

In this work we systematically develop a series of steps to create a methodology to compile projects in a source code repository. We used Sourcerer, a well-known database of open source projects, composed by more than 400 gigabytes of pre-processed information on *Java* projects.

Through our methodology, we were able to increase the initial 11.66% projects compiled to 33.18% projects compiled, and we show how through this process we obtain valuable information regarding the repository, for example, the exact number and list of dependencies, and the types of compile time errors encountered and their correlation with each other. We developed and worked on *Java* projects, but our techniques can be generalized to other scenarios.

1.2 Related Work

Various projects provide a combination of methodologies and repositories to analyze big corpus of open source software. An example is *Boa* [8], where a *DSL* is introduced with the specific purpose of providing scientists and researchers with a tool for mining information on big repositories. Other approaches with the same objective include [1, 2, 14], but these never touch the analysis from the perspective of compiling the projects in the repositories.

One lesson from this work is that dependencies play a big part in the success or failure of building open source projects. It is therefore normal that various other works focus on dependencies retrieval and have employed various techniques to mine open source repositories and obtain project dependency information.

Lungu et al. worked on extracting inter project dependencies with a specific focus on software ecosystems [15]. Zhang, Hanyu details four techniques for dependency resolution in C/C++ projects [25], through a) analyzing the source code of the project; b) parsing build scripts; c) reading binary files with the *ELF* file format and discovering dynamically linked libraries; and c) using a specification based technique that looks at the source specifications of projects recorded in *Debian* repositories. It is possible these techniques can be applied to *Java* projects, and can further help improve our success rate.

After the step presented in Section 4.3 we started to have the problem of having multiple sources of dependencies for the same project. Operating systems like *Debian* have package repositories that are interconnected between them, and various works were developed around these. Gonzalez-Barahona et al. studied the dependency evolution of such large software compilations [10]. Bavota et al. studied the evolution of inter dependencies in smaller software ecosystems like the *Apache* projects [6], while Seidl el al. model and analyze the evolution of platform software ecosystems like *Android* and *Eclipse* [19]. Sajnani et al. conducted an analysis of 2406 components in the *Maven* repository that were used across 55,191 open source *Java* projects [18].

API usage analysis, like [23], can be used to design alternative techniques for finding the right dependencies for projects. Lãmmel et al. extracted API usage and package footprint from large corpus of built java projects using AST based approaches. Here, missing packages required to build the java projects were obtained by searching the web for the jars using the unresolved package names as the search queries. They were able to improve their build success rate by 15% by using the resolution method in [13].

Barkmann et al. [5] conducted a quantitative evaluation of software quality metrics in open-source projects. While preparing the projects for analysis they encountered the problem of missing packages. They resolved these dependency errors by using both manual and automated techniques. Similar to our results, they also had problems both in syntax errors

and in ambiguous types that they could not resolved automatically, pointing out that there is a point from which it becomes harder to systematically tackle these problems.

Chapter 2

Repository and Tools used

To understand the difficulties of attempting to compile a huge corpus of open source software, we need a repository that provides us with easy and fast access to a huge set of software projects and its characteristics, and tools to compile the projects. We used Sourcerer as a repository and *Apache Ant* as a build-driving tool.

2.1 A Repository of Open Source Software - Sourcerer

Sourcerer [2] [3] is an infrastructure for large-scale collection and analysis of open source code, and contains a collection of large amounts of source code from open source repositories.

Sourcerer contains $74,341$ Java projects captured directly from open repositories on the internet: Google Code, Apache, SourceForge. Of these, $55,209$ *Java* projects were not empty and had a code base, and thus provided us with a perfect setting to explore the compilation of these artifacts, as well as analyze common compilation errors on these projects and hint at their resolution. For the rest of the paper we are ignoring the $19,132$ projects on Sourcerer that are empty.

A parser/feature extractor was built into Sourcerer to create a *MySQL* database in which the relational representation of pre-processed source code is stored in the form of an efficient platform. The schema for this database contains tables for entities such as packages, classes, methods or fields, together with their relations. Care was taken when developing this database to ensure that data access and retrieval is optimized, so throughout our analysis we could rapidly obtain information regarding the characteristics of the projects that are necessary when attempting their compilation, such as interdependencies among projects and dependencies with external libraries.

The tables in the Sourcerer database that have information required for the compilation of the *Java* projects are:

- *Entities* - This table contains all components found within the database, decomposing a *Java* project at every granularity, from local variables to methods or packages.
- *Files* - All files obtained by the crawler for a project are detailed in this table. Some dependencies of a project can be obtained from this table by looking for all jar files used by the project.
- *Jars* - Maintains a list of jar files, associating the repository path or the jar file with its hash. It helps us locate the physical resource from the information in Table *Files*.
- *Projects* - Table that holds information about all projects and libraries discovered by Sourcerer, with attributes such as the project name, the project path or description of the project, that help us generate a build file for the project.

The relations between these tables, as presented in the Sourcerer database, are shown in Figure 2.1. This figure shows tables with their attributes, relations, and shared identifiers.

General information required by the scripts to find the project source are contained in the table *Projects* and are marked in blue. This table includes all projects discovered by Sourcerer:

projects crawled from the Internet, *Jar* packages, *Maven* artifacts referenced by various projects, system projects, and internal *Java* libraries.

We use *project_type* to differentiate between different types of projects. The attributes marked in green help us obtain contextual information required to reference resources on the *Maven* artifact repository.

Attributes and tables marked in red identify *Jar* packages stored within the Sourcerer repository, and extract relevant information such as their path. The attribute *hash* represents relations between *Jar* files and projects.

For each project, we can find its exports by looking at all the *Files* that belong to the project and have the *file_type JAR*. These files can either be local *Jar* file or be pointing to external repositories. This can be differentiated by looking at *project_type* on the *Project* table.

The *Entity* table can represent any kind of entities, such as methods, classes or interfaces, and have associated to them a fully qualified name (*FQN*).

As we will see in the next sections, all this information will be important to resolve build failures. The information represented here, and the Sourcerer repository we used, is from the 2011 version.

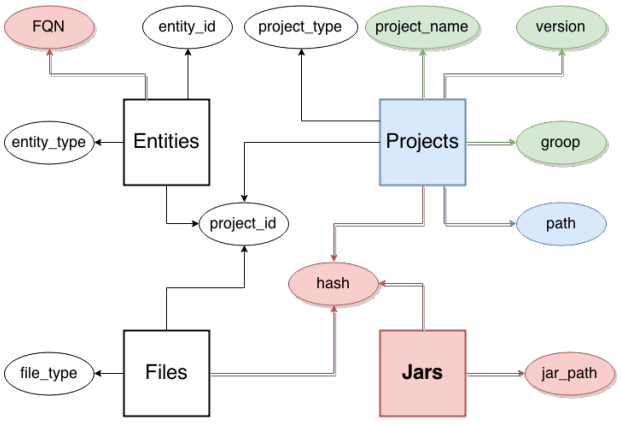

Figure 2.1: Relation between the tables *Entities*, *Files*, *Jars* and *Projects* in the Sourcerer database.

2.2 Using *Apache Ant* to Compile Software Projects

Apache Ant [11] is a compilation framework provided by the Apache Software Foundation. It is a tool for automating the software build process of software projects. It uses a standard *XML* file named *build.xml* to describe all the information necessary to compile a project, from the build process to its dependencies. It is an open source software, and can be freely used under the Apache License.

Ant is mainly used to build *Java* artifacts, and has built-in tasks to help in compiling, assembling, testing and running *Java* applications. This tool helped us resolve problems in sub-tasks when building our projects, where otherwise we would have to write custom scrips or perform actions manually (see Section 2.3). However, this tool can be used to build projects in other programming languages such as *C/C++*, and its mechanics can be used to drive any process that can be described based on a series of tasks and targets, which effectively describes any compilation process. This helps generalizing our approach to other programming languages.

Even though *Ant* has the potential to provide instructions for building software artifacts without restrictions on the programming language used, any other strategy that helps build software projects, and helps in organizing, compiling and linking source files with prioritization of these sub-tasks can be used on our approach. An example includes the famous *Make Utility* and its *Makefile* files. This is important because other repositories of software might exist where a big set of projects have incorporated their own build system, and with the generic approach we describe these build systems can be directly used, without the need to mine all the repository and create build instructions for all the projects from scratch.

In this project we used *Apache Ant* version 1.8.2, compiled on December 3 2011.

2.3 Other Tools and Repositories

Dependencies on external libraries play a big part when building most medium to large size software projects. We expected this to be a problem, and expected to solve it by querying Sourcerer for information that was present both as local *Jar* files or on external repositories of libraries.

In our context, we used the dependency management *Maven* [21], as an external repository that holds individual artifacts such as software libraries and modules for various *Java* projects. With *Maven*, a project that needs the (fictional) *Mondego* library only has to declare this library's coordinates, and both *Mondego* and its transitive dependencies will be automatically downloaded and stored in local repositories to be used when building a project.

Maven has a central repository[1], which of July 27 2011 had 286 gigabytes[2] of artifacts. By default this central repository is used, but *Maven* can be configured for use with private repositories (although this is not important in our scenario). Additionally, a search engine exists for *Maven*[3] that helps retrieve dependencies. All these features make *Maven* a good candidate for the use in our attempt to build *Java* projects.

As similar to *Ant*, the usage of *Maven* as an external repository to search for missing dependencies can be generalized to other scenarios, which typically contain repositories of existing libraries. If attempting to replicate our experience with other programming languages other online repositories can usually be found, such as *CPAN*[4] for *Perl*, *Hackage*[5] for *Haskell*, *PyPI*[6] for *Python* or the *Debian* package repositories[7] for (C/C++). These repositories are

[1] `http://central.sonatype.org`
[2] `http://blog.sonatype.com/2011/07/central-grows-up-see-the-history`
[3] `https://search.maven.org`
[4] `http://www.cpan.org`
[5] `https://hackage.haskell.org`
[6] `https://pypi.python.org`
[7] `https://www.debian.org/distrib/packages`

usually integrated with search engines.

Dependencies from *Maven* were referenced using *Apache Ivy*[8], a dependency manager specifically designed for *Ant*. In *Ivy* an external *XML* file, named *ivy.xml*, references projects both in privately and publicly available repositories (in our case, in *Maven*). *Ivy* resolves the dependencies, downloads the required resources and makes them available for use by *Ant*. This tool simplified our approach and aids in building *Java* projects. Similar tools exist for other languages (all the repositories previously mentioned have specific tools for automatically searching and downloading dependencies and transitive dependencies), but there might be scenarios were we need to write tools that perform the same actions. This should not, however, limit the replication of our experience for other programming languages/repositories

Additionally, we used *Python* scripts to automate processes when necessary, such as running all the build instructions for all the projects, and the standard *Java* compiler *javac* to build the projects. The versions of the software we used are as follows: *Apache Ivy* version 2.4.0, *Python* version 2.7.3 and *javac* version 1.8.0_05. As a *Java* run environment, we used the Java SE Runtime Environment, build $1.8.0\_05 - b13$.)

[8] `http://ant.apache.org/ivy`

Chapter 3

Building the Projects

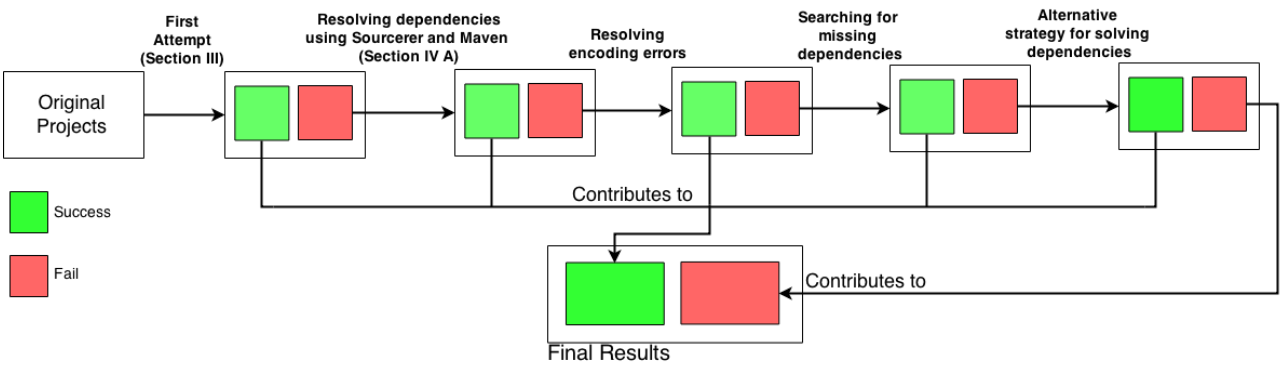

Figure 3.1: The steps taken to get maximum number of successfully compiling projects

In order to successfully compile as many projects as possible, we followed a series of steps. Each step was guided by the analysis of the results received in the previous step. Figure 3.1 shows the road map of these steps taken to improve the build percentages. The projects that failed to compile at each successive step were used for the subsequent step.

Our first attempt was to build the projects in Sourcerer by creating an *Ant* build file for each one. We expected a low success rate, but as we will see, the analysis of the main causes of failure provided by this first step were important in driving our experience and understanding which problems should be tackled first.

In each individual *Ant* file we inserted the description and the name of the project, which are optional fields but are potentially important for future analysis (and represent good programming practices). We also had the location of the project as a *path* in the Sourcerer repository and, since we are working with *Java* projects, we instructed *Ant* to use the traditional *Java* compiler *javac*.

With the help of *Python* scripts, we proceeded to try building all the projects, and captured the successful builds and the list of errors as provided by the output of *javac*.

We successfully built 11.66% of our total 55,209 projects, which totals to 6,439 projects built. This result shows a good number of projects had no external dependencies and have enough structural integrity to build.

The top 10 most common errors responsible for unsuccessful builds are presented in Table 3.1. These errors represent the following problems in the source code:

- *Cannot find symbol* - These errors arise when a symbol is used but is neither declared locally nor imported from external packages.
- *Package not found* - When the compiler is not able to find packages imported by the program, the build fails with this error type.
- *Unmappable character* - Text in programs can be written in any language using different encoding formats. The *Javac* compiler does not attempt to detect the encodings of files and defaults to *UTF-8*. The compilation fails with this error when it finds a character it cannot decode[1].
- *Duplicate class* - When more than one class with the same fully qualified name exists in the project, compilation fails with this error.

[1]As of May 13 2015, a bug exists on *javac* (`http://bugs.java.com/view_bug.do?bug_id=4508058`) where the Byte Order Mark (BOM) at the start of a text stream is not recognized in files, and the default *UTF-8* is always used unless specified otherwise. This bug, with the code *JDK*-4508058, was detected on versions 1.4.0 and 1.4.2_05 but was never solved. It is possible that this bug contributed for these type errors.

- *Override error* - These errors appear when the decorator *@Override* is used on a method, but the method does have a matching method in the super class.

- *Expected symbol not found* - Compilation fails with this error when a class, interface, or enum was expected and not found.

- *Static import only from classes and interfaces* - Static imports are those that import only static members from the specified class. If the specific symbol is not a compatible type, compilation fails and reports this error.

- *Incompatible Java version* - As newer versions of *Java* are released, some features are deprecated. Projects using such constructs can only be built with deprecated flags. Compiling without these flags fails with this error.

- *Illegal use* - This error is reported for some syntax errors when the offending code constructs are either at the wrong place or incorrectly written.

- *Incompatible types* - Compilation fails with this error when the type of the symbol used (in a method, or expression) does not match the expected type declared.

The main causes of build failure for the projects in Sourcerer were related to missing information, either from missing symbol declarations (*cannot find symbol*) or missing packages (*package not found*). This showed *Java* projects have a great dependency in external information, and our first approach should head towards solving this.

The third most common problem, that affected $\approx 25\%$ of the projects was *unmappable character*. This represents problems related to file encoding, and its position in the table can hint at problems in the online repositories where the projects are stored, users that upload projects where different files have different encodings or problems in Sourcerer itself.

The percentage of these problems do not sum up to 100% because frequently *javac* returned

Table 3.1: Build Errors - First Try

Error Type	**# projects**	**% projects**
Cannot find symbol	31018	56.184
Package not found	30871	55.918
Unmappable character	13866	25.116
Duplicate class	4211	7.628
Override error	3227	5.845
Expected symbol not found	2395	4.338
Static import only from classes and interfaces	2172	3.934
Incompatible *Java* code	1397	2.530
Illegal use	1203	2.179
Incompatible types	1125	2.038

more than one error type for a project. The two more common errors, both affecting ≈56% projects was found to have an almost complete overlap.

It is interesting to see that other types of errors such as problems related to class hierarchies (*Override error* and *Duplicated classes*) caused problems when building projects, and problems related to the encoding of source files (*Unmappable character*) also influence the compilation errors.

This step help us identify the main problems of building the *Java* projects: missing dependencies, and encoding errors. We expected the presence of the missing dependency problem, and this was validated in the two more common types of errors: *Package not found*, and *Cannot find symbol*. It is important to see these concerns confirmed, and these are the main problems we tackled in the next steps.

Chapter 4

Improving Build Success

A large cause of build failures in the previous step was explained by missing dependencies in projects, so we prioritize this problem in the next steps. With each step we took, we focused on only the projects that did not compile, ignoring the projects that were successfully built during the previous steps.

4.1 Resolving Dependencies using Sourcerer and Maven

We first approached this problem by querying the Sourcerer database for each individual project that failed to build in the previous step and collected the respective dependency information. For each of these package dependencies we tried to locate local *Jar* files and local *Maven* information.

Finding *Jar* files was a question of querying the Sourcerer database, and the organized nature of this repository greatly facilitated this task. We found all the entities in the database related to each individual project (in Sourcerer an entity can represent a myriad variety of artifacts, such as methods, classes or interfaces) and from the artifacts that were associated with each

specific project we extracted the ones that were registered as *Jar* files. We then obtained their local paths and added them to the project's local *Ant* file by updating the *build.xml* file with compile rules that reference them.

We followed a similar approach for the dependencies presented in *Maven*. For each individual project we search for entities that are *Maven* files and added this information to each individual project in the form of an *Ivy* file. Using this, *Ivy* would attempt to download the dependencies before *Ant* tried to build the projects.

This process is resumed as follows:

1. Project *GoodCode* build failed:
    (a) Search Sourcerer for *Jar* files that are associated with *GoodCode*, and add them to *build.xml*;
    (b) Search Sourcerer for *Maven* files that are associated with *GoodCode*, and add them to *ivy.xml*;
2. Try to build *GoodCode*, capture success or errors on failure.

This was automated with the help of scripting. It was executed for all the projects that failed to build, independent of the causes, but with the information from Table 3.1 we knew dependencies were a huge cause of failure.

With this step we build 687 new projects, to a total of $7,126$ Sourcerer projects successfully built. This represents 12.91% cumulative with the previous step, an increase of 1.25% from our previous step. From the new projects we were capable of building, there were a total of $248,366$ dependencies, with $195,885$ being dependencies from local *Jar* files, $14,105$ unique dependencies. $52,481$ dependencies were solved with *Maven*, with $3,104$ unique *Maven* dependencies being downloaded. On an average, the projects had 4.50 external dependencies

(an average of 3.55 local *Jar* files and 0.95 *Maven* dependencies). The project with the most number of dependencies was *Apache Geronimo* with 210 dependencies (80 local *Jar* files and 130 *Maven* dependencies downloaded).

These results by themselves provide interesting information about the *Java* projects in Sourcerer. We have the guarantee that the 7,126 projects that compile are syntactically correct, have structural and hierarchical integrity, they are written in a version of *Java* that works with modern compiler versions, and their dependencies are readily available. We also obtain values regarding, for example, the average number of dependencies per project and we can obtain statistics correlating them to the size of the projects (number of lines of code, methods or classes for example).

After this attempt we had 48,083 projects failing to build, whose top 10 errors can be seen in Table 4.1, where we also present the differentials on the number of projects affected by a particular error, by comparing them to Table 3.1.

This step also introduced 1 new error type to our top 10:

- *Unresolved dependencies* - *Apache Ivy* tries to resolve all *Maven* dependencies before the compilation starts. If a declared dependency fails to download then compilation is stopped and this error is reported.

Possible causes for this error includes incorrectly linked dependency due to incorrect data in the Sourcerer database, or the dependency was hosted in private repository to which we do not have access to. This error replaced the error type *illegal use*, which was no longer one of the top 10 reasons why projects fail to build.

We can also see that despite the top 2 error types having switched place, with *package not found* being now the main reason why builds fail, the percentage of projects affected by the top 2 error type is not only very high (around half the projects are affected by at least one

Table 4.1: Build Errors - Second Try

Error Type	**# projects**		**% projects**
Package not found	27244	(↓3627)	49.348
Cannot find symbol	27231	(↓3787)	49.324
Unmappable character	12578	(↓1288)	22.783
Unresolved dependencies	4207	(↑4207)	7.620
Duplicate class	3696	(↓515)	6.695
Override error	2524	(↓703)	4.572
Expected symbol not found	2119	(↓276)	3.838
Static import only from classes and interfaces	1842	(↓330)	3.336
Incompatible *Java* code	1284	(↓113)	2.326
Incompatible types	1109	(↓16)	2.009

of these two) but also very similar. This hints, like in our previous attempt, that the two errors types are overlapping for a large number of projects.

We found that there was an almost perfect overlap, with almost all the projects affected by *package not found* also being affected by *cannot find symbol*, and the cause of both is the same: incomplete dependencies. These two error types have a huge correlation.

Between the projects whose build failed, there are also a large number, around 6.7%, being affected by *duplicate class*, showing another smaller correlation. Duplicate class can arise when the source contains backup copies of source files which are expected to be excluded during compilation.

Projects whose errors were generated by *unmappable character* or *unresolved dependencies* are isolated and have practically no overlapping with projects whose problems have their cause in the other three previous errors (in the case of *unresolved dependencies*, no correlation at all). The dependencies unresolved against *Maven* create an immediate compilation error, and *Ant* refuses to continue building if the *Ivy* resolution fails. This explains the absence of

overlapping between *Unresolved dependencies* and the rest of the errors. Nevertheless, this still affects ≈22% and ≈8% projects, respectively. Interestingly, the top 6 error types, which are presented in this figure, account for around 90% of all projects, cumulatively.

We can see an evolution from the previous step, where we were able to build 687 more projects. Our approach to find local *Jar* files and *Maven* dependencies increases our success rate by 687. However, dependency problems still amount to 27,244 projects failing to build, so we continue to tackle this problem as our main focus.

4.2 Resolving Encoding Errors

After our second attempt, problems causing building errors branch into two main causes: encoding errors and problems caused by packages not found, with the *encoding errors* standing for the third largest cause of projects failure. There were a total of 12,578 projects with encoding mismatches. The *javac* compiler defaults to *UTF-8* as the encoding, and if the encoding is different it fails and throws an *unmappable character* error. We tried to fix this problem by auto detecting the encoding of the offending file(s) using the *Chardet*[1] package in *Python*; and configuring *javac* to expect this new encoding. The version of *Chardet* used was 2.3.0.

Chardet analyzes the file to give the best possible match of encoding format, and can detect more than 30 encoding formats. By changing the default encoding in *javac* and recompiling, we were able to successfully compile $1,614$ of the $48,083$ projects that failed at the previous step, bringing the total to $8,740$ or 15.83% of the projects in Sourcerer. This fix specifically targets the 12,578 projects that were failing due to character encoding issues.

[1] `https://pypi.python.org/pypi/chardet`

4.3 Searching for Missing Dependencies

The next approach used followed on our second attempt (Section 4.1) and our resolution of the encoding errors in the previous step (Section 4.2). In our second attempt we had tried to resolve missing dependencies by searching for *Jar* and *Maven* files with a relation to projects and increased the total percentage of projects successfully built to 12.91%, but there were still 27,244 projects where missing dependencies was the main cause of build failures. This might be caused by incomplete information in the projects as they are stored in the online repositories or errors in Sourcerer itself. We, however, did not rebuild the successful projects obtained by resolving encoding errors (Section 4.2).

Our next strategy was to find dependencies in Sourcerer, or try downloading them on *Maven*, that did not have a direct relation on the Sourcerer's database with the project we are trying to build. We instead try to take advantage of the size of the information in Sourcerer to match dependencies. To achieve this, we processed the building results of the projects that were failing for additional information.

We started by extracting the fully qualified name of packages that are dependencies of a project that we cannot build. This was done by processing the output resulting from *javac* in the cases where the compilation failed. An example of a fully qualified name is *"com.google.gwt.accounts.client"*. We then proceeded to search Sourcerer for *Jar* files and *Maven* information that relates to the qualified name of this package.

For *Jar* files, we search all the entities in Sourcerer that have the qualified name of the dependency, and from these we filtered the entities that were packages (remember, entities in Sourcerer can be methods, classes, *Jar* files, etc). The entities which are identified in Sourcerer as packages contain various items, and among them we find *Jar* files. We proceeded to add the paths of these *Jars* to the build instructions of the project that had the qualified package name as a dependency.

The intuition behind searching for packages in Sourcerer is that among these *Jar* files that are contained in the package we would encounter the information (namely, the implementation of the classes) needed to successfully resolve the missing dependency. This differs from our previous attempt, where we merely searched for entities which first: are *Jar* files; and second: have, in the Sourcerer database, a direct relation with the project we are searching dependencies for.

With the qualified names we applied another strategy to try to solve missing packages in project build failures. Returning to our example, we knew a project has a missing dependency with the fully qualified name *"com.google.gwt.accounts.client"*. We searched all the projects within Sourcerer that had this qualified name. Sometimes we found some, sometimes we did not, but for the latter cases we incrementally reduced the qualified name (for example, from *"com.google.gwt.accounts.client"* to *"com.google.gwt.accounts"* and continued searching for projects.

When we found in the Sourcerer universe a list of projects that contain the qualified name, we selected the one with the latest version and added them to the *Ivy* file of the original project, with the hope this project existed in *Maven* and the download of the project and its transitive dependencies would solve our problem.

This process is detailed as follows:

1. Project *GoodCode* build failed, and *javac* says one missing dependency is *"com.google.gwt.accounts.client"*;
2. Search Sourcerer for projects that have in their attributes information about the qualified name:
   (a) If one or more projects were found, jump to step 3);

(b) If no projects were found, reduce the qualified name to *"com.google.gwt.accounts"* and return to step 2);

3. From all the projects found, filter the unique ones with the latest version. For example, if the search returned:

   - project: *gwt-dev*
     version: 2.2.0
     qualified name:*"com.google.gwt"*
   - project: *gwt-user*
     version: 1.0.3
     qualified name:*"com.google.gwt"*
   - project: *gwt-user*
     version: 1.0.4
     qualified name:*"com.google.gwt"*

   use project *gwt-dev* *v*2.2.0 and *gwt-user* *v*1.0.4;

4. Add the projects *gwt-dev* *v*2.2.0 and *gwt-user* *v*1.0.4 to the *Ivy* file in the building instructions of *GoodCode*.

Hopefully, *Ivy* would locate and download *gwt-dev*, *gwt-user*, and their transitive dependencies and, in the process, the missing dependencies of *GoodCode* would be solved. Some times even after greatly reducing the qualified name we did not obtained any result. In this cases there was nothing we could do.

We used the project with the latest version so that we can get the most actual source code. This can be misleading (a project in version 1.0.0 can be more recent and a greater source of missing dependencies that a project on version 3.0, which can be very old), but we took this step because trying all the projects returned in step 2) would make our solution grow to

Table 4.2: Build Errors - Third Try

Error Type	**# projects**		**% projects**
Cannot find symbol	15262	(↓11969)	27.645
Package not found	9087	(↓18157)	16.460
Unresolved dependencies	8955	(↑4748)	16.220
Unmappable character	7921	(↓4657)	14.348
Duplicate class	3826	(↑130)	6.930
Incompatible types	2855	(↑1746)	5.171
Expected symbol not found	2238	(↑119)	4.054
Cannot be applied	2145	(↑1719)	3.885
Illegal access	1795	(↑1762)	3.251
Override error	1682	(↓842)	3.047

unmanageable complexity, and choosing the one with the largest version is generally better than picking one random, as in most cases it means there is a greatest effort in maintaining the project. Furthermore, as we will see, this strategy greatly improved our results.

We tried to apply the solution using *Maven* first and on the missing cases the solution using the *(*Jar) files, with which we were able to build 2,028 more projects. Afterwards we tried applying the the solution using the *Jar* files first and then the *Maven* projects to fill missing dependencies, and with this strategy we were able to build 6,128 new projects. This is a large improvement, making it by far the most successful among all our strategies. The results presented next were obtained by combining the successes of both strategies.

In total, after this step were were capable of building a total of 16,896 projects. This represents 30.60% of the Sourcerer repository. This was an increase of 14.77%, and we successfully built 8,156 more projects by mining the projects and programmatically searching for dependencies.

As similar to the previous sections we show the top 10 most common reasons for the failure of building projects, presented in Table 4.2. As is clear in this table, all the errors show a

significant decrease, with *Cannot find symbol* and *Package not found* being now present in the build failures of ≈28% and ≈16% projects, respectively. This was a huge decrease as these two errors appeared in ≈50% of the builds that failed.

This step also introduced two new error types to our top 10:

- *Cannot be applied* - This error is thrown when the type of the symbol used (in a method or expression) can not be inferred with the given type even though the fully qualified names are the same. Can also arise if the imported package is not the right version or type as the package expected.
- *Illegal access* - When the symbol accessed is either not present or not accessible due to the rules governing access modifiers like *private* and *protected*.

None the two error types that are new in our table are related to missing dependencies, but rather to problems in the source code of the projects. This confirms the trend that dependencies are starting to play a smaller impact in build failures.

Even with smaller percentages in Table 4.2 we saw overlaps in the errors. As we improved the problems related to dependencies, we start to see a shift away from dependencies and into other errors, namely *Incompatible types* and *Duplicate class*.

Two conclusions can be taken from Table 4.2: our strategies into solving dependencies are working, and resolving the $38,313$ projects that are still to be successfully built is becoming a harder task, as a larger percentage of projects are affected by a larger group of different problems. The projects that did not compile are getting distributed across new sets of errors.

4.4 A new strategy for Solving Dependencies

Even though the previous sections solved roughly a third of the original compilation problems, we still have 9,087 projects exclusively with dependency problems. We decided to try an alternative strategy for dependency matching to try and resolve these issues.

The compilation results from our first attempt (Section 3), gave us one really useful information. Since we did not include any external dependencies in the build process and just compiled the program as is, the error reports complaining about missing dependencies can have one of two sources. They can mean the project is incomplete and missing information, which can have various sources, such as problems when building Sourcerer or incomplete projects in the repositories themselves. In these cases there is nothing we can do. The other reason is related to missing external dependencies, and in these cases there is something we can do differently than our previous approaches.

Jars are simply compressed files containing a specific representation of a project packaged, organized in a structured format: in a *Jar*, a class defined by the fully qualified name *"foo.bar.baz"* has a package name *"foo.bar"* and, when digging inside a *Jar* file structure, is be stored at the location *"foo/bar/baz.class"*.

Sourcerer has in its repositories 147,243 *Jar* files, each one containing a series of classes. Using a *Jar* exploration tool provided by *Oracle*[2], we can list all the class files inside a *Jar* and extract the full qualified name for each one of them.

The next step would be to match each project to *Jar* files in Sourcerer, but doing so by comparing the fully qualified names of their dependencies with the fully qualified names obtained from build error logs. However, this would include a large number of unnecessary packages, since developers typically bundle more than one project into a single *Jar* package.

[2] `http://docs.oracle.com/javase/8/docs/technotes/tools/windows/jar.html`, as of May 15, 2015. This tool comes bundled with *Java SE* 8.

To avoid this problem, we split the *Jar* files into smaller Jars. We did this by recursively searching all the folders in side a *Jar*, and for each class file we found we create a new *Jar*. For example, if a specific *Jar* file contained the following classes:

1. foo/bar/baz/Foo.class
2. foo/bar/baz/Foo$1.class
3. foo/bar/baz/Foo$inf.class
4. foo/norf/Bar.class
5. foo/norf/quz/Bar$5.class
6. foo/norf/quz/Bar.class

we would obtain two packages. One jar would contain classes 1, 2 and 3 and would be associated with the fully qualified name *"foo.bar.baz"*. The other would contain classes 4, 5 and 6 and would be associated with the fully qualified names *"foo.norf"* and *"foo.norf.quz"*. We expected, with this step, to obtain dependencies that we could not find in the previous steps.

Using this method we split the original 147,243 *Jar* files in Sourcerer into 491,023 new smaller *Jar*s. Afterwards, we associated these to the 9,087 projects that still had dependencies missing. That required a total 91,904 packages (12,343 unique packages). For the projects that, even after this step, had missing dependencies, *Maven* dependencies were looked up in Sourcerer using the method detailed in Section 4.3.

We were not capable of locating all the packages for 5,467 projects. In cases where there were two *Jar* files with a fully qualified name as dependency of a project, we tried to choose the *Jar* whose set of fully qualified names covered better all the dependencies of the project.

Table 4.3: Build Errors - Fourth Try

Error Type	**# projects**		**% projects**
Cannot find symbol	17284	(↑2022)	68.042
Package not found	8113	(↓974)	31.938
Illegal access	6819	(↑5024)	26.844
Duplicate class	2575	(↓1251)	10.137
Incompatible types	2350	(↓505)	9.251
Override error	1889	(↑207)	7.436
Cannot be applied	1219	(↓926)	4.799
No suitable definition found	1183	(↓417)	4.657
Abstraction error	1012	(↓112)	3.984
Static import only from classes and interfaces	952	(↑385)	3.748

With this new step we were capable of building 1,425 new projects, which accounts for 33.18% of the total projects with which we started. We are at a total of 18,321 projects successfully built.

The top 10 errors encountered are shown in Table 4.3. The error type *Cannot find symbol* is now the top reason why projects build, and actually saw a increment (together with *Illegal access*). Most error types however decreased. This is good and proves the success of this approach.

With 68% errors in builds being now related to one error type, is it expected this starts playing a most significant role in failures, but doing in scenarios where it exists with a combination of other error types. We also see that *Unresolved dependencies* has reduced considerably. *Incompatible types* is not longer in the top few errors and has been replaced by *Illegal access*.

We also found a that the projects we have not built suffer from more varied sets of errors, making them harder to resolve with broad heuristics.

Chapter 5

Discussion and Conclusion

5.1 Final Analysis of Results

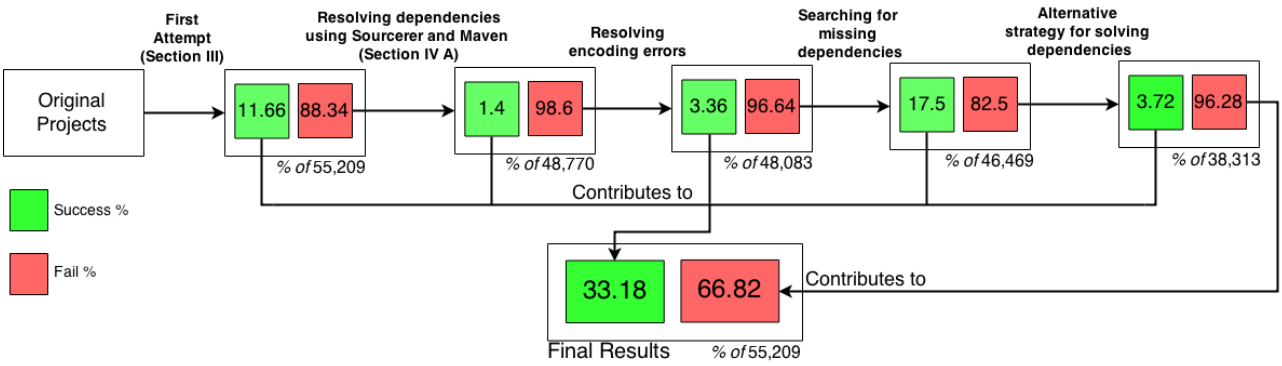

Figure 5.1: The success and failure of each step that we took.

As we transition from the first to the second compilation, and compare the results described in Table 3.1 to the results described in Table 4.1, we start seeing several trends. Resolving dependencies plays a significant role in successfully building the repositories and these problems became proportionally smaller and smaller and other error types become more present. More interestingly, we start seeing a large number of projects being individually affected by a greater number of problems.

The overlap of errors helps us understand the correlation between the different errors. For example, we see that *Package not found* and *Cannot find symbol* go hand in hand, and that there are few projects with just one of the error.

There are 844 projects that have *Package not found* and not *Cannot find symbol* and 831 projects that have *Cannot find symbol* and not *Package not found*, in comparison to 26,400 projects having both. *Override error* and *Duplicate class* also have high overlap with *Package not found* and are highly correlated.

After the third attempt we see a different picture. The overlap between *Package not found* and *Cannot find symbol* decreased considerably. The number of projects that have *Package not found* and not *Cannot find symbol* (430) is much less than the number of projects that have *Cannot find symbol* and not *Package not found* (6605).

It is interesting that there are projects that have *Package not found* errors but no *Cannot find symbol* errors. This would likely mean that these packages are imported needlessly into the projects and removing such imports could be a strategy for improving the success count. From the data in Table 4.2 we see that the number of projects throwing build errors due to missing dependency has substantially dropped. Which means our approach was able to resolve dependencies successfully.

Another lesson is that strategies can be successfully applied to build projects in Sourcerer only until a certain point, after which building projects becomes increasingly harder. As we resolve our larger problems, two trends start to appear. First, projects with incorrect dependencies start to have a series of problems with varied causes, and successfully building them implies tackle all the problems. Second, we start to see error types that are related to the actual source code, which can be missing information or code written with errors. It is hard to make a project compile if there are a few classes missing which are not generic, but are specific for that project. Similar results were obtained by [5] (see Section 1.2).

5.2 Generality of the Process

In this paper we describe the methods used and the results obtained when compiling the *Java* projects found in Sourcerer. These methods and results can be generalized to other programming languages if they meet the following criteria:

1. The language should have a compiler and the compiler should expose errors with enough detail to categorize and extract needed information from them. Most language compilers have sufficiently detailed information as they cater to the needs of the programmer. For example, in C/C++ if a header file *foo.h* is missing, the *GCC* compiler gives the error: *fatal error: foo.h: No such file or directory.* This can be parsed with regular expressions to determine that the C/C++ project does not have access to foo.h

2. The external packages used by the programming language should be in a format that allows for extracting information that can be used to identify them. In *Java*, *Jar* files could be explored using tools, the packages that they allow could be inferred from the folder structure. Similar techniques can be used for other languages: Header files found in the developer packages in software repositories, like the *Debian* package repository, can be used to determine the right packages to obtain for a *C/C++* project. DLL exploration tools[1] can be used to extract relevant information from *.NET* DLLs.

3. The language should have a build system that can automatically compile all the code, reference external dependencies, and provide us with the error logs. The *Ant* build system is a cross platform, language independent tool that can be used for this purpose. The process described however is not restricted to only *Ant*. Any build system that meets the mentioned criteria can be used. Examples of build systems for other languages include: *Make*, a classic Unix build tool usable for any language; *Cabal*, a

[1] `https://technet.microsoft.com/en-us/library/cc738370.aspx` is a Microsoft provided tool to explore a DLL

tool for building applications and libraries in *Haskell*; *MSBuild* for automating the build for *.NET* projects.

4. It should be viable to parse the language source code and build a code database like Sourcerer. The information obtained for the build process that was preprocessed and maintained by Sourcerer include: details about the project such as the name, description, location in the repository; the files used by the project as seen when crawled from the online code repository; and a database of every entity discovered by Sourcerer. All the three sets of information can be generated for other languages using crawlers and the right language parsers.

Programming languages like *C/C++*, *C#*, *Scala*, etc. that fit the requirements mentioned are easy candidates for the direct application of the methods described in this work. It would be harder to do the same for languages like *Python*, *Ruby*, *Bash*, etc. that do not conform to these requirements. Interpreted languages have at least two main problems that prevent us from using methods and analysis described in this work directly. First, all errors are only caught during executing. In order to detect and categorize errors, we would need to execute the code. This can lead to unintended side effects which in some cases could be dangerous as we could be inadvertently executing malicious or unsafe code. Second, for interpreted languages to execute, the environment has to be set up in the manner intended by the programmer. The absence of external artifacts like files, databases, system services, etc. are not a concern for languages that are compiled. In some cases there could be workarounds designed. For example, the tool py2exe [2] can be used to convert *Python* code into an *Windows* executable (*EXE* file) that can execute without the need of the *Python* interpreter.

[2] `http://www.py2exe.org/`

5.3 Conclusion

Open source software has moved from being pet projects for a minority to being the software used daily by the majority. For example, all the tools used in this work are open source software with their source code readily available. Open source code lends itself very well to software reuse. The meteoric growth of repository sites like GitHub creates problems that didn't exist before: while many projects are of excellent quality, a large number of open source projects are of poor quality in terms of purpose, completeness, maintenance, etc. The popularity of a project is not always a measure of its quality [18] and we can look to reuse even the smaller projects. As such, we need a way to filter out the good projects from the bad. We need to identify the source code that is complete and thus can be reused. A reliable attribute of a project to solve this problem is whether it can be compiled successfully.

Several conditions have to be met in order to compile one project successfully. Its external dependencies have to be identified and be made available during compilation. The project might also require special compiler flags. Some projects are written for compilation only in a specific environment. These are a few problems when compiling *one* project. It is impractical to manually repeat the process for all the projects in a repository. Even software companies, that rigorously maintain their internal repositories of code, occasionally see components failing to build successfully [12]. Thus, automatically compiling a large repository of open source code is challenging; we tackled it by designing a series of heuristics.

We used the projects indexed by Sourcerer as the target repository for designing and executing our heuristics. We tried to successfully compile as many projects in the repository as we could. We approached it by, first, finding and analyzing the errors when compiling, and then designing heuristics to tackle the most significant errors. This process was iterated five times and targeted the two most common types of error found viz. *package not found*, and *unmappable character*. Dependency errors (*package not found*) was by far the most common

error encountered. However, fixing external dependencies is not an easy task. Through our analysis of the build errors, we saw that matching packages to missing imports can lead to an increase in other types of errors. It is harder to know if these new errors are caused by a mismatch in the package (for example: wrong version of the package used) or are errors in the code that have now been exposed after successfully resolving all dependencies.

By executing every heuristic we designed on a repository of $55,209$ *Java* projects, we were able to successfully build $18,321$ or 33.18% of the projects. This gave us a new repository of compiled artifacts, and compilable projects that can be used in the studies to come.

Bibliography

[1] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 207–216, Piscataway, NJ, USA, 2013. IEEE Press.

[2] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 1–4, Washington, DC, USA, 2009. IEEE Computer Society.

[3] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79:241–259, 2014.

[4] R. Barbuti and S. Cataudella. Java bytecode verification on java cards. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 431–438, New York, NY, USA, 2004. ACM.

[5] H. Barkmann, R. Lincke, and W. Lowe. Quantitative evaluation of software quality metrics in open-source projects. In *Advanced Information Networking and Applications Workshops, 2009. WAINA'09. International Conference on*, pages 1067–1072. IEEE, 2009.

[6] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. The evolution of project inter-dependencies in a software ecosystem: the case of apache. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 280–289. IEEE, 2013.

[7] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. Spur: A trace-based jit compiler for cil. *SIGPLAN Not.*, 45(10):708–725, Oct. 2010.

[8] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press.

[9] A. Gal, C. W. Probst, and M. Franz. Java bytecode verification via static single assignment form. *ACM Trans. Program. Lang. Syst.*, 30(4):21:1–21:21, Aug. 2008.

[10] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[11] S. Holzner. *Ant: The Definitive Guide, Second Edition*. O'Reilly Media, Inc., 2 edition, 2005.

[12] N. Kerzazi, F. Khomh, and B. Adams. Why do automated builds break? an empirical study. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 41–50. IEEE, 2014.

[13] R. Lämmel, E. Pek, and J. Starek. Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1317–1324, New York, NY, USA, 2011. ACM.

[14] D. Landman, A. Serebrenik, and J. Vinju. Empirical analysis of the relationship between cc and sloc in a large corpus of java methods. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 221–230, Sept 2014.

[15] M. Lungu, R. Robbes, and M. 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.

[16] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, pages 1–30, 2013.

[17] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. ACM.

[18] H. Sajnani, V. Saini, J. Ossher, and C. V. Lopes. Is popularity a measure of quality? an analysis of maven components. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 231–240. IEEE, 2014.

[19] C. Seidl and U. Asmann. Towards modeling and analyzing variability in evolving software ecosystems. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 3:1–3:8, New York, NY, USA, 2013. ACM.

[20] S. Thummalapenta and T. Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[21] B. Varanasi and S. Belida. *Introducing Maven.* Apress, Berkely, CA, USA, 1st edition, 2014.

[22] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *Software Engineering, IEEE Transactions on*, 31(6):466–480, 2005.

[23] T. Xie and J. Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 54–57, New York, NY, USA, 2006. ACM.

[24] E. Yardimci and M. Franz. Mostly static program partitioning of binary executables. *ACM Trans. Program. Lang. Syst.*, 31(5):17:1–17:46, July 2009.

[25] H. Zhang. *A comprehensive approach for software dependency resolution.* PhD thesis, University of Victoria, 2011.