Automated Dependency Resolution for Open Source Software

Joel Ossher Sushil Bajracharya Cristina Lopes
Bren School of Information and Computer Sciences University of California, Irvine
Irvine, USA
{*jossher,sbajrach,lopes*}*@ics.uci.edu*

***Abstract*—Opportunities for software reuse are plentiful, thanks in large part to the widespread adoption of open source processes and the availability of search engines for locating relevant artifacts. One challenge presented by open source software reuse is simply getting a newly downloaded artifact to build/run in the first place. The artifact itself likely reuses other artifacts, and so depends on their being located to function properly. While merely tedious in the individual case, this can cause serious difficulties for those seeking to study open source software. It is simply not feasible to manually resolve dependencies for thousands of projects, and many forms of analysis require declarative completeness. In this paper we present a method for automatically resolving dependencies for open source software. It works by cross-referencing a project's missing type information with a repository of candidate artifacts. We have implemented this method on top of the Sourcerer, an infrastructure for the large-scale indexing and analysis of open source code. The performance of our resolution algorithm was evaluated in two parts. First, for a small number of popular open source projects, we manually examined the artifacts suggested by our system to determine if they were appropriate. Second, we applied the algorithm to the 13,241 projects in the Sourcerer managed repository to evaluate the rate of resolution success. The results demonstrate the feasibility of this approach, as the algorithm located all of the required artifacts needed by 3,904 additional projects, increasing the percentage of declaratively complete projects in Sourcerer from 39% to 69%.**

I. Introduction

The open source movement has fundamentally changed the software development process for many practitioners. The quantity of source code freely available online has made both component-based and pragmatic reuse viable alternatives to building programs entirely from scratch [1]–[3]. There are a variety of challenges that this approach presents to developers, ranging from locating relevant artifacts to understanding them and integrating them into a new system.

One of the most tedious of these challenges is simply getting a newly downloaded artifact to build or run in the first place. This is in a large part due to the artifacts themselves using other artifacts, potentially creating a long, and sometimes poorly documented, chain of dependencies. The work of Holmes and Walker aids users in exploring these dependencies [4], yet requires them to be resolved before this exploration can begin.

While there are a variety of things artifact developers can do to simplify this dependency resolution, the usability issues that affect open source software [5] in general are still present; just because developers can make things easier for users doesn't mean they do. Building or running an artifact can therefore vary widely in difficulty. It can involve anything from executing a single command to reading through documentation and online forum posts to importing source files into an IDE and manually resolving any missing dependencies. While the initial setup cost is negligible in comparison to the benefit that reuse can bring, it can discourage developers from adequately exploring their options. Furthermore, it greatly complicates those wishing to study open source software, as many analyses require declaratively complete programs. For example, in order to study the usage patterns of Java libraries in the open source community, one must be able to determine which projects make use of these libraries, and in what manner.

While manually resolving dependencies for a few projects is merely tedious, it is completely unfeasible for thousands. This leaves researchers with limited options. One approach is to pick a few projects to study and to manually ensure that they are declaratively complete. While effective, this approach is totally non-scalable, and fails to take advantage of the sheer breadth of the open source community. Another possibility is to modify the analyses being performed so as not to require declaratively complete programs. This can be quite successful, but is only possible on a case-by-case basis and necessarily introduces some degree of fuzziness. A related approach is to perform fuzzy type inference on the projects in order to fake declarative completeness [6], [7]. While this eliminates any need to modify the analyses themselves, it still introduces uncertainty.

In this paper we present a fourth approach, a technique for the automated resolution of missing dependencies in open source software. This is accomplished by cross-referencing missing type information statically extracted from a project with a database of commonly used artifacts. Our approach is scalable, as it is fully automated and designed to work with real-world data. It is also more accurate, as it locates the artifacts that were intended to be used with a project rather than attempting to reconstruct them.

This approach can simplify the process of producing reusable artifacts, in addition to consuming them. Properly specifying a project's build configuration for a tool such as Maven, while extremely helpful to the project's users,

978-1-4244-6803-4/10/$26.00 © 2010 IEEE

is non-trivial, especially if the project keeps evolving. In reducing the need for specification, our approach to automated resolution ultimately requires less work from everyone.

The contributions of this paper are threefold. First, it introduces a technique for automatically resolving missing dependencies in open source software. Second, it describes how this technique can be implemented on top of Sourcerer, an existing infrastructure for the large-scale analysis of open source software. Third, it presents a two-part empirical evaluation of the resolution system: a small-scale manual evaluation on 4 popular open source projects, and a large-scale automated evaluation on the 13,241 projects in the Sourcerer managed repository.

The remainder of the paper is organized as follows. Section II gives a brief overview of the Sourcerer infrastructure, while Section III describes our approach to automated dependency resolution. In Section IV we present our two-part evaluation of the system. Finally, we discuss the related work in Section V and conclude in Section VI.

II. Sourcerer Infrastructure

We implemented our dependency resolution algorithm as part of Sourcerer, an infrastructure for the large-scale indexing and analysis of open source code [8]. Sourcerer provides a foundation for empirical analysis and upon which state of the art code search engines and tools can easily be built.

As shown in Figure 1, the Sourcerer infrastructure is divided into three tiers. At the bottom is the core infrastructure, a collection of tools to handle the aggregation and indexing of open source code. The *code crawler* retrieves Java source code from a variety of locations, such as open source repositories, public web sites, and version control systems. This code is then parsed and analyzed by the *feature extractor*, and stored in Sourcerer in three forms: (i) the *managed repository* keeps a versioned copy of the original contents of the project and related artifacts such as libraries; (ii) the *code database* stores models of the parsed projects, based on the metamodel; and, (iii) the *code index* stores keywords extracted from the code for efficient retrieval.

The second tier consists of web-services for accessing the information in the lower tier. Direct access to the managed repository and code database is provided, as well as a code search service, and a service for automatically slicing an entity out from a project.

The top tier is composed of the applications built using Sourcerer. So far they include a web-based search engine and an Eclipse plug-in for test-driven code search.

Sourcerer uses Chen's entity-relationship modeling [9] to model the latest version of the Java language. The metamodel is an adaptation of Chen et al.'s C++ entity-relationship metamodel [10], and is similar to the FAMIX family of metamodels [11]. It is sufficiently expressive as

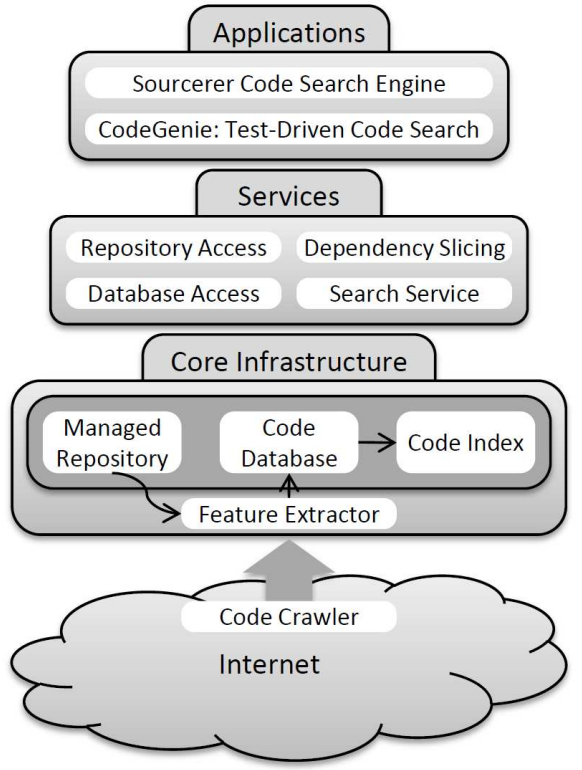

Figure 1. Sourcerer Infrastructure

to allow fine-grained search and structure-based analyses, yet also efficient and scalable enough to handle the large quantity of code present in Sourcerer. Following the metamodel, a project model element exists for every project contained in the managed repository, and a jar model element for every jar. Projects and jars are linked to the sets of entities contained within them, and to the relations that have these entities as their source. Entities are the defined types like CLASS or METHOD, while relations capture the interactions between the entities such as CALLS or INSTANTIATES. References external to a project, such as method calls to third-party components, are captured in the relations.

The feature extractor is the core infrastructure tool that handles the generation of the entity-relationship models from the source code. It is built as a headless Eclipse plug-in in order to take advantage of Eclipse's Java development tools [12]. For every file in every project and jar in the managed repository, the feature extractor builds an abstract syntax tree (AST) attributed with full binding information. The models are then constructed by doing a traversal of these ASTs. The feature extractor is designed to handle the messy data that results from Sourcerer's large-scale automation. It has mechanisms to filter out duplicate files from within a single project, and to handle unresolved types during extraction. However, these missing types introduce unknown reference

targets in the resulting model, and so are undesirable.

Further details on the Sourcerer infrastructure are available in our prior publications [8], [13].

III. Automated Dependency Resolution

Our approach to automated dependency resolution works by cross-referencing a project's missing type information with a repository of candidate artifacts. There are three main components to this approach. First, a collection of candidate artifacts must be created. Second, there needs to be a method for determining the missing types for a project. Third, there must be an algorithm for selecting artifacts from the candidate repository based on the missing type information. A detailed discussion of each of these three components follows.

A. Artifact Repository

The main requirement for the artifact repository is that it must be sufficiently comprehensive as to contain the missing dependencies for most open source projects. It also must be indexed in such a way that components can be easily retrieved by the types they contain.

We decided to use the Maven 2 Central Repository [14] as the basis for our artifact collection. Apache Maven is a set of Java tools for project management and build automation. One of its more interesting features is that users can specify component dependencies in a project's Project Object Model (POM) file which are then automatically downloaded when necessary whenever the project is built. In order to accomplish this, the Maven tool connects to a repository that contains a collection of artifacts organized by group, name and version. While users of Maven can create and host their own repositories, the open source community by and large uses the centralized Maven repository. As a result, this repository contains a very large set of commonly used artifacts, making it ideal for our purposes.

The online version of the Maven Central Repository is not indexed in such a way as to be directly usable for our form of dependency resolution. It supports artifact lookup, but not by the types that each provide. We therefore extracted and indexed all of the provided types for each artifact in the repository. This was accomplished using a stripped down version of the Sourcerer feature extractor. Instead of extracting the full set of entities and relations for a jar, we just extracted any entity that could be used externally: interfaces, classes, annotations, enums, fields and packages. These entities were then placed into an instance of the Sourcerer code database where they were indexed by fully qualified name.

B. Missing Type Identification

A necessary step in resolving missing dependencies is the identification of the missing types themselves. However, reliably determining the exact fully qualified name (FQN) of a missing type is not possible given the ambiguity in Java's import mechanism. Figure 2 contains a simple example to illustrate this ambiguity. Single type imports are the best case, as seen in line 3, as they contain a fully qualified name. Therefore even if `foo.Single` cannot be resolved, it is clear that the field on line 8 has the type `foo.Single`.

```
1  package example;
2
3  import foo.Single;
4  import bar.*;
5  import baz.Baz.*;
6
7  public class Example {
8    public Single a;
9    public OnDemand b;
10   public foo.OnDemand c;
11 }
```

Figure 2. Example of Import Ambiguity

On-demand imports, those with a * operator, do not fully specify which types they import, instead including all types within a given package or type. Lines 4 and 5 are both on-demand imports. This causes it to be unclear which package the type on line 9 belongs to. If a type named OnDemand cannot be found, there are three potential places it could be. It could be a top-level type in the package `bar`. It could be a public inner type in the type `baz.Baz` (assuming that `baz.Baz` is a type and not a package that fails to follow naming conventions). Or it could be a type in the package `example`. Import statements are not always necessary, as any type can be fully qualified. The type in line 10 is an example of this. Although it has the same simple name as the type in line 9, the reference in unambiguous.

We decided to take a relatively simple approach to handling this ambiguity. We restricted lookup to unresolved import statements and missing fully qualified type names; for on-demand imports, we restricted ourselves to the specified FQN, ignoring the *. This is effective in the vast majority of cases, as single type imports are the norm. Many development environments will even automatically convert on-demand imports into sets of single type imports when given the chance.

Not surprisingly, sidestepping the import ambiguity issue can cause some problems. For example, this approach will fail to locate any missing types within the same package. However, given our focus on identifying external dependencies rather over reconstructing type names for partial programs, we do not expect there to be significant overlap of package names. Another issue is that if only on-demand imports are used, and the package in question is found in multiple components, there is no way to pick the correct component.

One minor modification to solve these two issues is to generate all possible fully qualified names in cases of

ambiguity. Any missing simple name is therefore combined with every on-demand import as well as the current package to form candidate fully qualified names. This mirrors Java's type resolution system, and will be successful unless two of the candidate FQNs happen to match. While always possible, this does not seem to be especially likely.

When done manually, the missing type identification process is relatively simple. Either the compiler reports that some types cannot be resolved or the program terminates with an exception stating that a class definition could not be found. In both cases, the missing types must be located and the process iterated until no errors occur. This iteration is required because it is not always possible to statically determine all of the missing types in a single pass. Transitive dependencies are the main cause of this, though some type errors can also be masked by other errors, only appearing once the other errors have been fixed.

Rather than build a system to comb through compiler error messages, we modified the Sourcerer feature extractor to report missing types. As was described earlier, our feature extractor builds an attributed AST complete with resolved reference information in order to generate the entities and relations. Previously, if a reference could not be resolved the extractor created an UNKNOWN entity type with a limited FQN to represent that entity. With our new setting, if the extractor encounters any unresolved types it restricts itself to printing all candidate FQNs as discussed above. The missing type extraction is iterated along with the resolution algorithm until there are no longer any missing types or no more candidate components can be found.

C. Resolution Algorithm

Once the missing types have been identified, the final step is to match them against the artifacts in the candidate repository. We developed a simple greedy algorithm to perform this matching, which is described in pseudocode in Figure 3. It begins with a list of all the FQNs reported missing by the feature extractor. The nested loop on lines 1-8 collects the artifacts that can provide at least one of these missing FQNs, and marks each of those artifacts with the FQNs that it can provide.

The remainder of the algorithm repeatedly iterates until there are no more artifacts remaining in the collection. The for loop on lines 11 through 17 picks the artifact that matches the most missing FQNs, treating it as the next candidate for inclusion. The for loop on lines 19 through 28 then unmarks every FQN that this artifact provides from the remaining artifacts in the collection. If this causes an artifact to have no FQNs marked, it is removed from the collection. Every iteration of this loop will cause at least one artifact to be removed from the collection, ensuring termination. Once finished, line 31 returns the set of all artifacts that were chosen for inclusion.

Require: List L of missing FQNs, empty set REL of relevant artifacts, empty set $PICKED$ of artifacts.
Ensure: $PICKED$ contains a set of artifacts that cover as many missing FQNs as possible.

```
1:  for all fqn ∈ L do
2:      for all art such that art contains fqn do
3:          if art ∉ REL then
4:              Add art to REL
5:          end if
6:          Mark art as using fqn
7:      end for
8:  end for
9:  while REL not empty do
10:     bestMatch ⇐ null
11:     for all art ∈ REL do
12:         if bestMatch is null then
13:             bestMatch ⇐ art
14:         else if art has more marked FQNs than
            bestMatch then
15:             bestMatch ⇐ art
16:         end if
17:     end for
18:     Remove bestMatch from REL
19:     for all jar ∈ REL do
20:         for all fqn marked in bestMatch do
21:             if fqn marked in art then
22:                 Unmark art
23:             end if
24:         end for
25:         if art has no marked FQNs then
26:             Remove art from REL
27:         end if
28:     end for
29:     Add bestMatch to PICKED
30: end while
31: return PICKED
```

Figure 3. Component Matching Algorithm

In summary, the algorithm repeatedly picks the artifact that provides the largest number of missing types, discounting missing types already provided by previously included artifacts. It iterates until either no missing types or artifacts remain.

IV. Evaluation

We performed a two-stage evaluation of our automated dependency resolution system. First, as a small-scale proof of concept, we selected a few popular open source projects. We ran them through our system and manually compared the chosen jars to those expected to be used with those projects. Second, as a larger-scale evaluation, we applied the dependency resolution to the entire Sourcerer managed

Table I
SOURCERER MANAGED REPOSITORY GENERAL STATISTICS

General Stats	Count	Non-Empty	Disk Space
Projects	18,922	13,241	257.8GB
Project Jar Files	47,864	40,388	18.5GB
Maven Jar Files	55,135	51,293	21.5GB
Latest Maven Jars	10,725	9,707	4.1GB

repository. We automatically checked each project to determine if the dependency resolution was successful, and manually examined a subset of the failed cases to determine the cause.

A. Candidate Artifact Index

For both evaluations we used a candidate artifact repository derived from the Maven Central Repository. Due to resource constraints, rather than index the entire Maven repository, we limited the scope to the latest version of each artifact.

Table I contains some general statistics on the size of Sourcerer's managed repository and the Maven candidate artifact index. The first row describes the projects which we automatically captured from Apache, Java.net, Google Code Hosting and SourceForge. The existence of empty projects is primarily due to the crawler failing to download the code or incorrectly downloading a non-Java project. The second row is the jars that were included along with the projects. 40,000 jars for 13,000 projects indicates that, on average, each project came with nearly 4 jars. However, the distribution is significantly skewed, with many projects containing no jars at all while others contain hundreds. Row 3 describes our mirror of the Maven 2 Central Repository. Row 4 is the Maven mirror that has been restricted to the latest version of each artifact. In both case, the empty jars occur when our feature extractor is unable to read the jar file, either due to its being corrupted in some way, or simply not containing any class files.

Table II contains a more detailed breakdown of our candidate artifact index. With regards to the third row, a binary jar file is considered to have source code if the code is included inside the jar itself or if a corresponding source jar is co-located with it. For the forth row, the class file count excludes the class files generated by the Java compiler for inner or anonymous classes, and so closely corresponds to the number of source files.

The second half of the table contains details on the entities extracted from the jars. As can be seen by comparing the two columns, there is a large amount of duplication present in the candidate artifact index. We considered two entities to be duplicates if they have identical FQNs. While we had expected some duplication, the extent of it surprised us. Upon investigation, there appear to be two main causes. First, while we had limited the index to contain only the latest version of each artifact, we did not account for projects

Table II
CANDIDATE ARTIFACT INDEX STATISTICS

General Stats	Count	
Jar Files	10,725	
Non-Empty Jar Files	9,707	
Jar Files With Source	5,368	
Class Files	771,458	
Entity Breakdown	**Count**	**Unique Count**
Packages	78,950	43,199
Classes	774,937	433,237
Enums	6,877	4662
Interfaces	143,754	78,945
Annotations	6,848	2,627
Fields	3,323,417	1,777,234

that had changed their names. Any such artifact had its latest version included for each name, leading to duplication. Second, it appears that many artifacts attempt to limit the number of dependencies they require by packaging some of their dependencies into their jar. This is especially common for groups of artifacts from the same developers. Rather than requiring users to download a set of common utility classes as a separate jar, they are instead included into each distribution.

B. Small-Scale Evaluation

For the small-scale evaluation, we chose the four projects which were used to evaluate Dagenais et al.'s partial program analysis [6]: JFreeChart, Lucene, Jython, and Spring. We retrieved the source code for the latest version of each project from their respective repositories. Each project was individually loaded into Eclipse and run through the modified Sourcerer feature extractor under four conditions: no external artifacts (NA), resolve from no artifacts (RNA), project included external artifacts (PA), and resolve from project included external artifacts (RPA). In the first condition, the classpath was set to contain only the standard Java libraries. In the third, the classpath included the jars that came packaged along with the projects. For the second and fourth conditions, the feature extractor was told to dynamically modify the classpath based on detected missing types, starting from the no external artifact and project included external artifact conditions respectively. The extractor terminated when either no missing types remained, or it was unable to locate any further candidates for inclusion. The time to perform this dependency resolution varied depending on the size of the target project, with Spring taking the longest. Even then, it was only a matter of minutes; significantly less time than it would take to manually locate the necessary jars.

The results can be found in Table III, broken down by project and condition. The first row shows the number of unique missing types that were reported under the default classpath (no external artifacts) condition. As all the values

Table III
SMALL-SCALE EVALUATION RESULTS

	Project			
	JFreeChart	*Lucene*	*Jython*	*Spring*
NA Missing Types	68	55	115	1,048
RNA Added Jars	4	8	13	53
RNA Missing Types	0	13	22	65
Project Included Artifacts	3	13	155	240
PA Missing Types	58	32	38	47
RPA added artifacts	3	2	6	8
RPA missing types	0	13	19	17

Table IV
ARTIFACTS CHOSEN FOR JFREECHART

	Resolved Artifact by Condition	
Missing Artifact	*No Artifacts*	*Project Artifacts*
JUnit	JUnit	-
Eclipse SWT	Eclipse SWT / GWT-dev	Ecipse SWT
Java Servlets	GWT-dev	-
Pie Datasets	GWT-BV	JFreeChart
JCommon	GWT-BV	CLIF

are greater than zero, all of the projects were missing at least one type. The following two rows contain the number of jars that our missing type resolution algorithm added to each project's classpath, and the subsequent number of unique missing types reported. While artifacts were located for all four projects, only for JFreeChart were all the missing types eliminated. We will not discuss the reason for that here, as the large-scale evaluation thoroughly addresses why failures can occur. However, it should be noted that the unresolved missing types for these projects primary occurred in sandbox or experimental directories, which are not expected to be fully correct.

The second half of Table III covers the two conditions in which the project included artifacts were used. When added to the classpath, these artifacts reduced the number of missing types, but in no cases eliminated all of them. In this condition, the total number of missing types for both Jython and Spring was reduced relative to the resolution algorithm on its own, indicating that some of the project artifacts contain types that are not in our index.

Table IV contains a breakdown of which artifacts were chosen for JFreeChart under both resolution conditions. JFreeChart was missing libraries for JUnit, Eclipse SWT, and Java Servlets. In addition, two JFreeChart classes relating to pie datasets were missing from the subversion repository (they are present in the cvs repository). JCommon, while a dependency of the current stable release of JFreeChart, has been removed as a dependency from the version in the subversion repository. As a result, it is actually a transitive dependency introduced by the two missing pie dataset files. The JUnit and Java Servlet libraries were included in JFreeChart's repository, and so only needed to be resolved in the no artifacts case.

Table V
LARGE-SCALE EVALUATION RESULTS

Condition	**Unique (%)**	**Cumulative (%)**
No External Artifacts	2,608 (20%)	2,608 (20%)
Project Included Artifacts	2,578 (19%)	5,186 (39%)
Resolution Algorithm	3,904 (29%)	9,090 (69%)
Remainder	4,151 (31%)	13,241 (100%)

As the results in Table IV show, the resolution algorithm did not always pick the artifacts that one might expect. This is a consequence of the interaction between our resolution algorithm and the packaging of some of the artifacts in our candidate repository. Take, for example, GWT-dev, which is the developer package for the Google Web Toolkit. It contains some Eclipse SWT classes as well as a Java servlet implementation. Therefore, when the resolution algorithm was choosing artifacts, GWT-dev provided the most missing types, as it merged together two distinct artifacts.

In cases of single artifacts, our algorithm cannot distinguish between an original artifact and a secondary artifact that includes the original entirely. As a result, in neither condition was the JCommon artifact chosen. Instead, two artifacts that contained portions of JCommon were used. Similar results were seen for the other three projects.

Overall, the results of the small-scale evaluation indicate that our artifact resolution algorithm, while effective at eliminating most missing types, does so in a manner not entirely consistent with a manual approach. Further tuning of the selection heuristic or the index itself are needed to enable original artifacts to be chosen preferentially over copies.

C. Large-Scale Evaluation

The large-scale evaluation was performed in much the same manner as the small-scale one. The contents of the Sourcerer managed repository were run through Sourcerer's existing feature extraction system, and the output files were analyzed to evaluate the success of the extraction.

Each project was checked to determine if any missing types were reported during extraction, and if they were ultimately resolved. Projects were not checked for compilation errors in general. While this choice eliminates the impact of errors unrelated to type resolution, it introduces the possibility of another form of error. There is no way to know if the algorithm picked an artifact that resolved the missing types yet introduced other errors into the program. Such a problem seems most likely to occur if multiple incompatible versions of an artifact are present in the index. This remains an area for future exploration.

The results can be found in Table V. The first row describes the projects that require no external dependencies, which account for 20% of the repository. These projects have no missing types with the default classpath.

The second row shows those projects whose external dependencies were included along with the projects themselves. These projects had missing types when compiled with the default classpath. However, when the jars that were included along with the source code were added to the classpath, the projects no longer had any missing types. As our repository is primarily populated with snapshots from version control systems, this indicates that a large number of open source projects check in jar files. 19% of the projects fall into this category, which, when combined with the first category means that 39% of projects don't require any special attention with regards to dependency resolution.

The remaining 61% of the projects are the ones that cause problems for developers and researchers. They depend on external artifacts, but those artifacts are not included along with the projects. These dependencies are often expressed in documentation, in build files, or not at all. The third row of Table V shows the number of these projects for which our resolution algorithm was able to locate all of the missing dependencies. The 29% of projects in this category had missing types under the two previous conditions, but, when automatically selected jars from our artifact repository were added to the classpath, these missing types were resolved. Of the remaining 61% of projects, our approach was able to solve nearly half of them. This brings the total number of declaratively complete projects in the Sourcerer repository to 69%.

The final row in the table is the remaining projects for which our system was unable to locate all of the necessary artifacts. We randomly selected 100 projects from this category for manual examination. For each project we attempted to discern the reason for our system's inability to locate the correct artifacts. This was done by searching online for the types reported missing by the feature extractor; a process quite similar to what users currently do. In fact, in multiple cases our Google searches returned forum posts from users seeking to locate the same types as us.

We classified the 100 failed projects into 3 main categories based on the types of failures they represented: index failure, project failure and infrastructure failure. The results of this categorization can be found in Table VI. Index failure refers to those cases where the missing types should reasonably have been found, but were not in our index. Project failure is where the nature of the projects themselves prohibit using such an automated matching approach. Infrastructure failure is where our infrastructure itself seems to be at fault. Each category was broken into subcategories, which are described below.

Index Failure

1) Open source artifact not in Maven 2: These missing types were from open source projects that are not in the Maven 2 Central Repository. Most of these projects are hosted on common open source repositories, such as Google Code, SourceForge and GitHub. In some of the cases the missing types seem to occur between two related projects, possibly from the same group of developers, in which case their classification as external dependencies may be somewhat artificial.

2) Java library not in Maven 2: These missing types were from Java libraries, such as the Java Media Framework, Java OpenGL, and Java Advanced Imaging. Sun provides reference implementations for many of these, which are not in Maven.

3) Commercial artifact not in Maven 2: These missing types were from commercial artifacts not in Maven 2. We considered an artifact to be commercial if it is maintained by a company, has packages starting with `com`, and is not found in any open source repository. We did not consider licensing when categorizing, and so some of the artifacts may be released under open source and some may not. Examples we found include JBuilder (a commercial Java IDE), a Nokia library, Quicktime, and an Oracle JDBC driver.

4) Artifact in Maven 1: These missing types were from artifacts that appear in the Maven 1 Central Repository, but not in Maven 2. While one might expect the Maven 2 repository to subsume the Maven 1 one, this turns out not to be the case.

5) Version issue: These missing types were from projects that are in our artifact repository, yet from an earlier version than the one we indexed. This was the result of changes between versions that had caused incompatibilities. As we had only indexed the latest version, even with the correct artifact included there were still missing types.

Project Failure

6) Incomplete project: We considered a missing type to be caused by an incomplete project if it appeared that the authors of the project had forgotten to include some parts of the project in the online distribution. The primary indicator of this was when the missing types had the same package names as types from the initial project, and we were unable to locate the missing types through online searching. Our searching often only returned instances of the types being used in import statements, never in declarations.

7) Unable to locate artifact: These missing types could not be located by any of our searching. However, the package names did not match the originating project, so we decided not to consider these incomplete projects. While they might be cases of forgotten files, it is also possible that the necessary library is just obscure or has disappeared from the internet.

8) Generated code: These missing types seemed to refer to files that are automatically generated as part of the build process. They initially looked like incomplete projects, as types with matching packages were missing, yet closer investigation located html or xml files that appeared to be designed to be converted into Java source. It is quite

Table VI
CATEGORIZATION OF FAILURES

Category	Subcategory	Project Count
Index Failure	Open source artifact not in Maven 2	31
	Java library not in Maven 2	19
	Commercial artifact not in Maven 2	11
	Artifact in Maven 1	5
	Version issue	2
	Total	68
Project Failure	Incomplete project	17
	Unable to locate artifact	4
	Generated code	3
	Total	24
Infrastructure Failure	Crawler error	4
	Extractor error	2
	Total	6

possible that some of the projects categorized as incomplete projects might more correctly fall into this category.

Infrastructure Failure

9) Crawler error: These missing types occurred in projects where the missing types' package names matched those of the project and we were able to find the missing types in indexed online distributions of that project. This is distinct from the incomplete project condition because we were able to locate the missing types through searching. As the indexed online copies of the projects contained files that were not in Sourcerer's managed repository, it is unclear if Sourcerer's crawler missed parts of these projects, if the projects were updated after our crawl, or if our copy of the projects came from slightly different sources than the ones indexed online.

10) Extractor error: These missing types were caused by errors in the Sourcerer feature extractor. In both cases types were believed to be missing even though they were actually present. For example, one was an internationalization issue where Japanese characters had mistakenly led the extractor to miss certain definitions.

The distribution of the failures between the three categories is quite encouraging. The prominence of the index failure case suggests that simply by increasing the scope of the index we can greatly increase the proportion of projects for which our automated dependency resolution system is successful. Including more Java libraries, for example, could benefit many projects.

Improving performance in the project failure category is much more difficult. For incomplete projects there is little that can be done, but our system fares no worse than the traditional manual approach. There is room for improvement in cases involving automatically generated code, which is one area of planned future work.

D. Threats to Validity

The primary weakness of this evaluation approach is that we only checked for the presence of errors caused by missing types. There is no guarantee that our algorithm chose artifacts that would actually work; we only know that they eliminated the type errors. Even if we were to consider compilation errors in general, this still does not capture incompatibilities only seen in runtime behavior. This is especially an issue considering the repackaging of jars that we discovered in the small-scale evaluation.

In order to determine the functional correctness of the chosen artifacts, one possiblity is to evaluate projects that come with test cases. If the test cases execute correctly, then this suggests that the correct arifacts were chosen. This approach is clearly limited by the scope of the test cases, but goes a step further in validating our approach to automated dependency resolution.

The composition of the Sourcerer managed repository impacts the results of the evaluation. While we believe that our repository captures a representative snapshot of open source projects, it is possible that projects from repositories that Sourcerer does not crawl show different behavior with respect to automated dependency resolution. We would like to explore in more detail what types of projects are amenable to this approach.

V. RELATED WORK

There is a large body of work in both the industrial and academic communities on software reuse and the management of dependencies.

Component-based development is founded on the structured reuse of existing artifacts. Wren, a prototype environment to support component-based development, highlighted seven requirements for such environments, one of which was *reuse by reference* [15]. In order to address maintenance problems, the authors felt that component dependencies should be expressed by references to artifacts in remote, searchable component repositories.

That functionality is now provided by application-level package management systems, which are designed in part to automate artifact dependency resolution. They only function with the proper specifications, however, and so are useless when specifications are not provided.

When developers attempt to locate missing types, there are a variety of search engines available to them. Google Code Search [16], Koders [17], and Merobase [18] all can be used to find type definitions. findJAR [19] is especially relevant, since it indexes Maven with the specific aim of helping developers locate missing libraries. Our candidate artifact index effectively mirrors its functionality. They even provide an Eclipse plugin, yet it is limited to embedding their search interface into a tab. In contrast, our approach automates the task of detecting the missing types, searching for them, and checking if they successfully resolve the relevant errors.

Code Conjurer [3], a tool for test-driven code search, uses an automated dependency resolution system in order to compile and test recommended code. As with our approach, Code Conjurer repeatedly identifies missing types and searches for them. However, this is done on a file-level type-by-type basis, rather than considering artifacts as units. This is appropriate for locating the set of files within a project needed for an original file to compile, but will have difficulty at the artifact granularity, as it may incorrectly combine artifacts, and the number of necessary searches will increase dramatically.

Sourcerer is, in many ways, similar to Spars-J (and its successor Spars-R), a software component repository created by Inoue et al. [20]. As with Sourcerer, Spars-J preprocesses source code to extract reference information. While still belonging to projects, Spars-J treats components at the file-level, merging together components from different sources. When computing cross-project references, therefore, their system must perform some form of dependency resolution similar to what we propose, except at the file level.

In situations where compilation is not the final goal, fuzzy analysis techniques can be quite effective. Dagenais et al.'s [6] partial program analysis can generate fully-resolved abstract syntax trees in the face of missing types, albeit with a non-zero error rate. Thummalapenta et al.'s PARSEWeb [7] uses similar techniques. These approaches combine nicely with our automated dependency resolution, as they can be used in cases where dependencies cannot be located.

VI. Conclusion and Future Work

In this paper we have introduced an approach for automatically resolving dependencies for open source software. It works by cross-referencing a project's missing types with a repository of open source artifacts. We implemented this approach on top of Sourcerer, and performed an extensive evaluation. The evaluation demonstrated that our system is capable of locating all of the missing types for a significant number of open source projects. The results suggest that this type of automated system for dependency resolution is a viable alternative to the time consuming manual process, especially in the domain of large-scale source analysis.

The results of the evaluation suggest a number of areas for improvement. First, the scope of the index should be increased. The Maven central repository, while quite extensive, is not as comprehensive as we had hoped. We plan on integrating the projects from our managed repository into the candidate artifact index, as we discovered that many of the missing types were hosted on open source repositories that Sourcerer crawls.

Second, we must improve the identification of original artifacts. The artifact index contains a large amount of entity duplication, caused primarily by artifacts containing one another. This results in our system sometimes picking secondary artifacts to provide missing types. We intend to explore modifications to our selection heuristics and to the index structure to reduce this occurance.

Third, we need to handle incompatabilities between artifact versions better. One possibility is to modify the dependency resolution tool to check if compilation errors might be caused by the incorrect artifact version being used. We would also like to expand our evaluation to determine how our type resolution affects compilability.

In the near future we plan on creating an Eclipse plugin that developers can use to automatically locate artifacts using our system. Sourcerer, including the system discussed in this paper, can be found at http://github.com/sourcerer/Sourcerer.

VII. Replication Information

Sourcerer is entirely written in Java, and is comprised of a number of related Eclipse projects. As Figure 1 from Section II indicates, these projects are divided into three tiers. We will limit the discussion to the lowest tier, the infrastructure tools, as that is where automated dependency resolution is performed.

The source code for the Sourcerer infrastructure can be found at http://github.com/sourcerer/Sourcerer. The raw contents of the Sourcerer managed repository are not currently available online, though we are happy to share the data with anyone that is interested.

A. Creating a Managed Repository

For those wishing to create their own managed repository, the two projects found under *infrastructure/tools/core* are needed. The *code crawler* and *core-repository-manager* are used to crawl and build the managed repository. Instructions for their use, as well as a detailed description of the repository structure, can be found in the Git repository under *docs*.

Once a managed repository is in place, the following projects are needed to perform automated dependency resolution: *repository-manager*, *extractor*, *model*, *database*, and *utilities*. These projects can all be found under *infrastructure/tools/java*, except for utilities, which is found under *infrastructure*.

In order to use a generic Sourcerer managed repository with the Java tools, some preprocessing of the repository is necessary. The jar files from the projects must be aggregated and then indexed for quick access. This is done by running `edu.uci.ics.sourcerer.repo.Main` twice, with the *aggregate-jar-files* and *create-jar-index* flags respectively. In each case, the input repository (*input-repo*) must be specified.

B. Building a Candidate Artifact Index

The candidate artifact index contains the jar files used to resolve the missing dependencies. This index is constructed from any managed repository, which can contain whatever artifacts the user wants, such as a mirror of the Maven Central Repository. We recommend

contacting the people at Apache in order to obtain such a mirror, though a crawler and downloader for it can be found in the *repository-manager* project in the `edu.uci.ics.sourcerer.repo.maven` package.

Once the repository for the candidate artifact index is ready, the feature extractor is used to extract the types provided by these artifacts. The extractor is run as an Eclipse application, *Extractor.Extractor*, which is found in the *extractor* project. It can either be used directly in Eclipse, or as a headless plugin. The following must be specified in command-line arguments: the type of extraction (*extract-jars* to limit to jar files and *extract-binary* to ignore jar file source), the input repository (*input-repo*), and the output repository (*output-repo*).

After the extraction is complete, a database is populated with the type information. This is done using `edu.uci.ics.sourcerer.db.tools.Main` in the *datbase* project. The MySQL database connection information is specified in the following three command-line arguments: *database-url*, *database-user* and *database-password*. Before the import, the database must be initialized via the *initialize-db* flag. Then the import is performed via the *add-jars* flag, where *input-repo* must be specified. This database is then ready to be used as a candidate artifact index.

C. Automated Dependency Resolution

Once the candidate artifact index is complete, the feature extractor can perform automated dependency resolution. Dependency resolution is available for both jar and project extraction, simply by adding the *resolve-missing-types* flag to the inputs described earlier. If dependency resolution is used, the database containing the candidate artifact index must also be specified using the same arguments as described earlier.

D. Replicating the Evaluation

For the evaluation, we performed the extraction under four different conditions. The first, no external artifacts (NA), is achieved by running the extractor with *resolve-missing-types* set to false (its default value) and *use-project-jars* set to false (true is its default value). The second condition, resolve from no artifacts (RNA), is accomplished by changing *resolve-missing-types* to true. The final two conditions, project included external artifacts (PA) and resolve from project included external artifacts (RPA), are run by changing *use-project-jars* to true while alternating the value of *resolve-missing-types* accordingly.

`edu.uci.ics.sourcerer.repo.Main` can be run to evaluate the success of the extraction. The *extraction-stats* flag must be true, and *input-repo* set to the location where the extractor output its results. This tool will generate a report summarizing the extraction, which includes the total number of projects extracted, and the number of projects with missing types.

If one wants to manually examine the output of the dependency resolution process, the relevant files are located in the output repository of the extractor. Each extracted project or jar file had its own subdirectory containing a number of text files. *missing-types.txt* contains the FQNs of the types that could not be resolved. If dependency resolution is successful, or there are simply no missing types, this file is empty. *used-jars.txt* contains the list of jar files added to the classpath by the dependency resolution process. Each line starts with the md5 hash of the jar, and is followed by a space separated list of the FQNs of the missing types that the jar provided.

References

[1] A. Mockus, "Large-scale code reuse in open source software," in *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*. Washington, DC, USA: IEEE Computer Society, 2007, p. 7.

[2] R. Holmes, "Unanticipated reuse of large-scale software features," in *Proceedings of the 28th international conference on Software engineering*. Shanghai, China: ACM, 2006, pp. 961–964.

[3] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE Softw.*, vol. 25, no. 5, pp. 45–52, 2008.

[4] R. Holmes and R. J. Walker, "Supporting the investigation and planning of pragmatic reuse tasks," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 447–457.

[5] D. M. Nichols and M. B. Twidale, "The usability of open source software," 2003.

[6] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*. Nashville, TN, USA: ACM, 2008, pp. 313–328.

[7] 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*. Atlanta, Georgia, USA: ACM, 2007, pp. 204–213.

[8] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Min. Knowl. Discov.*, vol. 18, no. 2, pp. 300–336, 2009.

[9] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, 1976.

[10] Y.-F. Chen, E. R. Gansner, and E. Koutsofios, "A c++ data model supporting reachability analysis and dead code detection," *IEEE Trans. Softw. Eng.*, vol. 24, no. 9, pp. 682–694, 1998.

[11] S. Demeyer, S. Tichelaar, and S. Ducasse, "FAMIX 2.1the FAMOOS information exchange model," *Research report, University of Bern*, p. 11, 2001.

[12] eclipse java development tools, http://www.eclipse.org/jdt/.

[13] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes, "SourcererDB: an aggregated repository of statically analyzed and cross-linked open source java projects," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, 2009, pp. 183–186.

[14] maven 2 central repository, http://repo1.maven.org/maven2/.

[15] C. Lüer and D. S. Rosenblum, "Wren—an environment for component-based development," in *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2001, pp. 207–217.

[16] google code search, http://www.google.com/codesearch.

[17] Koders, http://www.koders.com.

[18] Merobase, http://www.merobase.com.

[19] findJAR, http://www.findjar.com.

[20] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *Software Engineering, IEEE Transactions on*, vol. 31, no. 3, pp. 213–225, March 2005.