Sourcerer - An Infrastructure for Large-scale Collection and Analysis of Open-source Code

Sushil K Bajracharya, Joel Ossher, Cristina V Lopes

University of California Irvine, USA

Abstract

A large amount of open source code is now available online, presenting a great potential resource for software developers. This has motivated software engineering researchers to develop tools and techniques to allow developers to reap the benefits of these billions of lines of source code available online. However, collecting and analyzing such a large quantity of source code presents a number of challenges. Although the current generation of open source code search engines provide access to the source code in an aggregated repository, they generally fail to take advantage of the rich structural information contained in the code they index. This makes them significantly less useful than Sourcerer for building state-of-the art software engineering tools, as these tools often require access to both the structural and textual information available in source code.

We have developed Sourcerer, an infrastructure for large-scale collection and analysis of open source code. By taking full advantage of the structural information extracted from source code in its repository, Sourcerer provides a foundation upon which state of the art search engines and related tools can easily be built. We describe the Sourcerer infrastructure, present the applications that we have built on top of it, and discuss how existing tools could benefit from using Sourcerer.

Keywords: Open Source, Internet-scale code retrieval, Data Mining, Sourcerer, Static Analysis, Software Information Retrieval

1. Introduction

The popularity of the Open Source Software movement has dramatically increased the general availability of free, high quality source code. The repositories that host open source software are growing at an exponential rate [30], and the software itself is seeing increasing usage, from both developers and the general public. With respect to developers, open source software is often used as part of the development infrastructure, as well as in the code itself, in the form of reusable libraries and frameworks [35, 5]. This growth in the usage and availability of open source software provides a rich opportunity for the creation of novel software engineering solutions. However, collecting, analyzing, and actually using such large quantities of source code is quite challenging, which makes building and evaluating any new techniques significantly difficult. In

Email addresses: sbajrach@ics.uci.edu (Sushil K Bajracharya), jossher@ics.uci.edu (Joel Ossher), lopes@ics.uci.edu (Cristina V Lopes)

this paper we present Sourcerer, an infrastructure developed to tackle these key challenges. The paper provides details on the models Sourcerer uses to represent open source code, the content Sourcerer stores, the tools that comprise Sourcerer, and the services that Sourcerer provides. The current version of Sourcerer is designed to work with open source projects developed using Java.

The proliferation of open source software has given rise to two recent trends in the software industry and the academic software engineering research community.

1. **Commercial code search engines**. The growth in the amount of open-source software in public repositories is reflected in the emergence of several commercial code search engines, such as Koders, Krugle and Google Code Search [67, 68, 11]. These code search engines allow developers to use a single site to search billions of lines of source code collected from various repositories on the Web.
2. **Research trends in leveraging large software repositories**. There has been considerable research effort in building specialized search tools for software developers. Studies of developers' activities reveal a routine use of web resources during development, in particular code examples and snippets [24, 36]. Supporting these observations several tools have been proposed that utilize code retrieved from repositories. Examples include recommending APIs [37], code-completion [25], finding reusable code fragments [41], finding and synthesizing API examples [36, 26], and finding application for prototyping [34].

The commercial code search engines go a fair way towards fulfilling software developers' needs for finding relevant open source code. However, their performance is often lacking, as any user can attest. As a result, the software engineering research community has created a number of novel approaches for large-scale code retrieval. A common theme among these approaches is the use of structural information along with the more traditional textual information extracted from the code. However, conducting research that requires this rich structural information presents three major challenges:

1. **Collection:** The primary challenge in collecting source code off the Internet is that there is no standard method of distribution. Open source projects are generally hosted by large open source repositories, such as Sourceforge [69], Google Code Hosting [66], and Apache [2], which rarely provide hooks for performing this type of collection. Ultimately, the best way to obtain the source code is to scrape the download links and version control systems from the project web pages. However, the use of different version control systems, download protocols, and constantly changing format/content of the web pages makes automating this collection process rather tedious.
2. **Analysis:** In order to leverage the structural information available in source code, one has to be able to first extract the information itself. In order to fully extract structural information from source code, the code must be declaratively complete; i.e. all the dependencies must be resolved. Unfortunately, when it comes to source code from the internet, there is no guarantee that the code is declaratively complete; missing dependencies and incomplete files are quite common. Scaling the analysis to thousands of projects further complicates matters, as it eliminates the feasibility of performing any type of manual processing.
3. **Application:** Due to the challenges posed by collection and analysis, it is often impossible to rapidly design and evaluate applications that make use of both textual and structural information extracted from large quantities of source code. The upfront cost of constructing a repository and implementing sound analysis tools makes research in large scale code retrieval non-trivial.

The Sourcerer project explores various applications of the large-scale collection and analysis of open source code. We have developed a research infrastructure that is principally driven by the need to tackle the three major challenges mentioned above. In summary, the Sourcerer infrastructure enables the following:

1. **Collection** of large amount of source code from open source repositories to build a reference repository for research in large scale source code analysis.
2. Automated **analysis** of arbitrary Java source code that exists out in the wild (on the internet).
3. Rapid development and evaluation of **applications** that leverage large amount of preprocessed, cross-linked and aggregated repository of source code.

This paper provides details on important aspects of Sourcerer that enable the above features. The paper is an extension of our earlier publications on Sourcerer [20, 61]. It includes more detailed description on the latest version of the Sourcerer infrastructure and makes the following contributions:

- Provides key details on the design and implementation of the Sourcerer infrastructure that enables collection and analysis of large amount of open source code.
- A summary of applications enabled by Sourcerer's infrastructure and a list of its major contributions in the area of large-scale code retrieval and source code data mining.
- Key details on implementation, configuration, and availability of Sourcerer to motivate its adoption by external researchers.

The Sourcerer Infrastructure primarily consists of five different components:

1. **Models:** A set of abstractions that capture the elements of textual and structural information in Sourcerer's source code collection.
2. **Tools:** A collection of loosely coupled tools responsible for the collection and analysis of source code.
3. **Stored Content:** Artifacts that are produced using various tools and stored locally in Sourcerer's repository.
4. **Services:** A layer of abstraction that enables access to the Stored Content.
5. **Applications:** that leverage the Stored Content accessed via various Services.

Paper Organization: The paper begins by covering the five components that constitute the Sourcerer infrastructure. Sections 2 through 5 capture how we built the infrastructure by describing the models, tools, and services. In Section 6, we discuss the applications built using Sourcerer, which validates the applicability and impact of the Sourcerer infrastructure. Section 7 provides information on availability of the Sourcerer infrastructure for use by other researchers. In Sections 8 and 9 we discuss related and future work, and cover how Sourcerer relates to existing similar tools and platforms. We conclude in Section 10. The Appendix contains details on implementation to motivate external users to obtain, use, and extend the Sourcerer's open source infrastructure.

2. Models

Three models define the basic mechanisms for storing and retrieving information from the source code available in Sourerer's repository.

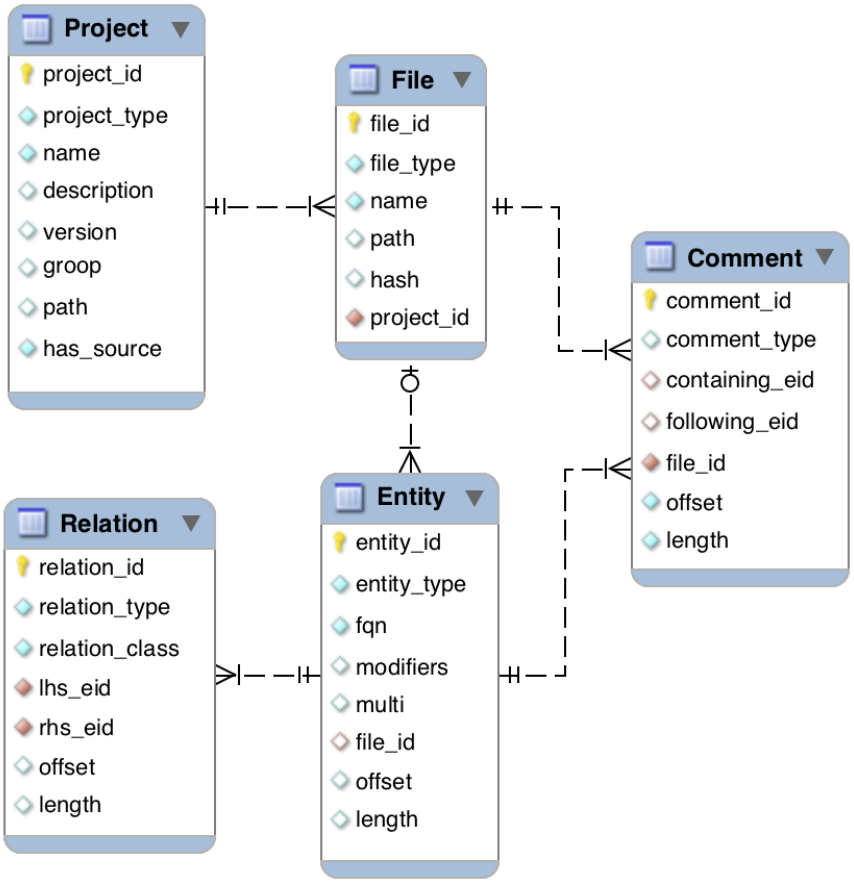

Figure 1: Sourcerer's Relational Model

2.1. Storage Model

The Storage Model captures the layout and structure of the physical files in Sourcerer's local repository. A layered directory structure was chosen for two main reasons. First, it allows projects from the same source to be grouped together, which makes adding or removing content more straightforward. Second, after initially implementing a flat version, we discovered that the file system did not adequately handle having tens of thousands of subdirectories in a single directory. The files collected from open source projects are stored in a folder according to the following template:

```
<repo_root>/<batch>/<id>
```

Above, `<repo_root>` is a folder assigned as the root of Sourcerer's file repository. Given the root folder, the individual project files are stored in a two-level directory structure defined by the path fragment `<batch>/<id>`. Each `<batch>` folder contains a semi-arbitrary collection of projects. For example, a batch could be a crawl from a specific online repository or a collection of fixed number of projects. Inside `<batch>`, another set of folders exist. Each second-level

folder in the local repository, indicated by `<id>` in the above template, contains the contents of a specific project. Each `<id>` directory contains a single file and two sub-directories, as shown below:

```
<repo_root>/<batch>/<id>/project.properties
<repo_root>/<batch>/<id>/download/
<repo_root>/<batch>/<id>/content/
```

Above, `project.properties` is a text file that stores the project metadata as a list of name value pairs. The `download` folder contains the compressed file packages that were fetched from the originating repository (e.g. a project's distribution in Sourceforge). The `content` directory contains the expanded contents of the downloads directory. Once the contents of the download directory have been expanded, the directory itself is usually emptied in order to free up space.

The project contents in the `content` directory can take two different forms, depending on its format in the initial repository. If the project contents are checked out from a remote software configuration management systems (such as svn and cvs), the file located at a relative path `path` in the originating repository (e.g Sourceforge) exists in Sourcerer's file repository at the following absolute path:

```
<repo_root>/<batch>/<id>/contents/<path>
```

If, instead, the project is fetched from a package distribution, a source file can be found in Sourcerer's file repository at the following absolute path:

```
<repo_root>/<batch>/<id>/contents
      /package.<i>/<path>
```

Above, `package.<i>` indicates a unique folder for each i^{th} package that is found in a remote repository. `path` indicates a relative path of a source code file that is found inside the i^{th} archived package, which is unarchived inside the `package.<i>` folder.

Project metadata: The `project.properties` file is a generic project description format that generalizes the project metadata from the online repositories. Many of the attributes in `project.properties` are optional, except for the following:

- `crawledDate`: indicates when the crawler picked up the project information
- `originRepositoryUrl`: URL of the originating repository; e.g. http://sourceforge.net
- `name`: project's name as given in the originating repository
- `containerUrl` : project's unique URL in the originating repository

And, one or both of the following: (i) Information on project's software configuration management (SCM) system indicated by `scmUrl` (ii) Information on project's source package distributed on the originating repository:

- `package.size` indicating total number of packages distributed
- `package.name.`i indicating name of the i^{th} package, where 1 <= i <= *size*, and i indicates a unique integer denoting a package number.

- `package.sourceUrl.`i indicating the URL to get the i^{th} package from the originating repository.

Two sample `project.properties` files showing metadata for two projects is given in Appendix A.

Jar Storage: In addition to the top-level `batch` directories described above, the local repository also contains a single `jars` directory. The jars directory is structured as follows:

```
<repo_root>/jars/project/<jar_path>
<repo_root>/jars/maven/<jar_path>
<repo_root>/jars/index.txt
```

The `project` subdirectory contains all of the jar files that come packaged with the projects in the main repository. This directory is populated by crawling through the repository itself, and copying every jar found. The copying is done so that these jar files can be modified, if necessary, without altering the original projects. The `maven` subdirectory contains a mirror of the Maven2 Central Repository. Lastly, `index.txt` contains an index that maps from the MD5 hash of a jar file to its location in the directory structure. This index is used to link the jar files from the projects to the files contained in the `jars` directory.

The Storage Model provides a standard for storing project files in Sourcerer and is not directly used by applications. The description above will be useful for those interesting in downloading and using the Sourcerer's reference file repository from [54]. Applications rely on other higher-level abstractions to access the contents stored in Sourcerer.

2.2. Relational Model

Sourcerer's Relational Model defines the basic source code elements and the relations between those elements. The metamodel is specific to Java, and is designed to capture its latest version. We decided to go with a language-specific metamodel, rather than a more generic metamodel such as FAMIX [29], as it allows us to more concisely represent Java-specific features. Our model supports a fine-grained representation of the structural information extracted from source code. It also links the code elements/relations with their locations in physical artifacts.

Two major goals guided the design of Sourcerer's relational model. First, it had to be sufficiently expressive as to allow fine-grained search and structure-based analyses. Second, it had to be efficient and scalable enough to include the large amount of code from thousands of open source projects. To meet these two goals we decided to use an adapted version of Chen et al.'s [27] C++ entity-relationship-based metamodel as Sourcerer's relational model for source code. In particular, their decision to focus on what they termed a *top-level declaration* granularity provides a good compromise between the excessive size of finer granularities and the analysis limitations of coarser ones.

Our metamodel has evolved two primary ways from its original version. First, it was adapted to include the features introduced by Java 1.5. While adding these features contributed a fair amount of complexity to the metamodel, we felt that it was necessary given the increasing prevalence of open source code using the features. Second, after receiving some feedback from users, we decided to add local variables to our metamodel, allowing for a slightly finer granularity in our analysis.

The relational model consists of the following five elements: Project, File, Entity, Comment, and Relation.

PACKAGE
CLASS
INTERFACE
ENUM
ANNOTATION
INITIALIZER
FIELD
ENUM CONSTANT
CONSTRUCTOR
METHOD
ANNOTATION ELEMENT
PARAMETER
LOCAL VARIABLE
PRIMITIVE
ARRAY
TYPE VARIABLE
WILDCARD
PARAMETRIZED TYPE
UNKNOWN

Table 1: Entity Types

A **project** model element exists for every project contained in Sourcerer's repository, as well as every unique Jar file. A project therefore contains either a collection of Java source files and jar files, or a collection of class files. A **file** model element represents these three types of files: source (`.java`), jar (`.jar`) or class (`.class`). Both source and class files are linked to sets of **Entities** contained within them, and to the **Relations** that have these entities as their source and target. Jar files, on the other hand, are linked to their corresponding jar projects, which in turn contains all of the **entities** and **relations**.

An **entity** model element either corresponds to an explicit declaration in the source code (e.g. Class, Interface, Method etc), a Java package[1], or Java types that are used but do not correspond to a known explicitly declared type (e.g. Array, Type Variable). An entity type is `UNKNOWN` when the type cannot be determined due to uncertainty in the analysis. Table 1 lists all entity model element types defined in Sourcerer. These types all adhere their standard meaning in Java, as defined in the Java Language Specification (JLS) [32].

A **relation** model element represents a dependency between two Entities. A dependency *d* originating from a source entity *s* to a target entity *t* is stored as a Relation *r* from *s* to *t*. Table 2 contains a complete list of the relation types with a brief description and example for each. All of the relations are binary, linking a source entity to a target. The source entity for a relation is smallest entity that contains the code that triggers that relation. While containment is clear for most of the entities, it should be noted that `FIELD`s are considered to contain their initializer code and `ENUM CONSTANT`s are considered to call their constructors. The source entity is always found within the project being examined. This is not necessarily true of the target entity. It can be a reference to the Java Standard Library or any other external jar. In fact, sometimes it is impossible to resolve the type of the target entity, due to missing dependencies.

A **comment** model element represents the comments defined in the Java source code.

[1]Packages are not considered to be standard declared entities as they do not have a single declaration

Relation	Description	Example
INSIDE	Physical containment	java.lang.String INSIDE java.lang
EXTENDS	Class extension	java.util.LinkedList EXTENDS java.util.AbstractSequentialList
IMPLEMENTS	Interface implementation	java.util.LinkedList IMPLEMENTS java.util.List
	Interface extension	java.util.List IMPLEMENTS java.util.Collection
HOLDS	Field type	java.lang.String.offset HOLDS int
RETURNS	Method return type	java.lang.String.toCharArray() RETURNS char[]
READS	Field read	...String.<init>(java.lang.String) READS java.lang.String.offset
WRITES	Field write	java.lang.String.<init>() WRITES java.lang.String.offset
CALLS	Method invocation	...String.indexOf(int) CALLS java.lang.String.indexOf(int,int)
INSTANTIATES	Constructor invocation	foo() INSTANTIATES java.lang.String.<init>
THROWS	Declared checked exception	java.io.Writer.write(int) THROWS java.io.IOException
CASTS	A cast expression	java.langString.equals(java.lang.Object) CASTS java.lang.String
CHECKS	An instanceof expression	java.langString.equals(java.lang.Object) CHECKS java.lang.String
ANNOTATED BY	Annotation	java.lang.Override ANNOTATED BY java.lang.annotation.Target
USES	Any reference	java.lang.String.<init>() USES char
HAS ELEMENTS OF	Array element type	char[] HAS ELEMENTS OF char
PARAMETRIZED BY	Associated type variables	java.util.List PARAMETRIZED BY <E>
HAS BASE TYPE	Generic base type	java.util.List<java.lang.String> HAS BASE TYPE java.util.List
HAS TYPE ARGUMENT	Generic type argument	java.util.List<java.lang.String> HAS TYPE ARGUMENT java.lang.String
HAS UPPER BOUND	? extends TYPE	<? extends java.util.List> HAS UPPER BOUND java.util.List
HAS LOWER BOUND	? super TYPE	<? super java.util.List> HAS LOWER BOUND java.util.List

Table 2: Relation Types

Figure 2 shows Sourcerer's relational model using an ER-diagram. It shows the five elements of Sourcerer's relational model and a set of attribute for each of them. Table 3 provides the details on all the attributes of the model elements. Figure 2 and Table 3 provide information on how the model elements are linked with each other, and how the attributes in the relational model link the relational model elements with the storage model. For example, Project element's 'path' attribute links it to the physical location defined by the storage model.

Various tools in Sourcerer make use of this information to connect the relational information with the textual content stored in the physical files.

2.3. Index Model

The Index Model complements Sourcerer's relational model by facilitating the application of information retrieval techniques to code entities. Sourcerer's information retrieval component is based on the popular Lucene [6] information retrieval engine, and therefore our index model follows Lucene's general approach. More details on Lucene's content model is available in [59].

Our index model matches a Lucene **document** to each entity in the relational model. A document is made up of a collection of **field**s, each field being a name/value pair. The simplest form of value is a collection of **term**s, where a term is the basic unit for search/retrieval. Terms are extracted from various parts of an entity, and stored in the fields of the document corresponding to that entity.

Fields in Sourcerer's index models can be categorized into five types:

1. Fields for *basic retrieval* that store terms coming from various parts of a code Entity
2. Fields for *retrieval with signatures* that store terms coming from method signatures and also terms that indicate number of arguments a method has
3. Fields storing *metadata*, for example the type of the Entity, so that a search could be limited to one or more types of entities
4. Fields that store information to facilitate *retrieval based on structural similarity* (e.g. fields storing fully qualified names (FQNs) of used entities and terms extracted from similar entities
5. Fields that pertain to some *metric* computed on an entity
6. Fields that store ids of entities for *navigational/browsing queries*

Being based on Lucene, Sourcerer's index model is quite flexible. Depending on a specific search application, an instance of a Sourcerer's index schema can have a subset of various field types listed above. Appendix B shows an example code index schema used in two of Sourcerer's search applications: Sourcerer Code Search engine [18] and CodeGenie [46, 44, 45, 47].

Table 4 presents a subset of the fields available in the Sourcerer index. Sourcerer's search index can be searched using Lucene's query language. The following Lucene query demonstrates how different fields are utilized to express a query that incorporates textual as well as structural information:

```
short_name: (week date)
  AND entity_type: METHOD
  AND m_ret_type_sname_contents: String
  AND m_args_fqn_contents: Date
```

The above query has the following meaning: find a method with the terms `week` and `date` in its short name (or simple name in JLS [32]), that returns a type with short name `String` and takes in an argument type with the term `Date` in its name.

	Description
Project	
project_id	unique identifier for a project
project_type	denotes whether this project represents a crawled project, or a Jar file
name	name of the project as it appears in the originating Internet repository
description	description of the project from the originating Internet repository
version	version of this project as extracted from originating Internet repository
groop	specific field applicable to Maven Jars
path	corresponds to the `<batch>/<id>` path fragment as defined by the storage model
has_source	denotes whether the project contains source files
File	
file_id	unique identifier for a file
file_type	denotes the file's type - source, Jar, class
name	name of the file in the file system
path	corresponds to either `<batch>/<id>/contents/<path>`, or `jars/<jar_path>` as defined by the storage model
hash	unique MD5 hash, applicable for Jars only
project_id	project_id that this file belongs to
Entity	
entity_id	unique identifier for an Entity
entity_type	one of the several code entity types. E.g. CLASS, METHOD etc
fqn	Fully qualified name (FQN) of the entity
modifiers	modifiers defined for the code entity
multi	denotes array dimension, applicable for ARRAY types only
file_id	file_id that this entity is extracted from
offset	start position of this entity in the source file
length	length of this entity in the text (source file)
Relation	
relation_id	unique identifier for a relation
relation_type	one of the several code relation types. E.g. CALLS, EXTENDS etc
relation_class	denotes whether the relation terminates to a library or a local entity
lhs_eid	the source entity that the relation originates from
rhs_eid	the target entity that the relation terminates into
offset	start position in the source entity's corresponding file where this relation exists
length	length of the text in source code where this relation spans
Comment	
comment_id	unique identifier for a comment
comment_type	denotes the comment's type - Javadoc, Block, Line
containing_eid	the immediate code entity that contains this comment
following_eid	the immediate code entity that follows this comment
file_id	file where this comment is found
offset	start position of comment in the source file
length	length of this comment in text (source file)

Table 3: Sourcerer's Relational Model Elements Details

Index Field	Description
Fields for basic retrieval	
fqn_contents	Tokenized terms from the FQN of an entity
short_name	right most fragment of the FQN (w/o method arguments for methods)
Fields for retrieval with signatures	
m_args_fqn_contents	method's formal arguments tokenized into terms after passing each of them through keyword extractor
m_ret_type_sname_contents	short name of the method's return type tokenized
Fields Storing metadata	
entity_type	String representation of entity type. Eg; "CLASS"
Fields for navigation	
fan_in_mcall_local	entity ids of all local callers for a method from the same project

Table 4: Sample search index fields

Our recent publication [21] provides a more elaborate discussion on how Sourcerer's index model is used in a code-retrieval application.

3. Stored Content

The Sourcerer infrastructure maintains a collection of stored content corresponding to each of the three models.

A **File Repository** keeps a collection of files downloaded and fetched from open source repositories in the Internet. The structure of the code repository follows the storage model.

Two different databases store the relational information about the contents in the file repository. First, **ArtifactDB** stores limited information about the jar files found in the repository in order to enable the automated resolution of missing dependencies [62]. Second, **SourcererDB** stores the relational information on all projects, files and code entities that exist in the code repository. Both of the code databases exist as MySql databases whose schemas confirm to Sourcerer's relational model.

A Lucene based **Search Index** is available that stores information about terms extracted from each code entity in the corresponding documents and fields. The search index uses a code index schema following the index model.

4. Services

All the artifacts managed and stored in Sourcerer are accessible through a set of web services. These services provide a layer of abstraction and programmatic access to rapidly build applications that can leverage the underlying content stored in Sourcerer.

Relational Query: Both ArtifactDB and SourcererDB are implemented as MySql databases. They provide direct access to query the underlying structural/relational information in Sourcerer using standard SQL.

Repository Access: This service provides access to the textual content of four of Sourcerer's model elements: files, entities, relations and comments. Repository access is a simple HTTP-based web service that returns the full text when given a unique id.

Dependency Slicing: This service provides dependency slices of the code entities in SourcererDB. A dependency slice of an entity is a program (collection of Java source files) which includes that entity as well as all the entities upon which it depends. Requested slices are packaged into zip files, and should immediately be compilable. The dependency slicing service can take in one or more entity ids and return a zip file containing the collection of sliced/synthesized Java files that the given set of entities depend on.

Code Search: This service implements a query processing and retrieval facility. Client applications (such as CodeGenie [46, 44, 45, 47]) can send queries as a combination of terms and fields and the service returns a result set with detailed information on the entities that matched the queries. The query language is based on Lucene's implementation using which clients can express structural information in the queries. The current code search service is implemented as a customization of the Solr search engine [15]. Solr is a front end for Lucene, and supports a wide range of retrieval functions such as: basic retrieval of code entities based on the vector space model, faceting based on fields defined in the code index schema, similarity searches etc. All these features are available over Sourcerer's code search index. The matching and scoring (ranking) of entities follow Lucene's implementation. Further details on how Lucene/Solr match the query terms in index fields and scores the matched entities is available from our previous publication [21]; a more definitive source is [10]. In summary, a boolean retrieval is performed based on a Lucene query as shown in Section 2.3, then all matched entities (documents) are ranked using the TF-IDF measure [58].

Similarity Calculation: The Similarity Calculation service takes in an entity_id of an entity 'e' and returns a list of other entities that are similar to 'e'. Currently, the similarity calculator can suggest similar entities based on three different measures of usage similarity. For this purpose, the similarity calculator uses the usage information stored in SourcererDB. The Structural Semantic Indexing technique that we presented in [21] makes use of the similarity calculation service. Further details on similarity calculation is available in [21].

Except the Relational Query service, all other services are simple HTTP based services. Currently three services are open to the public. A detailed description of how to use these services is available online [40].

5. Tools

A number of loosely coupled tools are available in the Sourcerer infrastructure. These tools are primarily responsible for collecting and analyzing source code and producing Sourcerer's stored contents.

Code Crawler: Sourcerer consists of a multithreaded plugin-based code crawler that can crawl the web pages in online source code repositories. To adapt with the changes and differences with web pages in different Internet repositories, the crawler follows a plugin based design. A separate plugin can be written targeting the crawl of a repository. This makes it possible to just update the plugin or add new plugins when a different or new web site has to be crawled. Currently the crawler consists of plugins for Sourceforge [69], Java.net [3], Tigris [4], Google Code

Hosting [66], and Apache [2]. The crawler takes a set of root URLs as an input and produces a list of download URLs and version control links along with other project specific metadata. This project specific metadata is in the form as specified by (the `project.properties` file in) the storage model.

Repository Creator: The Repository Creator tool is responsible for parsing the output list from the Code Crawler, filtering noise from the list (e.g. removing duplicate links), and downloading the contents from the online repositories to Sourcerer's local file repository. Given a local file repository's root folder, the repository creator creates the required folder structure and places the contents as specified by Sourcerer's storage model. The repository creator first creates the two level folder structure based on the number of projects it needs to add to the repository. Second, it creates the `project.properties` file describing each project. Third, it fetches the files from remote/original repositories. `project.properties` has metadata about two content sources in remote repositories: (i) Source Configuration Management systems such as svn and cvs, and (ii) downloadable packages such as compressed distributions (zips, tars etc). When an information on a SCM repository is available, the repository creator first tries to check out contents from the respective SCM system. If errors are encountered or if the SCM check out brings no contents, then the repository creator downloads all the packages, given that the information on links to the packages exist in `project.properties`. After the download, the repository creator explodes the archives inside the `content` folder corresponding to the project. The end result of this process is a local Sourcerer repository based on the storage model and that contains contents fetched from remote open source repositories.

Repository Manager: The repository manager tool is responsible for two tasks: (i) library management, and (ii) optimizing the local repository for feature extraction. Under library management, the repository manager creates and maintains a local mirror of all jar files from the Maven2 central repository [2] [13]. It also aggregates all of the jar files from the individual projects into the jars directory, as described above. It then creates an index of all the unique jar files in the repository. These jars can be used to provide missing types to projects in Sourcerer's file repository during feature extraction if needed. Under optimizing the local repository, the repository manager performs tasks such as compressing the contents inside a project's folder, and cleaning the jars' manifest files to avoid problems due to unexpected classpath additions.

Feature Extractor: The Feature Extractor in Sourcerer is responsible for extracting the detailed structural information from the source code files stored in Sourcerer's file repository. The feature extractor is built as a headless Eclipse plug-in, to make use of Eclipse's AST Parser. Before running the feature extractor, the source code is preprocessed to detect missing libraries using import statements. Some additional heuristics are used to be able to fully resolve the bindings in the source code types and links to the libraries. These heuristics are fully explained in our earlier publication [62]. The Repository Manager and the Feature Extractor together implement the required techniques for Automated Dependency Resolution, a key feature available in the Sourcerer infrastructure, that enables feature extraction from large number of open source projects despite missing dependencies and errors. In summary automated dependency resolution works as follows. First, the feature extraction runs through the available projects to detect missing types. It creates the AST representation of code available in the projects and generates a list of missing types reported by the underlying Eclipse parser. From the list of missing types, the feature extractor generates a list of possible FQNs for those types to be found. It then looks

[2]Maven is a build system for Java that provides the facility to fetch required libraries from a central repository [12].

up the ArtifactDB for possible Jar files where the missing FQNs could be found. While doing so it selects the jar files that can provide the maximum number of missing FQNs. Once the jars are selected, they are included in the classpath of the project with missing types and then the feature extractor runs again. This process is repeated until all missing types are found or if no jars could be located for remaining missing types. After this step, the feature extraction does a full extraction of entities and relations from the projects. Our evaluation of automated dependency resolution has shown that it can increase the percentage of declaratively complete projects in Sourcerer from 39% to 69%. Full details of automated dependency resolution is available in our previous publication [62].

Database Importer: This tool allows importing the Feature Extractor's output into the code databases: ArtifactDB and SourcererDB.

Code Indexer: The code indexer tool is responsible to index all code entities in Sourcerer's repository using the textual and structural information available for the entities. The code indexer obtains this information using three services, the File Access Service - to obtain the full text corresponding to a code entity, SourcererDB to retrieve entities and comments related to a code entity being indexed, and Similarity Calculation service to retrieve similar entities. As a result of the indexing process, the code indexer produces a semi-structured full text index based on Lucene [6]. Currently the code indexer is implemented as a customization of the Solr [15] indexing and search system. The code index schema varies based on a particular code search application. To index a code entity, the code indexer can retrieve all or some the following data: the full-text for the corresponding entity, the fully qualified names (FQNs) of related entities, comments of the used libraries, and FQNs of used entities. The search index schema will consist of fields to store the terms corresponding to these data types. The terms are extracted from the FQNs and full-text using code-specific analysis techniques (e.g. camel case splitting, removing language keywords as stop words etc). The code indexer tools consists of several of these code-specific analyzers. Appendix B provide further details on configuring the code indexer.

6. Applications

The services and stored contents in Sourcerer have been used to develop several code retrieval systems and data mining techniques on source code. The data collected in Sourcerer's file repository have enabled large-scale inter-project analysis techniques that have helped strengthen the capability of the infrastructure itself. This section reviews Sourcerer's key features that facilitated its application in the area of large scale code search, analysis and data mining on source code.

Table 5 lists 6 applications of Sourcerer that have produced major research contributions. The applications are listed in a chronological order. Major publications corresponding to each application is listed in the fourth column. These references provide full details on the specific contributions these applications of Sourcerer has made in the area of Internet-scale code retrieval and source code data mining.

The first column in Table 5 denotes the Sourcerer milestone, indicating the version of the Sourcerer's infrastructure used in these applications. The difference in Sourcerer's infrastructure between milestones M1 and M2 is explained at the end of this section.

The last column in Table 5 indicates whether the corresponding Sourcerer application is available online, and provides the reference to the URL for those that are available online.

6.1. Impact

Among the 6 applications in Table 5 Sourcerer Code Search (SCS) Engine, CodeGenie, Structural Semantic Indexing (SSI), and Sourcerer API Search (SAS) fall under the application category of *Internet-scale code retrieval*. The work done in Topic Modeling Source Code is an example of *source code data mining* applications that Sourcerer enables. Finally, Sourcerer Reference Collection (SRC) is an effort towards building *reference collection* for replicable research in large scale code analysis. Together, these applications demonstrate Sourcerer's impact and contribution since its inception.

6.2. Applications Enabled by Infrastructure Features

Reference Collection: Sourcerer's file repository, storage model, and a simple attribute based project metadata format (see Appendix A) provides a basis for creating a large code repository that can store and describe contents fetched from thousands of open source projects. This enabled us to create and release a recent Sourcerer file repository as a reference collection of source code for research in large-scale analysis of source code. The reference repository is available at [54].

Data Mining: The storage model, file repository, relational model, and relational access to SourcererDB allows creating a text-based corpus of source code documents. Each document can have terms extracted from code entities, along with terms extracted from entities that they depend on. This allows rapid construction of a corpus with the bag-of-words representation of documents. The topic modeling work was enabled by the capability of Sourcerer to create many variations of such corpora with little effort.

Code Indexing: The index model, repository access, and the relational query service facilitate rapid construction of a search index. The code indexer, being based on Solr, allows building the search index in a declarative way. Two XML files are needed that specify the exact search index (index schema) fields, and data sources to be used to produce a search index (data sources). A wide range of code-specific text analyzers, implemented as part of the Sourcerer infrastructure, can be specified in the index schema to generate terms required for the index fields. Appendix C provides an example of Sourcerer's declarative code indexing.

Code Retrieval/Search: All of the five Sourcerer's services enable many required features for building code search applications. First, the code search service can process any standard Lucene query to retrieve a set of code entities from the search index. Again, using Solr as the search server facilitates many advanced search techniques. The format of the search result can be customized as per application's need. Usually the search result contains a subset of matched entities called 'hits', each representing a code entity. Each hit is associated an entity id, which can be used to look up information with the other services. This service-based design facilitates building code search tools that are (i) deeply integrated with developers' working environment, and (ii) leverage textual/structural information extracted from a large amount of source code. Codegenie is an excellent example. Codegenie uses the code search service to first search the entities. It then, using the repository access service, fetches the desired code for each entity returned as a search result. Finally, it gets compilable slices of code using the dependency slicing service that can be merged back into a developer's project workspace. Full details on how Codegenie is built on top of Sourcerer's services is given in [46].

Inter-project Structural Analysis: The collection of large number of source code for projects, libraries and automated dependency resolution technique have enabled large scale structural/dependency analysis across projects. The first example of this application is the computation of

global Coderank [19, 18, 48], a measure of popularity for a code entity based on adaptation of Google's Pagerank [43] algorithm on the code-graph created using SourcererDB. In this code-graph, entities represent nodes and relations represent edges. Cross-project links are made by finding the source code implementation for libraries that projects refer to. For example, when a code entity has a relation terminating to a binary (library) entity l, the corresponding source entity l_s can be found in the SourcererDB. This allows us to create a global graph of program dependencies that span projects. The details on creating such cross-project links are available in our previous publication [61].

6.3. Sourcerer Milestones

Both the design and implementation of Sourcerer has been constantly evolving. Because of this, earlier applications of Sourcerer use slightly different versions of the models and the repository. At large, the changes in Sourcerer can be divided into two milestones M1 and M2, based on five key features. Table 6 shows these features. Sourcerer M1 existed from 2006 - 2009, and the relational model only captured the entities and relations in Java as defined in Java Language Specification 1.3. These entities and relations were extracted using a custom Java parser built using the JFlex/CUP software packages. Search applications built with Sourcerer M1 used the schema shown in Appendix B, Table C.7. The repository used with Sourcerer M1 had 4632 projects.

Sourcerer M2 is a major rewrite of the entire Sourcerer infrastructure. The original design was kept intact, but the implementation was different, producing a set of loosely coupled tools (described earlier Section 5) unlike a monolithic tool that implemented the functionalities of all current tools in Sourcerer M1. Sourcerer M2's relational model captures the entities and relations as defined Java 1.5. Sourcerer M2 also contains a new and bigger source repository with around 18,000 projects. Sourcerer M2, has been open sourced since 2009. The work on code retrieval done with Sourcerer M2 uses a different code index schema that is described in [21].

7. Availability and Access

The implementation of the Sourcerer infrastructure is available online as an open source project [9]. Some of the services are available online as listed in Table 5. As mentioned earlier, we have recently released Sourcerer M2's file repository as a reference collection to be used for research in large scale analysis of source code [54]. Currently, researchers from four different universities have downloaded and are using the repository. Appendix D describes a workflow of using Sourcerer; starting from crawling code to running evaluations using tools available in its implementation. We believe this will motivate other researchers to use and extend Sourcerer in their research.

8. Related Work

The work done in code analysis and relational model for source code in Sourcerer is quite similar to the approach taken by many reverse engineering tools and models. For example, FAMIX is a language independent model for describing the static structure of object-oriented software systems [29], and is conceptually compatible with the relational model we use. FAMIXs primary purpose is to support the exchange of information between multiple tools. Where as

Milestone	Application	Key Idea/Contribution	Publications	Web
M1	Sourcerer Code Search Engine	Coderank; Including Structural Information in code retrieval	[19, 18, 48]	Yes [17]
	Topic Modeling Source Code	Extracting concepts and author-topic association	[49, 50, 51, 48]	
		Information-theoretic model of Aspects	[23]	
	CodeGenie	Test-driven code search Dependency Slicing	[46, 44, 45, 47]	Yes [8]
M2	Inter-project Structural Analysis	Cross-project linking for determining global usage statistics	[61]	
	Structural Semantic Indexing	Effective retrieval of API usage examples leveraging usage similarity	[21]	
	Sourcerer API Search	Prototype for Exploratory Code Search using SSI	[22]	Yes [1]
	Sourcerer Reference Collection	Reference Collection for research in large scale analysis of source code		Yes [54]

Table 5: Various Applications of Sourcerer Infrastructure

Milestone	Period	JLS	Index Model	Repo. Size	Feature Extractor	Open Sourced
M1	2006 - 2009	1.3	SCS	4K	Homegrown	No
M2	2009 - 2010	1.5	SSI	18K, ~350	Eclipse-based	Yes

Table 6: Sourcerer Milestones

Sourcerer's relational model's purpose is to represent the structural information in code at the right level of granularity and scale.

Linton's OMEGA system was one of the first to model source code in a relational manner, when back in 1984 he used a relational schema to describe a Pascal-like language [53, 52]. As discussed earlier, Sourcerer's relational model is most closely related to that of Chen et al. [27]. It is also nearly identical to the one used by the back-end repository for [37].

Sourcerer's approach to large scale collection of open source code is, in many ways, similar to Spars-J, a software component repository created by Inoue et al. [42]. Spars-J contains structure and reference information similar to SourcererDB, with the addition of various software metrics. Spars-J also merges similar components, except at the entity, rather than project, level. However the Spars-J public demo appears to be limited to pre-Java 1.5. Although their web-interface allows for searching and browsing of individual files and packages, as well as for references to be followed, it provides no support for direct database or web-service access, making it unsuitable as a foundation to build applications. In addition to Spars-J, there are many other component repositories and code search engines. Merobase, for example, is a commercial component repository that provides a developer API to access its structure-based search [14]. Though it lacks any reference-based information, this still puts it ahead of many other code search engines, such as Google Code Search [11] and Koders [67].

9. Future Work

This section presents existing software engineering tools from a few different areas, and describes how they could have benefited from the Sourcerer infrastructure. We also discuss important areas of improvement that could be made to the Sourcerer infrastructure itself.

9.1. Possible Benefits to Existing Tools

A natural extension to Sourcerer's applications would be to reimplement or interface existing software engineering tools with Sourcerer's repository and services. This can not only ease and improve the development of these tools (or future tools like these) but also provide opportunities for fair and more scientific evaluation/comparison of these tools with a common underlying repository.

Example Recommendation: Holmes et al.'s Strathcona [37] is a tool for using a developer's current structural context to recommend source code examples. Strathcona attempts to match the structural information in the current context against examples from its repository. The information stored in Strathcona's repository is sufficiently similar to that in Sourcerer's that Strathcona could be implemented on top of the Sourcerer infrastructure. This would focus Strathcona's development on the matching heuristics and client integration, while immediately providing access to a very large repository.

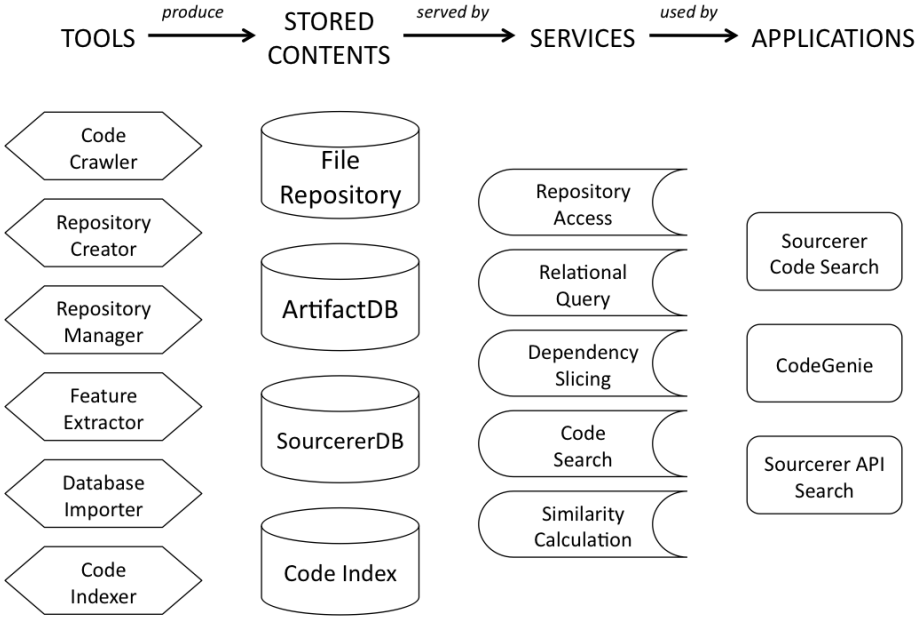

Figure 2: Sourcerer's Overall Architecture

XSnippet [63], Prospector [57] and PARSEWeb [64] are all systems designed to provide examples of object instantiation. Although implementation on top of Sourcerer would provide some benefit to all of them, PARSEWeb would be dramatically improved. Currently PARSEWeb uses Google Code Search to find and download likely examples of object instantiation. These snippets are then analyzed to determine if they contain appropriate invocation sequences. This analysis is complicated by the fact the code snippets are missing most of their external references. PARSEWeb is forced to utilize a variety of heuristic techniques to guess the missing types. Sourcerer is ideally suited for this sort of use, as it can provide snippets where the external references are present, eliminating the errors introduced by the fuzzy analysis.

Information Mining: Both SpotWeb [65] and CodeWeb [60] are tools for detecting API hotspots. If they were to use Sourcerer, hotspots could be detected directly simply by ordering the entities in a jar by the number of incoming relations.

Pragmatic Reuse: Holmes and Walker's approach to reuse [38] shares many similarities with our dependency slicing. While our approach is fully automated, drawing in all necessary dependencies, theirs permits a greater level of customization, allowing developers to exclude dependencies they do not want. In order to achieve this customization, however, a developer must download and import the full project into his workspace. This creates a fair amount of manual overhead, for if there are multiple candidate projects for reuse, the process must be repeated for each one. Furthermore, any unresolved dependencies in the initial project download will remain unresolved in the final result. The combination of their approach with the Sourcerer infrastructure has the potential to eliminate many of these problems. One could construct a reuse plan on a slice returned by our system, further reducing its size, without having to worry about downloading the full project or unrelated or unresolved dependencies.

9.2. Extending Sourcerer

Another area of future work in Sourcerer is to address some of its current limitations, and find opportunities to integrate it with complementary platforms/infrastructure.

Multiple Languages: Currently Sourcerer is designed explicitly to work with the Java programming language. Although Java is quite popular as a language of choice to develop software applications today, there are other languages that are equally popular. Even in a single project, it is common to use multiple languages today. For example, a Java project itself might include scripts written in dynamic languages such as Ptyhon and Groovy. Projects also include other declarative mechanisms to generate and build code that make extensive use of languages such as XML. Therefore, models and tools to analyze and store multiple languages, and more importantly, dependencies between them seem to be an important area that an infrastructure such as Sourcerer needs to improve on. In this regard, existing approaches such as XIRC [31] that allows some form of cross artifact analysis in development environments seem relevant. One strategy could be to apply techniques used in XIRC and scale it up to make it work with Sourcerer.

Addressing Evolution: The current design of the Sourcerer infrastructure does not have a strong support to address the evolution of the source code. Source code in open source repositories are constantly modified and updated, at least in active projects. Sourcerer's current design requires it to create a separate project for each unique version of project it needs to store. In terms of storage and analysis this is not the most efficient approach to deal with evolving software. A careful survey of the requirements and application of evolutionary data on source code can guide Sourcerer's future approach in dealing with evolution. One possibility would be to extend Sourcerer's current models with elements that describe evolution, as seen in the meta-model used by the Small Project Observatory project [56].

Considering non-code artifacts: Source code is not the only artifact that is available in open source repositories. It is well known that during a software development lifecycle various kinds of non-source artifacts are produced and used. For example, data on issues, bugs, documentation, authorship, developer's activities/ history etc. While Sourcerer is not primarily designed to address these kinds of non-source artifacts, it is important to find an approach to connect Sourcerer's models and services with repositories (for example Hipikat [28] that store these non-code artifacts.

Integrating with other open source analysis platforms: Another avenue of extension would be to integrate Sourcerer with open-source quality monitoring platforms, such as Alitheia Core [33], that aim to collect source code metrics from open source projects on the Internet. Such integration could enable yet new applications, that would leverage open-source quality metrics with information available in Sourcerer's stored contents.

The FLOSSmole project is a collaborative effort to collect and analyze large amount of open source project data [39]. Sourcerer is more comprehensive in terms of tools, services and depth of information it covers for analyzing the source code available in open source project. FLOSSmole's database include more project specific metadata, and had information on larger number of open source projects. On one hand, the data from FLOSSmole project could be used, instead of the data produced by Sourcerer's code crawler, to construct a Sourcerer file repository. On other hand, the depth of information that can be produced using Sourcerer can further enhance the kinds of analyses that FLOSSmole currently supports. Therefore, integrating Sourcerer with FLOSSmole could widen the scope and impact of both projects.

10. Conclusion

This paper presented the design and implementation of the Sourcerer infrastructure. The primary components and the basic working principle of Sourcerer can be summarized in the architecture diagram presented in Figure 2. Sourcerer's architecture consists of a set of loosely coupled tools that produce various stored contents. These stored contents confirm to the design specified by three models: Storage, Relational and Index. Five different services provide a layer of abstraction for programmatic access to the underlying stored content. These services enable the development of applications that require access to preprocessed information from large amount of source code (in both textual and structural form).

The Sourcerer infrastructure makes three primary contributions in the area of large-scale collection, analysis and application of open source code:

1. **Collection**: The storage model, a common metadata format for describing open source projects across various online repositories, and the plugin-based design of the code crawler enable collection of large-amount of code from the Internet.
2. **Analysis**: A large collection of libraries in Sourcerer's file repository, and the automated dependency resolution implemented in the feature extractor increases the number of declaratively complete projects to 69% from 39% [62]. This makes it possible to extract fine-grained structural information from a large number of projects even in the presence of missing dependencies.
3. **Applications**: Sourcerer consists of several services that provide a layer of abstraction and programmatic access to textual, structural, and information retrieval models of source code. This enables rapid development and evaluation of novel techniques for source code retrieval and data mining. By providing the required models, tools, and services - it serves as a testbed for rapid implementation and evaluation of large scale code analysis tools such as Internet-scale code search tools. The successful application of Sourcerer in applications such as Sourcerer Code Search engine [19, 18, 48] , Codegenie [46, 44, 45, 47], and Structural Semantic Indexing validates [21] Sourcerer's impact in this area.

Besides above three contributions, Sourcerer provides key resources to conduct replicable research in large scale code analysis by (i) making available a large collection of source code in a standard format [54], and (ii) by making the infrastructure's implementation available as an open source project [9]. Finally, we hope that the details on the Sourcerer's implementation available in the Appendix will motivate external usage and contribution to the Sourcerer project.

Acknowledgements: Sourcerer is a large systems project, incorporating over 8 people (up to 4 at a given time). The first two authors are responsible for all work done in Sourcerer since milestone M2. We acknowledge contributions made by other in the earlier versions of Sourcerer infrastructure.

Contributors to Sourcerer milestone M1: Yimeng Dou implemented the first basic version of the Sourcerers crawler that was replaced by a more robust and plugin-based code crawler by Huy Huynh. Trung Ngo was instrumental in contributing to the first design and implementation of the overall Sourcerer infrastructure and developed the context sensitive analysis, core indexing components, and the 'Coderank' [18, 19, 48] technique that was part of the graph-based heuristic to improve code retrieval in SCS. Paul Rigor contributed to deployment and running our tools on various system configurations ranging from a single machine to clusters of machines. Erik Linstead was responsible for running the experiments required for evaluation of retrieval schemes

in SCS [19, 48], and was the primary contributor to data-mining applications of Sourcerer [49, 50, 51, 48, 23]. Otavio Lemos, contributed to the idea of of Test-driven Code Search (TDCS) and implemented CodeGenie [46, 44, 47, 45]. Last but not the least, Pierre Baldi led the data-mining research in Sourcerer and supported the infrastructure's implementation since its inception.

References

[1] Sourcerer wiki page on sourcerer api search tool *http://wiki.github.com/sourcerer/Sourcerer/sas*.

[2] Web Site for Apache Software Foundation. *http://apache.org*.

[3] Web site for Java.net. *http://java.net*.

[4] Web site for Tigris. *http://tgris.org*.

[5] Black Duck's web page with Koders usage information. http://corp.koders.com/about/, February 2010.

[6] Lucene web site. *http://lucene.apache.org*, Jan 2010.

[7] Web Location of Galago Search Evaluation Tool. *http://code.google.com/p/galagosearch/source/browse/tags/galagosearch-1.04/galagosearch-core/src/main/java/org/galagosearch/core/eval/Main.java*, July 2010.

[8] Web page for codegenie. *http://sourcerer.ics.uci.edu/codegenie*, July 2010.

[9] Web page for Sourcerer's github repository. *http://github.com/sourcerer/Sourcerer*, June 2010.

[10] Web Page on Apache Lucene Scoring. *http://lucene.apache.org/java/2_4_0/scoring.html*, Mar 2010.

[11] Web site for Google Code Search. *http://www.google.com/codesearch*, July 2010.

[12] Web site for maven. *http://maven.apache.org*, July 2010.

[13] Web site for maven's central repository. *http://repo1.maven.org/maven2/*, July 2010.

[14] Web site for merobase. *http://www.merobase.com/*, July 2010.

[15] Web Site for Solr *http://lucene.apache.org/solr/*, July 2010.

[16] Web site for Sun Grid Engine. *http://gridengine.sunsource.net*, July 2010.

[17] Web page for sourcerer project and the sourcerer code search engine. *http://sourcerer.ics.uci.edu*, July.

[18] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. pages 681–682, New York, NY, USA, 2006. ACM Press.

[19] S. Bajracharya, T. Ngo, E. Linstead, P. Rigor, Y. Dou, P. Baldi, and C. Lopes. A study of ranking schemes in internet-scale code search. Technical Report UCI-ISR-07-8, UCI ISR, November 2007.

[20] 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*, pages 1–4. IEEE Computer Society, 2009.

[21] S. Bajracharya, J. Ossher, and C. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. 18th International Symposium on the Foundations of Software Engineering, 2010.

[22] S. Bajracharya, J. Ossher, and C. Lopes. Searching api usage examples in code repositories with sourcerer api search. In *SUITE 2010: Second International Workshop on Search-driven Development - Users, Infrastructure, Tools and Evaluation*, 2010.

[23] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 543–562, Nashville, TN, USA, 2008. ACM.

[24] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the 27th international conference on Human factors in computing systems*, pages 1589–1598, Boston, MA, USA, 2009. ACM.

[25] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceegings of FSE*, pages 213–222, Amsterdam, The Netherlands, 2009. ACM.

[26] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Fundamental Approaches to Software Engineering*, pages 385–400. 2009.

[27] Y. Chen, E. R. Gansner, and E. Koutsofios. A c++ data model supporting reachability analysis and dead code detection. *IEEE Trans. Softw. Eng.*, 24(9):682–694, 1998.

[28] D. Cubranic, G. Murphy, J. Singer, and K. Booth. Hipikat: a project memory for software development. *Software Engineering, IEEE Transactions on*, 31(6):446–465, 2005.

[29] S. Demeyer, S. Tichelaar, and S. Ducasse. Famix 2.1 — the famoos information exchange model. Technical report, University of Bern, 2001.

[30] A. Deshpande and D. Riehle. The total growth of open source. In *Fourth Conference on Open Source Systems*. Springer Verlag, 2008.

[31] M. Eichberg, M. Mezini, K. Ostermann, and T. Schafer. Xirc: A kernel for cross-artifact information engineering

in software development environments. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 182–191, Washington, DC, USA, 2004. IEEE Computer Society.

[32] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The*. Addison Wesley, 3 edition, June 2005.

[33] G. Gousios and D. Spinellis. Alitheia Core: An extensible software quality monitoring platform. In *Proceedings of the 31st International Conference on Software Engineering*, pages 579–582. IEEE Computer Society, 2009.

[34] M. Grechanik, K. M. Conroy, and K. A. Probst. Finding Relevant Applications for Prototyping. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 12. IEEE Computer Society, 2007.

[35] J. Hammond. What developers think. http://www.drdobbs.com/architect/222301141, Jan 2010.

[36] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 13–22, Newport, Rhode Island, USA, 2007. ACM.

[37] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 117–125, New York, NY, USA, 2005. ACM Press.

[38] R. Holmes and R. J. Walker. Lightweight, Semi-automated Enactment of Pragmatic-Reuse Plans. In *Proceedings of the 10th international conference on Software Reuse: High Confidence Software Reuse in Large Systems*, pages 330–342, Beijing, China, 2008. Springer-Verlag.

[39] J. Howison, M. Conklin, and K. Crowston. FLOSSmole: A collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, 2006.

[40] http://sourcerer.ics.uci.edu/services. Sourcerer web services.

[41] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Softw.*, 25(5):45–52, 2008.

[42] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: relative significance rank for software component search. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 14–24, Washington, DC, USA, 2003. IEEE Computer Society.

[43] R. M. Lawrence Page, Sergey Brin and T. Winograd. The pagerank citation ranking: Bringing order to the web. *Stanford Digital Library working paper SIDL-WP-1999-0120 of 11/11/1999 (see: http://dbpubs.stanford.edu/pub/1999-66)*.

[44] O. Lemos, S. K. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, 2009.

[45] O. A. L. Lemos, S. K. Bajracharya, and J. Ossher. CodeGenie:: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 917–918, Montreal, Quebec, Canada, 2007. ACM.

[46] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, P. C. Masiero, and C. V. Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Information and Software Technology*, (To appear).

[47] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. CodeGenie: using test-cases to search and reuse source code. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 525–526, Atlanta, Georgia, USA, 2007. ACM.

[48] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, Apr. 2009.

[49] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining concepts from code with probabilistic topic models. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 461–464, Atlanta, Georgia, USA, 2007. ACM.

[50] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining eclipse developer contributions via Author-Topic models. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 30. IEEE Computer Society, 2007.

[51] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining internet-scale software repositories. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 929–936, Cambridge, MA, 2008. MIT Press.

[52] M. A. Linton. Queries and views of programs using a relational database system. Technical Report UCB/CSD-83-164, EECS Department, University of California, Berkeley, 1983.

[53] M. A. Linton. Implementing relational views of programs. In *SIGPLAN Not.*, page 132–140, New York, NY, USA, 1984. ACM Press.

[54] C. V. Lopes, S. K. Bajracharya, J. Ossher, and P. F. Baldi. UCI Source Code Data Sets. [http://www.ics.uci.edu/~lopes/datasets/] Irvine, CA: University of California, Bren School of Information and Computer Sciences.

[55] Lucid Imagination. Lucidworks for solr certified distribution reference guide. http://www.lucidimagination.com/Downloads/LucidWorks-for-Solr/Reference-Guide, 2010.

[56] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The Small Project Observatory: Visualizing software ecosystems. *Science of Computer Programming*, 75(4):264–275, Apr. 2010.
[57] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 48–61, New York, NY, USA, 2005. ACM Press.
[58] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 1 edition, July 2008.
[59] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action*. Manning Publications, 2 edition, July 2010.
[60] A. Michail. Code web: data mining library reuse patterns. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 827–828, Toronto, Ontario, Canada, 2001. IEEE Computer Society.
[61] 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*, pages 183–186, 2009.
[62] J. Ossher, S. Bajracharya, and C. Lopes. Automated dependency resolution for open source software. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 130–140, Cape Town, South Africa, 2010.
[63] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 413–430, New York, NY, USA, 2006. ACM Press.
[64] 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*, pages 204–213, Atlanta, Georgia, USA, 2007. ACM.
[65] S. Thummalapenta and T. Xie. SpotWeb: detecting framework hotspots and coldspots via mining open source code on the web. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 327–336, 2008.
[66] Web Site for Google Code Hosting. *http://code.google.com/projecthosting*.
[67] Web site for Koders. *http://www.koders.com*, 2010.
[68] Web site for Krugle. *http://www.krugle.com*, 2010.
[69] Web Site for Sourceforge. *http://sourceforge.net*.

Appendix A. Sample project.properties files

The example below shows metadata description for a project crawled from Google code hosting.

```
00 #Thu Sep 24 16:15:01 PDT 2009
01 releaseDate=null
02 name=dlctarea1
03 category=DLC, Java, Netbeans, FileChooser
04 languageGuessed=Java
05 versionGuessed=$SCM
06 scmUrl=svn checkout
     http\://dlctarea1.googlecode.com/svn/trunk/
       dlctarea1-read-only
07 license=GNU General Public License v2
08 keywords=null
09 sourceUrl=null
10 exractedVersion=$SCM
11 projectDescription=Tarea n\uFFFD 1
12 fileExtensions=null
13 originRepositoryUrl=http\://code.google.com
14 containerUrl=http\://code.google.com/p/dlctarea1/
15 contentDescription=null
16 crawledDate=2009-Sep-23
```

The example below shows metadata description for a project crawled from Sourceforge. This description includes list of downloadable packages along with the SCM URL, unlike above that only has the SCM URL.

```
00 #Tue Sep 22 23:53:26 PDT 2009
01 versionGuessed=$SCM
02 containerUrl=http\://sourceforge.net/cvs/?group_id\=25954
03 projectDescription=\  The Virtual Data Center project is
    building an operational, open-source, digital library to
    enable the sharing of quantitative research data, and the
    development of distributed virtual collections of data
    and documentation.
04 contentDescription=cvs
05 package.releaseDate.2=2006-04-17 20\:23
06 package.releaseDate.1=2006-04-17 20\:23
07 license=GNU General Public License GPL
08 languageGuessed=Java , Perl
09 keywords=null
10 exractedVersion=$SCM
11 package.sourceUrl.2=
    http\://downloads.sourceforge.net/thedata/
      VDC-1.0.4-11.Fedora.4.tar
12 crawledDate=2009-Feb-28
13 package.sourceUrl.1=
    http\://downloads.sourceforge.net/thedata/
      VDC-1.0.4-11.RedHat.4AS.tar
14 package.versionGuessed.2=VDC 1.0.4 Final
15 package.versionGuessed.1=VDC 1.0.4 Final
16 fileExtensions=null
17 category=Education , Dynamic Content , Indexing/Search ,
    Other/Nonlisted Topic , Scientific/Engineering , Archiving
18 package.name.2=thedata-vdcfedora
19 package.extractedVersion.2=VDC 1.0.4 Final
20 package.name.1=thedata-vdcredhatas
21 package.extractedVersion.1=VDC 1.0.4 Final
22 sourceUrl=null
23 releaseDate=null
24 originRepositoryUrl=http\://sourceforge.net
25 scmUrl=
     cvs -d\:pserver\:anonymous@thedata.cvs.sourceforge.net\
       :/cvsroot/thedata login;
     cvs -z3 -d\:pserver\:anonymous@thedata.cvs.sourceforge.net\
       :/cvsroot/thedata co -P modulename
26 package.size=2
27 name=thedata
```

These two examples show that Sourcerer's project metadata format enables description of projects and contents across various online repositories.

Appendix B. Sourcerer Code Index Schemas

Table C.7 shows Sourcerer's code index model used in Sourcerer Code Search and CodeGenie.

Appendix C. Declarative Code Indexing with Solr

This section shows how Sourcerer provides a declarative way of defining and creating a code index using Solr. The syntax, specification and declarative approach is due to the use of Solr.

Three kinds of specifications are required for creating a search index in Sourcerer. First, every index field needs to define a field type.

Field Declaration: The XML fragment below shows a field declaration named `sname_contents` that has a field type of `FT_sname_contents`. The field stores the terms extracted from a simple name of code entity, the text source from which the field is populated is a FQN of a code entity. This is retrieved from SourcererDB, details are shown later in this section.

```
<field name="sname_contents" type="FT_sname_contents"/>
```

Field Types: A number of fields can be declared once a field type is defined. A field type specifies the analysis done on the textual content that will be stored in the corresponding field. For example, the field type `FT_sname_contents` defined below denote the field type to store the terms extracted from the simple name of Java code entities. These terms are extracted from the FQNs of entities that are retrieved from SourcererDB.

```
01 <fieldType name="FT_sname_contents"
02       class="solr.TextField">
03   <analyzer type="index">
04     <tokenizer
05       class="solr.KeywordTokenizerFactory" />
06     <filter class="sourcerer.FqnFilterFactory"
07       extractSig="0" shortNamesOnly="1" />
08     <filter class=
09      "sourcerer.NonAlphaNumTokenizerFilterFactory"/>
10     <filter class=
11       "sourcerer.CamelCaseSplitFilterFactory"/>
12     <filter class=
13       "sourcerer.LetterDigitSplitFilterFactory"
14       preserveOriginal="1" />
15     <filter class="solr.LowerCaseFilterFactory"/>
16   </analyzer>
17   <analyzer type="query">
```

The field type definition shown above indicates that a FQN (an input text to the field) is treated as a Keyword token (due to the use of the class shown in Line 05). Therefore, the analysis starts with the full FQN in its original form. The FQN is then piped through five filters. Each filter takes a token stream (a stream of terms), processes it and produces a new token stream. The terms produced at the end is what gets stored in a field. The values of the attribute 'class' in the XML fragment above (Lines 06 - 15) denote a Java implementation of a token filter to use. Classes with the prefix 'sourcerer' are code specific token filters available as part of Sourcerer's implementation. In the above example, the first filter (lines 06 and 07) extracts the simple name from the FQN, a second filter (Line 08) splits the simple name using non-alphanumeric characters as delimiters, the third filter does a camel case split on the new token stream, the fourth filter split the term further based on letter-digit transition while keeping the terms already present in the token stream, and the last filter converts every term to a lowercase version. The output from this last filter gets stored as terms corresponding to a simple name of a Java entity.

Data Source Configuration: Another step in declarative specification for indexing is the configuration of data sources that provide the content from which terms will be extracted and stored in the index fields. The XML snippet below shows how SourcererDB and the repository access service is used to feed data into some index fields.

```
01 <document name="entity">
02   <entity name="code_entity" pk="entity_id"
03     query="SELECT entity_id, fqn FROM entities
04       WHERE entity_type in
05           ('CLASS','METHOD','CONSTRUCTOR')''
06     transformer="sourcerer.Transformer"
07     code-server-url=
08       "http://sourcerer-url/file-server">
09   <field column="fqn" name="sname_contents" />
```

```
10   <field column="fqn" name="fqn"/>
11   <field column="entity_id" name="entity_id" />
12   <field column="code_text" name="full_text"/>
```

Above, Line 02 indicates that a document in the index represents a code entity. Line 03 specifies a SQL query that will fetch a set of columns from SourcererDB. Lines 09 - 12 indicate the number of fields that will be stored for the corresponding document. The values for the attribute 'column' specifies the column name corresponding to the SQL query that start in Line 03. The values for the attribute 'name' indicates the index field the contents from the column will be stored into. Line 12 indicates that the column to be used to get the content for the index field `full_text` is named 'code_text' that does not exist in the SQL query starting at Line 03. This column is added by the class named `sourcerer.Transformer`, which acts as a data transformer. The class is responsible to fetch the source code corresponding to an entity. It uses the value from the column `entity_id`, sends an HTTP request to the repository access service available at the URL specified in Line 07, and provides the text as an input to be stored for index field `full_text`.

This demonstrates how easily a searchable index can be created once the services are accessible in Sourcerer. The XML snippets shown above are simplified versions of what exists in Sourcerer's indexing tool implementation. The first two snippets come from the schema definition file (named schema.xml in a Solr installation), the third snippet comes from the data import file (named db-data-config.xml in a Solr installation). More details on configuring these files are available from standard references on Solr [55, 15].

Appendix D. Example Sourcerer Workflow

This section presents how various parts of Sourcerer's implementation (available in Github at http://github.com/sourcerer/Sourcerer) can be used in evaluation of a code retrieval scheme. It starts from running the code crawler to build a file repository to running a client application. The workflow resembles the setup/scenario of evaluation done for Structural Semantic Indexing as described in [21].[3]

Creating a Sourcerer File Repository: The two projects found under *infrastructure/tools/core* are needed to create the Sourcerer file repository. The projects *codecrawler* and *core-repository-manager* are used to crawl and build the file repository. *codecrawler* implements the code crawler tool, and *core-repository-manager* implements the Repository Creator tool. Binaries to run these tools are found in the Github repository folder *bin*.

Once a file repository is in place, the following projects are needed to perform automated dependency resolution: *repository-manager* - that implements the Repository Manager tool, *extractor* - that implements the Feature Extractor, *model* - that implements the Relational Model, *database* - that implements the Database Importer tool, 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 Sourcerer file repository created using the Repository Creator Tool, 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,

[3]The repository used in [21] was partly created manually. The files were added to a folder structure confirming Sourcerer's storage model.

Index Field	Description
Fields for basic retrieval	
contents	Default field to be searched. combination of fqn_contents, fqn_fragments and fqn
fqn	Fully qualified name of an entity, untokenized
fqn_contents	Tokenized terms from the FQN of an entity.
fqn_fragments	untokenized form of FQN fragments. Eg: for a FQN; foo.bar.SomeOne.method(int) - the fragments are; foo bar SomeOne method
short_name	right most fragment of the FQN (w/o method arguments for methods)
short_name_contents	tokenized form of short_name
comments	The collected text (untokenized) from an entity's comments
Fields for retrieval with signatures	
m_sig_args_fqn	method's formal arguments FQN in format org.foo.Arg1,x.y.arg2,z.arg3,...,Y.argn
m_sig_args_sname	method's formal arguments short name in format arg1,arg2,arg3,...,argn
m_sig_ret_type_sname	short name of the method's return type, rightmost of FQN
m_sig_ret_type_fqn	FQN of the method's return type
m_args_arity	Number of arguments a method has
m_args_fqn_fragments	method's arguments' FQN fragments (not ordering info)
m_args_fqn_contents	method's formal arguments tokenized into terms after passing each of them through keyword extractor
m_args_sname_contents	Terms extracted from the short names of each method argument
m_ret_type_fqn_fragments	FQN fragments of return type
m_ret_type_contents	method's return type FQN tokenized into terms after passing it through keyword extractor
m_ret_type_sname_contents	short name of the method's return type tokenized
Fields Storing metadata	
modifiers	modifiers applied on this entity
is_binary	true for entities coming from jars/classes and those that are missing, false for entities coming from source code
entity_type	String representation of entity type. Eg; "CLASS"
Fields storing metrics	
complexity	currently mapped to lines of code
coderank_local	local coderank
Fields for navigation	
entity_id	the entity id of this entity in the code database
parent_id	the parent entity that contains this entity (via the inside relation)
fan_in_all_local	all incoming entity ids
fan_out_all_local	all outgoing entity ids
fan_in_mcall_local	entity ids of all local callers for a method from the same project
fan_out_mcall_local	entity ids of all local callees for a method from the same project

Table C.7: Fields in Sourcerer's code index schema used in Sourcerer Code Search and Codegenie

Step	Task	run *Script*/`Class`/service	found in *folder* / `package`	part of..
1	crawl projects	*run-code-crawler.sh*	bin/	Code Crawler
2	create repository layout	*repo-folder-creator.sh*	bin/	Repository Creator
3	populate repository	*content-fetcher.sh*	bin/	Repository Creator
4	prepare repository for extraction	`Main.java`	`edu.uci.ics.sourcerer.repo`	Repository Manager
5	feature extraction for ArtifactDB	`Extractor.java`	`edu.uci.ics.sourcerer.extractor`	Feature Extractor
6	populate ArtifactDB	`Main.java`	`edu.uci.ics.sourcerer.db.tools`	Database Importer
7	run ArtifactDB	MySql Database		Relational Access
8	full feature extraction	`Extractor.java`	`edu.uci.ics.sourcerer.extractor`	Feature Extractor
9	populate SourcererDB	`Main.java`	`edu.uci.ics.sourcerer.db.tools`	Database Importer
10	run SourcereDB	MySql Database		Relational Access
11	get raw usage data for Hamming Distance/Tanimoto Coefficient based similarity	*run-raw-usage-writer.sh*	bin/	Similarity Calculation
12	get final usage data for Hamming Distance/Tanimoto Coefficient based similarity	*run-filtered-usage-writer.sh*	bin/	Similarity Calculation
13	prepare index for TF-IDF similarity	*run-index.sh*	infrastructure/services/solr-config	Code Indexer
14	run TF-IDF based similarity service	Solr Instance	infrastructure/services/solr-config	Similarity Calculation
15	run HD/TC based similarity service	`SimilarityServer.java`	`edu.uci.ics.sourcerer.server.similarity`	Similarity Calculation
16	run Repository access	`FileServer.java`	`edu.uci.ics.sourcerer.server.file`	Repository Access
17	run final indexing	*run-index.sh*	infrastructure/services/solr-config	Code Indexer
18	run code search service	Solr Instance		Code Search
19	run retrieval evaluation for SSI	`EvalSnippetsEntryPoint.java`	`edu.uci.ics.sourcerer.evalsnippets.client`	Applications

Table C.8: An example Sourcerer Workflow

with the `aggregate-jar-files` and `create-jar-index` flags respectively. In each case, the input repository (`input-repo`) must be specified.

Building ArtifactDB: As specified by the storage model, the `jars` folder in the file repository contains the jar files used to resolve the missing dependencies. The contents of this folder and the corresponding Jar-index can be 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 file repository is ready with the Jar-index, the Feature Extractor tool is used to extract the types provided by these artifacts. The Feature Extractor is run as an Eclipse application, class `Extractor` inside package `edu.uci.ics.sourcerer.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 file repository (`input-repo`), and the output repository (`output-repo`). After the extraction is complete, an instance of ArtifactDB can be populated with the type information. This is done using class `Main`, inside package `edu.uci.ics.sourcerer.db.tools`, in the *database* project.

Automated Dependency Resolution: Once ArtifactDB is ready, the Feature Extractor tool 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 `Extractor`. If dependency resolution is used, the database containing the ArtifactDB must also be specified as an argument to `Extractor`.

Building SourcererDB: Populating SourcererDB is done in exactly the same way as ArtifactDB.

Running Services: The File Repository and SourererDB make up two content sources needed for indexing. To proceed with indexing, first the file repository needs to be served using the Repository Access service. An implementation of this service is available in the Github repository under *infrastructure/ services/ file-server*.

To create the search index required for retrieval evaluation described in [21], two different similarity calculation services are needed. First, that computes the Tanimoto Coefficient/Hamming Distance based similarity. This part is implemented in *infrastructure /services /similarity-server*. The *similarity-server* requires a data source with information on API usage. This can be provided as a plain text file. A tool to generate this usage information is available inside project *infrastructure /tools /core /machine-learning*. Two classes produce the required usage statistics. First, class `UsagePreCalculatorRunner` inside package `edu.uci.ics.sourcerer.ml.db.tools` calculates the usage details on all entities and APIs. Second, class `FilteredUsageCalculator` found in the same package finally prepares the required usage statistics by filtering outliers (e.g.; APIs that are used by only one entity).

The second similarity calculation service that computes the TF-IDF based similarity requires a running instance of a minimal code search service that is available under *infrastructure /services /solr-config*. The procedure required to run the indexing tool and the code search service is same as for running these tools to do the final retrieval evaluation. The only difference lies in writing the schemas and data configuration specification.

Indexing: The implementation of the code indexer and code search tools is available under *infrastructure /services /solr-config*. To run the code index tool, first a Solr installation needs to be configured by writing schemas as described Appendix B. The script that runs the search tool is available in the Github project *solr-config*, and is named *runindex.sh*. This script runs the indexing in a cluster of machines running the Sun Grid Engine distributed computing platform [16]. The script can easily be modified to run the indexing in a single machine.

Retrieval and Evaluation: The retrieval step requires four different services to be running: Repository Access (*file-server* in Github), Relational Query (an instance of SourcererDB), similarity server for HD/TC similarity (*similarity-server*) in Github, and similarity server for TF-IDF similarity (a configured instance of *solr-config*). Once these services are up and running, an instance of the retrieval evaluation tool can be run to execute different queries and collect the required statistics that get generated. The retrieval tool is available in Github under project *infrastructure /apps /codesearch*. The implementation of the retrieval tool used in [21] is available as a GWT (Google Web Toolkit) based web application starting at class `EvalSnippetsEntryPoint` in package `edu.uci.ics.sourcerer.evalsnippets.client`.

The output of running the retrieval evaluation tool can be processed using a tool available in Galago Search project [7]. Finally, the output from Galago Search project provides the necessary data that can be analyzed to measure the retrieval performance. To ease this analysis, handy scripts to be used with the R statistical programming environment are available in Sourcerer's Github location *research /api-location /evaluation*.

Summary: The workflow described above demonstrates that Sourcerer provides an end-to-end solution for large scale collection, analysis and retrieval of code. A collection of tools and services are required for that purpose. The implementation of these tools and services are found in Sourcerer's github repository but can be difficult to trace through in the order they need to be used. Therefore, in Table C.8, we present a summary of the workflow described above in 19 steps that need to be performed in the given order. The table shows the name and location of the implementation for scripts, tools and services corresponding to each step. It also shows the part of the Sourcerer's architecture that each

implementation belongs to.