Meditor: Inference and Application of API Migration Edits

Shengzhe Xu
Computer Science
Virginia Tech
Blacksburg, United States
shengzx@vt.edu

Ziqi Dong
Software Engineering
Northeastern University
Shenyang, China
alanziqidong@gmail.com

Na Meng
Computer Science
Virginia Tech
Blacksburg, United States
nm8247@vt.edu

***Abstract*—Developers build programs based on software libraries. When a library evolves, programmers need to migrate their client code from the library's old release(s) to new release(s). Due to the API backwards incompatibility issues, such code migration may require developers to replace API usage and apply *extra edits (e.g., statement insertions or deletions)* to ensure the syntactic or semantic correctness of migrated code. Existing tools extract API replacement rules without handling the additional edits necessary to fulfill a migration task. This paper presents our novel approach, $Meditor$, which extracts and applies the necessary edits together with API replacement changes.**

$Meditor$ has two phases: inference and application of migration edits. For edit inference, $Meditor$ mines open source repositories for migration-related (MR) commits, and conducts program dependency analysis on changed Java files to locate and cluster MR code changes. From these changes, $Meditor$ further generalizes API migration edits by abstracting away unimportant details (e.g., concrete variable identifiers). For edit application, $Meditor$ matches a given program with inferred edits to decide which edit is applicable, customizes each applicable edit, and produces a migrated version for developers to review.

We applied $Meditor$ to four popular libraries: Lucene, CraftBukkit, Android SDK, and Commons IO. By searching among 602,249 open source projects on GitHub, $Meditor$ identified 1,368 unique migration edits. Among these edits, 885 edits were extracted from single updated statements, while the other 483 more complex edits were from multiple co-changed statements. We sampled 937 inferred edits for manual inspection and found all of them to be correct. Our evaluation shows that $Meditor$ correctly applied code migrations in 218 out of 225 cases. This research will help developers automatically adapt client code to different library versions.

***Index Terms*—API migration edits, program dependency analysis, automatic program transformation**

I. Introduction

As software libraries evolve, migrating client code between library releases can be difficult and time-consuming. A recent article reported that Google developers spent about 9 years migrating all their codebases from `proto-1` to `proto-2` APIs [2]. Such difficulty of code migration is mainly due to *API backwards incompatibility* issues: when library developers evolve software, they sometimes introduce *API breaking changes* that make client code fail to compile or run [24], [33], [49], [53]. To handle the compilation or execution errors, developers of client code have to manually locate the usage of breaking APIs and explore alternative code for replacement.

Manually migrating code between library releases is tedious and error-prone. Even though some libraries provide change logs or release notes [14] to document how a new release (e.g., L_n) is different from the prior release (e.g., L_{n-1}), such documentation is insufficient. This is because while release notes focus on differences between *adjacent* library versions, we observed client code to be often migrated between *nonadjacent* versions. When there is no sufficient documentation providing the needed guidelines, developers have to extensively search for solutions or discuss issues on technical websites [3], [4], [10]–[12]. Even though developers went though such painful process, they could still make mistakes when migrating code and introduce bugs to previously mature code [47].

Existing tools provide limited support for automatic API migration [23], [25], [40], [45], [52], [56]. They compare versions of a library or client code to infer API mappings without handling any surrounding edit required by the API replacement. Cossette et al. studied the nature of API incompatibilities and identified some API migration patterns not supported by any existing tool [24]. For instance, when a method API evolves to take an additional parameter, e.g., "$foo() \rightarrow foo(int\ v)$", current tools only capture the API correspondence but do not care about how to prepare a value for v before the function call [24]. Additionally, existing techniques only suggest API mapping rules without automatically migrating code.

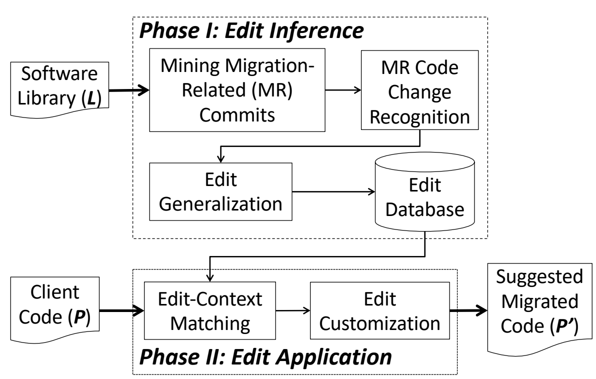

Fig. 1: $Meditor$ consists of two phases: Phase I infers API migration edits given versions of a library (L), and Phase II applies the inferred edits to a given client application (P).

This work was supported by ONR Grant N00014-17-1- 2498.

This paper presents a new approach, *Meditor*, to generate and apply **API migration edits (i.e., API replacements + related edits)** based on developers' migration changes in open source projects. As shown in Fig. 1, *Meditor* contains two phases. Given versions of a library L, Phase I identifies open source projects that use the library, and locates any migration-related (MR) commit in projects' version history. For each MR commit, *Meditor* identifies and groups MR code changes via syntactic program differencing [30] and program dependency analysis. After generalizing each group of MR changes by abstracting concrete identifier usage, *Meditor* derives API migration edits and saves them to a database.

Phase II takes in a client program (P) that uses L. It enumerates all edits in the database to decide which edit is applicable to P. If the tentative context matching between any edit and P succeeds, *Meditor* applies those edits and produces a migrated version (P') accordingly. In this way, *Meditor* can help inexperienced developers find the API migration edits applied by other developers, and apply similar edits to transform client code.

We applied *Meditor* to four widely used libraries: Lucene [13], CraftBukkit [9], Android SDK [5], and Commons IO [7]. *Meditor* generated 153, 931, 268, and 16 unique edits for individual libraries. Among these edits, (1) 885 edits require for single statement updates, (2) 189 edits involve multi-statement changes that modify program data flows but preserve the control flows, and (3) 294 edits modify both control and data flows. We sampled 937 inferred edits for manual checking and found all of them to be correct, which indicates *Meditor*'s great capability of edit inference.

To evaluate the edit application capability of *Meditor*, we created a data set of 87 examples. Each example includes multiple code snippets showing the same migration pattern. Within each example, we used *Meditor* to generate an edit from one snippet and applied the edit to the other snippets. *Meditor* correctly migrated code in 218 out of 225 cases.

In summary, this paper makes the following contributions:

- We developed *Meditor*, a novel approach to infer and apply API migration edits. Different from prior work, *Meditor* infers API replacement rules together with co-applied edits and automates edit application.
- We developed a novel algorithm that flexibly locates and groups MR code changes in commits where migration-related changes are co-applied with unrelated changes.
- We conducted a large-scale study with *Meditor* and observed interesting phenomena, including (1) *Meditor* revealed undocumented rules and (2) many programs were migrated between nonadjacent library releases.
- *Meditor* correctly inferred and applied edits in most scenarios. By inferring domain knowledge from human-written changes in migrated code, *Meditor* can mimic the coding practices to suggest similar code migration.

II. A Motivating Example

This section overviews our approach with exemplar changes drawn from open source projects [6], [8]. Suppose that a developer Alex wants to migrate code between releases of Lucene. To obtain migration suggestions from *Meditor*, Alex needs to first provide versions of the library such that *Meditor* can retrieve any related migration edit stored in the database. If there is no edit extracted so far, *Meditor* crawls GitHub projects to search for any project that (1) uses Lucene and (2) has any commit updating the release information of Lucene.

```
1. - boolean recreate = !IndexReader.indexExists(indexPath);
2. - indexWriter = new IndexWriter(indexPath, getAnalyzer(analyzer), recreate);
3. + Directory dir = FSDirectory.open(new File(indexPath));
4. + Analyzer an = getAnalyzer(analyzer);
5. + IndexWriterConfig iwc = new IndexWriterConfig(LUCENE_VERSION, an);
6. + iwc.setOpenMode(OpenMode.CREATE_OR_APPEND);
7. + indexWriter = new IndexWriter(dir, iwc);
```

Fig. 2: Migrating H2Fulltext.java from `lucene-2.3.2` to `lucene-4.7.0` [16]

Fig. 2 presents the code changes applied in one identified commit drawn from revisions to nexeo [8]. In this example, multiple statements were changed together because nexeo was migrated from Lucene 2.3.2 to Lucene 4.7.0. We highlight the deleted code with red and mark it with "-". Similarly, the added code is highlighted with green and marked with "+". According to the figure, the old version invokes:

- two Lucene APIs (i.e., `IndexReader.indexExists(...)` and `IndexWriter` constructor), and
- one user-defined method `getAnalyzer(...)`.

However, the new version invokes:

- four Lucene APIs (i.e., `FSDirectory.open(...)`, `IndexWriterConfig` constructor, `IndexWriterConfig.setOpenMode(...)`, and `IndexWriter` constructor),
- one user-defined method `getAnalyzer(...)`, and
- one JDK API `File` constructor.

Compared with the old `IndexWriter` constructor (see line 2), the new constructor (see line 7) takes two instead of three parameters. Consequently, when updating the usage of this API, developers also modified the parameter preparation logic (line 1 and lines 3-6).

Edit Inference. To infer the migration edit or pattern demonstrated by Fig. 2, *Meditor* first extracts any replaced API whose signature belongs to the old release but not to the new one (e.g., the `IndexWriter` constructor), and then exploits control and data dependencies to correlate the API replacements with surrounding co-applied edit operations (e.g., statement insertions and deletions). In this way, *Meditor* obtains a cluster of MR edited statements Ch. Next, to generalize an MR edit that is applicable to programs using different variables, *Meditor* replaces concrete identifiers used in Ch with symbolic names. As shown in Fig. 3, the created symbolic names (e.g., `v_7` and `v_7_boolean`) not only preserve the data flows of original identifiers, but also record type information to facilitate later edit application. *Meditor* stores all inferred edits in a database to enable edit query and comprehension.

Edit Application. Given a program P to migrate between versions L_i and L_j of Lucene, *Meditor* queries the database

t_o:

```
boolean v_7 = !IndexReader.indexExists(v_0_String);
v_6 = new IndexWriter(v_0_String, m_0(v_2_String), v_7_boolean);
==========Replaced by==========
```

t_n:

```
Directory v_1 = FSDirectory.open(new File(v_0_String));
Analyzer v_3 = m_0(v_2_String);
IndexWriterConfig v_5 = new IndexWriterConfig(c_0_Version,
 v_3_Analyzer);
v_5_IndexWriterConfig.setOpenMode(OpenMode.CREATE_OR_APPEND);
v_6=new IndexWriter(v_1_Directory, v_5_IndexWriterConfig);
```

Fig. 3: A migration edit generated by $Meditor$. Notice that `v_7` and `v_7_boolean` actually correspond to the same concrete variable. We attached type information to the latter one to facilitate template comprehension.

```
1. - boolean create = !IndexReader.indexExists(_directory);
   ... // unchanged edit-irrelevant code
2. - idxWriter = new IndexWriter(_directory, analyzer, create);
3. + Directory v_1 = FSDirectory.open(new File(_directory));
4. + Analyzer v_3 = analyzer;
5. + IndexWriterConfig v_5 = new IndexWriterConfig(c_0_Version, v_3);
6. + v_5.setOpenMode(OpenMode.CREATE_OR_APPEND);
7. + idxWriter = new IndexWriter(v_1,v_5);
```

Fig. 4: Code migration changes suggested by $Meditor$

for any edit matching the version numbers. For each found edit, $Meditor$ establishes context matching between P and the edit; if the matching succeeds, $Meditor$ concretizes the edit for migration suggestion. Fig. 4 presents an exemplar set of migration changes suggested by $Meditor$ for a program that Alex intends to migrate from Lucene 2.3.2 to Lucene 4.7.0. According to Fig. 4, P is different from the original inference example in Fig. 2 in two ways. First, the used variables are different (e.g., `_directory` vs. `indexPath`). Second, no user-defined method is invoked by P. Dispite the differences, $Meditor$ managed to suggest code changes for Alex to review.

Although existing migration tools at most infer and suggest many-to-many API mappings between L_o and L_n, they are insufficient for two reasons. First, the mappings do not indicate how the data and control dependencies among old APIs are replaced by those among new APIs. Second, current tools do not automate edit application to further reduce developers' workload. $Meditor$ overcomes both limitations.

III. Approach

As shown in Fig. 1, there are two phases in $Meditor$. In this section, we first summarize the steps in each phase and then describe each step in detail (Section III-A-Section III-E).

Phase I: Edit Inference

- Given versions of a library L, $Meditor$ mines open source projects on GitHub for any commit that updates the version number of L in a build file (e.g., pom.xml), obtaining commits $C = \{c_1, c_2, \ldots, c_l\}$.
- $Meditor$ processes each commit to identify and cluster **MR code changes**. Each cluster of MR edited statements demonstrate one migration pattern, denoted as $Ch = \{G_o, G_n\}$, where G_o and G_n are edited statement groups separately from the old and new versions.
- From Ch, $Meditor$ abstracts away project-specific details (e.g., concrete variable identifiers) and derives a general **API migration edit** $E =< t_o, t_n >$, where t_o and t_n are code templates in the old and new versions.

Phase II: Edit Application

- Given P to migrate from L_i to L_j, $Meditor$ queries its database for edits between the versions. For each found edit $E = \{t_o, t_n\}$, $Meditor$ tentatively matches P with t_o; if a matching is found, $Meditor$ records the mappings of constants, variables, methods, and expressions.
- With those mappings, $Meditor$ concretizes t_n to create updated code, suggesting a revised version P' for review.

A. Mining Migration-Related (MR) Commits

Given the jar files of multiple releases for L, $Meditor$ searches the software repositories of a list of 602,249 Java projects on GitHub [38]. In each project repository, $Meditor$ scans the latest version of software for any usage of L in the build file. Different build systems (e.g., Ant, Maven, and Gradle) require developers to use distinct build files to specify library dependencies. Our research focuses on the pom.xml files in Maven projects and build.grade files in Gradle projects because of the popularity of Maven and Gradle [1], [54].

```
    <dependency>
        <groupId>org.apache.lucene</groupId>
        <artifactId>lucene-core</artifactId>
-       <version>3.0.2</version>
+       <version>4.0-SNAPSHOT</version>
        <type>jar</type>
        <scope>compile</scope>
    </dependency>
    <dependency>
```

Fig. 5: A pom.xml file with a library version updated [17]

In particular, if the build file of a project refers to L, $Meditor$ explores the repository to find any commit updating the version number of L. Intuitively, when developers update library version information, they may also apply API migration changes in the same commit. We gathered such commits as candidate MR commits, denoting them as $C = \{c_1, \ldots, c_n\}$. Fig. 5 shows an exemplar updated pom.xml file, which replaces Lucene 3.0.2 with Lucene 4.0-SNAPSHOT.

B. MR Code Change Recognition

Suppose that the before- and after- versions of each MR commit are (V_o, V_n), and they are separately based on two library releases (L_o, L_n). To precisely locate MR code changes in one commit, we need to solve two technical challenges:

- **Tangled Changes** are unrelated changes applied in one commit for multiple tasks, such as bug fixing, library migration, and feature addition [20], [34]. *Untangling MR code changes and irrelevant changes is challenging but crucially important for precise edit inference.*
- **Change Intent** explains why developers change code in certain ways. When developers use new APIs to replace

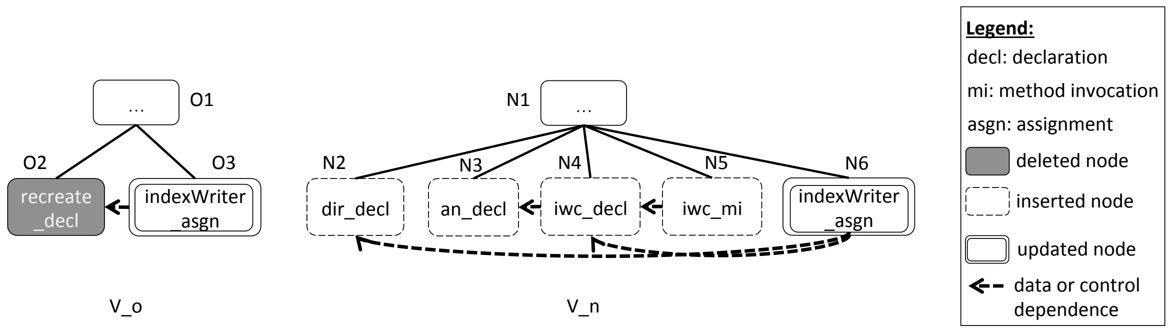

Fig. 6: Syntactic program differencing between V_o and V_n

old API usage, the code changes may be applied for tasks other than library migration. *Inferring the change intent of API replacement is also challenging but vital.*

To tackle the challenges, $Meditor$ first uses syntactic program differencing to extract Java code changes from each commit (Section III-B1). Among the changes, $Meditor$ locates any statement update caused by the library version change and considers them to be MR (Section III-B2). Next, $Meditor$ uses the MR updated statements as centers to cluster relevant statement insertions or deletions via control or data dependencies, separating MR changes from dependency-irrelevant changes (Section III-B3). Finally, for each cluster of MR code changes, $Meditor$ checks whether the changes were applied in a semantic-preserving way; if not, $Meditor$ does not further infer any MR edit because those changes might be applied to fulfill other tasks instead of library migration (Section III-B4).

*1) **Syntactic Program Differencing**:* To detect and represent Java code changes, $Meditor$ uses ChangeDistiller [30]—a tree differencing algorithm—to compare the old and new versions of each changed Java source file. In particular, ChangeDistiller creates an AST for each version, i.e., ($tree_o$, $tree_n$), and compares the trees; it generates an AST edit script that may contain four kinds of statement-level changes.

- **delete (Node u)**: Delete node u.
- **insert (Node u, Node v, int k)**: Insert node u and position it as the $(k+1)^{th}$ child of node v.
- **move (Node u, Node v, int k)**: Delete u from its current position and insert u as the $(k+1)^{th}$ child of v. This operation changes u's position.
- **update (Node u, Node v)**: Replace u with v. This operation changes u's content.

The scripts produced by ChangeDistiller serve two purposes. First, if a script is empty, we can remove the corresponding Java file from further processing because no syntactic change was applied. Second, when a script is non-empty, we focus our analysis on the identified edited statements. For the motivating example in Fig. 2, ChangeDistiller compares ASTs (as shown in Fig. 6) and outputs the following edit operations:

1. `update (O3, N6)`
2. `insert (N2, N1, 0)`
3. `insert (N3, N1, 1)`
4. `insert (N4, N1, 2)`
5. `insert (N5, N1, 3)`
6. `delete (O2)`

*2) **Identification of Obvious MR Edit Operations**:* Some edit operations are obviously related to migration because they use obsolete APIs. To recognize outdated APIs used in V_o, $Meditor$ scans the APIs invoked by each updated statement and resolves the type bindings. If a statement in V_o calls an API that belongs to L_o but not L_n, the related statement update is MR. To resolve bindings, $Meditor$ uses Eclipse ASTParser to generate an AST for each Java source file, and queries the generated AST for binding information. For our example in Fig. 2, one updated statement invokes the old `IndexWriter` constructor in V_o (line 2) and calls the new constructor in V_n (line 7). As the invoked APIs are separately defined by L_o and L_n, we consider this statement update to be MR.

```
- aboutBody.append(Html.fromHtml(getString(R.string.about_text,
    app_name, versionName, YAPI.GetAPIVersion())));
+ String text = getString(R.string.about_text, app_name,
    versionName, YAPI.GetAPIVersion());
+ Spanned html;
+ if (android.os.Build.VERSION.SDK_INT >=
    android.os.Build.VERSION_CODES.N) {
+   html = Html.fromHtml(text, Html.FROM_HTML_MODE_LEGACY);
+ } else {
+   html = Html.fromHtml(text);
+ }
+ aboutBody.append(html);
```

Fig. 7: Migrating AboutDialog.java from Android Level 23 to Level 25 [15]

Additionally, we found some releases of the Android library, which deprecate APIs without removing them but still require for MR changes. To successfully detect such MR changes, we implemented another heuristic in $Meditor$. As illustrated by Fig. 7, when an `if`-statement is added to check whether a version number field (e.g., `VERSION.SDK_INT`) satisfies a condition, $Meditor$ scans both branches to decide whether (1) one branch purely invokes APIs declared in both V_o and V_n, and (2) the other branch invokes any API uniquely defined by V_n. If so, the `if`-statement insertion is also MR.

*3) **Edit Operation Correlation**:* To reveal any additional statement insertion or deletion required by API replacements or obvious MR edit operations, $Meditor$ correlates identified MR edit operations with co-applied edit operations leveraging the program dependencies (i.e., data and control dependencies) between edited statements in V_o and V_n:

- *Data Dependence:* Statement x is data dependent on statement y if x uses a variable defined by y. In Fig. 2, line 2 is data dependent on line 1 because line 2 uses the variable `recreate` declared in line 1.
- *Control Dependence:* Statement x is control dependent on y if x may or may not execute depending on a decision made by y. In Fig. 7, line 5 is control dependent on line 4, because whether the `then`-branch executes depends on the evaluation of the `if`-condition.

If an edited statement $stmt$ has direct data or control dependence relation with an MR edited statement, $Meditor$ considers $stmt$ to be MR as well. For the example in Fig. 2, because line 3 was inserted to prepare the value of `dir`—a variable used for migration-related API replacement, the inserted statement is also MR. In this way, $Meditor$ reveals MR edited statements iteratively until (1) all edited statements are labeled as MR, or (2) no more edited statement depends on or is depended on by any revealed MR edited statement. We denote the correlated edited statements with $Ch = \{G_o, G_n\}$, where G_o and G_n are groups of edited statements separately from V_o and V_n. This step intends to filter out migration-irrelevant changes before $Meditor$ infers MR edits. In our implementation, we exploited a widely used static analysis framework—WALA [18]—to identify the control and data dependencies between statements.

4) ***Semantic Checking****:* We believe that MR edits usually refactor code to solve API backwards incompatibility issues. Given $Ch = \{G_o, G_n\}$, this step checks whether G_o and G_n are semantically equivalent in order to infer developers' change intent. If the program semantics are different, developers might have replaced API usage not purely for library migration. For such cases, $Meditor$ does not infer any MR edit from the given recognized MR changes.

It is almost impossible to accurately reason about the semantic equivalence between two arbitrary code snippets. Therefore, we developed an intuitive approach to approximate semantic equivalence checking by comparing the input and output variables of code snippets. Given a snippet s, the **input variable set (I)** includes the variables defined elsewhere but used by s, while the **output variable set (O)** contains variables defined by s but used elsewhere. Intuitively, if code changes preserve semantics, both the input and output variable sets should match between G_o and G_n.

$Meditor$ uses data flow analysis to create the input and output sets of G_o and G_n: (I_o, O_o) and (I_n, O_n). If both the input and output sets match, $Meditor$ concludes that the applied MR changes preserve semantics. For our example in Fig. 2, since $I_o = I_n = \{\texttt{indexPath}, \texttt{analyzer}\}$ and $O_o = O_n = \{\texttt{indexWriter}\}$, $Meditor$ considers the two versions equivalent. Although our semantic checking approach is not sound or complete, based on our experience, $Meditor$ usually infers MR edits without errors.

C. Edit Generalization

This step generalizes an API migration edit $E = < t_o, t_n >$ from each cluster of edited statements $Ch = (G_o, G_n)$. Notice that the clustered edited statements contain concrete identifiers for API-irrelevant methods, variables, literals, and expressions. To ensure the generality of any inferred edit, $Meditor$ abstracts away such unimportant program-specific editing details. Specifically, $Meditor$ first checks the binding information of each identifier. If an identifier refers to a library API, $Meditor$ keeps the identifier as is because the reference demonstrates API usage; otherwise, a symbolic name is generated to replace all occurrences of the concrete identifier. For instance, as shown in Fig. 2 and Fig. 3, the statement

```
indexWriter = new IndexWriter(indexPath, getAnalyzer(
            analyzer), recreate);
```

is generalized to:

```
v_6 = new IndexWriter(v_0_String, m_0(v_2_String),
      v_7_boolean);
```

In the generalization, variables (e.g., `indexWriter`) are consistently replaced by symbolic names starting with “`v`” (e.g., `v_6`). The library API `IndexWriter(...)` is kept as is. The user-defined method `getAnalyzer(...)` is replaced by a symbolic name starting with “`m`” (i.e., `m_0`), such that this project-specific method information is not propagated to the template.

$Meditor$ saves all generalized edits in a database for edit search and improvement. When novice developers are curious about the migration edits between certain library releases, they can query the database with release numbers. When experienced developers find some migration edits to be improperly represented or missing in the database, they can also manually modify the inferred edits or insert new ones to the database based on their domain knowledge.

D. Context Matching

Given P to migrate from L_o to L_n, $Meditor$ queries the database for any mined edit from L_o to L_n. For each obtained edit $E = < t_o, t_n >$, $Meditor$ matches P with t_o in two steps.

1) Statement Matching: $Meditor$ compares t_o with P at the statement level. For any statement template $s_t \in t_o$, $Meditor$ identifies all library APIs used by s_t and searches for any statement $s_p \in P$ invoking the same set of APIs. For each statement s_p identified in this way, if the concrete identifiers and expressions in s_p can also match the symbolic names in s_t, $Meditor$ records the pair (s_t, s_p) as a candidate match. Correspondingly, the matches between abstract identifiers in s_t and concrete identifiers/expressions in s_p are also recorded. For instance, the first statement in Fig. 3 matches line 1 in Fig. 4. Thus, the corresponding identifier mappings are recorded as `(v_7, create)` and `(v_0_String, _directory)`.

2) Dependency Matching: When t_o contains multiple statements and each statement s_t has one or more matches in P, $Meditor$ further leverages the dependency edges in t_o to query for any correspondence in P. In particular, after matching individual statements between Fig. 3 and Fig. 4,

TABLE I: Client project data extracted for four subject libraries

	Lucene	CraftBukkit	Android SDK	Commons IO	Total
# of commits with MR code changes	49	556	136	10	751
# of client projects holding the refined commits	36	299	120	10	465
# of snippets with MR code changes	247	1,864	328	19	2,458

$Meditor$ retrieves the control and data dependencies between statements in t_o, which information is illustrated by the dashed lines in Fig. 6. Then $Meditor$ applies program dependency analysis to the code snippet in Fig. 4 to check whether the concrete statements have the same dependency relationship as template statements; if so, t_o and P also have their program dependencies matched.

Once all statements and dependencies in t_o are consistently matched by at least one code snippet in P, $Meditor$ considers E to be applicable to P. This step serves two purposes. First, it determines whether an edit is applicable to a program. Second, if an edit is applicable, the created mappings between concrete and abstract statements will enable edit customization. Notice that since $Meditor$ uses program dependencies to link individually matched statements, it can flexibly match t_o with noncontinuous statements in P if the unmatched statements standing between matched statements have no dependency relation with any edited code. As shown in Fig. 4, even if there is edit-irrelevant code between the edited code in P, $Meditor$ matches code with t_o to enable further edit application.

E. Edit Customization

To suggest a program P' after migration, $Meditor$ replaces all symbolic names in t_n with concrete identifiers to customize the edit; it then replaces the matching code of t_o in P with the customized code. For instance, if a statement $s_t \in t_o$ is replaced by multiple statements in t_n, then the concrete code matching s_t is also replaced by the related customized code.

IV. Evaluation

This section describes our data set (Section IV-A), and presents our evaluation on $Meditor$'s effectiveness of edit inference (Section IV-B) and edit application (Section IV-C).

A. Data Set

To create the data set, we conducted a preliminary study. We blindly crawled program commits in GitHub projects for any library version update in `pom.xml` files. If (1) the version numbers of a library are frequently updated in such commits, and (2) there are code changes co-applied in these commits to replace API usage, then we included the library into our data set. In this way, we found four libraries: Lucene [13], CraftBukkit [9], Android SDK [5], and Commons IO [7].

Table I presents the extracted client project data for these libraries. In total, we identified 49, 556, 136, and 10 commits for individual libraries, which contain MR code changes. The extracted commits distribute among 36 Lucene-based projects, 299 CraftBukkit-based projects, 120 Android SDK-based projects, and 10 Commons IO-based projects. These numbers indicate that MR code changes popularly exist in open source projects. Because each commit can have multiple groups of MR code changes co-applied, we located 247, 1,864, 328, and 19 snippets with MR changes applied.

B. Effectiveness of Edit Inference

From the extracted code snippets with MR code changes (see Section IV-A), $Meditor$ infers 153, 931, 268, and 16 unique MR edits for different libraries.

To ensure the quality of extracted edits, the first two authors checked 153 edits for Lucene, 500 edits for CraftBukkit, 268 edits for Android SDK, and 16 edits for Commons IO. They manually compared the inferred edits with corresponding MR code changes to decide whether each edit is correctly generated. When unsure about certain edits, we had discussions to achieve consensus. We found all these 937 edits to be correctly inferred. It means that the automatic approach aligns well with our manual practice of generalizing edits from MR changes.

Finding 1: *We sampled 937 unique MR edits inferred by $Meditor$ and found all of them to be correct.*

To further characterize the inferred edits, we (i) classified them into three categories based on the extraction complexity, (ii) identified the most frequent release pairs for migration, and (iii) compared the documented edits with inferred edits between a pair of library releases.

1) Edit Categorization: To facilitate discussion of the extracted edits, we classified them into three categories based on how complex it was to extract the edits.

Single *(Sin):* Only one single statement or expression is updated, such as modifying an API name. Existing approaches can detect such changes.

Block *(Blo):* A block of statements (e.g., one or more contiguous statements) are replaced by another block of statements. The extraction of such multi-statement edits involves data dependency analysis, but no control dependency analysis. Existing tools cannot fully handle such edits, because they focus on API invocation replacements but ignore any surrounding MR change (e.g., line 4 in Fig. 2).

Multi-Blocks *(MB):* One block of statements are replaced by multiple blocks of statements or vice versa, with the control flow changed. The extraction of such multi-statement edits involves both control and data dependency analysis. No existing tool handles such edits because they do not track how MR changes influence control or data dependencies.

Table III presents the numbers of edits extracted for different libraries. Among the three categories, Sin contains the largest number of edits. This is understandable, because library developers usually try to simplify migration tasks for client code when API breaking changes have to be introduced. On

TABLE II: The 10 most frequent library release pairs of migration

	Lucene		CraftBukkit		Android SDK		Commons IO	
	Release pair	**# of snippets**	**Release pair**	**# of snippets**	**Release pair**	**# of snippets**	**Release pair**	**# of snippets**
1	3.0.2-4.0	24	1.5.1-1.5.2	129	19-21	63	2.1-2.4	7
2	3.6.2-4.8.1	16	1.6.4-1.7.2	122	21-22	41	2.0-2.5	3
3	3.6.0-4.1.0	15	1.6.2-1.6.4	109	22-23	25	1.1-1.2	2
4	3.6.2-4.9.0	15	1.6.1-1.6.2	102	21-23	19	1.3.2-2.4	2
5	4.1.0-4.2.0	14	1.5.2-1.6.1	100	17-23	18	2.0.1-2.4	2
6	3.0.2-3.1.0	12	1.7.2-1.7.5	96	18-19	18	1.0-2.4	1
7	2.9.2-3.2.0	11	1.7.9-1.7.10	60	19-20	14	1.3-1.4	1
8	4.0-3.6.0	11	1.4.6-1.4.7	57	19-23	12	2.2-2.4	1
9	3.6.1-4.0	10	1.7.5-1.7.8	56	23-24	10		
10	2.3.1-2.9.3	8	1.4.5-1.4.6	53	24-25	10		

TABLE III: Edits extracted for different libraries

	Lucene	**Craft-Bukkit**	**Android SDK**	**Commons IO**	**Total**
Sin	92	621	159	13	885
Blo	29	129	31	0	189
MB	32	181	78	3	294
Total	153	931	268	16	1,368

the other hand, there are still many Blo and MB edits that are challenging for previous tools to handle. Developers are quite likely to need more support to apply Blo and MB edits than Sin, because these edits involve complicated interactions between multiple APIs and surrounding context.

Finding 2: *35% of the extracted edits belong to either Blo or MB. Different from prior work, $Meditor$ can extract all these types of nontrivial edits, demonstrating great capability of edit inference.*

2) Most Frequent Release Pairs with Migration Edits: When projects were migrated between library releases, some of the releases required for more MR changes than the others. In the scenario where a project is migrated from one release to another, we name the original release **migration source**, and the new release **migration target**. Such source and target releases delimit the edits required to fulfill migration tasks.

Table II presents the 10 most frequent release pairs in different libraries that require for migration edits. We observed three interesting phenomena. First, *not every library provides official release notes to describe migration changes*. For instance, CraftBukkit has no release note. Developers of client code are on their own to explore migration edits. This observation implies the necessity of $Meditor$, which infers the domain knowledge of migration edits from some developers' code and applies the knowledge to help other developers.

Second, *migrations seldom occurred between consecutive releases*. We compared Lucene's top 10 release pairs with the software official release list [14], and found only one pair to contain consecutive releases: `lucene4.1.0-lucene4.2.0`. The other nine pairs consist of nonconsecutive releases. This phenomenon indicates the importance of our research. While library release notes document migration edits between adjacent releases, the edits revealed by $Meditor$ can help with migrations between nonadjacent releases.

Third, *migrations sometimes downgraded the library usage.* Although most migration tasks began with libraries' lower releases (more dated) and ended up with higher releases (more recent), there are tasks that updated API usage in the opposite direction. For instance, the 8^{th} most frequent pair of Lucene has the source `lucene-4.0` and the target `lucene-3.6.0`. While existing release notes focus on library upgrading changes, $Meditor$ can also help developers downgrade library usage.

Finding 3: *$Meditor$ can help developers migrate code when (1) there is no library release note, (2) developers migrate code between nonadjacent releases, or (3) they downgrade library usage.*

3) Case Study: With a pair of adjacent library releases, we are curious how the edits inferred by $Meditor$ compare with rules documented in the release note. Thus, we conducted a case study for a frequent release pair mentioned in Table II. We compared the extracted edits for `lucene4.1.0-lucene4.2.0` with the release note of Lucene 4.2.0 [14]. This version pair was chosen because (i) the two releases are consecutive; (ii) there are a good number of edits (i.e., 14) inferred by $Meditor$; and (iii) a comparative number of edits (i.e., 16) are mentioned in the note.

Table IV presents the edit distribution among different change categories. The 16 edits (12 changes in backwards compatibility policy + 4 API changes) in the release note belong to 6 categories of changes: 2 categories of type API changes, 2 categories of method API changes, and 2 categories of field API changes. In comparison, the 14 edits extracted by $Meditor$ correspond to 3 categories of changes: 2 categories of method API changes and 1 category of field API change. Interestingly, **there is no content overlap between the edits from different sources**. The edits inferred by $Meditor$ complement those edits mentioned in the release note.

For each documented edit in the release note, there is always a corresponding patch (i.e., textual diff file) attached to illustrate how library implementation is modified. Such edit descriptions and related patch files focus on *how library developers edited code, instead of how application developers should edit their client code for migration*. With such documentation, application developers need to decide (1) which li-

TABLE IV: Comparison between the documented edits and automatically extracted edits

Source	API Category	Change Type	# of Edits	Exemplar Edit
Release Note	Type	Change API definition	10	*Ex1.* FacetsCollector is changed from a concrete class to an abstract class.
		Remove API definition	1	*Ex2.* Remove FacetRequest.SortBy (an enum)
	Method	Change return value	1	*Ex3.* A FacetRequest on a non-existent field now returns an empty FacetResult instead of skipping it
		Add parameter	2	*Ex4.* DrillDown.query now takes Occur
	Field	Change default value	1	*Ex5.* The default category delimiter character was changed from U+F749 to U+001F
		Remove field API	1	*Ex6.* FacetResultNode no longer has a residue field
Inferred Edits by $Meditor$	Method	Change return type	12	*Ex7.* public BytesRef fill(BytesRef, long) ==========Replaced by========== public void fill(BytesRef, long)

		Remove method API	1	*Ex8.* v_0_long += RamUsage.NUM_BYTES_ARRAY_HEADER + v_1_Reader_inst.getBlocks().length;
				for (byte[] v_2 : v_1_Reader_inst.getBlocks() {v_0_long += v_2.length;}
				==========Replaced by==========
				v_0_long += c_0_long;

	Field	Change field API	1	*Ex9.* Version.LUCENE_41 ==========Replaced by========== Version.LUCENE_42

TABLE V: Effectiveness of $Meditor$'s Edit Application

	Lucene			CraftBukkit			Android SDK			Commons IO		
	Sin	Blo	MB	Sin	Blo	MB	Sin	Blo	MB	Sin	Blo	MB
# of snippets for edit generation	22	1	0	27	0	10	26	0	0	1	0	0
# of snippets for edit application	91	1	0	58	0	15	57	0	0	3	0	0
# of code snippets modified by the tool	91	1	0	58	0	15	57	0	0	3	0	0
# of code snippets transformed partially correctly	0	1	0	0	0	6	0	0	0	0	0	0
# of code snippets transformed fully correctly	91	0	0	58	0	9	57	0	0	3	0	0

brary modification influences their code, and (2) how to adjust client code to solve any API backwards compatibility issue. In comparison, the extracted edits by $Meditor$ demonstrate how developers migrated code between library releases with program transformation templates. *Instead of focusing on the edits within libraries, the inferred edits focus on the migration practices conducted by application developers.*

> **Finding 4:** *The extracted edits are very different from documented edits in terms of their categories and content. It means that the edits inferred by $Meditor$ can well complement the information in release notes.*

C. Effectiveness of Edit Application

To evaluate how well $Meditor$ applies inferred edits, we constructed a data set of 87 edits from the 2,458 snippets with MR code changes. Specifically, we created the set based on recurring migration patterns—edits repetitively applied to multiple snippets. When multiple code change examples (e.g., $Ch1$ and $Ch2$) illustrate the same migration edit (e.g., E), we used one example (e.g., $Ch1$) for $Meditor$ to infer the pattern, and used the remaining examples (e.g., $Ch2$) to evaluate how $Meditor$ applies the pattern. By manually comparing the tool-generated versions with human-crafted versions, we determined whether $Meditor$ transformed code correctly.

Table V presents our evaluation results. In total, $Meditor$ inferred edits from 87 examples to acquire 87 unique edits, and then applied the edits to 225 given code snippets. Ideally, if $Meditor$ can perfectly apply all inferred edits, all these snippets should be transformed fully correctly. Specifically, 209 of the 225 snippets require Sin edits, 1 snippet requires for Blo edit, and 15 snippets require for MB edits. Although it would be better if we have a balanced data set with equal numbers of different categorized edits, we could not control the distribution of snippets among the categories. As the current data set contains edits of all three categories, it is still helpful for us to evaluate $Meditor$'s capability of edit application in a variety of scenarios.

According to Table V, $Meditor$ applied all Sin edits fully correctly. It applied the single Blo edit partially correctly, because the edit uses an undeclared variable or unknown constant into the migrated code. As shown in Fig. 4, a constant identifier `c_0_Version` was introduced by $Meditor$ because the original example uses a project-specific constant (i.e., `LUCENE_VERSION`). Similarly, for the 15 snippets requiring MB edits, $Meditor$ transformed 9 snippets fully correctly and 6 snippets partially correctly. The main reason for partial correctness is still the usage of undeclared variables/constants. By reviewing the suggested code migration, developers can (1) learn what APIs to use to replace outdated APIs, and (2) apply extra edits as needed to efficiently complete migration.

> **Finding 5:** *$Meditor$ fully correctly applies edits for 218 out of 225 cases and applies edits partially correctly for the remaining 7 cases, manifesting great capability of migrating code between library releases.*

D. Discussion

We open sourced our project at https://bitbucket.org/shengzhex_research/meditor. Our research will shed light on related areas, such as automatic detection and fixing of API misuses. Recent work shows that software practitioners sometimes misuse security APIs and produce vulnerable code [22], [29]. As the next step, we will extend $Meditor$ to infer any fixing pattern for security API misuses, and apply those patterns to automatically patch vulnerable code.

Currently, we inspected code changes together with the inferred patterns to decide the correctness of migration edits. Such edit validation process is time-consuming and subject to human bias. To improve the process, we plan to use regression testing and automatic test generation techniques to check whether the migrated code compile and execute successfully.

V. Related Work

This section describes related work on API usage mining and suggestion, inference and application of migration rules, and empirical studies on library-related software evolution.

A. API Usage Mining and Suggestion

Although library APIs are widely used in software development, API usage is often poorly documented. Researchers built a variety of approaches to mine specifications from source code or documentation, and to provide coding suggestion accordingly [19], [28], [31], [32], [36], [41], [46], [48], [51], [55], [60]. For instance, Engler et al. mined API usage invariants like method `lock()` must be invoked together with `unlock()`, and then checked code for any violation of the invariants [28]. Khairunnesa extended the research to mine for any precondition of using certain APIs, such as the valid value range of a passed-in parameter [36]. Gu et al. extracted API usage sequences and the first sentence of corresponding document comments to train a deep learning model with RNN, and suggested API usage given a natural language query [32].

Raghothaman et al. mapped natural language queries to relevant APIs by learning a statistical model from the clickthrough data of Bing search [51]. Then they mined API usage patterns from open-source code repositories. When a user searches for the implementation of a certain task, their tool SWIM can automatically synthesize an exemplar implementation with proper API usage. Nguyen et al. mined frequent co-applied API usage changes in software repositories with statistical learning, and recommended API code completion based on the given program context [41]. These approaches focus on how to use APIs appropriately instead of how to adapt the API usage between library releases.

B. Inferring and Applying API Migration Rules

Prior research proposed several tools to infer or apply API migration rules [23], [25], [33], [40], [45], [52], [56], [59]. Specifically, Chow and Notkin proposed a semi-automated mechanism to update client applications in response to library changes [23]. When library maintainers modify function interfaces of a library, they are required to annotate the changed functions with specifications. Such specifications are then used to generate tools that can update client code. However, the proposed method only works for simple changes like updating API signatures. Catchup! records API refactoring actions as a library maintainer evolves an API, and then replays the refactorings to update client applications accordingly [33]. Nevertheless, this approach only fully supports three types of refactorings: renaming types, moving Java elements, and changing method signatures.

SemDiff compares different versions of a library to analyze how the library applies adaptive changes to its API evolution [25]. If a method call is frequently replaced by another method call, SemDiff recommends such API method replacement to client code. LibSync compares different versions of a library to locate the changed APIs, and then compares versions of migrated client code to extract the associated API usage adaptation patterns like renaming a method or changing parameters [45]. All these tools focus on API mappings.

Some researchers mined API translation rules between Java and C# [42]–[44], [61]. For instance, Zhong et al. aligned the client code of different libraries based on textual similarity, constructed API usage graphs for each pair of aligned code, and inferred API usage mappings accordingly [61]. Nevertheless, this approach only infers API mappings. Nguyen et al. tokenized source code, and leveraged statistical machine translation to infer the correspondence between Java code and the equivalent C# implementation. With the established correspondence, the researchers then translated a given Java program to C# [42]–[44]. Although these approaches map both API usage and the surrounding code, they do not handle code migrations between releases of the same library.

Different from prior work, $Meditor$ applies static program analysis to changed Java source files. This analysis allows $Meditor$ to align many-to-many statements between versions based on the data and control dependencies among statements. It also allows $Meditor$ to infer API replacement operations together with other related editing operations, and to safely ignore migration-irrelevant details for edit generalization purpose. By extracting API replacements together with related statement insertions or deletions, $Meditor$ can help developers migrate code with fewer syntactic and semantic errors, improving programmer productivity and software quality.

C. Empirical Studies on API Evolution

Several studies examined how library APIs evolve [27], [37], [50], [58]. For example, Dig and Johnson manually inspected API changes based on change logs and release notes, and found that 80% of API breaking changes were introduced by code refactorings. Xing and Stroulia used UMLDiff [57] to compare the program structures of library versions, and concluded that about 70% of structural changes were refactorings. Kim et al. investigated function signature-change patterns, and observed correlations between signature changes and other types of changes like LOC and function body changes [37]. Raemaekers et al. analyzed seven years of library release history, and found that one third of all releases introduce at least one breaking change [50]. None of these studies explores how client code should co-evolve with API changes.

Some researchers investigated the impact of API evolution on client software evolution [21], [26], [35], [39], [47]. Specifically, Padioleau et al. studied how Linux device driver code collaterally evolved with kernel library APIs [47]. They found that an API evolution and dependent collateral evolutions might take several years to complete and could introduce bugs into previously mature code. Bavota et al. used the build files of Apache projects to analyze (1) how library dependencies change over time; (2) whether a dependency upgrade is due to different kinds of factors, and (3) how an upgrade impacts on a related project [21]. Dietrich et al. identified problems in client code caused by library upgrades [26]. Hora et al. [35] and McDonnell et al. [39] separately investigated the Pharo and Android Ecosystem, to understand how client code reacted to API changes in an ecosystem.

Our characterization study on inferred edits is different from all prior studies, because we examined low-level details of migration patterns. Our evaluation results also complement prior research by exposing a variety of nontrivial migration edits already applied by developers. Cossette et al. conducted an empirical study to manually distill the API migration edits applied in libraries' version history [24]. They classified the edits into three categories: fully automatable (e.g., API renaming), partially automatable (e.g., implementing a newly declared class), and hard to automate (e.g., inserting data preparation logic for an added method parameter). This piece of work by Cossette et al. motivated our research. By correlating API replacement changes with surrounding co-applied statement insertions or deletions, $Meditor$ is able to handle some hard-to-automate cases (e.g., adding a method parameter and removing a method API) mentioned in that paper.

VI. Threats to Validity

$Meditor$ currently generates and applies migration edits for only Java-based libraries. Its methodology can be similarly implemented to handle programs written in other object-oriented languages, when we exploit tools to conduct syntactic differencing and static analysis for other languaged programs. $Meditor$ detects MR commits by checking for any version change in two types of build files: pom.xml and build.gradle. There are still other formatted build files used in Java projects like build.xml. By expanding the types of build files to process, we will be able to detect more MR changes.

Currently, $Meditor$ processes the code changes co-applied with build-file changes to reveal MR code changes. It can miss relevant changes when developers intentionally submitted build-file changes in one commit and submitted migration code changes in follow-up commits. In the future, we plan to overcome this limitation by defining a sliding window to scan any N commits ($N \geq 1$) checked in after the commit with build-file changes.

$Meditor$ focuses on edits relevant to method and field APIs, and simple edits related to type APIs (e.g., class rename or move). It cannot handle complicated type API changes such as replacing an interface with an enum, or replacing a concrete class with an abstract one. The main challenge is that such changes may require for extra project-specific implementations to define new methods. $Meditor$ can infer project-agnostic migration changes, but does not handle project-specific edits because such edits usually vary with projects. $Meditor$ currently ensures the quality of inferred edits by comparing the input and output variable sets of edited regions. We used such comparison to approximate semantic equivalence checking, although this approximation is neither sound nor complete. We will explore more ways to compare the program semantics between code revisions.

VII. Conclusion

This paper presents the design and implementation of $Meditor$, a novel approach to infer and apply migration edits leveraging program dependency analysis and semantic equivalence checking. Compared with existing approaches, $Meditor$ is unique in several aspects. First, $Meditor$ extracts edits purely from client code instead of from the evolution history of libraries themselves. While the library evolution only demonstrates how libraries use their own APIs, multiple client projects can use APIs in a more diverse way and thus embody various migration patterns. Second, $Meditor$ generalizes program transformations instead of solely inferring API mapping rules. When statement insertions or deletions are co-applied with API replacements, $Meditor$ is especially helpful because it keeps track of the data or control dependencies between program co-changes, and clusters statement insertions or deletions together with API usage updates. Third, $Meditor$ applies edits automatically.

Our evaluation reveals several interesting insights about migration edits. First, a considerable number of the generalized edits (483 out of 1,368) apply correlated changes to multiple statements, indicating the necessity of using program dependence analysis to identify and cluster MR changes. Second, the source and target library releases of most migration tasks are nonconsecutive, with several releases standing in between. This means that existing library release notes are usually not helpful. Third, according to our case study, the edits $Meditor$ inferred well complement the documented knowledge in library release notes. Fourth, $Meditor$ applied edits fully correctly to 218 of 225 snippets, and transformed the remaining 7 snippets partially correctly. Our future work includes building techniques to (1) extract more edits from repositories, (2) generate more complicated migration edits involving project-specific logic, and (3) conduct more rigorous semantic equivalence checking.

Acknowledgement

We thank reviewers for their insightful comments. We also thank Prof. André van der Hoek for his valuable feedback.

References

[1] The Ultimate Java Build Tool Comparison: Gradle, Maven, Ant + Ivy. https://zeroturnaround.com/rebellabs/java-build-tools-part-2-a-decision-makers-comparison-of-maven-gradle\-and-ant-ivy/, 2014.

[2] API Evolution and Migration at Google. http://academicscode.com/posts/2017/05/wapi-api-evolution-and-migration-at-google//, May 2017.
[3] A definitive guide to API-breaking changes in .NET. https://stackoverflow.com/questions/1456785/a-definitive-guide-to-api-breaking-changes-in-net, 2018.
[4] Always broken, inconsistent and non versioned, welcome to API hell. https://thoughts.t37.net/always-broken-inconsistent-and-non-versioned-welcome-to-api-hell\-a26103b31081, 2018.
[5] Android SDK. https://stuff.mit.edu/afs/sipb/project/android/docs/sdk/index.html, 2018.
[6] BGCX261/zoie-svn-to-git. https://github.com/BGCX261/zoie-svn-to-git, 2018.
[7] Commons IO. https://commons.apache.org/proper/commons-io/, 2018.
[8] Content management platform to build modern business applications. https://github.com/nuxeo/nuxeo, 2018.
[9] CraftBukkit. https://getbukkit.org/download/craftbukkit, 2018.
[10] How do you get Incompatible API's to work? https://forums.pmmp.io/threads/how-do-you-get-incompatible-apis-to-work.2283/, 2018.
[11] Incompatible change to sessions.restore API in Firefox 54. https://blog.mozilla.org/addons/2017/05/10/incompatible-change-sessions-restore-api-firefox-54/, 2018.
[12] [jgit-dev] [VOTE] Making API incompatible changes. https://www.eclipse.org/lists/jgit-dev/msg02191.html, 2018.
[13] Lucene. https://lucene.apache.org, 2018.
[14] Lucene Change Log. http://lucene.apache.org/core/4_2_0/changes/Changes.html, 2018.
[15] "New build based on revison 27127". https://github.com/yoctopuce/yoctolib_android/commit/f73f800087a198fa7764210bc148835a45ee5b9d, 2018.
[16] NXP-14091: update to lucene 4.7.0. https://github.com/nuxeo/nuxeo/commit/9cbf49dc3799ddc3683706080cc52eacd39f3567, 2018.
[17] Starting a port to Lucene 4.x. https://github.com/behas/lucene-skos/commit/821ef75dcca4124d284c44e8f99e5369bf187fcf, 2018.
[18] WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page, 2018.
[19] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.
[20] M. Barnett, C. Bird, J. a. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 134–144, Piscataway, NJ, USA, 2015. IEEE Press.
[21] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *2013 IEEE International Conference on Software Maintenance*, pages 280–289, Sept 2013.
[22] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags. How reliable is the crowdsourced knowledge of security implementation? In *ICSE*, 2019.
[23] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 359–, Washington, DC, USA, 1996. IEEE Computer Society.
[24] B. E. Cossette and R. J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 55:1–55:11, New York, NY, USA, 2012. ACM.
[25] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 481–490, New York, NY, USA, 2008. ACM.
[26] J. Dietrich, K. Jezek, and P. Brada. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 64–73, Feb 2014.
[27] D. Dig and R. Johnson. The role of refactorings in api evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398, Sept 2005.
[28] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.
[29] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack Overflow considered harmful? The impact of copy&paste on Android application security. In *38th IEEE Symposium on Security and Privacy*, 2017.
[30] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.
[31] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.
[32] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 631–642, New York, NY, USA, 2016. ACM.
[33] J. Henkel and A. Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 274–283, New York, NY, USA, 2005. ACM.
[34] K. Herzig, S. Just, and A. Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21(2):303–336, Apr 2016.
[35] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente. How do developers react to api evolution? the pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 251–260, Sept 2015.
[36] S. S. Khairunnesa, H. A. Nguyen, T. N. Nguyen, and H. Rajan. Exploiting implicit beliefs to resolve sparse usage problem in usage-based specification mining. *Proc. ACM Program. Lang.*, 1(OOPSLA):83:1–83:29, Oct. 2017.
[37] S. Kim, E. J. Whitehead, and J. J. Bevan. Properties of signature change patterns. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 4–13, Sept 2006.
[38] L. Martie, T. D. LaToza, and A. v. d. Hoek. Codeexchange: Supporting reformulation of internet-scale code queries in context (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 24–35, Nov 2015.
[39] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 70–79, Washington, DC, USA, 2013. IEEE Computer Society.
[40] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 353–363, June 2012.
[41] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 511–522, New York, NY, USA, 2016. ACM.
[42] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 457–468. ACM, 2014.
[43] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 651–654, New York, NY, USA, 2013. ACM.
[44] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 585–596, Nov 2015.
[45] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. *SIGPLAN Not.*, 45(10):302–321, Oct. 2010.
[46] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The*

Foundations of Software Engineering, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[47] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 59–71, New York, NY, USA, 2006. ACM.

[48] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language api descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 815–825, Piscataway, NJ, USA, 2012. IEEE Press.

[49] J. H. Perkins. Automatically generating refactorings to support api evolution. *SIGSOFT Softw. Eng. Notes*, 31(1):111–114, Sept. 2005.

[50] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, SCAM '14, pages 215–224, Washington, DC, USA, 2014. IEEE Computer Society.

[51] M. Raghothaman, Y. Wei, and Y. Hamadi. Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 357–367, New York, NY, USA, 2016. ACM.

[52] T. Schäfer, J. Jonas, and M. Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 471–480, New York, NY, USA, 2008. ACM.

[53] L. Shi, H. Zhong, T. Xie, and M. Li. An empirical study on evolution of api documentation. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, FASE'11/ETAPS'11, pages 416–431, Berlin, Heidelberg, 2011. Springer-Verlag.

[54] M. Sulír and J. Porubän. A quantitative study of java software buildability. *CoRR*, abs/1712.01024, 2017.

[55] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *ASE 2009: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 295–306, Los Alamitos, CA, Nov. 2009. IEEE Computer Society.

[56] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. Aura: A hybrid approach to identify framework evolution. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 325–334, New York, NY, USA, 2010. ACM.

[57] Z. Xing and E. Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM.

[58] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Sept 2006.

[59] Z. Xing and E. Stroulia. Api-evolution support with diff-catchup. *IEEE Trans. Softw. Eng.*, 33(12):818–836, Dec. 2007.

[60] H. Zhong and H. Mei. An empirical study on api usages. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[61] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 195–204. ACM, 2010.