Migrating Client Code without Change Examples

Hao Zhong
Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
zhonghao@sjtu.edu.cn

Na Meng
Department of Computer Science
Virginia Polytechnic Institute and State University, USA
nm8247@cs.vt.edu

Abstract—API developers evolve software libraries to fix bugs, add new features, or refactor code. To benefit from such library evolution, the programmers of client projects have to repetitively upgrade their library usages and adapt their codebases to any library API breaking changes (e.g., API renaming). Such adaptive changes can be tedious and error-prone. Existing tools provide limited support to help programmers migrate project clients from old library versions to new ones. For instance, some tools extract API mappings between library versions and only suggest simple adaptive changes (i.e., statement updates); other tools suggest or automate more complicated edits (e.g., statement insertions) based on user-provided exemplar code migrations. However, when new library versions are available, it is usually cumbersome and time-consuming for users to provide sufficient human-crafted samples in order to guide automatic migration.

In this paper, we propose a novel approach—AUTOUPDATE—to further improve the state of the art. Instead of learning from change examples, we designed AUTOUPDATE to automate migration in a compiler-directed way. Namely, given a compilation error triggered by upgrading libraries, AUTOUPDATE exploits 13 migration operators to generate candidate edits, and tentatively applies each edit until the error is resolved or all edits are explored. We conducted two evaluations. The first evaluation involves migrating 371 tutorial examples between versions of 5 popular libraries. AUTOUPDATE reduced migration-related compilation errors for 92.7% of tasks. It eliminated such errors for 32.4% of tasks, and 33.9% of the tasks have identical edits to manual migrations. In the second evaluation, we applied AUTOUPDATE to migrate two real client projects of Lucene. AUTOUPDATE successfully migrated both projects, and the migrated code passed all tests.

I. INTRODUCTION

Application programming interface (API) libraries provide reusable APIs and are widely used in software development. As API developers constantly repair bugs and implement new features, it is likely that client code can benefit from new versions of libraries. Despite the potential benefits, it is usually tedious and error-prone for client-code programmers to upgrade their projects, because new libraries contain many API breaking changes [30], [53]. For example, when an old API foo() does not exist in the new library version, programmers have to adapt client code after library upgrade, and replace any usage of foo() with that of new API(s).

The challenges. Migrating a project to call a newer library has the following challenges:

1. API mappings alone are insufficient to migrate clients. Lamothé and Shang [31] observed more complicated modifications than renaming code names and changing parameters during migration. Some such modifications were observed by other researchers. For example, Dig and Johnson [17] identified complicated API changes such as new method contracts and new interfaces. It is difficult to define them as mappings.

2. There are multiple ways to migrate an API call. As one API element (a type, a method, or a field) can be split to multiple ones, the call sites of an API can have different ways to be migrated.

3. Many replacements have no documentation. When APIs become deprecated, their replacements can be not documented. The prior studies report that 26% [31] to 60% [11] replacements are not described in documents.

4. Resolving a compilation error can lead to more errors. For example, after we replace a missing type with a new type, more incompatible method errors are introduced, because the new methods change their parameters.

The state of the art. To reduce the manual effort of client code migration, researchers proposed various approaches (Section VII). Based on their migration capabilities, these approaches fall into either of the following categories:

1. Mining API mappings. API mappings provide hints to replace old API calls with their replacements. Nguyen et al. [41]–[43] combine graphs with statistical learning to mine API mappings from two versions of clients. Chen et al. [12] compare the source code and comments in clients to mine API mappings. When libraries are updated, some APIs stay unchanged. Wu et al. [62] call such unchanged APIs as anchors, and analyze how the anchors are called by other changed APIs to obtain more mappings. Meng et al. [37] compare the revisions of libraries to mine API mappings. Lamothé and Shang [31] criticize that API mappings alone are insufficient to migrate clients. As these approaches focus on only one aspect of migrating clients, researchers either do not migrate clients (e.g., [37], [62]), or they admit that mined mappings only reduce migration efforts (e.g., [68]) in their evaluations. Here, two versions of libraries and clients can be considered as code change examples.

2. Migrating with change examples. Based on the types of change examples, these approaches can be roughly divided into two research lines. The first line of approaches analyzes the examples of library changes, and infer the corresponding adaptive changes in client code. For example, Xing and Stroulia [63] break the UMLs of two libraries down, and infer their corresponding code changes on clients. The second line of approaches analyzes the examples of client code changes, and infer transformations that can be applied to new code locations. For example, Meng et al. [56] learn an edit script
from a migration example, and apply the script to similar code locations. As migration examples are expensive to be manually constructed, Fazzini et al. [18] propose an approach to search for such examples from a code repository.

In summary, the state-of-the-art approaches mainly use change examples to resolve the above challenges, but it is very expensive to construct such examples. **Our contributions** are as follows:

1. A novel research direction of building complicated migration edits from simple ones. Instead of breaking complicated changes, we construct complicated changes from simple edits. To achieve this goal, we reduce the construction into an optimization problem of searching for a best combination of simple edits. The target of an optimization problem [9] is to search for the best solution from all feasible solutions.

2. The first approach, called AUTOUPDATE, that attacks the migration problem with synthesis. To determine which one is the best, there must be an algorithmic way of measuring the fitness values of solutions. Our fitness function calculates the number of compilation errors, and our optimized solution is a migrated project whose compilation errors are all resolved.

Our evaluation results show that AUTOUPDATE made the following highlights:

1. Two real projects were successfully migrated, and the migrated projects passed all their test cases. AUTOUPDATE migrated two open-source Java projects—ESA [57] and FleaDB [15]—from an old version of the lucene library to a new version, although the migration tasks involved complex modifications.

2. The compilation errors of 92.7% tasks were reduced, and those of 32.4% tasks were eliminated. For each version of another five libraries, we created a migration task, in which the API tutorial examples of $v_{old}$ were added to the source scope and the binary files of $v_{new}$ (i.e., jar files) were added to the library scope. Each task is challenging because the examples call many APIs. In total, we used AUTOUPDATE to migrate 371 tasks. AUTOUPDATE reduced the compilation errors in 92.7% of tasks, and removed the compilation errors in 32.4% of tasks.

3. 39.3% migrated tasks are identical to manual updates. We compared the modifications of AUTOUPDATE with manual updates. Our results show that (1) in 39.3% migration tasks, AUTOUPDATE produces migrations that are identical to manual ones, and (2) the compilation errors in 54.1% migration tasks are never resolved by programmers, but were fully migrated by AUTOUPDATE.

II. MOTIVATING EXAMPLE

In this section, we construct a migration task to show the challenges of code migration, and to illustrate the benefits of AUTOUPDATE. More compilation errors are introduced, when a target library becomes more different from the original one. To construct a more challenging task, we compile an API tutorial example (simple authentication) of cassandra 1.0.0 with the binary files of cassandra 3.0.0. In total, the task has 20 compilation errors. We analyzed these errors, and we found the instances of all our listed four challenges:

**Challenge 1. More edits than replacements.** We found that the example implements the IAuthenticator interface. Compared with the one of 1.0.0, the interface of 3.0.0 defines five additional methods. As the example project does not implement the five methods, the compiler produces error messages, and one such message is “The type SimpleAuthenticator must implement the inherited abstract method IAuthenticator.setup()”. It is insufficient to handle this problem with only replacements. To handle this challenge, AUTOUPDATE generates stub methods for each needed method.

**Challenge 2. More than one-to-one mappings.** The example calls the FBUtilities.hexToBytes method. In 3.0.0, the FBUtilities class is split into a new FBUtilities class and a ByteBufferUtil class, and the hexToBytes method is moved to the later one. As a result, the type is mapped to two new types. To handle this type of problems, AUTOUPDATE implements more migration operators than renaming API calls and enumerates other operators until one works. In particular, AUTOUPDATE finds the replacement method, because the method has the most similar name and the identical parameter list to the original one.

**Challenge 3. Undocumented replacements.** As introduced in the previous paragraph, the hexToBytes method is moved to a new class. However, we find that neither the documentation of 1.0.0 nor the documentation of 3.0.0 mention that the method is moved. As cassandra provides thousands of API methods, it is difficult for programmers to manually locate the replacement of this method. To handle this challenge, we implement a tool to automatically mine the mappings from API documentation, and even without mined mappings, AUTOUPDATE implements other operators to update code. In this example, AUTOUPDATE finds the correct replacement by matching their method names and parameter types.

**Challenge 4. More compilation errors after one is resolved.** This example project implements the IAuthority interface. In 3.0.0, the interface is merged with IAuthenticator, and the new interface is named as IAuthenticator. As JDT does not find the IAuthority interface, it reports a compilation error. This problem cannot be resolved by a single renaming action. After we replace the interface with the correct one, JDT reports more compilation errors. The new interface defines more methods than the old one. After the old class is replaced, the compiler reports that the added methods must be implemented. To fix the problem, it must combine multiple migration operators with proper steps. AUTOUPDATE uses a migration algorithm to guide its migration process.

When programmers update the example project, they also have to resolve all the compilation errors. As resolving one problem can lead to more errors, we have to modify more locations than the initially reported 20 compilation errors. As migrating this example needs more than mappings, only mined mappings are insufficient to migrate the example. Meanwhile, it is difficult to find its change examples, and writing change examples is tedious. AUTOUPDATE spent only several minutes...
to migrate the example, and its solution is identical to our manual migration. The results show two benefits:

**Benefit 1. AutoUpdate updates client code in a fully automatic manner.** The prior approaches [24], [37], [62], [63] mine API mappings, but Lamothe and Shang [31] show that it is insufficient to migrate clients with only replacements. AutoUpdate handles more issues than replacing APIs with its migration operators (Section IV-B).

**Benefit 2. AutoUpdate is able to migrate code locations that require multiple and complicated modifications without change examples.** Lamothe and Shang [31] complain that some changes in migration are complicated (e.g., many-to-many mappings), and suspect that such migrations must be learnt from examples. Although AutoUpdate does not require migration samples, it overcomes the challenges, since it generates complicated changes.

In summary, AutoUpdate presents a way to resolve all the four challenges in Section I without any change examples.

### III. Research Roadmap

In this section, we analyze our research roadmap.

**The comparison to prior approaches.** For the simplicity of our discussion, we use a function, \( f(x) \rightarrow y \), to denote an approach, where \( x \) is the input, \( y \) is the output, and \( f \) is its technique. After this approach is proposed, an researcher can propose their \( f' \) to attack the identical problem. To show that \( f' \) is better than \( f \), the researcher shall present empirical evidences. In this scenario, the researcher shall conduct controlled experiments to compare \( f \) with \( f' \). In this controlled experiment, the researcher shall align the input \( x \); switch the treatments between \( f \) and \( f' \); and compare their output \( y \). Although we agree that controlled experiments are the foundation of the scientific research [7], it is infeasible or unnecessary to conduct such an experiment in some cases. For example, if a novel approach takes quite different inputs from the prior ones, it becomes infeasible to align the inputs. As a result, even if \( f \) produces better outputs than \( f' \) does, it cannot conclude that \( f \) is better than \( f' \). In this case, it is infeasible and unnecessary to conduct an controlled experiment to show the improvements. For example, most approaches mine specifications from clients [51], but many APIs do not have sufficient clients for mining [66]. Zhong et al. [69] proposed the first approach to mine specifications from API documents. As it is infeasible to align their inputs to the prior approaches, they did not conduct any controlled experiments in their evaluations. As we remove change examples from the inputs of migrating clients, it becomes infeasible to align the inputs of AutoUpdate with the prior approaches. Even if AutoUpdate completes more migration tasks than a prior approach, a critical researcher will complain that change examples are insufficient, but including change examples is already unfair to AutoUpdate, since it does not need any change examples at all. Indeed, eliminating the effort to preparing change examples is our major contribution, but the effort is not even considered if a compared approach uses such examples. As a result, in our evaluation, we did not compare with any prior approaches.

**The comparison of follow-up approaches.** As there is no constraints on how a library can evolve, the deltas from a library to its future versions are not a constant set. As a result, it is infeasible to derive a complete set of operators, and the quests for better operators and migration algorithms are endless. From the research angle of operators, we envisage that researchers can obtain more effective migration operators from four sources: (1) After the prior approaches (e.g., [27], [36]) learn transformation patterns from change examples, it becomes feasible to mine those common ones as migration operators; (2) it is feasible to infer better migration operators from the prior empirical studies on API breaking changes and behavioral backward incompatibilities [26], [39], [50]; (3) As Cossette and Walker [12] show that on average only 20% code changes can be realized by the prior migration approaches, it can be feasible to design migration operators based on their observed failure cases; (4) software engineering thesauruses [10], [58] are useful to design better migration operators; and (5) the code change types identified by prior studies [19] are useful to design migration operators. Still, even if operators are sufficient, their migration capability can be unleashed with more advanced algorithms (e.g., the swarm algorithm [54]) can be more effective to guide the migration process than ours. Our dataset can be released to construct a benchmark for follow-up researchers, and on this benchmark, they can conduct controlled experiments to show their improvements. Furthermore, they shall present their migration capability beyond only benchmarks.

**The good intentions of modifications.** As an automatic approach, we need a measure to guide the migration process, and it is natural to compilation errors as the measure. Indeed, when programmers manually update code, as the first step, they also have to manually resolve compilation errors. Researchers [48], [49] show that it is feasible to cheat an automatic measure. A recent study [61] shows that even experienced programmers may not identify vulnerabilities in pull requests, if they are maliciously and deliberately embedded. To ensure the good intentions of modifications, the results of an automatic measure (e.g., fewer compilation errors) alone are insufficient to prove an improvement over ours. We suggest that researchers shall describe their operators clearly, and release their changed code and logs as we did, so that other researchers can manually inspect and confirm the reliability of their improvements.

**The work after migration.** Even if all modifications are in good intentions, migrated code can still be buggy. Zhong et al. [67] show that mapped APIs have semantics differences. For example, two mapped APIs can produce different output when inputs are corner cases (e.g., null). Although such differences can cause bugs in migrated code, to the best of our knowledge, no migration tools guarantee the semantic correctness of migrated code. Still, only after the compilation errors of a migrated project are fully removed, researchers can define other fitness functions to further improve the migrated project. For example, automatic program repair and repairing...
corrupt data (e.g., [59]) use the number of passing test cases to guide their repair process. If a project still has compilation errors, the above approaches cannot remove its bugs, since test cases will fail on code with compilation errors.

IV. APPROACH

To reduce compilation errors in migrated code, AUTOUPDATE prepares the inputs for updating (Section IV-A), and defines a set of migration operators to update code (Section IV-B). The process is guided by our migration algorithm (Section IV-C).

A. Preparing Inputs for Updating

As the first step, AUTOUPDATE prepares the inputs such as error locations, API documentation, and API mappings.

1. Extracting error locations with JDT. An error location is a code element, which introduces a compilation error after API libraries are updated. AUTOUPDATE uses JDT [2] to extract compilation errors, since we focus on updating Java code and JDT is the built-in compiler of the Eclipse IDE. When JDT locates a compilation error, it reports the code range of the error. Based on each code range and its corresponding Abstract Syntax Tree (AST), AUTOUPDATE locates its code element with errors.

Besides code ranges, JDT reports the types of compilation errors. For example, if it fails to resolve a type named A, JDT reports its error message: “A cannot be resolved to a type.” AUTOUPDATE parses these compilation errors to determine their error types. For each type of compilation errors, AUTOUPDATE applies corresponding migration operators as defined in Section IV-B.

2. Extracting API elements and their mappings from their API documentation. As Lamothe and Shang [31] report that API documents describe most API replacements, we choose to extract API elements and their mappings from their documents. In Java, as API documents are generated by the Javadoc tool [1], they follow strict formats, and AUTOUPDATE extracts API elements (i.e., types, methods, and fields) by matching these formats. After the elements are extracted, AUTOUPDATE uses the Hungarian algorithm [29] to extract their mappings. The Hungarian algorithm is a classical algorithm to extract the best mappings between two sides of items. The algorithm needs a function to define the distance between two items, and it finds mappings that can minimize the overall distance. For two API elements of the same type, AUTOUPDATE uses the Levenshtein edit distance of their full code names to calculate their distance. One of our migration operators (i.e., MA1 in Section IV-B) uses mappings to update client code. Although the algorithm mines only one-to-one mappings, AUTOUPDATE can migrate code that requires many-to-many mappings, since it can combine simple edits to complicated ones (see Section IV-C). Compared with other API mapping approaches (e.g., [12]), our extraction is simple. AUTOUPDATE can achieve even better results, if follow-up researchers use more advanced techniques to mine API mappings.

B. Resolving Compilation Errors

Table I shows the mappings between our migration operators and their target compilation errors. We implement our migration operators on Spoon [46], which is a library that allows the analysis and modifications of Java code. Although updating libraries introduces compilation errors in source files, it does not introduce syntactical errors. As a result, it is still able to generate correct modifications.

We design these migration operators with the following steps. First, we tried to update libraries of several projects, and collected their compilation errors. Second, we tried to recreate the challenges that are mentioned in the prior studies [14], [16], [24], [31], [40], [56], and collected their compilation errors. After we collected as many as possible compilation errors, we analyzed how to resolve each type of compilation errors, and designed their migration operators. In particular, by tentatively replacing API(s) based on each type of mappings, we check whether the related compilation issue(s) are resolved, and thus construct our migration operators.

We can obtain a complete set of migration operators, if we break changes into the finest edits. After all, all modifications are the additions and deletions on a limited number of letters, but such a complete set of operators makes the search space extremely large. A practical tool must balance between the effectiveness and performance. As the first approach in this research line, we cannot guarantee that our migration operators are complete. Still, our positive results show a novel and effective way to resolve migration issues (see Section V), and we further discuss this issue in Section VI.

1) Missing API elements: This category includes undefined types, undefined methods, undefined fields, invisible code elements, and incompatible methods. AUTOUPDATE implements the following migration operators to resolve this type of compilation errors:

MA1. Replacing undefined API elements with mappings. As introduced in Section IV-A, we have mined mappings from API documents. For an undefined element, MA1 queries mined API mappings using the full name of the code element as the keyword. If a replacement is found, MA1 replaces the undefined element with the replacement. For the example in Section IV-A, AUTOUPDATE replaces the missing IAuthority interface with the IAuthenticator interface, since the mapping IAuthority → IAuthentication is extracted.

MA2. Replacing undefined API elements with compatible ones. If a method or a field is undefined, MA2 searches the code elements of the newer library, to locate its compatible matches. A compatible code element has a similar name, and introduces no more compilation errors, after it replaces the missing one. Here, we consider only the top ten similar items, and the similarity is the inverse of the Levenshtein edit distance between code names. For example, the FBUtilities type in 1.0.0 is split into the FBUtilities type and the ByteBufferUtil type in 3.0.0. As AUTOPDARE mines only one-to-one mappings, the old FBUtilities class is mapped to the new FBUtilities class. The mined mapping is insufficient to resolve the compilation error. After searching the methods
TABLE I: Our migration operators.

| Category         | Id  | Migration action                                      | Target compilation error                |
|------------------|-----|-------------------------------------------------------|----------------------------------------|
| Missing API      | MA1 | Replacing undefined API elements with mappings        | undefined types, methods, and variables|
| elements         | MA2 | Replacing undefined API elements with compatible ones | undefined methods and variables        |
|                  | MA3 | Replacing undefined fields with getters/setters       | undefined fields                       |
|                  | MA4 | Replacing undefined constructors with creators        | undefined constructors                 |
| Incompatible     | MA5 | Generating explicit conversions                       | class hierarchy changes (incompatible types) |
| API elements     | MA6 | Reducing or swapping method parameters                 | incompatible methods                   |
|                  | MA7 | Replacing static calls with instance calls            | undefined and incompatible methods      |
|                  | MA8 | Exploring declared fields and methods                 | incompatible actual parameters          |
| More or fewer    | MA9 | Generating method stubs                               | unimplemented methods                  |
| API calls        | MA10| Handling exceptions                                  | unhandled exceptions                   |
|                  | MA11| Removing API calls                                    | undefined methods and variables         |
| Other issues     | MA12| Resolving ambiguous types                             | ambiguous types                        |
|                  | MA13| Replacing invisible fields with getters and setters    | invisible fields                        |

in 3.0.0, MA2 locates the `ByteBufferUtil.hexToBytes` as the correct replacement.

**MA3. Replacing undefined fields with getters or setters, and vice versa.** If a field is missing, MA3 will replace it with its getters or setters. When this happens, a field name is typically similar to the names of its getter and setter. For example, the sample of Section II uses the `username` field of the `AuthenticatedUser` type. In 3.0.0, this public field is changed to private. To handle the problem, AUTOUPDATE replaces the field with the `getName` method. Here, it first removes `get` or `set` from getters and setters. After that, it calculates the Levenshtein edit distances between the remaining method names and the field names, and select the one with the least distance. Meanwhile, if a getter or a setter of a field is deleted, AUTOUPDATE will try to replace it with the field.

**MA4. Replacing undefined constructors with creators.** To implement a Factory design pattern, API developers can delete or hide the constructors of a class, and implement creators for the class. MA4 will replace such deleted and hidden constructors with their creators. For example, the samples of `cassandra 0.8.8` call the constructor of the `ColumnFamily` type, but the constructor is hidden in later versions. Instead, `ColumnFamily` implements a set of static methods to create the type. AUTOUPDATE replaces the constructor with these creators to resolve the compilation error.

2) **Incompatible API elements:** This category includes conversion errors, and incompatible/undefined methods. AUTOUPDATE implements the following migration operators:

**MA5. Generating explicit conversions.** For type conversion errors, MA5 adds a cast expression to explicitly convert it. For example, the migration task in Section II involves an assignment:

```java
authorized = Permission.ALL;
```

JDT reports a type mismatch error. To resolve the problem, AUTOUPDATE adds an explicit cast:

```java
authorized = (EnumSet<Permission>) Permission.ALL;
```

**MA6. Reducing or swapping method parameters.** When the signature of a method is changed, JDT can report incorrect overridden methods if the method is overridden, or incompatible methods if the method is directly called. Lamothe and Shang [31] also notice the problem, and introduce a case (i.e., the `queue(Bytebuffer, int)` method), in which the second actual parameter must be removed. Following their suggestions, AUTOUPDATE compares the new signature of its actual parameters of a call site. If a parameter is deleted, AUTOUPDATE removes its corresponding actual parameter from the call site. Alternatively, if the parameter order is changed, AUTOUPDATE reorders the parameters based on parameter types.

**MA7. Replacing static calls with instance calls, and vice versa.** If a static method is deleted, MA7 will replace it with instance methods, and vice versa. For example, the samples of `cassandra 0.8.0` call `ByteBufferUtil.bytes(key)` to obtain the bytes of `key`, but later versions delete the static method. The type of `key` is `Text`, and in the later versions, an instance method `getBytes()` is added to `Text` to obtain the bytes of `key`. To resolve the error, AUTOUPDATE replaces the static method call with `key.getBytes()`.

**MA8. Exploring declared methods and fields of an incompatible type.** If a method has an actual parameter whose type is `Type1` but the desirable one is `Type2`, MA8 explores all the fields and methods that are declared by `Type1`. If the type of a field is `Type2`, MA8 modifies the actual parameter to reference this field. Similarly, if the return type of a method is `Type2`, MA8 modifies the actual parameter to call this method.

3) **More or fewer API calls:** Programmers can add or delete API calls to resolve compilation errors during migration. AUTOUPDATE imitates the process to resolve these errors.

**MA9. Generating method stubs.** Suppose that a client-code class (CC) extends an API abstract class (AC) and implements an interface (AI), and in a follow-up version, a new abstract method is added to AC, or a new method is added to AI. In both cases, CC must implement the newly added method to avoid compilation errors. If JDT reports that a type (c) does not implemented a method (a), MA8 searches the super classes of c for the signature of a. With the found signature, AUTOUPDATE generates a method stub for c. For example, the example of 0.8.0 has a `SimpleAuthenticator` class, which implements the `IAuthenticator` interface. A later
Algorithm 1: the migration Algorithm

Input:

\( p \) is the project whose library is updated.

Output:

\( p' \) is the migrated project.

1: errors ← compile(p); pool \( ^{\text{add}} \) p; nobetter ← 0; generation ← 0;
2: while errors \( \neq 0 \) and nobetter <10 and generation <100 do
3: \( \text{nextpool} \leftarrow 0 \); for \( p \in \text{pool} \) do
4: \( \text{errors} \leftarrow \text{compile}(p) \); for error \( \in \text{errors} \) do
5: \( \text{error} \leftarrow \text{operator}(\text{error}); \text{solution} \leftarrow \text{apply}(\text{op}, \text{p}); \)
6: \( \text{nextpool} \leftarrow \text{add} \) solution; \( \text{end for} \)
7: \( \text{end for} \)
8: if compile(\text{nextbest}) < compile(\text{best}) then
9: \( \text{nobetter} \leftarrow \text{nobetter}+1 \);
10: else
11: \( \text{errors} \leftarrow \text{compile}(\text{nextbest}) \);
12: \( \text{nobetter} \leftarrow \text{nobetter}+1 \);
13: \( \text{end if} \)
14: generation ← generation+1; \( \text{pool} \leftarrow \text{top10} \) nextpool;
15: \( \text{end while} \)

After the method stubs are generated, programmers have to implement its method body. As this migration operator indicates, programmers have to implement new code, when they migrate code to call newer APIs. We further discuss this issue in Section VI.

MA10. Handling exceptions. If the thrown exceptions of a method are modified, JDT can report exception-related errors. To remove such errors, MA9 adds throw expressions or try-catch statements at error locations. For example, the keyspace method of cassandra 1.2.0 throws two exceptions, but later versions throw three exceptions. As a result, when we migrate the examples of cassandra 1.2.0 to later versions, JDT reports a compilation error that an exception is not handled. AUTOUPDATE modifies the corresponding client code to handle the new exception.

MA11. Removing API calls. Programmers can remove API calls, since their corresponding API elements are removed. For example, the examples of poi 3.16 call the following method:

```java
boldFont.setBoldweight(HSSFFont.COLOR_NORMAL.
BOLDWEIGHT_BOLD);
```

As the later versions delete BOLDWEIGHT_BOLD, the above code produces a compilation error during migration. We found that programmers simply delete the above code line to resolve the compilation error. This migration operator imitates this process. To reduce the possibility of removing useful code lines, AUTOUPDATE removes at most one line each time. As removing useful code lines introduces compilation errors, the guidance algorithm of AUTOUPDATE will penalize the abuse of this migration operator.

4) Other issues: AUTOUPDATE implements several migration operators to handle other issues of updating client code.

MA12. Resolving ambiguous types. This migration operator is designed to resolve ambiguous types. If JDT reports that a type is ambiguous, MA12 generates a precise import statement to resolve the problem. As AUTOUPDATE works in a try-and-validate manner, it does not have to determine which type is the correct type. Instead, it generates a solution for each ambiguous type. For example, the examples of cassandra 1.2.0 import all the types of two packages:

```java
import org.apache.cassandra.db.*;
import org.apache.cassandra.thrift.*;
```

As the later versions implement two ConsistencyLevel types under the above packages, JDT reports that the type is ambiguous. AUTOUPDATE replace the above statements with:

```java
import org.apache.cassandra.db.ConsistencyLevel;
```

AUTOUPDATE generates a solution for each possible type. In the other solution, it replaces the above statement with the ConsistencyLevel type under the thrift package. This solution is discarded, since it introduces more errors.

MA13. Replacing invisible fields with getters and setters. An updated library can hide a visible field. In such cases, JDT reports that fields are invisible. To handle these errors, MA12 tries to replace such field accesses with getters and setters. For example, the examples of poi 3.15 have the following code:

```java
if (attachment.attachLongFileName != null) {
```

The later versions hide the attachLongFileName field, and implement a getter called getAttachLongFileName(). AUTOUPDATE replaces the above field access with the corresponding getter to resolve this issue. This operator is similar to MA8. We implement a separate operator, since their error types are different.

C. Guiding the Migration Process

As introduced in the beginning of this section, the purpose of AUTOUPDATE is to reduce the compilation errors that are introduced in code migration. We reduce this purpose to a search problem, in which we minimize the compilation errors of a client whose libraries are updated. Genetic algorithms [60] are widely used to solve optimization problems. Algorithm 1
TABLE III: The examples of cassandra 0.8.0-beta1.

| Name            | File | LOC | Class | Method | Field |
|-----------------|------|-----|-------|--------|-------|
| client          | 1    | 196 | 46    | 18     | 28    |
| hadoop word count | 3   | 495 | 115   | 67     | 68    |
| simple authentication | 2  | 306 | 47    | 14     | 21    |
| Total           | 6    | 997 | 208   | 99     | 117   |

shows our algorithm to guide the migration process, and it is inspired by genetic algorithms.

In our algorithm, Line 1 initiates our variables, and Line 2 shows our termination condition. It stops when (1) a solution without compilation errors is synthesized; (2) no better solutions are found in the recent ten generations; or (3) it fails to resolve all the compilation errors within one hundred generations. For each solution in our pool, Line 5 compiles it to collect its compilation errors. For each compilation error, Line 7 selects a migration operator, and applies the operator to generate a new solution. If multiple migration operators are feasible, AUTOUPDATE randomly picks one of them. Line 8 adds new solutions to a new pool. Line 11 obtains the best solutions of both pools. Line 12 compares the best solutions from the two pools. If the new pool does not contain a better solution, Line 13 increases nobetter. If it contains, Line 16 resets the variable and assigns the better solution to $p'$. Line 18 increases generation, and selects the top ten unique solutions from the new pool. It determines that two solutions are duplicated, if their compilation errors are identical.

V. EVALUATION

We conduct evaluations to explore the research questions:

(RQ1) How effectively does AUTOUPDATE resolve compilation errors that are introduced by updating libraries to newer versions (Section V-A)?

(RQ2) What are the differences between the code updated by AUTOUPDATE and the code updated by programmers (Section V-B)?

(RQ3) How effectively does AUTOUPDATE migrate real code (Section V-C)?

(RQ4) How effective is AUTOUPDATE, if it uses only API mappings (Section V-D)?

For RQ1, our results show that AUTOUPDATE reduced compilation errors of 92.7% tasks; eliminated the compilation errors of 32.4% tasks; and do not increase the errors of tasks.

For RQ2, our results show that 39.3% of our fully migrated tasks are identical to manual code changes.

For RQ3, our results show that AUTOUPDATE eliminated all the compilation errors of two real clients which were introduced by updating their libraries to a more recent version.

For RQ4, our results show that no task was migrated with only mappings.

More details of our evaluations are listed on our website:
https://anonymous.4open.science/r/fa1c702b-5ccd-47c3-985e-ee92acefd949/

A. RQ1. Overall Effectiveness

1) Setup: Table II shows the subjects of our evaluations. We select five widely used Java libraries: accumulo provides a sorted and scalable data storage; cassandra is a scalable database; karaf is an enterprise application runtime; lucene is a search engine; and poi is a library to manipulate Microsoft documents. All the projects are collected from the Apache Foundation, since they provide the archives of all their versions. Column “FV” lists the first versions. Column “LV” lists the last versions. Column “TV” lists the number of the total versions. We select all the versions of these libraries.

All the five libraries provide API examples to illustrate how to call their APIs. Column “LOC” lists the total lines of API examples. Although API examples do not have many lines of code, they call many APIs of our subject libraries. For example, Table [III] shows the examples of cassandra 0.8.0-beta1. Columns “Class”, “Method”, and “Field” show the numbers of unique called API classes, methods, and fields that are declared by cassandra. As shown in Table III, the three examples have only six files, but they call two hundred API classes of cassandra. The prior studies [33], [66] show that most clients call a few hot spots APIs. As a comparison, most real projects do not call many APIs from a single library like cassandra. As for API migration, it is even more challenging to migrate examples than most real code.

Given the examples of the v1 version and the next v2 version, we build a project (p), in which the source files of v1’s examples are added to the source scope of p and the binary files of v2 are added to the class path of p. The examples of v1 call the APIs of v1, and the binary files of v2 provide the APIs of v2. As a result, the p project illustrates a task, in which programmers migrate the examples from v1 to v2. If v2 contains breaking changes, the p project will have compilation errors. In such cases, programmers have to manually resolve all the compilation errors, if they migrate the examples from v1 to v2. In this way, we construct a migration task for each version of a library.

We use AUTOUPDATE to migrate our tasks, and count the number of reduced compilation errors as the measure for the effectiveness of a migration tool. We select this measure, because reducing such errors is a critical step of migrating client code. Indeed, when programmers manually migrate projects to call newer libraries, they also have to resolve all compilation errors before they can test and debug bugs that are introduced in their migration process. We notice that some researchers (e.g., [68]) also use this measure to analyze the effectiveness of their approaches. Although superficial modifications can resolve compilation errors, they can lead to semantic bugs in migrated code. As a result, fewer compilation errors do not necessarily indicate better migration results. To resolve this issue, we compare our migrated tasks with the existing migrate projects in Section V-B. Furthermore, in Section V-C we use AUTOUPDATE to migrate two real projects, and execute their test cases to ensure that its migrations are correct.

We did not compare AUTOUPDATE with prior approaches, because it is difficult to obtain change examples. For example, Xing and Stroulia [63] requires UML changes, but most libraries do not present their UMLs. As it is infeasible to align their inputs, any comparison is illustrative but not conclusive.
TABLE IV: The overall result and the result of only mappings.

| Name     | Total | Clean | AUTOUPDATE | MA1 |
|----------|-------|-------|------------|-----|
|          |       |       | - s + m % | error solution | - s + m % | error solution |
| accumulo | 28    | 4     | 22 11 45.8 | 3.6 17.4 | 0 23 1 0 0.0% -1.3 0.2 |
| cassandra| 161   | 89    | 72 1 39.7 | 3.9 18.6 | 3 66 3 0 0.0% 0.0 0.2 |
| karaf    | 71    | 68    | 2 1 33.3 | 96.0 26.0 | 0 3 0 0 0.0% 0.0 0.0 |
| lucene   | 91    | 78    | 9 4 15.4 | 13.0 25.6 | 3 10 0 0 0.0% 6.3 0.6 |
| poi      | 20    | 9     | 9 2 36.4 | 16.4 41.7 | 1 9 1 0 0.0% 2.0 0.6 |

2) Result: Table IV shows the result. Column “Total” shows the number of total tasks. Column “Clean” shows the number of tasks whose projects have no compilation errors. These projects have no compilation errors, since their API changes are all compatible. For the other projects, Column “AUTOUPDATE” lists our results. Subcolumns “-”, “+”, and “m” show the number of the tasks where compilation errors of projects are reduced, unchanged, and increased, respectively. AUTOUPDATE reduces the compilation errors for more than 90% tasks, without increasing such errors of any tasks. Subcolumns “migrated” shows the number of tasks whose compilation errors are eliminated. In total, AUTOUPDATE fully migrated 61 tasks. Subcolumns “%” are calculated as \[
\text{migrated} = \frac{\text{m}}{\text{total} - \text{clean}}
\]
In total, AUTOUPDATE eliminates the compilation errors of 33.2% tasks. Subcolumns “error” show the averages of reduced compilation errors per task. Subcolumns “solution” show the averages of generated solutions per tasks, before the best one is found.

From Table IV we identify two types of API evolutions. In the first type of evolutions, most next versions have trivial breaking changes. As a typical example, in accumulo, 24 out of its 28 projects have compilation errors. AUTOUPDATE is able to eliminate their compilation errors, for about half of the projects, and it needs to resolve 3.6 compilation errors per task. In the second type of evolutions, fewer near releases have breaking changes, but if they have such changes, their changes are nontrivial. As a typical example, in karaf, 68 out of 71 projects have no compilation errors, but the other three projects have hundreds of compilation errors. On average, for karaf, AUTOUPDATE resolved 96 compilation errors per task, which are much more than those for accumulo and cassandra, but it still fails to fully resolve the three projects.

Table IV highlights the importance of our migration algorithm. On average, AUTOUPDATE has to try more than twenty solutions, before it finds the best ones. Without a migration algorithm, if a tool resolves one error for each task, it can still reduce compilation errors in projects, but it is unlikely to generate the best solutions for most tasks.

In summary, our results show that AUTOUPDATE reduces the compilation errors of more than 90% tasks, and 32.4% tasks are fully resolved. AUTOUPDATE is more suitable to update client code whose libraries have continuous and trivial changes, but its effectiveness is reduced when handling libraries with sudden and nontrivial changes. In addition, most best solutions are not synthesized by single migration operators, but the combinations of them.

B. RQ2. Comparison with Manual Migrations

1) Setup: In total, 61 tasks were fully migrated. We manually inspected all these tasks to understand the differences between manual migrations and those of our tool. In each task, a project is built by combining the example of v1 and the binary files of v2. We checked whether the examples of v2 and its later versions have the same compilation errors. If programmers migrated such errors, we checked whether their modifications are consistent with what AUTOUPDATE did on the projects. Here, we ignore superficial edits (e.g., refactoring) and edits on comments.

2) Result: Based on our inspection results, we classified the tasks into four categories:

1. In 24 tasks, our migrated projects are identical to manual migrations. For example, the project of cassandra 1.0.0 has an error: “The method hexToBytes(String) is undefined for the type FBUtilities”. AUTOUPDATE replaces the method call with hexToBytes(String) of the ByteBuffersUtil type, which is identical to manual migrations.

2. In 33 tasks, we cannot compare our migrated projects with manual migrations, since their compilation errors are never resolved by programmers. Typically, API tutorials do not provide any APIs. As programmers can pay less attention in updating examples, we find that these compilation errors are never resolved. For example, the project of cassandra 0.8.0 includes the CassandraBulkLoader type that calls the constructor:

   ```java
   baseColumnFamily = new ColumnFamily(ColumnFamilyType .
   Standard , DatabaseDescriptor . getComparator ( keyspace .
   columnFamily ) , DatabaseDescriptor .
   getSubComparator ( keyspace , columnFamily ) ,
   CFMetaData . get1d ( keyspace , columnFamily ) );
   ```

   As the constructor is deleted in the later versions, the above code produces a compilation error. The most recent version of the type appears in cassandra 0.8.10, but even in that version, this compilation error is not removed. AUTOUPDATE replaces the call with a creator to resolve the problem:

   ```java
   baseColumnFamily = ColumnFamily . create ( CFMetaData . get1d ( keyspace , columnFamily ) );
   ```

3. In two tasks, the buggy code is deleted. For example, the project of cassandra 1.2.0-beta3 has an example, in which a code line is as follows:

   ```java
   EnumSet < Permission > authorized = Permission . NONE ;
   ```

   As the type of Permission . NONE is modified to Set-<Permission> in the later versions, the above code example
produces a type mismatch. AUTOUPDATE adds an explicit cast to resolve it:

```java
1 EnumSet<Permission> authorized = (EnumSet<Permission>)
   Permission.NONE;
```

We found that the later versions delete this example, so we cannot compare our modifications with manual migrations.

4. In two tasks, API code is modified to resolve their compilation errors. For example, the p project of lucene 4.0.0 has an error: “The constructor TextField(String, BufferedReader) is undefined”, since its next version deletes this constructor. AUTOUPDATE modified client code to resolve this issue, but lucene programmers roll back the deletion to resolve the issue, and their client code is not changed.

In summary, we find that 39.3% migrated tasks are identical to manual modifications, and in 54.1% tasks, compilation errors are ignored by programmers but are resolved by AUTOUPDATE. In four tasks, programmers present other ways to resolve update issues, e.g., updating API code.

C. RQ3: Real Migrations

1) Setup: In this section, we focus on the clients of lucene, since it is a popular library. In total, there are 427 projects under the lucene topic of Github [3]. Among those projects, we included projects into our evaluation if any project P satisfies all of the following five criteria: (1) P uses any of lucene’s versions lower than 7.4.0; (2) P uses Maven as the build system, because Maven allows us to easily change the version information of any library dependency; (3) P can be successfully built without any compilation error; (4) P is not a toy project according to the project’s description; and we also require that P should be forked and have stars; and (5) P has unit test cases because after migration, we run those test cases to detect possible errors. With this filtering process, we ended up with two client projects: ESA and FleaDB.

According to semantic versioning [4], the first number of a version indicates that this version makes incompatible API changes. For each selected project, we changed the version of its lucene library (v1) to the next version whose first number is different from v1. This setting analyzes to what degree AUTOUPDATE can resolve such incompatible API changes.

2) Updating ESA: ESA [57] is the abbreviation of Explicit Semantic Analysis. It leverages Wikipedia dumps to compare the semantic similarity between two given texts. It is a research tool, and its papers are published in the top AI venues [20], [21]. Google scholar reports that the two papers have more than 3,000 citations. The latest ESA has 1,514 lines of code, and is built on lucene 4.8.1.

In this migration task, we replaced its lucene library from 4.8.1 to 5.0.0. After that, ESA produced 23 compilation errors. These compilation errors include 9 unresolved variables, 6 undefined constructors or methods, 6 incompatible parameters, and 2 unimplemented methods. For example, an error message is “LUCENE_48 cannot be resolved to a variable”, because LUCENE_48 is deleted in 5.0.0.

AUTOUPDATE migrated ESA in only several minutes, and all the compilation errors are resolved. After inspecting the migration log, we find that MA6 resolved 16 compilation errors. For example, MA6 made a modification as follows:

```java
1 -queryParser=new QueryParser(LUCENE_48, TEXT_FIELD, analyzer);
2 +queryParser=new QueryParser(TEXT_FIELD, analyzer);
```

When lucene 5.0.0 removes the LUCENE_48 constant, it also removes the first formal parameter of the QueryParser constructor. MA6 removed the actual parameter of the old version to resolve this compilation error.

MA8 resolved 5 compilation errors, and one is as follows:

```java
1 -... = FSDirectory.open(termDocIndexDirectory){
2 +... = FSDirectory.open(termDocIndexDirectory.toPath()){
```

In the above patch, the type of termDocIndexDirectory was File, but the type of the formal parameter was changed to Path. MA8 resolved the problem by calling the toPath() method of the actual parameter. Besides the above modifications, MA9 generated stubs to resolve the remaining 2 unimplemented methods. ESA has two test methods, and our migrated code passed both test methods.

3) Updating FleaDB: FleaDB [13] is a database that supports the persistence of objects. It has 2,611 lines of code, and is built on lucene 4.10.3. The programmer of FleaDB explains that the APIs of lucene often change and FleaDB can provide more stable APIs.

In this migration task, we also replaced its lucene library with 5.0.0. The replacement introduced 6 compilation errors, including 2 undefined methods and 4 incompatible parameters. AUTOUPDATE resolved all the six compilation errors. After inspecting its migration log, we find that MA6 resolved 3 compilation errors; MA8 resolved 2 compilation errors, and MA11 resolved 1 compilation error. For example, a modification is as follows:

```java
1 -TopFieldCollector.create(this, sort, 1, null, false, false, false);
2 +TopFieldCollector.create(this, sort, 1, false, false, false, false);
```
In 5.0.0, the third formal parameter of the `create` method is deleted. MA8 removed the corresponding actual parameter to solve the problem. `FleADB` implements 15 test methods, and our migrated code passed all the test methods.

In summary, `AUTOUPDATE` migrated two projects to call a more recent version of `lucene`, and its migrated code passed all the test methods of both projects. The results show the significance of `AUTOUPDATE`, in that some migration edits are complicated as shown in the above examples.

D. RQ4. Migrating with Only API Mappings

1) Setup: As Lamothe and Shang [31] report that many approaches mine API mappings, it is interesting to explore the effectiveness of our approach, if migrates code with only such mappings. In this research question, we enable only MA1 of `AUTOUPDATE` to update code.

2) Result: In Table IV, Column “MA1” lists the migration results of only mappings. Compared with Column “AUTOUPDATE”, MA1 does not change the compilation errors for more than 90% tasks, and for five tasks, it even increases such errors. Although MA1 reduces the compilation errors of seven tasks, it fails to fully migrate any task. As a comparison, `AUTOUPDATE` resolves more compilation errors and generates more solutions than MA1 does per task.

E. Threats to Validity

The threats to external validity include our subjects. We select only Apache projects, since their websites provide all their releases. To improve the generality of our results, a tool needs to select projects from more sources, if such projects also provide the archives of all the releases and provide code examples to illustrate their API usages. The construct threat to validity includes that as programmers can forget to update their API examples, the compilation errors in our migration tasks are more than the cases where client code is up-to-date. As a result, the true effectiveness of `AUTOUPDATE` can be underestimated. The problem can be mitigated by migrating client code from more reliable sources. The threat to interval validity includes the parameters of our migration algorithm, which can be less optimized and can be improved.

VI. THE POTENTIAL OF OUR DIRECTION

Our work opens a new research direction to migrate obsolete client code, and our evaluation results show that for the first time, our tool is able to migrate real projects without change examples. As it is tedious and even infeasible to find change examples for all possible migrations, our achievement is a critical step towards practical migration tools. Still, there is still sufficient space for improvements, and the space can attract and incubate a small research community. Besides what were already discussed in Section III, some other research opportunities are as follows:

1. Migrating other languages. Another programming language can call APIs in a way that is different from Java or other object-oriented languages, and its migration process can involve more issues. To enable the migration on other languages, researchers shall design other migration operators, and the migration algorithm can be modified according to its new features.

2. Tuning our treatments and independent variables. Tuning the treatments and independent variables of `AUTOUPDATE` can squeeze its last bit out. For example, trying other search algorithms [47], [54] and tuning our parameters [6] can improve our effectiveness. As another example, when a driver produces a compilation error, Lawall et al. [32] propose an approach to locate its related patches of kernels, and similar techniques can recommend better migration operators.

VII. RELATED WORK

**Code migration.** With evolution of libraries, some APIs may become incompatible across library versions. To address this problem, Henkel and Diwan [24] proposed an approach that captures and replays API refactoring actions to update the client code. Xing and Stroulia [63] proposed an approach that recognizes the changes of APIs by comparing the differences between two UMLs of libraries. Kalra et al. [28] match traces of client code in two versions for their mappings. Balaban et al. [8] proposed an approach to migrate client code when mapping relations of libraries are already available. Xu et al. [64] learn edit scripts from migration commits and apply them on new migration tasks. `AUTOUPDATE` is the first approach to combine simple edits to complicated migrations.

**Search-based software engineering.** Harman et al. [23] conducted a review on the search-based software engineering. As a typical search algorithm, genetic algorithms have been widely used in various applications such as generating test cases [55], release planning [22], repairing bugs [59], refactoring [45], and estimating development cost [25]. Our migration algorithm is inspired by search algorithms, but it resolves in a different research problem, complementing the above approaches.

**Learning transformations from change examples.** Given an original file and its modified file, Andersen et al. [5] extract a set of term replacements from change examples. Meng et al. [35], [36] learn edit scripts from change examples. Rolim et al. [52] search for a transformation that is consistent with all change samples. Long et al. [34] infer AST templates from patches. Nguyen et al. [44] mine graph change patterns from change examples. Given a change example, Jiang et al. [27] mine where to apply its transformation. Given change examples illustrate migrations, Fazzini et al. [18] learn transformations to migrate code. Zhang et al. [65] conduct an empirical study on compiler errors in continuous integration. Meshbah et al. [38] learn a model from past fixes to repair compilation errors. `AUTOUPDATE` is not a learn-based approach, since it does not require change examples as its inputs.

VIII. CONCLUSION

The prior approaches typically migrate client code with migration samples, but it is difficult to obtain such samples. In this paper, we propose `AUTOUPDATE`. Compared with the state of the art, `AUTOUPDATE` presents a new research
direction, in which client code can be migrated without migration samples. Its basic idea is to combine simple edits of migration operators under its migration algorithm, until a complex migration is synthesized. Our evaluation results show that AutoUpdate resolves all the compilation errors of 32.4% migration tasks; 39.3% migrated tasks are identical to manual migrations; and two real projects were fully migrated.

REFERENCES

[1] Javadoc. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html, 2019.
[2] JDT. http://www.eclipse.org/jdt/, 2019.
[3] Lucene clients on Github. https://github.com/topics/lucene, 2019.
[4] Semantic Versioning. https://semver.org/, 2019.
[5] Javadoc. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html, 2019.
[6] M. Angelova and T. Pencheva. Tuning genetic algorithm parameters to improve convergence time. International Journal of Chemical Engineering, 2011, 2013.
[7] R. Baker, Y. Elad, and I. Chet. The controlled experiment in the scientific method with special emphasis on biological control. Physiopathology, 74(9):1019–1021, 1984.
[8] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In Proc. OOPSLA, pages 265–279, 2005.
[9] A. Beck. On the linear search problem. Israel Journal of Mathematics, 2(4):221–228, 1964.
[10] S. Beyer and M. Pinzger. Synonym suggestion for tags on stack overflow. In Proc. IJcAI, pages 94–103, 2015.
[11] G. Brito, A. Hora, M. T. Valente, and R. Robbes. Do developers deprecate APIs with replacement messages? a large-scale analysis on Java systems. In Proc. SANER, pages 360–369, 2016.
[12] C. Chen, Z. Xing, Y. Liu, and K. L. X. Ong. Mining likely analogical methods across third-party libraries via large-scale unsupervised api semantics embedding. IEEE Transactions on Software Engineering, 2019.
[13] B. E. Cossette and R. J. Walker. Seeking the ground truth: a retroactive evaluation approach to improve convergence time. International Journal of Chemical software engineering, 2013.
[14] D. Greer and G. Ruhe. Software release planning: an evolutionary and iterative approach. Information and software technology, 46(4):243–253, 2004.
[15] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys, 45(1):11, 2012.
[16] J. Henkel and J. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In Proc. ICSE, pages 274–283, 2005.
[17] S.-J. Huang and N.-H. Chiu. Optimization of analogy weights by genetic algorithm for software effort estimation. Information and software technology, 48(11):1034–1045, 2006.
[18] K. Jezek and J. Dietrich. API evolution and compatibility: A data corpus and tool evaluation. Journal of Object Technology, 16(4):2–1, 2017.
[19] J. Jiang, L. Ren, Y. Xiong, and L. Zhang. Inferring program transformations from singular examples via big code. In Proc. ASE, pages 255–266, 2019.
[20] S. Kalra, A. Goel, D. Khanna, M. Dhawan, S. Sharma, and R. Purandare. POLLUX: Safely upgrading dependent application libraries. In Proc. ESEC/FSE, pages 290–300, 2016.
[21] H. W. Kuhn. The hungarian method for the assignment problem. Naval research logistics quarterly, 2(1-2):83–97, 1955.
[22] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue. Do developers update their library dependencies? Empirical Software Engineering, 23(1):384–417, 2018.
[23] M. Lamothe and W. Shang. Exploring the use of automated API migration techniques in practice: An experience report on Android. In Proc. MSR, pages 503–514, 2018.
[24] J. Lawall, D. Palinski, L. Gnirke, and G. Muller. Fast and precise retrieval of forward and back porting information for linux device drivers. In Proc. USENIX, pages 15–26, 2017.
[25] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In Proc. MSR, pages 2–11, 2014.
[26] F. Long, P. Amidon, and M. Rinard. Automatic inference of code transformations for patch generation. In Proc. ESEC/FSE, pages 727–739, 2017.
[27] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In Proc. PLDI, pages 329–342, 2011.
[28] M. S. Meng, M. Kim, and K. S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In Proc. ICSE, pages 502–511, 2013.
[29] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In Proc. ICSE, pages 353–363, 2012.
[30] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian. Deepdelta: learning to repair compilation errors. In Proc. ESEC/FSE, pages 925–936, 2019.
[31] S. Mostafa, R. Rodriguez, and X. Wang. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In Proc. ISSTA, pages 215–225, 2017.
[32] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In Proc. ICSE, pages 353–363, 2012.
[33] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian. Deepdelta: learning to repair compilation errors. In Proc. ESEC/FSE, pages 925–936, 2019.
[34] S. Mostafa, R. Rodriguez, and X. Wang. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In Proc. ISSTA, pages 215–225, 2017.
[35] A. T. Nguyen, H. A. Nguyen, T. N. Nguyen, and D. Dig. Api code recommendation using statistical learning from fine-grained changes. In Proc. ESEC/FSE, pages 511–522, 2016.
[36] A. T. Nguyen, H. A. Nguyen, T. N. Nguyen, and T. N. Nguyen. Statistical learning approach for mining api usage mappings for code migration. In Proc. ASE, pages 457–468, 2014.
[37] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In Proc. ICSE, volume 1, pages 858–868, 2015.
[38] A. T. Nguyen, T. N. Nguyen, and T. N. Nguyen. Lexical statistical machine translation for language migration. In Proc. ESEC/FSE, pages 651–654, 2013.
[39] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In Proc. ICSE, pages 819–830, 2019.
[40] M. O Keeffe and M. O. Cinnéide. Search-based refactoring for software maintenance. Journal of Systems and Software, 81(4):502–516, 2008.
[41] R. Pavlak, M. Monperrus, N. Petitprez, C. Nogueira, and L. Seinturier. Spoon: A library for implementing analyses and transformations of java source code. Software: Practice and Experience, 46:1155–1179, 2015.
[42] M. O Keeffe and M. O. Cinnéide. Search-based refactoring for software maintenance. Journal of Systems and Software, 81(4):502–516, 2008.
[43] R. Pawlak, M. Monperrus, N. Petitprez, C. Nogueira, and L. Seinturier. Spoon: A library for implementing analyses and transformations of java source code. Software: Practice and Experience, 46:1155–1179, 2015.
[44] W. Price. Global optimization by controlled random search. Journal of Optimization Theory and Applications, 40(3):333–348, 1983.
[45] Y. Qiu, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In Proc. 36th ICSE, pages 254–265, 2014.
[46] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In Proc. ISSTA, page to appear, 2015.
[47] S. Rahm, A. van Deursen, and J. Visser. Semantic versioning and impact of breaking changes in the maven repository. Journal of Systems and Software, 129:140–158, 2017.
[48] M. P. Robillard, E. Boddé, D. Kuvirykov, M. Mezini, and T. Ratchford. Automated api property inference techniques. IEEE Transactions on Software Engineering, 39(5):613–637, 2012.
