Untangling Spaghetti of Evolutions in Software Histories to Identify Code and Test Co-evolutions

Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, Jean-Marc Jézéquel

Untangling Spaghetti of Evolutions in Software Histories to Identify Code and Test Co-evolutions

Quentin Le Dilavrec
Univ. Rennes, IRISA, INRIA
Rennes, France
quentin.le-dilavrec@irisa.fr

Djamel Eddine Khelladi
CNRS, IRISA, Univ. Rennes
Rennes, France
djamel-eddine.khelladi@irisa.fr

Arnaud Blouin
INSA, IRISA, Univ. Rennes
Rennes, France
arnaud.blouin@irisa.fr

Jean-Marc Jézéquel
Univ. Rennes, IRISA, INRIA
Rennes, France
jean-marc.jezequel@irisa.fr

***Abstract*—Version Control Systems are key elements of modern software development. They provide the history of software systems, serialized as lists of commits. Practitioners may rely on this history to understand and study the evolutions of software systems, including the co-evolution amongst strongly coupled development artifacts such as production code and their tests. However, a precise identification of code and test co-evolutions requires practitioners to manually untangle spaghetti of evolutions. In this paper, we propose an automated approach for detecting co-evolutions between code and test, independently of the commit history. The approach creates a sound knowledge base of code and test co-evolutions that practitioners can use for various purposes in their projects. We conducted an empirical study on a curated set of 45 open-source systems having Git histories. Our approach exhibits a precision of 100 % and an underestimated recall of 37.5 % in detecting the code and test co-evolutions. Our approach also spotted different kinds of code and test co-evolutions, including some of those researchers manually identified in previous work.**

***Index Terms*—Code evolution, Tests co-evolution**

I. Introduction

Software testing plays an important role for quality verification in today software development. In well-tested software systems, tests can even get larger than production code. For example, the Apache Common Collections library has a ratio of one method for three tests, with a total of 13 677 test cases for 3552 methods and 85 % coverage [1]. In this library, a given method can be directly or indirectly called by up to hundreds of tests. This naturally brings a tight coupling between production and test code: they should be jointly evolved, *i.e., co-evolved*. Changes in production code, *i.e., the cause* of the co-evolution, can impact multiple tests so that a practitioner may have to correct them in consequence, *i.e., the repair* of the co-evolution.

Since such a manual maintenance task can be time-consuming and error-prone, the development of reliable automated co-evolution techniques could be very helpful, for instance in a refactoring context. The development of such techniques first requires the understanding of what characterizes a co-evolution, in particular its *cause* and the corresponding *repair*. Yet, test and code co-evolutions are seldom studied at large scale [2], [3], [4], [5]. This paradox takes root in the difficulty for practitioners in understanding characteristics of co-evolutions. This prevents the development of reliable automated co-evolution techniques, because of two major issues. First, existing evolution detection techniques are not able to identify co-evolutions, so that practitioners have to manually untangle spaghetti of changes to try to relate *causes* and corresponding *repairs*. Second, this manual detection is hardly feasible due to the combinatorial number of possibilities to evaluate and the error-proneness of this task: 1) even while studying a single commit, a practitioner has to identify a code change that negatively alters test verdicts, to then identify test changes that fix these tests; 2) the empirical study we conducted shows that on the analyzed projects, one commit contains a median of 10 changes, which leads to 2^{10} possible combinations to evaluate on average; 3) quite often commits are not atomic with respect to co-evolutions: code might have been modified in one commit in such a way that it breaks its tests, but the tests themselves would only be updated in subsequent commits; 4) similarly to evolutions [6], detecting co-evolutions with high precision is necessary: even a moderate proportion of false positive would jeopardize the conclusions of empirical studies and would make worthless build tools that would rely on co-evolution detection.

In this paper, we propose an automated approach for precisely finding code and test co-evolutions in Java software systems and their code history. Our approach supports both atomic and complex evolutions, such as refactorings.

It first performs a static analysis of Git histories and Java code to identify evolutions. It then applies several heuristics to reduce the combinatorial explosion of candidate co-evolutions. The approach finally narrows down these results with a dynamic analysis, based on test execution and compilation, which evaluates the effects of evolutions on test verdicts and compilation results. This dynamic analysis is one novelty of our approach that permits to qualify with high confidence the causal relation between two groups of evolutions in production and test code, *i.e.,* the cause and repair of a co-evolution. Moreover, as another novelty in contrast to [2], [3], [4], [5], we refine the definition of code and test co-evolution to distinguish two types, namely *complete* and *partial* co-evolutions. The former is when the co-evolved test passes, and the later is when it does not, hence, missing some co-evolution ingredients.

We conducted an empirical study on a curated set of 45 open-source systems having Git histories. Our approach found 612 co-evolutions among which 500 are contained in single commits and 112 are scattered on two separated commits. In particular, our approach detected 202 complete co-evolutions for which it exhibits a precision of 100 % and an underestimated

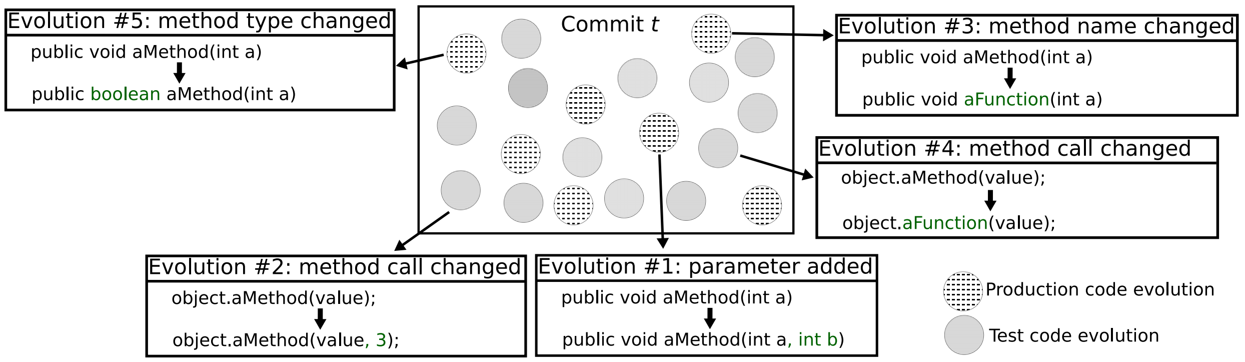
Commit t
Evolution #5: method type changed
public void aMethod(int a)
public boolean aMethod(int a)
Evolution #3: method name changed
public void aMethod(int a)
public void aFunction(int a)
Evolution #4: method call changed
object.aMethod(value);
object.aFunction(value);
Evolution #2: method call changed
object.aMethod(value);
object.aMethod(value, 3);
Evolution #1: parameter added
public void aMethod(int a)
public void aMethod(int a, int b)
Production code evolution
Test code evolution

Fig. 1. Example of evolutions in a commit identified by evolutions detection tools such as RefactoringMiner [6], [7].

recall of $37.5\,\%$ in detecting the code and test co-evolutions. Our approach also spotted the different kinds of code and test co-evolution that researchers manually identified in previous work [2], [3], [5].

In summary, the contributions of the paper are as follows:

- The first, to the best of our knowledge, automated approach for detecting tests and code co-evolutions in single commits (immediate) and in separated commits (delayed);
- We conducted an empirical study on 45 open-source software systems to evaluate the performance of our approach;
- The study confirms the existence of immediate and delayed co-evolutions. It also give evidences and permits the novel characterization of *complete and partial co-evolutions*;
- We bundled the set of detected co-evolutions as a knowledge base for further investigation on this topic;
- We provide a comprehensive dataset for replicating the empirical study. https://github.com/quentinLeDilavrec/ICSME2021

The remainder of this paper is organized as follows: Section II motivates the research problem and provides definitions. Section III details the approach. Section IV presents the evaluation of the approach. Section V discusses related work. Section VI concludes the paper and gives research perspectives.

II. Definitions and Motivating Example

This section defines the core concepts used in the paper and motivates the work through an illustrative example where in a specific commit, a developer replaced *object.aMethod(value)* with *object.aFunction(value, 3)*.

A. Definitions

Def. 1 – Evolution. We refer to changes in production or test code as evolutions spotted from code histories, for example, between commits or releases. The literature classifies evolutions into two groups that correspond to two levels of granularity:

- Atomic evolutions in the form of insertions, deletions, or updates of an AST element [8];
- Complex evolutions (*i.e.,* composition of atomic evolutions) that take the form of, for example, refactoring operations [9], such as *extract class/method* or *push/pull method*, or functional evolutions that consists of newly added code (*i.e.,* a set of atomic insertions).

Multiple algorithms and approaches exist to detect those atomic evolutions from code histories. Tree difference algorithms permit to detect atomic evolutions, such as in *GumTree* [8] or *ChangeDistiller* [10], [11]. More specialized algorithms detect complex evolutions, such as *RefactoringMiner* [6], [7] or *RefDiff* [12]. Note the difference between a change a developer applied and the set of atomic evolutions such tools detect. Consider Figure 1, showing that in the commit t a developer replaced *object.aMethod(value)* with *object.aFunction(value, 3)*. An evolution detection tool decomposes such a change into atomic evolutions as illustrated by the evolutions #1, #3, and #5. More precisely, the commit t in Figure 1 is composed of a set of atomic evolutions that either concern production or test code. *Evolution #1* depicts an evolution in production code on a method that now has a new parameter. *Evolution #2* depicts an evolution in test code where a method call on an object now has a new parameter value. Similarly, *Evolution #3* and *Evolution #5* are production code evolutions that respectively consists of a method renaming and of a method type changing. *Evolution #4* is a test code evolution where the name of a method call changed.

Def. 2 – Dependency graphs. Static and dynamic analyses can extract different types of graphs for representing dependencies among elements of an AST (*Abstract Syntax Tree*), in particular:

- a *call graph* relates methods/constructor/lambda declarations to calls;
- a *flow graph* relates variables and built-in operations such as additions and assignments;
- a *type hierarchy graph* relates type declarations to, for example, variable declarations or return type declarations.

For example, *Evolution #1* declares the method *aMethod*, while *Evolution #2* calls this method on the object *object*. This consists of a call graph.

Def. 3 – Effect of evolutions on tests. Code evolutions (on production or on test code) can affect observable properties of the systems under study. In this work we use two properties: production and test code compilation (*i.e.,* a compilation passes or fails); and test execution verdicts (*i.e.,* a test passes or fails).

We classify code evolution effects on these two properties into three categories, namely:

- *Impacting Evolution*. This is an evolution that now makes a test fail or prevents code to compile.
- *Repairing Evolution*. This is a code evolution that now makes a test pass. This implies that production and test compilation pass.
- *Effectless/Neutral Evolution*. This is a code evolution that does not change the state of tests or compilation.

For example, *Evolution #1* and *Evolution #3* are each an *impacting* evolution as they prevent tests from compiling. Tests get *repaired* in *Evolution #2* and *Evolution #4* respectively. *Evolution #5* is a *neutral* evolution: its does not alter the compilation and the execution of tests.

To qualify the effects on compilation or on test execution, we rely on the following states:

1) **PCFAIL** *(level 1)*: production code compilation fails (test code not compiled or executed);
2) **TCFAIL** *(level 2)*: production code compilation passes but test code compilation fails;
3) **TFAIL** *(level 3)*: both compilations pass and the statically impacted test fail.
4) **TPASS** *(level 4)*: both compilations pass and the statically impacted test pass;

Def. 4 – Code and test co-evolution.

A code and test co-evolution is composed of: a first set of evolutions in production code that breaks tests, *i.e.,* it is the *cause* of the co-evolution; a second set of evolutions in code and/or tests that fixes tests broken by the *cause*, *i.e.,* it is the *repair* of the co-evolution. In a timeline, the *cause* is first applied, followed by the *repair*. We distinguish two types of co-evolution, namely *complete* and *partial*, that have their own definition of the *cause* and the *repair*.

Def. 4.1 – Complete co-evolutions. A complete co-evolution: 1/ starts at TPASS (before the cause); 2/ moves to TFAIL, TCFAIL, or PCFAIL after the cause (and before the repair); 3/ moves back to TPASS after the repair (applied after the cause).

For example, the tuple <*Evolution #1*, *Evolution #2*>, respectively the cause and the repair, is a complete co-evolution, since the test code does not compile anymore (because of *Evolution #1*) until *Evolution #2* is applied.

Def. 4.2 – Partial co-evolutions.

A partial co-evolution corresponds to all the scenarios that differ from starting to TPASS and ending to TPASS, and imply a degradation when applying the cause. For example, one partial co-evolution scenario is: 1) it starts at TFAIL (before the cause); 2) moves to TCFAIL after the cause (and before the repair); 3) moves back to TFAIL after the repair (that does not strictly repair in such cases). Partial co-evolutions relax criteria to obtain and study other interesting cases of co-evolutions.

Finally, when both the cause and repair are located in: 1) the same commit, we refer to it as an *immediate co-evolution*; 2) several commits, we refer to it as a *delayed co-evolution*.

B. *Challenges in detecting test and code co-evolutions*

Classifying the effects of code evolutions on tests with good precision is a real challenge. To find a code and test co-evolution, a practitioner indeed *has to find and isolate impacting groups of evolutions*. This practitioner can hardly do this task by hand since *it requires to identify and test each possible combination of evolutions,* i.e., *to compile code and run for each combination the tests to evaluate potential impacting and repairing effects.* Then, the practitioner *has to associate an impacting evolution to its possible repairing evolutions*, following the same process. For 10 evolutions in a single commit, this implies 2^{10} possible combinations to evaluate.

III. Overall Approach

This section presents our approach for finding code and test co-evolutions from code histories. Figure 2 shows the overall process of our approach. After cloning the evolution history of a given project, it detects evolutions $\boxed{A}$. Then, a static analysis computes dependency graphs between evolutions in production code and test code $\boxed{B}$. Next the approach organizes evolutions $\boxed{C}$. Through a dynamic analysis, it then qualifies the effects of each group of evolutions on tests (as impacting, repairing, or neutral) $\boxed{D}$. Finally, the approach assembles the production code evolutions that *impact* test code, with the test code evolutions that *repair* tests $\boxed{E}$.

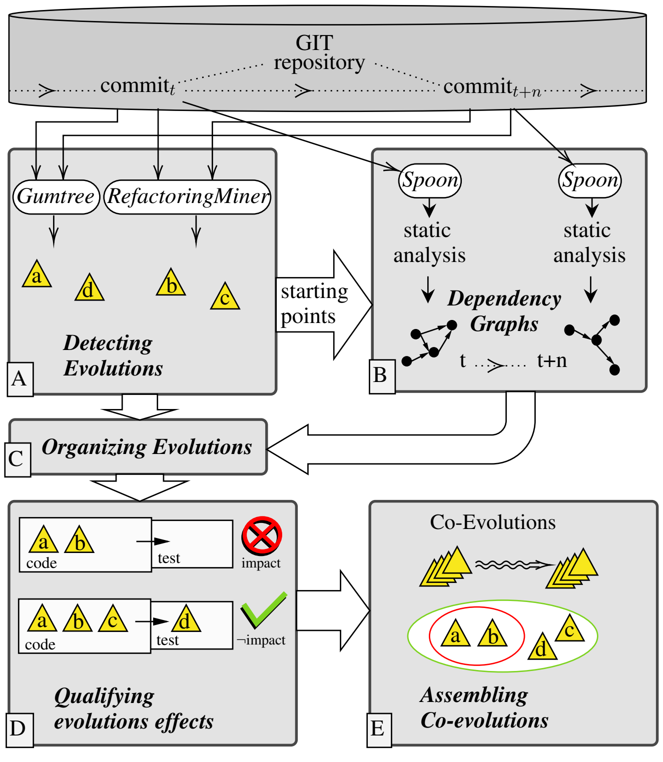

Fig. 2. Overall process of the approach.

A. *Detecting evolutions*

The first step $\boxed{A}$ of the approach consists in detecting all the evolutions between two commits t and $t + n$, with $n \geq 1$.

The approach relies on state-of-the-art techniques for detecting atomic and complex evolutions, respectively with *Gumtree* [8] and *RefatoringMiner* [6], [7].

B. Extracting dependency graphs

The second step $\boxed{B}$ of the approach consists in identifying evolutions in production code that *might* have impacts on tests. This step involves a static analysis that explores three kinds of dependency graphs (see Section II-A) from each identified evolution. For each evolution e located in the production code, the static analysis identifies the corresponding elements $e_0 \ldots e_n$ in the AST and extracts the three kinds of graphs from them. The construction of each graph stops when reaching an AST element located in test code.

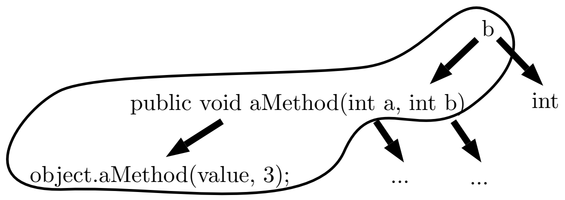

Fig. 3. Illustration of the static analysis, based on Figure 1. The bubble represents one dependency from an AST element of an evolution in production code to an AST element in test code.

Figure 3 illustrates the static analysis based on the example of Figure 1 and its evolution #1 that consists of a new parameter *b* in the declaration of the method *aMethod*. The static analysis starts from *b* to explore its type (*int*) and its method *aMethod*. The analysis then gathers all the calls to *aMethod*. Evolution #2 depicts one of those calls located in test code, so that the analysis stops the exploration for this AST path: *it has found an evolution in production code that may have an impact on test code.* The analysis however continues to explore the AST based on the other calls to *aMethod* to find other potential impacts in test code.

C. Organizing evolutions

The static analysis can only find *potential* impacts on tests. We need a dynamic analysis to qualify those potential impacts with precision. To do so, we observe compilation and test execution results.

A naive analysis would require to evaluate $2^n \times m$ cases, where n is the total number of evolutions and m is the total number of tests: for each identified evolution, we would have to compile and run all the tests. For a large number of evolutions and tests in real histories this would not scale. Moreover, all evaluated cases might not be useful to find co-evolutions. Our third step $\boxed{C}$ thus aims at reducing this combinatorial number of cases. We developed an optimization based on the dependency graph (computed in the previous step) and complex evolutions. This optimization consists in organizing atomic evolutions into groups that we are then able to explore efficiently (see Section III-D).

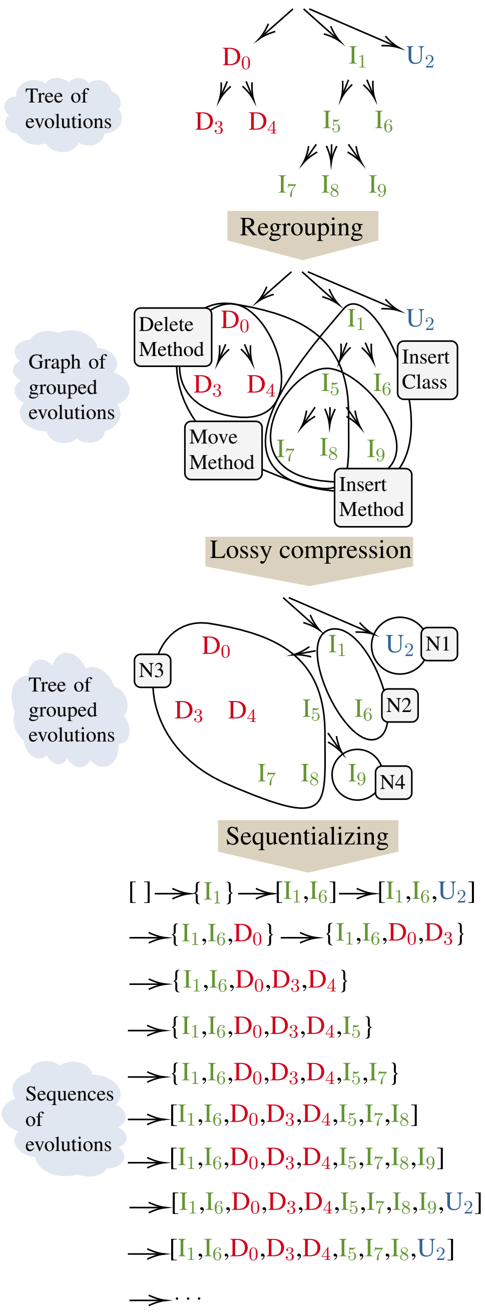
Tree of evolutions
D0
I1
U2
D3
D4
I5
I6
I7
I8
I9
Regrouping
Graph of grouped evolutions
Delete Method
Insert Class
Move Method
Insert Method
Lossy compression
Tree of grouped evolutions
N1
N2
N3
N4
Sequentializing
Sequences of evolutions
[ ] → {I1} → [I1,I6] → [I1,I6,U2]
→ {I1,I6,D0} → {I1,I6,D0,D3}
→ {I1,I6,D0,D3,D4}
→ {I1,I6,D0,D3,D4,I5}
→ {I1,I6,D0,D3,D4,I5,I7}
→ [I1,I6,D0,D3,D4,I5,I7,I8]
→ [I1,I6,D0,D3,D4,I5,I7,I8,I9]
→ [I1,I6,D0,D3,D4,I5,I7,I8,I9,U2]
→ [I1,I6,D0,D3,D4,I5,I7,I8,U2]
→ ...

Fig. 4. The different steps for organizing and sequentializing the evolutions to be then qualified. Legend:
D_x, I_x, U_x: AST element deletion, insertion, update (respectively)
[...]: sequence of evolutions to evaluate
{...}: intermediate sequence of evolutions (not to evaluate)

1) Grouping statically related evolutions: The static analysis of the previous step computed a set of dependencies from production code to test code, as illustrated by Figure 3. Instead of considering each dependency path individually we can group those that are statically related. We consider evolutions not statically dependent on each other as unlikely to be part of a same co-evolution. The top graph of evolutions of Figure 4 shows an example of a statically related group of evolutions.

2) Grouping evolutions by source: Based on the groups of evolutions produced during the previous grouping step, the second grouping step simply consists in grouping those groups using their location in the production code or tests.

3) Grouping using complex evolutions: The approach handles atomic evolutions in the grouping step $\boxed{C}$ because atomic evolutions are: i/ the basic evolution operations so that they also represent complex evolutions; ii/ the most accurate evolutions to represent the transition from a state to another. For the detected complex evolutions, to be able to apply them and qualify their effects, we thus represent them as a group of atomic evolutions. So, a complex evolution is applied if all its atomic evolutions are applied.

For example, moving a method is a complex evolution composed of multiple atomic ones: it consists of deleting the method from its original location and inserting the same method in a different location, as depicted by the "*Graph of grouped evolutions*" in Figure 4: the deletion D_0 implies D_3 and D_4 so that they are grouped. The same reasoning applies for the insertions where: I_5 (resp. I_7, I_8) corresponds to its opposite operation D_0 (resp. D_3, D_4). The operation I_9 is not part of *Move Method* since it is a post-operation applied after the move to customize the insertion.

4) Lossy evolution graph compression: The previous grouping step produces as output an oriented graph of grouped evolutions that is still too large and not usable for an automated and efficient exploration. This new step transforms the graph of grouped evolutions into a *tree of evolutions* (see Figure 4). To do so, the transformation starts from the root node and analyzes its children, namely D_0, I_1, and U_2. The transformation then considers their upper group of evolutions and parenthood that links those groups:

- U_2 has no upper group so it is left alone ($N1$).
- The *Move Method* group requires the target class to exist so that the *Insert Class* group must be its parent (so $N2$ is the parent of $N3$).
- Three atomic evolutions of *Insert Class* (I_5, I_7, and I_8) are related to the *Move Method* group, so that the *Insert Class* can be reduced to I_1 and I_6 (group $N2$) and have as unique child the *Move Method* group (group $N3$).
- I_9 refers to an additional insertion done after the move, so that it is left alone in $N4$ and has as parent $N3$.

Compared to the initial graph of atomic evolutions, this step significantly reduces the number of cases to consider.

D. Qualifying evolutions effects

1) Sequentializing evolutions: In step $\boxed{D}$ (Figure 3), to evaluate the functional effects of each group of evolutions on tests, we need to apply those evolutions to then run the tests. To do so, our approach now takes as input the tree of groups of evolutions produced by the step '*Lossy compression*' (Section III-C4). The approach now *sequentializes* the compressed tree (see Figure 4), *i.e.,* it transforms this tree into sequences of evolutions to apply. This transformation uses a gray code[1] to go through all possible combinations of leafs in the tree of evolutions. Then, parents evolutions are interleaved, following two cases: if the leaf corresponds to an insertion then its parents should be inserted recursively (if not already done); if the leaf corresponds to a deletion then its parents with no child must be removed. Note that the entire group of evolutions must be applied before we can run the tests.

For example, with Section III-C4, at the root node the sequence is empty. The first child is I_1, but this sequence is incomplete (I_6 missing) so that it cannot be evaluated. I_6 is then added to complete this sequence. Recursively, atomic evolutions of N3 complete another sequence to evaluate: $[I_1, I_6, D_0, D_3, D_4, I_5, I_7, I_8]$. I_9 then completes another sequence.

2) Qualifying sequences of evolutions: The qualification is based on compilation and test execution results. For example with Figure 4, we assume that those changes compose the commit t_x. The approach applies on the code at t_{x-1} (*i.e.,* before applying t_x) the first possible sequence: $[I_1, I_6]$. This sequence gives as output one of the qualifications we introduced previously: PCFAIL, TCFAIL, TFAIL, or TPASS.

It then applies the next sequence: $[I_1, I_6, U_6]$. In an optimization purpose, the approach does not go back to t_{x-1} and apply the full sequence. Instead, it applied on $t_{x-1} + [I_1, I_6]$ the missing operations, namely U_2.

When the next sequence to apply does not contain operations applied during the previous sequence, the approach revert those operations. For example, with the last two sequences of Figure 4: $[\ldots, I_8, I_9, U_2]$ and $[\ldots, I_8, U_2]$. The approach must revert the operation I_9 to obtain the latest sequence. Because our approach relies on the three basic evolution operations (insert, delete, update), to cancel one of these operations the approach applies its opposite: for an insert it applies a delete (vice versa); the opposite of an update is an update that automatically applies its changes in the reverse order.

E. Assembling co-evolutions

Based on the effects of each group of evolutions, this step $\boxed{E}$ (see Figure 4) can now combine evolutions to form co-evolutions. For example, given a group of evolutions A that breaks a test T and an additional group of test evolutions B that fixes back T. Our approach then identifies a co-evolution where A is the cause of the co-evolution and where B is the repair of the co-evolution. Based on Def. 4.1 and 4.2 in Section II, we can qualify the types of co-evolution as *complete* or *partial.*

To find delayed co-evolutions spanning over multiple commits, we search for a group of evolutions that breaks a test in some commits, then we search the same test (*i.e.,* using the signature, move and update evolutions) in a later commit where we locate groups of evolutions that repair the test. It should be noted that, here, we need to decide what to do with evolutions in intermediary commits *i.e.,* whether they are breaking, repairing

[1]An optimal ordering algorithm for sequences of evolutions. See "Section 22.3. Gray Codes" in Press *et al.* [13].

or not involved. This is left as a future investigation for our approach.

F. Implementation

We implemented our approach to support Java Maven projects that use Git. Our implementation is open-source and freely available in our companion Web page[2].

Technically, it is implemented in Java and interfaces with *Jgit*, *GumTree* [8], *RefactoringMiner* [6], [7], and *Spoon* [14] for performing the steps $\boxed{A}$ to $\boxed{C}$ (see Figure 2. The tool stores all the results in a *Neo4j* graph database.

For qualifying groups of evolutions (step $\boxed{D}$) the tool relies on Maven and its build lifecycle phases that permits to: compile production code, compile test code, run specific tests, and get the status of these phases. For example, to qualify complete co-evolutions, Maven must gives us: 1) TPASS before applying the *cause* evolutions; 2) TFAIL or TCFAIL right after those evolutions; 3) TPASS again after having applied the *repair* evolutions.

One can query all the results in the database using the Cypher query language [15]. We did so to extracting data we discuss in the next section dedicated to the evaluation of the proposal based on this implementation.

IV. Evaluation

This section presents the evaluation of our approach. All the materials and data of this empirical study are freely available on the companion Web page. First, we present the research questions. Then, we present the data set and evaluation process before we discuss the obtained results. We finally discuss the threats to validity and the scope of the approach.

We ran the implementation of our approach on the following hardware configuration: *2 x Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz; 187Gb ram; 1 T SSD*, running *Ubuntu*.

A. Research Questions

RQ1: Can we detect code and test co-evolutions with precision? This aims to assess the ability of our approach in finding co-evolutions, by looking both at their quality and representativity.

RQ2: Can we detect immediate and delayed co-evolutions and in what ratio? This aims to assess whether our approach can find both types of co-evolutions and to shed light on their frequency.

RQ3: To what extent delayed and immediate co-evolutions are similar or different? Through this question we aim at scrutinizing those co-evolutions to characterize them.

RQ4: What is the time performance of the approach? We aim at discussing to what extent the approach scales.

B. Experimental Protocol

To the best of our knowledge, our approach is the first one that permits to automatically find code and test co-evolutions from histories. So, we had to select representative software systems to analyze. This sub-section details the selection criteria we used and the resulting data set of software systems. We then detail the dependent variables.

Data Set.

The source of our data set is GitHub. We aimed to select Java projects that we can compile, are well tested and having a rich evolution history. To gather such data, we took the following steps:

1) We queried GitHub[3] to retrieve Java projects with more than five stars and that use Maven (*i.e.,* have a *pom.xml* file) so that dependencies, builds, and tests are automatically handled. We then made an intersection with the qualitative data set[4] of Allamanis *et al.* [16] who aimed to study coding and testing practices in Java. With this first step, we found 3588 repositories that should now be filtered.
2) We then selected projects that have at least two releases on GitHub. This selection has multiple beneficial effects: a) We consider that as a possible maturity threshold; b) Zaidman *et al.* [3] have shown clear development cycles between releases; c) It filters out initial commits, which are often difficult to interpret due to their size and uniqueness; d) We made the hypothesis that the longest a code history is, the more it might contain test and co-evolutions (because of maintenance and evolution operations); This filtering resulted in 3588 repositories.
3) Even though we selected projects with releases, it is still not a guarantee that they compile and build. Thus, for each project we tried to construct the AST and build the ten latest releases and filtered those that did not build at least once. Thus, we kept at the end 395 repositories with 5439 releases in total (with an average of 14 releases per repository).
4) This work focuses on code and test co-evolution. Tests are also crucial in our approach since it uses tests to qualify the nature of the effect of evolutions. So, we further selected projects that have a significant number of tests (JUnit tests in our case). We fixed this minimal threshold of tests to 50. This resulted in 164 repositories to analyze.
5) As we consider complex and refactoring evolutions in this paper, we aimed at detecting co-evolutions involving refactorings. As such, we found 30 122 refactorings on 91 projects. Then, we extracted the ones with at least 10 refactorings, thus, **reaching** 45 **repositories**.

Dependent Variables.

- **Precision.** For measuring the precision, we first applied our automated approach to obtain a set of possible co-evolutions. We then check them manually, but since this is a very time-consuming task, we only scrutinized *complete* co-evolutions to state whether they were indeed valid ones. On these *complete* co-evolutions, we computed the precision as follows.

$$precision = \frac{|coevolutions_{correct}|}{|coevolutions_{detected}|} \times 100$$

[2]https://github.com/quentinLeDilavrec/ICSME2021

[3]Using *PyGithub.py* library https://github.com/PyGithub/PyGithub

[4]http://groups.inf.ed.ac.uk/cup/javaGithub/

- **Recall.** For measuring the recall, we need to have a ground truth of test and code co-evolutions. Such a ground truth would permit the identification of the co-evolutions our approach does not detect. However, to the best of our knowledge our approach is the first one that automatically detects code and test co-evolutions, so that no ground truth exists.
  Manually detecting co-evolutions in the analyzed repositories to obtain a ground truth is not possible: for immediate co-evolutions, this would require to manually test the effects of several evolutions together. So for a commit with n atomic evolutions, this would imply $2^n \times m$ cases to evaluate. It is even more challenging to start looking for delayed co-evolutions as the number of atomic evolutions to consider mechanically increases. The evaluation of one single case requires to run each test before and after the code evolutions, and after the tests evolutions to qualify the evolutions as co-evolutions or not. This becomes infeasible to perform manually.
  Nonetheless, we can compute the minimal recall to have an underestimation of it. To do so, when a code evolution statically impacts a test, we make the overestimating hypothesis this test is part of a code and test co-evolution. So, given $|tests_{impacted}|$ the number of tests we found statically impacted by evolutions, and $|tests_{coevolved}|$ the number of such tests already part of the co-evolutions we identified, we computed the recall this way:
  $$recall = \frac{|tests_{coevolved}|}{|tests_{impacted}|} \times 100$$
- **Execution time.** Through the execution time we aim at evaluating the scalability of the proposed approach (RQ4). This variable computes the mean execution time of the approach on: one commit; one release; one repository. Given tt the overall execution time in seconds spent for analyzing all the repositories, $|repos|$ the total number of analyzed repositories, $|commits|$ the total number of commits computed from all the repositories, we compute $time_p$ (time per repository) and $time_c$ (time per commit) as follows:
  $$time_p = \frac{tt}{|repos|} \qquad time_c = \frac{tt}{|commits|}$$

C. Results

From the 45 repositories, we analyzed in total 6738 commits and 78 million lines of Java code.

RQ1: Can we detect code and test co-evolutions with precision?

Table I reports the co-evolutions found in the analyzed repositories.

Regarding complete co-evolutions, in total, our approach automatically detected 202 co-evolutions among which 88 co-evolutions involved at least one refactoring evolution (first line of Table I). These values concern both immediate and delayed co-evolutions (we discuss in details about these two different types of co-evolutions in RQ2). We computed a precision of 100 % and a recall of 37.5 % for complete co-evolutions. To compute the recall, the value $|tests_{impacted}|$ equals 4997, while the value $|tests_{coevolved}|$ equals 1877. As previously discussed, our recall result is underestimated. To precise this value, we encourage researchers to replicate our experiment to potentially find missing co-evolutions. The co-evolutions we identified would serve them as the minimal ground truth for computing their recall.

Regarding partial co-evolutions, Table I reports the results for all detected co-evolutions in the different cases of the initial states of the co-evolved tests. We found 111 additional partial co-evolutions where the tests were failing, then impacted by the code evolution before to be repaired to their initial state. We also observed 276 passing tests and 23 failing tests where they respectively were degraded but did not come back to their initial state after being partially repaired.

TABLE I
COMPLETE AND PARTIAL CO-EVOLUTIONS DETECTION RESULTS.

Status of tests		Co-evolutions		Type
Initial	Final	All	With ≥1 refactoring	
TPASS	TPASS	202	86	Complete
TFAIL	TFAIL	111	30	Partial
TPASS	TFAIL	79	13	Partial
TPASS	TCFAIL	197	115	Partial
TFAIL	TCFAIL	23	1	Partial
	Total	612	245	

Regarding the semantic of partial co-evolutions, we identified two different facets. First, a partial co-evolution can be part of a complete co-evolution (our implementation has detected or not), *i.e.,* a step towards completing a co-evolution (we discuss the co-evolutions our tool may miss in the threats to validity section). Such partial co-evolutions are interesting to understand how complete co-evolutions are formed. For example, partial co-evolutions can be used as a contrast to complete co-evolutions for drawing lists of best and bad practices in code and test co-evolutions, analogously to design pattern and anti-pattern [17]. Second, a partial co-evolution may exist because some tests did not pass, *i.e.,* developers never fixed them. We think that such partial co-evolutions are in minority in our results because of the criteria we used to select mature projects.

RQ1 insights:

- Our approach detects complete code and test co-evolutions with a precision of 100 % and a recall of at least 37.5 %.
- We give evidences of the existence of *partial co-evolutions*. Detecting partial co-evolution shows our approach flexible ability to cover different scenarios.
- Thanks to those co-evolutions, we produced a knowledge base that practitioners can use, for example, as a ground truth in experiments.

RQ2: Can we detect immediate and delayed co-evolutions and in what ratio?

TABLE II
ALL DETECTED CATEGORIES OF CO-EVOLUTIONS.

Status of tests		**Immediate Co-evolutions**		**Delayed Co-evolutions**		**Type**
Initial	**Final**	**All**	**With ≥1 refactoring**	**All**	**With ≥1 refactoring**	
TPASS	TPASS	140	82	62	6	Complete
TFAIL	TFAIL	62	28	49	2	Partial
TPASS	TFAIL	78	12	1	1	Partial
TPASS	TCFAIL	197	115	0	0	Partial
TFAIL	TCFAIL	23	1	0	0	Partial
	Total	500	238	112	3	

Table II displays our results for immediate and delayed co-evolutions for both complete and partial co-evolutions. Regardless of the type of co-evolutions (complete, partial, immediate, delayed), our approach found a total of 612 co-evolutions. The total number of immediate co-evolutions (complete and partial) is 500, so 81.69 %. The total number of delayed co-evolutions (complete and partial) is 112, so 18.30 %. Moreover, we observe a slightly different ratio on the 202 complete co-evolutions with a bit more of immediate ones. In particular, 140 (69.30 %) are immediate complete co-evolutions and 62 (30.69 %) are delayed complete co-evolutions.

Finally, we calculated the gap in delayed co-evolutions as the number of commits between the commit of the cause and the repair. For example, a gap of 2 means that the cause is in a first commit and the repair in the next commit.

We observed that 69, 3, 1, 13, and 26 of the delayed co-evolutions (both complete and partial), respectively, come with a gap of 2, 3, 4, 5, 8. Regarding complete delayed co-evolutions, 59 come with a gap of 2, while 3 others, respectively, come with a gap of 3, 4, and 5 commits. Note that in order to scale when detecting delayed co-evolutions, we limited our search up to a gap of 30 commits.

RQ2 insights:

- Our approach is able to automatically spot *immediate* and *delayed* co-evolutions.
- We observed a ratio of 69.31 % of immediate complete co-evolutions and 30.69 % of delayed complete co-evolutions, suggesting that developers seem to co-evolve their tests while performing evolutions on production code.
- We observed mostly a distance of two commits in delayed co-evolutions while the rest varied between four and eight commits.

RQ3: To what extent delayed and immediate co-evolutions are similar or different?

We now look in depth at what are the differences or similarities of immediate and delayed co-evolutions.

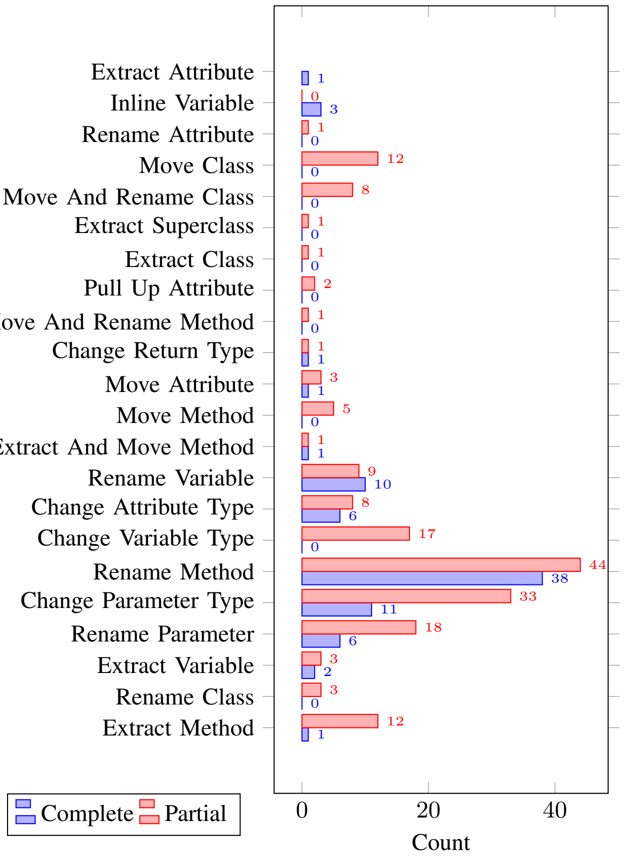

Fig. 5. Refactoring evolutions in immediate co-evolutions.

First, we had a look at the evolutions constituting the detected co-evolutions, in particular at what refactoring evolutions are part of both types of co-evolutions. Figures 5 and 6 give the types of refactoring and their frequency we found in respectively the spotted 272 refactorings in immediate and 14 refactorings delayed co-evolutions.

We observed that when a refactoring evolution is part of a code and test co-evolution, it is most likely to occur within an immediate co-evolution (272), whereas, delayed co-evolutions rather implicate atomic changes and few refactorings (14). This suggests or can be explained as when developers perform a significant evolution to the code, they may tend to ensure their consistency with the tests immediately, hence, performing immediate co-evolutions. Whereas, developers may miss impacts of small or minor code evolutions on the tests, hence, performing delayed co-evolutions.

Moreover, we investigated the length of the dependency chains (in edges) in both types of co-evolutions. Our hypothesis is that the shorter the dependency chain is, the likelier the co-evolution to be immediate, and vice versa. On average, the length of the dependency chain for an immediate and a delayed co-evolution, respectively, is 10.84 and 10.44. Thus, our hypothesis is rejected based on the results. This is surprising, because one may think as the more close the code evolution to the tests, the likelier developers will see it and fix it immediately

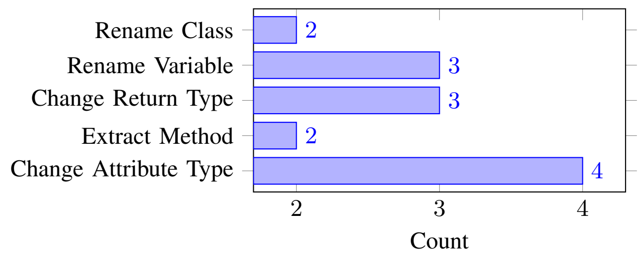

Fig. 6. Refactoring evolutions in delayed Co-evolutions.

and not delay it in further commits. Nonetheless, future research remains necessary to further investigate other reasons behind the delayed co-evolutions.

RQ3 insights:

- Immediate co-evolutions tend to have far more refactoring evolutions (272) than in delayed co-evolutions (14). This may suggest how developers handle co-evolutions based on the size of the committed evolutions.
- Our hypothesis that the immediate co-evolutions tend to have shorter dependency paths between the cause and the repair than the delayed co-evolutions is rejected.

RQ4: What is the time performance of the approach?

The overall execution time (tt) our tool spent for analyzing the whole 164 ($|repos|$) repositories[5] is $90\,720\,\mathrm{min}$ ($1512\,\mathrm{h}$). The value $|commits|$ is $15\,954$.

So $time_p$, the average time per repository, is $553.17\,\mathrm{min}$ ($9.22\,\mathrm{h}$). $time_c$, the average time per commit, is $5.67\,\mathrm{min}$. Of course these mean values depend on: the number of commits per repository; the number of evolutions per commit. As an indication, Figure 7 depicts a boxplot of the number of evolutions per commit, with a median value of 10 atomic evolutions per commit.

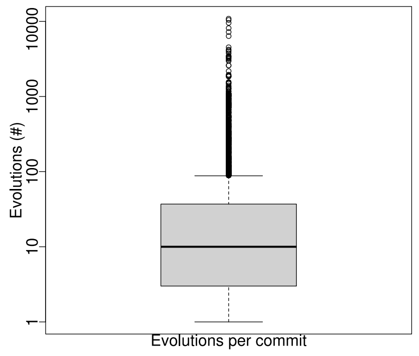

Fig. 7. Number of atomic evolutions per commit

[5]We had to apply the approach on the 164 to perform the filtering step 5) on the data set. We consider here those 164 repositories instead of the curated set of 45 repositories to have more representative results.

RQ4 insights:

- The computed durations show that our proposal scales but still requires considerable resources to perform large-scale studies.
- The time it takes to automatically analyze the code base typifies the extreme difficulty to perform such analyzing tasks manually, thus motivating our proposal.

D. Threats to Validity

Beyond the threats to validity that may affect our experiment, this section also discusses the scope of our proposal.

External validity. We used selection criteria to find a large representative panel of Java code repositories, stored on Github. We consider Maven projects, a build automation tool widely used in the industry. We do not think that the use of another build automation tool, such as Gradle, would have an impact on detection of code and test co-evolutions.

We use Github releases during the selection process of projects to analyze. A Github release is associated to a Git tag. This helped us in filtering Git tags of interests. We do not think that the use of other version control systems (VCS) such as Mercurial would prevent the use of our proposal since it is based on atomic evolutions common to all VCS. The main challenge is related to RefactoringMiner that our approach relies on for detecting refactorings and that works with Git. We would have to find a similar tool for other VCS. Finally, we cannot generalize our results to software repositories in other languages than Java.

Internal validity. We conducted manual analyses to measure the precision of the approach. Such a manual analysis is error-prone, so to overcome this threat two persons performed all the manual analysis. They then compared their results. On divergent results, these two persons discussed to converge to a final decision. These persons are authors of the paper with a high expertise in detecting code and test co-evolutions. Note that as non-experts of the analyzed projects, these two persons cannot detect tests of poor quality, *i.e.,* tests that still pass while they should not (missing assertion, flaky tests, *etc.*). To limit this issue, we designed criteria for selecting relevant projects.

The exact computation of the recall is not technically possible due to the combinatorial explosion of the number of cases to evaluate when establishing ground truth. To overcome this issue we underestimated the real recall of our approach by over-estimating the possible existing co-evolutions. This way, we do not overclaim on the performance of our approach.

Construct validity. Our approach relies on *Gumtree* and *RefactoringMiner* for detecting atomic and complex evolutions. The precision and recall of these tools have an impact on the performance of our proposal. To limit this threat we validated by hand the co-evolutions found by our approach.

To qualify evolutions, we rely on code compilation and test execution verdicts. However, a passing test suite after a production code change does not certify that this change has no effect on the test suite: a test of low quality might not spot regressions. We consider this threat while designing the selection criteria of the projects to analyze. Moreover, one can consider other assessable properties that compilation result and test verdict to spotting co-evolutions. We think that compilation results and test verdicts are still good enough to cover a large majority of test and code co-evolutions.

Closely related, the relevance of the selected projects has an impact on the evaluation results. To limit this threat, we designed rigorous selection criteria.

Regarding the recall of our approach, we use our static analyzing technique for spotting tests that are statically impacted by evolutions. Our technique might miss some cases, in particular because our approach does not consider code reflexivity.

Regarding the detection of delayed co-evolutions, we use a maximal threshold of 30 commits between commits to analyze because of resource limitations. If such a threshold may miss complete co-evolutions we think that its value 30 is high enough to limit this effect.

Finally, Git and Github have several strategies for merging branches that may affect the computation of immediate and delayed co-evolutions. For example, the *Squash* technique squashes all the commit of the branch into a single commit. The *Rebase* technique individually adds each commit of the branch into the base branch. Studying commits and branches structures is another combinatorial challenge. We only considered the most direct sequences of commits between releases.

V. Related Work

Co-evolution is a hot topic in software engineering. Multiple approaches addressed co-evolution of various artifacts, such as models [18], [19], [20], [21], [22], [23], [24], [25], constraints [26], [27], [28], [29], [30], model transformations [31], [32], [33], [34], [35], and code [36], [37], [38], [39], [40]. In this section, we present the main related work w.r.t. detecting and investigating co-evolution of code and test.

Few empirical studies were conducted on this topic. Zaidman *et al.* [2], [3] performed an empirical analysis to understand the co-evolution of tests on three software projects and focused on changes at files level and not actual evolutions of code and test. Thus, in contrast to us, they do not identify actual code and test co-evolutions, but abstract co-evolution behavior at the files level. As motivated in our work, identifying code and test co-evolutions with precision is hardly feasible because of the number of possible combinations to evaluate.

Lubsen *et al.* [4] proposed to use association rules between code and test to study their co-evolution. Marsavina *et al.* [41] proposed to mine association rules on five software projects before detecting six co-evolutions based on atomic changes. Levin *et al.* [5] also studied tests co-evolution empirically on 61 software projects and focused on impacts of atomic changes in the code.

However, all these studies [2], [3], [4], [5] considered only fine-grain and atomic changes in the code and test co-evolution. In contrast to our approach, we consider both atomic and complex evolutions, such as refactoring operations. In addition, we propose to detect code and test co-evolutions fully-automatically in contrast to [2], [3], [4], [5] who did it manually. Finally, none attempted to quantify their precision or recall regarding the observed co-evolutions, and none even consider to distinguish between complete and partial co-evolutions. We not only are the first to shed light on complete and partial co-evolutions, but we also provide evidence for their presence.

Other works aimed to detect co-evolutions not for code and test, but for class co-evolution [42], component co-evolution [43], co-evolution of business processes [44], and co-evolution of comments and code [45].

To the best of our knowledge, our work is the first to attempt detecting automatically code and test co-evolutions. One novelty in our approach is it relies on dynamic analyzes (*i.e.,* compilation and test execution) to detect qualitative co-evolutions. Another novelty is related to the types of co-evolution, *i.e.,* complete and partial, while detecting them in both immediate and delayed settings.

VI. Conclusion

We proposed, to the best of our knowledge, the first approach for automatically detecting code and test co-evolutions in object-oriented code. Our approach can detect co-evolutions contained in a single commit (immediate co-evolution) but also co-evolutions scattered over multiple commits (delayed co-evolution). We implemented our approach in a tool that analyzes Java code stored in Git repositories. Our implementation is fully open-source and available for replicating studies.

We evaluated our approach by conducting an empirical study on a curated data set of 45 repositories using our implementation. Our approach found 202 complete and 410 partial co-evolutions. We also detected 140 immediate and 62 delayed complete co-evolutions. For delayed co-evolutions, the gap between the cause and repair commits varied from two to 8. Finally, we also observed that the dependency chains tend to be the same in immediate and in delayed co-evolutions. The evaluation permitted the creation of the first ground truth of test and code co-evolutions that practitioners can use in their future work.

In our future work, we envision the use of this ground truth to design a recommendation engine for code and test co-evolution. We also aim at poring over the ground truth to analyze, understand, and explain at a large scale how developers do test and code co-evolutions. We also plan to consider other software artifacts in addition to code and test, such as issues and dependency configurations. In particular, considering dependencies may increase the quantity of found co-evolution.

Acknowledgments.

The research leading to these results has received funding from the *RENNES METROPOLE* under grant *AIS no. 19C0330* and from *ANR* agency under grant *ANR JCJC MC-EVO*^{2} *204687*.

References

[1] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "A comprehensive study of pseudo-tested methods," *ESE*, vol. 24, no. 3, pp. 1195–1225, 2019.

[2] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *2008 1st international conference on software testing, verification, and validation*. IEEE, 2008, pp. 220–229.

[3] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *ESE*, vol. 16, no. 3, pp. 325–364, 2011.

[4] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production & test code," in *MSR*. IEEE, 2009, pp. 151–154.

[5] S. Levin and A. Yehudai, "The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes," in *ICSME*. IEEE, 2017, pp. 35–46.

[6] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *ICSE*. ACM, 2018, pp. 483–494.

[7] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *TSE*, 2020.

[8] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ASE*, 2014, pp. 313–324.

[9] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, 1999.

[10] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *TSE*, vol. 33, no. 11, pp. 725–743, 2007.

[11] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, no. 1, pp. 26–33, 2009.

[12] D. Silva, J. Silva, G. J. D. S. Santos, R. Terra, and M. T. O. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *TSE*, 2020.

[13] W. H. Press, H. William, S. A. Teukolsky, A. Saul, W. T. Vetterling, and B. P. Flannery, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[14] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.

[15] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *International Conference on Management of Data*, 2018, pp. 1433–1445.

[16] M. Allamanis and C. Sutton, "Mining Source Code Repositories at Massive Scale using Language Modeling," in *MSR*. IEEE, 2013, pp. 207–216.

[17] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh, and M. Zulkernine, "Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults," *Empirical Software Engineering*, vol. 21, no. 3, pp. 896–931, 2016.

[18] W. Kessentini, M. Wimmer, and H. Sahraoui, "Integrating the designer in-the-loop for metamodel/model co-evolution via interactive computational search," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, 2018, pp. 101–111.

[19] W. Kessentini, H. Sahraoui, and M. Wimmer, "Automated metamodel/-model co-evolution: A search-based approach," *Information and Software Technology*, vol. 106, pp. 49–67, 2019.

[20] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*. IEEE, 2008, pp. 222–231.

[21] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "Cope-automating coupled evolution of metamodels and models," in *ECOOP 2009–Object-Oriented Programming*. Springer, 2009, pp. 52–76.

[22] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin, "Managing model adaptation by precise detection of metamodel changes," in *Model Driven Architecture-Foundations and Applications*. Springer, 2009, pp. 34–49.

[23] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *ECOOP*. Springer, 2007, pp. 600–624.

[24] R. F. Paige, N. Matragkas, and L. M. Rose, "Evolving models in model-driven engineering: State-of-the-art and future challenges," *JSS*, vol. 111, pp. 272–280, 2016.

[25] R. Hebig, D. E. Khelladi, and R. Bendraou, "Approaches to co-evolution of metamodels and models: A survey," *TSE*, vol. 43, no. 5, pp. 396–414, 2016.

[26] E. Batot, W. Kessentini, H. Sahraoui, and M. Famelis, "Heuristic-based recommendation for metamodel—ocl coevolution," in *MODELS*. IEEE, 2017, pp. 210–220.

[27] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais, "Metamodel and constraints co-evolution: A semi automatic maintenance of ocl constraints," in *International Conference on Software Reuse*. Springer, 2016, pp. 333–349.

[28] D. E. Khelladi, R. Bendraou, R. Hebig, and M.-P. Gervais, "A semi-automatic maintenance and co-evolution of ocl constraints with (meta) model evolution," *JSS*, vol. 134, pp. 242–260, 2017.

[29] A. Correa and C. Werner, "Refactoring object constraint language specifications," *Software & Systems Modeling*, vol. 6, no. 2, pp. 113–138, 2007.

[30] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, J. Schoenboeck, W. Schwinger, and M. Wimmer, "Systematic co-evolution of ocl expressions," in *11th APCCM 2015*, vol. 27, 2015, p. 30.

[31] W. Kessentini, H. Sahraoui, and M. Wimmer, "Automated co-evolution of metamodels and transformation rules: A search-based approach," in *International Symposium on Search Based Software Engineering*. Springer, 2018, pp. 229–245.

[32] D. E. Khelladi, R. Kretschmer, and A. Egyed, "Change propagation-based and composition-based co-evolution of transformations with evolving metamodels," in *Conference on Model Driven Engineering Languages and Systems*. ACM, 2018, pp. 404–414.

[33] K. Garcés, J. M. Vara, F. Jouault, and E. Marcos, "Adapting transformations to metamodel changes via external transformation composition," *Software & Systems Modeling*, vol. 13, no. 2, pp. 789–806, 2014.

[34] J. García, O. Diaz, and M. Azanza, "Model transformation co-evolution: A semi-automatic approach," *SLE*, vol. 7745, pp. 144–163, 2013.

[35] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, W. Schwinger, and J. Schonbock, "Consistent co-evolution of models and transformations," in *ACM/IEEE 18th MODELS*, 2015, pp. 116–125.

[36] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux, "Maintaining invariant traceability through bidirectional transformations," in *ICSE*. IEEE, 2012, pp. 540–550.

[37] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution," in *ICSE*. IEEE, 2005, pp. 274–283.

[38] 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," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 302–321, 2010.

[39] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *TOSEM*, vol. 20, no. 4, p. 19, 2011.

[40] J. Andersen and J. L. Lawall, "Generic patch inference," *Automated software engineering*, vol. 17, no. 2, pp. 119–148, 2010.

[41] C. Marsavina, D. Romano, and A. Zaidman, "Studying fine-grained co-evolution patterns of production and test code," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 195–204.

[42] Z. Xing and E. Stroulia, "Data-mining in support of detecting class co-evolution." in *SEKE*, vol. 4. Citeseer, 2004, pp. 123–128.

[43] L. Yu, "Understanding component co-evolution with a study on linux," *ESE*, vol. 12, no. 2, pp. 123–141, 2007.

[44] T. Bodhuin, R. Esposito, C. Pacelli, and M. Tortorella, "Impact analysis for supporting the co-evolution of business processes and supporting software systems." in *CAiSE Workshops (2)*. Citeseer, 2004, pp. 146–150.

[45] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, 2009.