Automated Testing of Metamodels and Code Co-evolution

Zohra Kaouter Kebaili, Djamel Eddine Khelladi, Mathieu Acher, Olivier Barais

▶ **To cite this version:**

Zohra Kaouter Kebaili, Djamel Eddine Khelladi, Mathieu Acher, Olivier Barais. Automated Testing of Metamodels and Code Co-evolution. Software and Systems Modeling, 2024, pp.1-32. 10.1007/s10270-024-01245-2 . hal-04932177

HAL Id: hal-04932177
https://hal.science/hal-04932177v1

Submitted on 6 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Automated Testing of Metamodels and Code Co-evolution

Zohra Kaouter Kebaili[1*], Djamel Eddine Khelladi[1†], Mathieu Acher[2†], Olivier Barais[3†]

[1*]CNRS, Univ. Rennes 1, IRISA, INRIA, Rennes, 35000, France.
[2*]INSA, IUF, IRISA, Inria, Rennes, 35000, France.
[3*]Univ. Rennes 1, IRISA, INRIA, Rennes, 35000, France.

*Corresponding author(s). E-mail(s): zohra-kaouter.kebaili@irisa.fr;
Contributing authors: djamel-eddine.khelladi@irisa.fr;
mathieu.acher@irisa.fr; olivier.barais@irisa.fr;
†These authors contributed equally to this work.

Abstract

Metamodels are cornerstone in MDE. They define the different domain concepts and the relations between them. A metamodel is also used to generate concrete artifacts such as code. Developers then rely on the generated code to build their language services and tooling, e.g., editors, checkers. To check the behavior of their client code, developers write or generate unit tests. As metamodels evolve between releases, the generated code is automatically updated. As a consequence, the additional developers' code is impacted and is co-evolved accordingly for each release. However, there is no guarantee that the co-evolution of the code is performed correctly. One way to do so is to re-run all the tests after each code co-evolution, which is expensive and time-consuming.

This paper proposes an automatic solution for tracing impacted tests due to metamodel evolution. Thus, we end up matching metamodel changes with impacted code methods and their corresponding tests both in the original and evolved versions of a given project. After that, we map the two versions of the impacted tests and compare them to analyze the behavior of the code before and after its evolution due to the metamodel evolution. In particular, we implemented an Eclipse plugin that allows tracing, mapping, execution, and reporting back the results to the developers for easier in-depth analysis of the effect of metamodel evolutions rather than analyzing the whole test suite.

We first ran a user study experiment to gain evidence on the difficulty or not of the manual task of tracing impacted tests. We found that manually tracing the

tests impacted by the evolution of the metamodel is a hard and error-prone task. Not only the participants could not trace all tests, but they even wrongly traced non-impacted tests. We then evaluated our approach on 18 Eclipse projects from OCL, Modisco, Papyrus, and EMF over several evolved versions of metamodels. For the 14 projects without manual tests, we generated a test suite for each release with the state-of-the-art tool EvoSuite. The results show that we successfully traced the impacted tests automatically by selecting 1608 out of 34612 tests due to 473 metamodel changes. When running the traced tests before and after co-evolution, we observed cases indicating possibly both behaviorally correct and incorrect code co-evolution. Finally, we reached gains representing, on average, a reduction of 88% in the number of tests and 84% in the execution time.

Keywords: Metamodel evolution, code co-evolution, Unit tests, testing co-evolution

1 Introduction

Model-driven engineering (MDE) is a state-of-art software engineering approach for supporting the increasingly complex construction and maintenance of large-scale systems [1–3]. In particular, MDE allows domain experts, architects, and developers to build languages and their tools that play an important role in all phases of the development process [4].

A central artifact in MDE when building languages is the *metamodel* that defines the aspects of a business domain, i.e. the main concepts, their properties, and the relationships between them [5]. A metamodel is the cornerstone to not only specify model instances, constraints, or transformations, but also the code when building the necessary language tooling, e.g., editor, checker, compiler, data access layers, etc. In particular, metamodels are used as inputs for complex code generators that leverage the abstract concepts defined in metamodels. *Eclipse Modeling Framework (EMF)* [6] is a prominent example that supports the generation of Java code consisting of a core code API for creating, loading and manipulating the model instances, adapters, serialization facilities, and an editor, all from the metamodel elements. This generated code is further enriched by developers to offer additional functionalities and tooling, such as validation, transformation, simulation, or debugging. A metamodel and its generated code API are, hence, a cornerstone when building a language and its tooling. For instance, UML[1] and BPMN[2] Eclipse implementations rely on the UML and BPMN metamodels to generate their corresponding code API before building around it all their tooling and services in the additional code.

One of the foremost challenges to deal with in *MDE*, is the impact of the evolution of metamodels on its dependent artifacts, in particular, the impacted code. Indeed, when a metamodel evolves between two releases, and as the core API is re-generated again, the additional code implemented by developers can be impacted. As a consequence, it is co-evolved accordingly by the developer in the next release. However, while developers co-evolve their code either manually or automatically, they cannot

[1] https://www.eclipse.org/modeling/mdt/downloads/?project=uml2
[2] https://www.eclipse.org/bpmn2-modeler/

ensure that the code co-evolution is behaviorally correct, i.e., without altering the behavior of their impacted code. Especially, when there is alternative co-evolution of the same impacted code. One way to check this is to re-run all the tests after each code co-evolution, which is expensive and time-consuming. It is also tedious and error-prone when the developer checks the output of the tests' execution and manually maps them between the original and evolved versions. Whereas several existing approaches automate the metamodels and code co-evolution [7–14], to the best of our knowledge, they do not focus on checking the behavioral correctness of the co-evolved code and do not trace till the impacted tests.

This paper proposes a new fully automatic approach to check the behavioral correctness of the code co-evolution between different releases of a language when its metamodel evolves. We leverage the test suites of the original and evolved versions of the language, and hence, its metamodels and code. Test suites are usually used to check the code is behaviorally correct. In our work, we use unit test suites before and after code co-evolution to check that the co-evolution did not alter the behavior of the code. The approach first takes as input the metamodel evolution changes and then parses the code to compute the code call graph (CCG). With the changes and the CCG, we first locate all usages of the metamodel elements in the generated code. For example, a getter/setter of a metamodel attribute/reference, interface, the class implementation, etc. After that, we recursively trace the code usages of the metamodel elements in the CCG throughout the methods calls in the additional code until reaching the test methods. Thus, we end up matching the metamodel changes with impacted code methods and their corresponding tests. We perform this step on both releases corresponding to the original and evolved metamodels and code to be able to check the behavioral correctness of the code before and after co-evolution. We implemented our approach in an Eclipse plugin that allows to trace the tests, map them with state-of-the-art solution GumTree [15] and execute them. Then, we report them back in a form of diagnostic to the developers for an easier in-depth analysis of the effect of metamodel evolution rather than re-running and analysing the whole test suite.

A first part of the evaluation consisted of an user study experiment to gain evidence on the difficulty or not of the manual task of tracing impacted tests after metamodel evolution and co-evolution. We found that tracing manually the tests impacted by the evolution of the metamodel is a hard and error-prone task. Not only the participants could not trace all tests, but they even wrongly traced non-impacted tests. The post-questionnaire results after a demonstration of our automatic approach suggest its high usefulness and adoption likelihood. We then evaluated our approach on 18 Eclipse projects from OCL, Modisco, Papyrus, and EMF over several evolved versions of metamodels. For four projects we had manually written tests. For the 14 projects without manual tests, we generated a test suite for each release with the best available state-of-the-art tool EvoSuite [16]. Results show that we automatically traced 1608 out of 34612 tests based on 473 metamodel changes. When running the traced tests before and after co-evolution, we observed the two cases, indicating possibly both behaviorally incorrect and correct code co-evolution. Thus, helping the developer to locate code co-evolution to investigate in more details. In addition, our approach provided gains representing, on average a reduction of 88% in the number of tests and 84% in execution time.

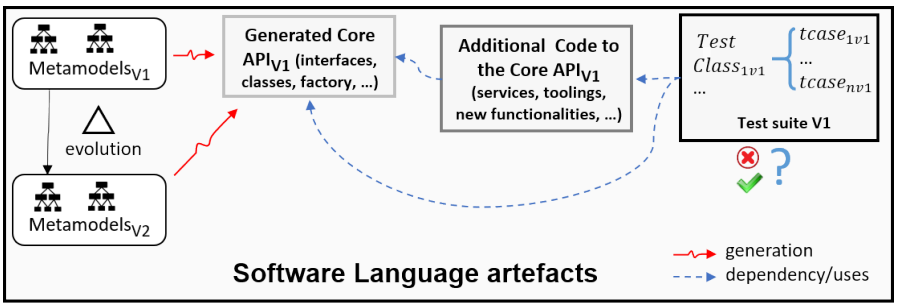

Fig. 1 Evolution of metamodels and related artifacts of a software language

This paper significantly extends our previous vision paper [17] where we laid out the base for the test tracing. In addition to a more detailed approach, we extend this work by additional core contributions of 1) the mapping of impacted tests between original and evolved versions, 2) executing them programmatically, and 3) reporting the result of the mapped tests and the causing metamodel changes in a GUI to the user. Moreover, the preliminary evaluation of the vision paper [17] was extensively improved both 1) in research questions with three added structured questions, 2) in the number of case studies that was extended from 4 projects to 18 projects with both manually written tests and automatically generated tests, and 3) a user study experiment with 8 participants.

The rest of the paper is structured as follows. Section 2 discusses the background. Section 3 presents our approach for test tracing while Section 4 evaluates it. Sections 5 and 6 discuss threats to validity and related work. Finally, Section 7 concludes the paper and discusses future work plan.

2 Background and Example

This section gives a background on how metamodels play a significant role when building software languages and their tooling for a better comprehension of the current work. It then discusses the scenario of co-evolution that arises between metamodels and code, and the need for testing its behavioral correctness.

2.1 Key Concepts

Metamodels are a cornerstone in MDE. It serves to create model instances, constraints, or transformations. In our work, we focus on the relation between metamodels and code. Figure 1 depicts a software language structure and its usage as in practice in the Eclipse platform. Once the metamodels are carefully defined and validated in a given version. The core API code is generated [6] consisting of the class implementations of the metamodel classes, a factory and package classes, etc. All this generated code allows parsing the AST of the metamodels' models instances, navigate in it and modify it. The generated code is enriched with additional code to offer more advanced functionalities. For instance, methods defined in classes in the metamodel are generated without their bodies, which developers must implement. Developers also integrate

additional classes to implement advanced functionalities, such as language services like validation and language tooling like an execution engine. Finally, a test suite is added on top of the generated and the additional code to test the implemented functionalities. This can be done manually or with the existing techniques for automated test generation [16, 18, 19].

However, the generated code, additional code and tests hold for a single version of the metamodel. With metamodel evolution comes the challenges of co-evolution and its correctness. For example, when a metamodel evolves, model instances must be co-evolved. One way to check the models' correctness is to rely on the OCL constraints to verify the static semantic of the models [5, 20]. Thus, one can compare the constraints before and after the models' co-evolution. In our case, when the metamodel evolves, the API code can be re-generated again. As a consequence, the additional code manually integrated by developers must be co-evolved accordingly as well. Unfortunately, there is also no guarantee that the code co-evolution is correct. Usually, the test suite is used to identify possible bugs in the new evolved version of the code. In this work, similarly to the practice of regression testing, we leverage the test suites in both the original and evolved versions of the code to check particularly the behavioral correctness of the co-evolution.

Indeed, in regression testing, the goal is to re-run tests after any code changes to ensure that the software still works as intended [21–23]. In this paper, we intend to follow a similar methodology by tracing the impacted tests that must be re-run to compare their results before and after code co-evolution.

2.2 Motivating Example

This section introduces a motivating example to illustrate the challenge of metamodel and code co-evolution and testing it.

Figure 2 shows an excerpt of the "Modisco Discovery Benchmark" metamodel[3] consisting of 10 classes in version 0.9.0. It illustrates some of the domain concepts `Discovery`, `Project`, and `ProjectDiscovery` used for the discovery and reverse engineering of an existing software system. From these metaclasses, a first code API is generated, containing Java interfaces and their implementation classes, a factory, a package, etc. In version 0.11.0, the Modisco metamodel evolved with several significant changes, among which we find: 1) Renaming the property *totalExecutionTimeInSeconds* to *discoveryTimeInSeconds* in metaclass `Discovery`, followed by 2) Moving the property *discoveryTimeInSeconds* (after its renaming) from metaclass `Discovery` to `DiscoveryIteration`

Listing 1 shows a directly impacted test from the class `DiscoveryImpl_ESTest` after the evolution of Modisco metamodel. It tests directly the evolved method. In the class `DiscoveryIterationImpl_ESTest` of Modisco 0.11.0, the test shown in Listing 2 is impacted indirectly by the same change. Indeed, the method *totalExecutionTimeInSeconds* after its rename and move is used in the method *eSet* as shown in Listing 3, which is in turn used in the unit test shown in Listing 2. It tests indirectly the evolved method.

[3] https://git.eclipse.org/r/plugins/gitiles/modisco/org.eclipse.modisco/+/refs/tags/0.12.1/org.eclipse.modisco.infra.discovery.benchmark/model/benchmark.ecore

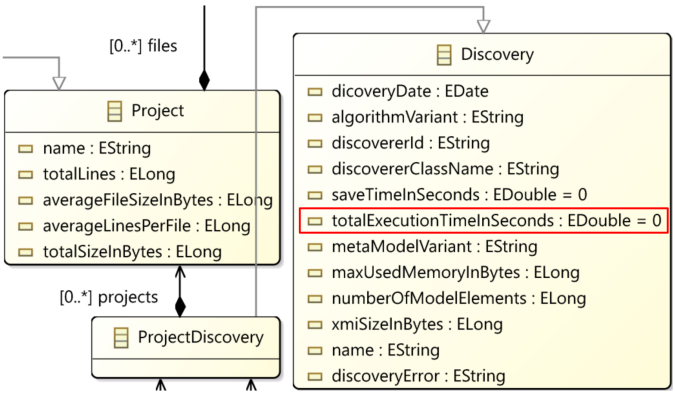

Fig. 2 Excerpt of Modisco Benchmark metamodel in version 0.9.0.

The above examples show the direct and indirect impact of the metamodel evolution on the code and on the tests. However, manually tracing the impact of multiple metamodel evolutions at once till the test is tedious, error-prone and time-consuming. In particular, this tracing must be done before and after the metamodel evolution and then to map the traced tests to investigate the code co-evolution correctness.

The next section presents our contribution for an automated tracing of the tests impacted by the evolution of the metamodel that allows later to check the behavioral correctness of the metamodel and code co-evolution.

Listing 1 Excerpt of a directly impacted test in Modisco.

```
1   @Test(timeout=4000)
2   public void test000() throws Throwable {
3    DiscoveryIterationImpl discoveryIterationImpl0 = new DiscoveryIterationImpl();
4      ...
5      assertFalse(discoveryIterationImpl0.eIsProxy());
6      assertEquals(0.0,  discoveryIterationImpl0.getDiscoveryTimeInSeconds(), 0.01);
7      assertTrue(discoveryIterationImpl0.eDeliver());
8      assertEquals(0.0, discoveryIterationImpl0.getSaveTimeInSeconds(), 0.01);
9      assertEquals(0L, discoveryIterationImpl0.getMaxUsedMemoryInBytes());
10     assertTrue(discoveryIterationImpl0.eDeliver());
11     ...
12}
```

Listing 2 Excerpt of an indirectly impacted test in Modisco.

```
1  @Test(timeout = 4000)
2   public void test16()  throws Throwable  {
3       DiscoveryIterationImpl discoveryIterationImpl0 = new DiscoveryIterationImpl()
            ;
4       EList<Event> eList0 = discoveryIterationImpl0.getMemoryMeasurements();
5       try {
6         discoveryIterationImpl0. eSet(30, (Object) eList0);
7         fail("Expecting␣exception:␣ClassCastException");
8
9       } catch(ClassCastException e) {
10        verifyException("org.eclipse.modisco.infra.discovery.benchmark.impl.
              DiscoveryIterationImpl", e);
11      }
12      assertEquals(0.0, discoveryIterationImpl0.getSaveTimeInSeconds(), 0.01);
13      ...
14  }
```

Listing 3 Excerpt of an impacted method in Modisco.

```
@Override
    public void eSet(int featureID, Object newValue) {
        switch (featureID) {
            ...
            case BenchmarkPackage.DISCOVERY_ITERATION__DISCOVERY_TIME_IN_SECONDS:
                setDiscoveryTimeInSeconds((Double)newValue);
                return;
            case BenchmarkPackage.DISCOVERY_ITERATION__SAVE_TIME_IN_SECONDS:
                setSaveTimeInSeconds((Double)newValue);
                return;
            case BenchmarkPackage.DISCOVERY_ITERATION__MAX_USED_MEMORY_IN_BYTES:
                setMaxUsedMemoryInBytes((Long)newValue);
                return;
    ...
    }
```

3 Approach

This section presents our proposed overall approach. It first gives an overview. Then, it describes how to detect the metamodel changes and how to trace their impacts until the tests and map them. Finally, it details our prototype implementation.

3.1 Overview

The overall objective of our approach is to help developers in checking the behavioral correctness of the code co-evolution when metamodels evolve, as the co-evolution may be done incorrectly or in an incomplete way (i.e., referred to as partial co-evolution in [24, 25]). Several ways exist, such as using formal methods, manual code review, or unit tests, etc. Our scope lies in tracing the impact of the metamodel changes till the tests and rely on them as an indicator for behavioral correctness of the code co-evolution, similarly as in a regression testing method [21–23]. Our vision is rather than letting the developers execute all test suite in both versions and manually analyzing them, we can reduce the set of tests to be analyzed to the only minimum necessary one. Thus, saving effort and time for developers.

Figure 3 depicts the overall approach workflow. We first compute the difference between the two metamodel versions each of them having a generated code and an additional code (step 1). In the original version, the additional code is the impacted one, and in the evolved version, the additional code is the co-evolved one. After that, we run the impact and the test tracing analysis to link the metamodel changes to the impacted and co-evolved code and their respective tests (step 2). Therefore, a developer can run the traced tests before and after the code co-evolution to check their behavioral correctness. Finally, to ease this task, we map the traced tests and execute them to report them back in a form of a diagnostic to the developers for an easier in-depth analysis of the effect of metamodel evolution rather than analyzing the whole test suite (step 3). Therefore, in a nutshell, there are no particular preconditions to our approach except having available code and tests from both before and after co-evolution along with the delta of the metamodel changes.

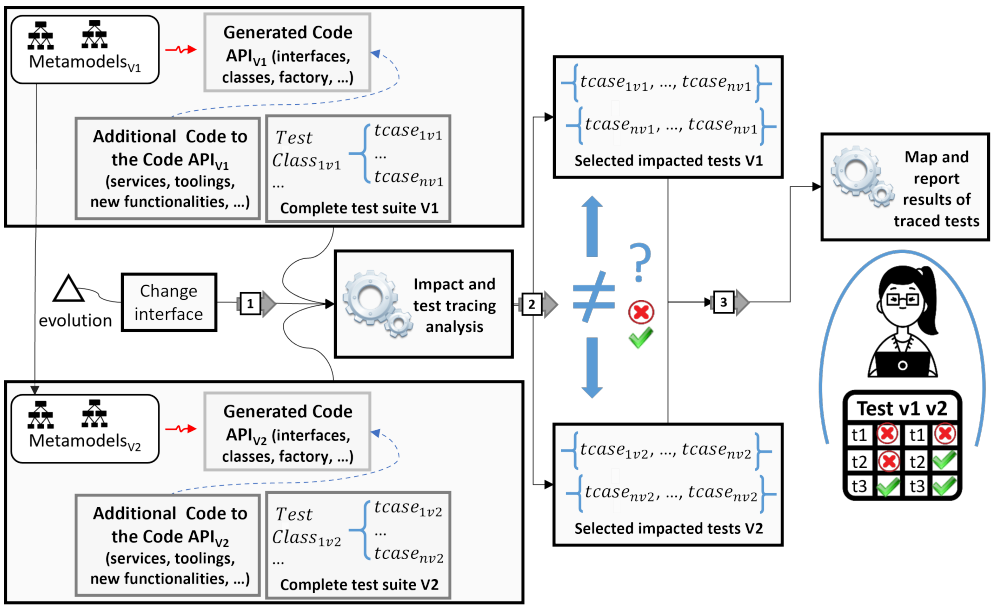

Fig. 3 Overall approach

3.2 Detection of metamodel Changes

Software artifacts continuously evolve over time [26]. As any artifact, metamodels evolve as well. Two types of changes are known and considered in the literature for metamodel evolution: *atomic* and *complex* changes [27]. Atomic changes are additions, removals, and updates of a metamodel element. Complex changes consist of a sequence of atomic changes combined together [28, 29]. For example, move property is a complex change where a property is moved from a source class to a target class. This is composed of two atomic changes: delete a property and add a property [27]. Several existing approaches allow to automatically detect metamodel changes between two versions, such as [28, 30–34].

In this work, we use an interface specification of changes 1 that is a connection layer to our test tracing approach with the existing change detection approaches. It basically defines each metamodel change as a class with its necessary information. Therefore, in practice, any detection approach [28, 30–34] can be integrated by bridging its changes to our interface and the rest of our approach can be performed independently.

In practice, we focus on the impacting metamodel changes that will require co-evolution of the code and not on the non-impacting changes. For example, a delete change or a change of type will impact the code and possibly its behavior that can be observed with its tests. However, addition changes, although non-breaking, can be traced back to their newly added tests. Thus, we also consider them to observe their behavior. The list of metamodel changes [32, 35] we consider for tracing their impact up to the tests is as shown in the first column of Table 1.

Table 1 List of metamodel changes and how they are traced up to the tests in the original and evolved versions.

Metamodel changes	Tests treatment	
	In original version (V1)	In evolved version (V2)
◇ Delete property p in class `C`	Search for usages of p in `C`	n/a
◇ Delete class `C`	Search for usages of `C`	n/a
◇ Add property p in class `C`	n/a	Search for usages of$p$ in `C`
◇ Add class `C`	n/a	Search for usages of `C`
◇ Rename element e to e' in class `C`	Search for usages of e in `C`	Search for usages of e' in `C`
◇ Change multiplicity of property p in class `C`	Search for usages of p in `C`	
◇ Change type of property p from `S` to `T`	Search for usages of p in `C`	
◇ Move property p_i from class `S` to `T` through *ref* ◇ Extract class of properties p_1, ..., p_n from `S` to `T` through *ref*	Search for usages of all p_i in `S`	Search for usages of all p_i in `T`
◇ Push property p_i from class `Sup` to $\texttt{Sub}_1$,...,$\texttt{Sub}_n$	Search for usages of all p_i in `Sup`	Search for usages of all p_i in all $\texttt{Sub}_i$
◇ Pull property p_i from classes $\texttt{Sub}_1$,...,$\texttt{Sub}_n$ to `Sup`	Search for usages of all p_i in all $\texttt{Sub}_i$	Search for usages of all p_i in `Sup`
◇ Inline class `S` to `T` with properties $p_1, ..., p_n$	Search for usages of all p_i in `S`	Search for usages of all p_i in `T`

For each version of the tests, we use different information provided by each metamodel change, depending on whether we are tracing the impacted tests in the original or the evolved versions. Columns 2 and 3 of table 1 detail the treatments of each metamodel change in the original and evolved versions. For example, for a rename element e to e', we search for e and e', respectively, in the original and evolved versions. Similarly, for the other changes, such as Move, Pull, Push, etc. where the source and target classes are different in the original and evolved versions. Only the impact of delete changes is searched in the original version, while the impact of addition changes is only searched in the evolved version.

3.3 Tracing the Impacted Tests

Our approach traces the impact of metamodel changes up to the test. To do that, we structure the code source to better navigate in it. Before starting, we parse the code source including tests and build the Code Call Graph (CCG) at the methods level. It consists of nodes $\mathbb{N}$ that are methods, and edges $\mathbb{E}$ that are calls between methods. For a given method, the CCG allows us to retrieve its callers, hence, tracing the call methods recursively up to the tests. After that, we apply Algorithm 1 on the built CCG. Overall, for each detected metamodel change, the algorithm computes the list of direct and indirect impacted tests that can be traced to the given metamodel change.

First, we analyze the AST of the code to identify the code usages of the evolved metamodel element. The information concerning the metamodel element before and after evolution, which is included in the metamodel change (see Table 1), allow us to spot the impacted code usages (Line 1). For example, for a rename property *id*, the algorithm will first find its usages, such as *getId()* or *setId()*[4]. Then, we filter these impacted code usages by keeping only the ones found inside a method declaration. Let us call the found method declaration using the impacted code *IM()* (Line 5). If *IM()* is a test method, that means that we found a direct impacted test (Line 7). Otherwise, thanks to the CCG, we retrieve *IM()*'s parents *parentsOfIM*, which are all the method declarations invoking IM() (Line 9-17). Afterwards and recursively, we check each parent of *IM()* if it is a test method or not (Line 14). The process is finished if we reach a method declaration that has no parents in the CCG that is either a test or not, or if the reached method declaration is already treated in the *parentsOfIM*. Therefore, by design, after browsing all the impacted code usages, Algorithm 1 traces the list of all impacted tests, without missing any if impacted. It is worth noting that tracing all impacted tests holds syntactically and w.r.t. static semantics. Possible side-effect will require further advanced dynamic analysis and is left for future work. Listing 1 presents an example of an impacted test in the Eclipse Modisco.infra.discovery.benchmark project. As described in Section 3.2, we detect that the attribute setTotalExecutionTimeInSeconds is renamed and moved from the class Discovery to the class DiscoveryIteration. After that, Algorithm 1 detects the code usage *getDiscoveryTimeInSeconds*. Then, it traces it to the method *test000*. As it has the *@Test* annotation, we conclude that *test000* is an impacted test due to the detected move change. Note that a test can be impacted by multiple metamodel changes, and one metamodel change can impact many tests. Algorithm 1 will detect either cases.

3.4 Mapping of impacted tests

After having traced the impacted tests, we further assist developers in analyzing the output of our approach. We provide a diagnostic in a form of a visualized report. This report displays the mapping of impacted tests between the original and evolved versions, along with the verdict of their execution (i.e., pass, fail, and error) and the corresponding impacted change. As an input to generate the diagnostic, the developer selects two impacted traced test classes in the original and evolved versions. The mapping between the two sets of test cases for these classes is performed using a state-of-the-art tool, namely GumTree [15]. It parses both test classes into a tree structure to enable the matching of the test cases. Herein, we distinguish three cases, namely: 1) tests that exist in both versions, 2) tests that exist in the original version but not in the evolved one, and 3) tests that exist in the evolved version but not in the original one. The impacted tests are then executed programmatically using JUnit runner. To facilitate the analysis and the tracing of impacted tests, we include the corresponding impacting metamodel changes in an additional column of the diagnostic report.

[4]Note that the knowledge about the generated code elements from the metamodel elements (e.g., getter/setter for EAttribute, class/interface for EClass, etc.) is so far hard-coded in the implementation of Algorithm 1. The mappings must be provided for our approach to be able to trace the tests.

Algorithm 1 Impacted tests detection

```
Require: codeCallGraph, change
impactedUsages ← match(AST, change)
impactedTests ← φ
for (impactedUsage ∈ impactedUsages ) do
    /* Find the method declaration using impactedUsage*/
    IM ← getIM(impactedUsage, codeCallGraph)
    if (isTest(IM)) then
        impactedTests.add(IM) /*If not already added*/
    else
        parentsOfIM ← getParents(IM, codeCallGraph)
        nextRound.add(parentsOfIM)
        while (nextRound.hasNewIMs()) do
            IM ← nextRound.get()
            if (isTest(IM) then
                impactedTests.add(IM)/*If not already added*/
            else
                parentsOfIM ← getParents(IM, codeCallGraph)
                nextRound.add(parentsOfIM)
            end if
        end while
    end if
end for
```

Figure 4 illustrates a screenshot of the test tracer report on a toy example "Employee Management Project". After selecting the class of tests that have been traced before code co-evolution, and the class of tests that have been traced after code co-evolution, the user clicks on "Map tests" to display the table of mapped tests with their verdict of execution. To illustrate the verdict of the test execution, we made: the passing test in green, the failing test in blue, and erroneous tests in red. For example, the change "Delete Class Contact" impacts two tests, test11 which passes, and test06 which fails. The verdict of the execution of the tests has no relation with the change itself but with the test that uses the code elements impacted by the metamodel change. Those tests do not exist anymore in the evolved version since the class Contact is absent and cannot be tested anymore.

3.5 Tool implementation

We implemented our solution as an Eclipse Java plugin handling Ecore/EMF metamodels and their Java code. We rely on our approach [29] to perform the detection of the metamodel changes bridged with the change interface. The test tracing, technically, consists of parsing the java code and manipulating its AST using JDT eclipse plugin[5] to construct the Code Call Graph (CCG). After that, we navigate within the methods calls until we either reach a test or not. Finally, for each impacted *TestClass*, we create

[5]Eclipse Java development tools (JDT): https://www.eclipse.org/jdt/core/

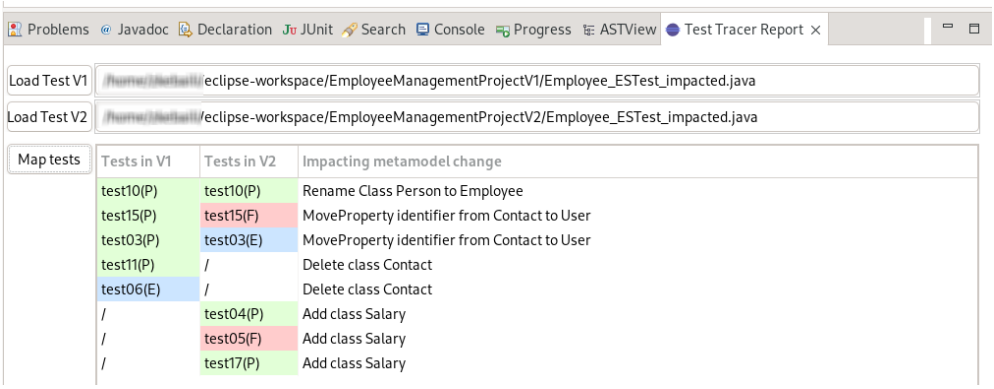

Fig. 4 A snippet of the diagnostic report view to visualize and analyze the traced impacted tests.

a copy *TestClass_Impacted* where we only include the impacted traced test cases. The developer can then launch the traced tests in both the original and evolved versions to investigate the behavioral correctness of the co-evolved code. We further implemented the diagnostic report as a view that shows the mapped tests with GumTree [15], their verdict retrieved programmatically using JUnit Runner[6] and impacting metamodel changes, as shown in Figure 4. The goal is to ease the developers' in-depth analysis of the effect of metamodel evolutions rather than rerunning and analyzing the whole test suite.

4 Evaluation

This section evaluates our automatic approach of checking the behavioral correctness of the metamodel and code co-evolution. First, we present the data set and the evaluation process. Then, we set the research questions we address and discuss the obtained results.

4.1 Data Set

This section presents the used data set in our evaluation to be found in the attached supplementary material[7].

We evaluate our approach on four case studies of language implementations in Eclipse, namely OCL [36], Modisco [37], Papyrus [38], and EMF [39] project. OCL is a standard language defined by the Object Management Group (OMG) to specify First-order logic constraints. Modisco is an academic initiative to support development of model-driven tools, reverse engineering, verification, and transformation of existing software systems. Papyrus is an industrial project led by CEA[8] to support model-based simulation, formal testing, safety analysis, etc. The EMF project is a modeling framework and code generation facility for building tools and other applications based

[6] https://junit.org/junit4/javadoc/4.13/org/junit/runner/Runner.html
[7] https://figshare.com/s/b6251b9e47fa82983ce5
[8] http://www-list.cea.fr/en/

Table 2 Details of the metamodels and their evolutions.

Evolved metamodels	Versions	Atomic changes in the metamodel	Complex changes in the metamodel
Pivot.ecore in project *ocl.examples.pivot*	3.2.2 to 3.4.4	*Deletes:* 2 classes, 16 properties, 6 super types *Renames:* 1 class, 5 properties *Property changes:* 4 types; 2 multiplicities *Adds:* 25 classes, 121 properties, 36 super types	1 pull property 2 push properties
Pivot.ecore in project *ocl.pivot*	6.1.0 to 6.7.0	*Deletes:* 0 classes, 4 properties, 4 super types *Renames:* 0 class, 1 properties *Property changes:* 49 types; *Adds:* 5 classes, 47 properties, 7 super types	n/a
ExtendedTypes.ecore in project *papyrus.infra.extendedtypes*	0.9.0 to 1.1.0	Deletes: 10 properties, 2 super types Renames: 3 classes, 2 properties Adds: 8 classes, 9 properties, 8 super types	2 pull property 1 push property 1 extract super class
Benchmark.ecore in project *modisco.infra.discovery.benchmark*	0.9.0 to 0.13.0	Deletes: 6 classes, 19 properties, 5 super types Renames: 5 properties Adds: 7 classes, 24 properties, 4 super types	4 moves property 6 pull property 1 extract class 1 extract super class
Ecore.ecore in project *org.eclipse.emf*	2.37.0 to 2.37.0'	Deletes: 1 class, 2 properties Renames: 2 properties	1 move property 1 pull property

on a structured data model [6]. Thus, the four case studies cover standard, academic, and industrial languages that have evolved several times for more than 10 years of continuous development period.

Moreover, we aimed at selecting meaningful evolutions that do not consist in only deleting metamodel elements, but rather include complex evolution changes. We also aimed to select long and short evolution intervals in the selected releases versions to stress test our approach in different scenarios. This is the case for the OCL, Modisco, and Papyrus case studies. However, they do not have manually written tests. Thus, we added the EMF case study that have manually written tests, but its metamodel is stable with no evolutions. Therefore, we had to simulate a set of metamodel evolution changes similar to those real-world changes observed in our three first case studies and we co-evolved their impacts with our previous work [13].

Table 2 gives details about the selected case studies, in particular about their metamodels and the changes applied during evolution. The total of applied metamodel changes was 452 atomic changes and 21 complex changes in the five metamodels. Table 3 gives details on the size of the projects in terms of code and tests of the original and evolved versions. We collected a total of 18 projects to evaluate our approach on.

4.2 Evaluation Process

We evaluate our approach by: 1) investigating its usefulness compared to the manual tracing of the impacted tests with user study, 2) measuring its ability to automatically trace the impacted tests due to impacting metamodel changes both in the original and evolved version of the project, 3) assessing its ability to give an indication about the correctness of the code and metamodel co-evolution, and finally 4) measuring the gains of its usage in terms of reduction of tests and their execution time. Note that as

the metamodel changes are taken as input of our automatic test tracing, we studied the original and evolved versions to confirm the metamodel changes. Therefore, we do not take an incorrect input of metamodel changes that would mislead our traced tests, which would mislead the behavioral checking of the code co-evolution.

Regarding the tests, only the EMF case study had manually written tests. Thus, we had to generate tests for OCL, Modisco, and Papyrus case studies. We used a state-of-the-art tool, namely EvoSuite [16]. EvoSuite is a search-based tool for Unit Tests generation. It uses a heuristic algorithm, particularly, genetic algorithm in the Test Suit generation. In their used approach Fraser et al. [16] aimed to maximize the coverage metric and mutation score which guarantees good-quality tests. EvoSuite is largely used and it was evaluated not only in literature but also in the industrial context. Firhard et al. [40] compared it with DSpot, a state-of-the-art tool for test amplification, their results show that EvoSuite achieves a statistically better mutation score. Herculano et al. found that EvoSuite's generated tests can successfully help to identify faults during maintenance tasks [41]. In industry, Rozière et al. use automated tests generated with EvoSuite to filter invalid code translations in the context of their work done for Meta [42]. Gruber et al. [43] further showed the quality and robustness of the generated tests. It showed that while flakiness is at least as common in generated tests as in developer-written tests, EvoSuite is effective in alleviating this issue giving 71.7% fewer flaky tests. Thus, EvoSuite is appropriate in our work to generate robust tests in the original code and in the co-evolved code to compare their results, i.e., check behavioral correctness. We simply let EvoSuite run to generate Junit test classes for the selected projects with the following parameters: *-DmemoryInMB=2000 -Dcores=4 -DtimeInMinutesPerClass=10 evosuite:generate evosuite:export*. It uses up to 2GO of RAM, 4 CPU cores, and 10 minutes per test class. Generating tests is a best practice, in particular w.r.t. its efficiency in test generation and at a large scale for all public methods [44, 45]. EvoSuite generates tests for all public methods, whereas developers tend to manually write a few tests for only some targeted methods. Thus, relying only on manually written tests increases the risk of not assessing the behavioral correctness of many cases of code co-evolutions that are not covered by test cases. Generating tests alleviates this risk. Indeed, this is observed when computing the coverage metric for each of our considered projects. Table 4 shows that the highest coverage (69% to 95%) is obtained on projects with automatically generated tests and the lowest coverage (17% to 33%) were on the two projects with manually written tests.

4.3 Research Questions

This section sets the research questions (RQs) to assess our work. The research questions are as follows:

RQ0. *To what extent can developers manually trace the tests impacted by the evolution of the metamodel?* This aims to asses if developers can manually trace the tests that are impacted by the changes of the metamodel. This also aims further to assess our approach's usefulness through main observations.

RQ1. *To what extent does our automatic approach trace the impact of the metamodel evolution to the tests?* This aims to assess on real-world case studies the ability

Table 3 Details of the projects and their tests.

Projects co-evolved in response to the evolved metamodels	N^o of packages	N^o of classes	N^o of test packages	N^o of test classes	N^o of LOC	N^o of tests
$[P1_{V1}]$ ocl.examples.pivot	22	439	22	290	74002	7322
$[P1_{V2}]$ ocl.examples.pivot	22	480	22	220	89449	4990
$[P2_{V1}]$ ocl.examples.base	12	181	12	119	17617	2320
$[P2_{V2}]$ ocl.examples.base	12	181	12	118	17596	2133
$[P3_{V1}]$ ocl.pivot	60	1006	55	598	142236	8795
$[P3_{V2}]$ ocl.pivot	63	1090	58	683	153613	6396
$[P4_{V1}]$ papyrus.infra.extendedtypes	7	37	7	19	2057	135
$[P4_{V2}]$ papyrus.infra.extendedtypes	7	51	7	26	2570	248
$[P5_{V1}]$ papyrus.infra.extendedtypes.emf	5	25	4	14	1145	104
$[P5_{V2}]$ papyrus.infra.extendedtypes.emf	5	25	4	14	1145	104
$[P6_{V1}]$ papyrus.uml.tools.extendedtypes	5	15	3	9	726	75
$[P6_{V2}]$ papyrus.uml.tools.extendedtypes	5	15	3	9	725	75
$[P7_{V1}]$ org.eclipse.modisco.infra.discovery.benchmark	3	28	3	15	2333	524
$[P7_{V2}]$ org.eclipse.modisco.infra.discovery.benchmark	3	30	3	15	2588	619
$[P_{ecore\_V1}]$ org.eclipse.emf.ecore	13	168	/	/	142586	0
$[P_{ecore\_V2}]$ org.eclipse.emf.ecore	13	166	/	/	141434	0
$[P8_{V1}]$ org.eclipse.emf.test.core	/	/	19	141	40858	322
$[P8_{V2}]$ org.eclipse.emf.test.core	/	/	19	141	40544	322
$[P9_{V1}]$ org.eclipse.emf.test.xml	/	/	6	27	12088	64
$[P9_{V2}]$ org.eclipse.emf.test.xml	/	/	6	27	12088	64

Table 4 Coverage metric of each evaluation project.

Projects	[P1]	[P2]	[P3]	[P4]	[P5]	[P6]	[P7]	[P8]	[P9]
Coverage V1	66.9%	86.1%	80.1%	95.6%	95.1%	89.5%	91.3%	18%	33.4%
Coverage V2	66.2%	85.4%	74.1%	95.2%	95.2%	89.2%	87.5%	17.2%	33%

and applicability of our automatic approach to trace the metamodel changes with code elements till their tests.

RQ2. *What is the observed behavioral correctness level of the code co-evolution?* This aims to assess through running the selected tests the effect of the code co-evolution, whether it keeps the tests' results stable, or instead degrade or improve them.

RQ3. *What are the observed gains (w.r.t. test case reduction and execution time) obtained from our approach of tracing impacted tests by metamodel evolution ?* This aims to highlight the benefit of our approach compared to when not using it and relying on the whole test suite as a baseline.

4.4 Results

We now discuss the results w.r.t. our research questions.

4.4.1 RQ0

The first goal of our evaluation is to gain evidence on the difficulty or not of the manual task of tracing impacted tests. Thus, we designed and ran a user study experiment.

RQ0 Set Up

We first describe our user study experiment.

Subjects selection. The experiment was conducted with 8 participants (2 females and 6 males), including PhD students and research engineers in IRISA Laboratory, at the University of Rennes. The participants have a varying level of experience in programming (from 3 to 12 years with an average of 7 years), and a varying level in model-driven engineering (MDE) (from 0 to 7 years with an average of 2 years and 6 months).

Experiment Task. The experiment aims to evaluate the ability of participants to trace manually and analyze the affected unit tests before and after the metamodel evolution. We prepared two Eclipse workspaces. The first one contains the original version of the project `org.eclipse.emf.test.core`, and the second workspace contains the evolved version of the same project. The number of tests is 322 in both versions. The number of tests that must be traced is respectively 173 and 17 in the original and evolved versions. We then give in the guideline of the experiment the list of the changes that details the evolution of the metamodel (see last row of Table 2) with a description of the metamodel and project. We also explain what type of code elements are generated from each metamodel element. Each participant had then to identify impacted unit tests in the original and evolved version of the project `org.eclipse.emf.test.core`. This procedure not only highlights the direct impacts of the metamodel changes but also requires the consideration of indirect impacts. Additionally, the study explores the usefulness and potential adoption of our approach as an automatic support tool for tracing the impacted tests. After the end of this task, we presented our automatic tracing approach to the participant then we ran a post-questionnaire.

Variables. Our user study aimed to measure to what extent can developers trace impacted tests. The independent variable we controlled was the impacting metamodel changes. We covered seven different types of changes with both atomic and complex changes. We then observed the dependent variable of the traced tests by the participants.

RQ0 Results

When we analyzed the answers of each participant, we found that they were able to trace only a few tests. In the original version of the project, the total number the manually traced tests varied between 1 and 18, with an average of 6 tests out of the 173 impacted tests. In the evolved version of the project, the total number the manually traced tests varied between 2 and 32, with an average of 11 tests out of

the 17 impacted tests. While we first observe that none of the participants traced all tests, they also did not correctly trace the tests.

Indeed, the number of correctly manually traced tests varies between 1 to 18 in the original version of the project, with an average of 5 traced tests. In the evolved version, the number of correctly manually traced tests varies between 0 to 10 tests with an average of 4 tests. We had five participants out of eight who wrongly traced eight tests in the original version of the projects, varying between one or two tests for each of them. In the evolved version of the project, all participants have wrongly traced between one to 26 tests that are not impacted by the evolution of the metamodel. We investigated the cause of this incorrect tracing. We found that one reason was the tracing of tests that contain a commented impacted code. Another reason was traced the wrong overload method `getEEnumLiteral(EInt)` of the actually evolved method `getEEnumLiteral(EString)`. Another reason was to simply include sibling tests in the class of a trace test.

In addition, we found that five participants considered both the direct and indirect impact of the metamodel evolutions on the tests, while the 3 remaining participants considered only the directly impacted tests. The results of the user study show the difficulty of manually tracing the tests with the evolved metamodel. Not only the participants could not trace all tests, but they even wrongly traced non-necessary tests.

Regarding the answers of the participants about usefulness[9] of our approach as an automatic support tool for tracing the impacted tests. One participant graded our approach as 'Somewhat useful', five out of eight graded it as 'Very useful', and two graded it as 'Extremely useful'. The last question was about their potential adoption[10]of our approach as an automatic support tool for tracing the impacted tests. Two participants out of eight answered 'Somewhat likely', three participants answered 'Likely', and the three remaining participants answered 'Very likely'. The finding of this experiment emphasizes the adoption likelihood and usefulness of our approach (discussed in RQ4).

> ***RQ*$_0$ insights:** From our user study experiment, we observe that tracing manually the tests impacted by the evolution of the metamodel is a hard and error-prone task. The post-questionnaire results after a demonstration of our automatic approach suggest its high usefulness and adoption likelihood.

4.4.2 RQ1

Following the evaluation protocol, we executed our approach on our case studies in Table 3. Figure 5 depicts the number of traced tests due to the impacting metamodel changes in Table 2. We first observe that we can trace tests successfully. We traced a total of 1608 out of 34612 tests due to 473 metamodel changes, distributed in 1106 and 502 tests in the original and evolved versions. Thus, we can isolate for the developers the tests that must be executed and looked at to check the behavioral correctness of the co-evolution.

[9]Between 'Useless – Little useful – Somewhat useful – Very useful – Extremely useful'.
[10]Between 'Very unlikely - Unlikely - Somewhat likely - Likely - Very likely'.

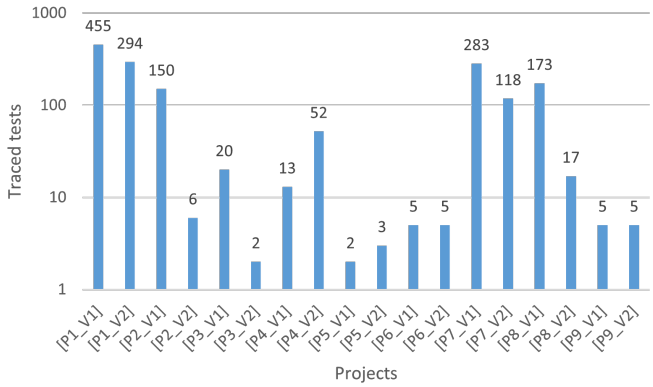

Fig. 5 Traced tests due to metamodel evolution in each project.

We also observe that the more the number of evolution changes between the original and evolved versions of the metamodel increases, the more the number of tests we trace increases as well. In particular, when a lot of tests are available for analysis. This is true for the OCL Pivot project [$P1$] and Modisco project [$P1$]. We also observe no overall difference between projects with automatically generated tests and projects with manually tests. We traced similar ratios of tests.

Moreover, as several deletions of classes and properties occurred in the evolution changes, several tests are not generated in the evolved version, which explains why we trace more tests in the original version than in the evolved version. This is observed in most of the projects except the [$P4$], where [$P4_{V2}$] had more tests. We double-checked this case, and we found that there were strangely more tests generated by EvoSuite in [$P4_{V2}$] than in [$P4_{V1}$] for the same classes, likely due to more dependencies available in V2.

Finally, regarding the overhead, the time performance varied from 5 minutes in projects Papyrus Extendedtypes with 42 metamodel changes and 726 LOC up to 60 minutes in projects OCL Pivot with 117 metamodel changes and 142236 LOC. This is of course to be compared to manual tracing of the tests in both original and evolved versions before and after co-evolution, which can be tedious and time-consuming. In particular, when thousands of tests exist as in the project of OCL Pivot. However, our prototype traces the impact of the metamodel changes sequentially as a proof-of-concept for feasibility and applicability. Time performance can further be improved by parallelizing the tracing for the metamodel changes. This is left as future work.

***RQ*$_1$ insights:** We could successfully trace the tests that must be executed before and after the co-evolution regardless of whether they are manually or automatically written. This would help developers to immediately check the code co-evolution by executing the subset of relevant traced tests among the whole test suite.

Table 5 Selected tests before and after code co-evolution.

Projects	$[P1_{V1}]$ to $[P1_{V2}]$	$[P2_{V1}]$ to $[P2_{V2}]$	$[P3_{V1}]$ to $[P3_{V2}]$	$[P4_{V1}]$ to $[P4_{V2}]$	$[P5_{V1}]$ to $[P5_{V2}]$	$[P6_{V1}]$ to $[P6_{V2}]$	$[P7_{V1}]$ to $[P7_{V2}]$	$[P8_{V1}]$ to $[P8_{V2}]$	$[P9_{V1}]$ to $[P9_{V2}]$
N^o class tests	114 - 57	51 - 4	8 - 2	5 - 10	2 - 3	2 - 3	11 - 12	32 - 8	1 - 1
N^o pass	106 - 97	124 - 0	14 - 1	10 - 22	2 - 3	2 - 2	206 - 68	146 - 10	5 - 5
N^o fail	2 - 5	1 - 1	0 - 0	1 - 3	0 - 0	0 - 0	1 - 0	0 - 1	0 - 0
N^o error	347 - 192	25 - 5	6 - 1	2 - 27	0 - 0	3 - 3	76 - 50	27 - 6	0 - 0

4.4.3 RQ2

After tracing the tests, we could execute them to observe their effect before and after co-evolution of the code. Table 5 depicts the results for the original and evolved projects' versions. The second line gives the number of traced class tests and the rest of the lines categorizes the tests. We overall observe no significant difference between projects with automatically generated tests and projects with manually tests.

The most interesting project is the first $[P1_{V1}]$ to $[P1_{V2}]$. Even though the tests decreased by 161 (455 - 294 from Figure 5), the number of passing tests decreased only by 9 (106 - 97 from Table 5). Regarding the error tests, they decreased by 155. However, the failing tests increased by 3 from 2 to 5, as shown in Table 5. There was also the appearance of one failing test in $[P8]$. These cases of increased failing tests indicate that the code co-evolution may be not completely behaviorally correct.

Moreover, in the other projects $[P2][P3][P7][P8]$, many tests that existed in the original version were not in the evolved version due to the delete changes of the metamodels. This is actually a sign of behavioral correct co-evolution, as indeed the tests should be removed following the removal of the generated code for those deleted metamodel elements. In the rest of the projects $[P4][P5][P6][P9]$, roughly the same number of tests in the original version behaved the same in the evolved version, suggesting again a behaviorally correct co-evolution. These results should help developers to further check the code co-evolution rather than simply accepting them in particular when it is fully automated.

***RQ*$_2$ insights:** Our traced tests could hint in two projects that the co-evolution may not be entirely correct due to more failing tests and fewer passing tests. The rest of the project would hint on rather a correct co-evolution due to delete metamodel changes. Overall, automating the help for checking of the behavioral correctness of the code co-evolution for developers, regardless of whether they tests are manually or automatically written.

4.4.4 RQ3

With the traced tests due to the metamodel evolution, we can assess the gains in terms of test reduction and execution time compared to the whole test suite as a baseline. Indeed, rather than re-running the whole test suite both in the original and evolved versions, or worse, not considering the tests at all. We provide developers with a zoomed view of traced impacted tests by the metamodel evolution, hence, focusing on assessing the code co-evolution.

The first line of Table 5 already gives the number of traced impacted test classes that are always less than the original number of test classes.

Table 6 further illustrates the differences between the original test suite and the traced impacted tests in terms of number of test cases and execution time. Columns 2 and 3 give the number of original tests and of traced tests. Column 4 depicts the gains in terms of test reduction percentage. On average, we observe a reduction gain of 88% of test cases, varying from 46% to 99.9%. Out of 34612 tests in the 18 projects, we traced 1608 impacted tests representing an absolute 95% reduction.

This naturally leads to a gain in terms of execution time reduction of the tests. Columns 5 and 6 give the execution times for the whole test cases and the traced ones. Herein, we measured the execution time through IDE runner for the Junit tests. Column 7 depicts the gains in execution time of the traced tests compared to the whole test suite. On average, we observe a reduction of 84%, varying from 69% to 99%. Overall, we observe no significant difference in benefit of reducing tests and the gain in execution time between projects with automatically generated tests and projects with manually tests. Respectively, we observe a reduction gain in tests of 88% versus 81% and a reduction gain in execution of 82% versus 95.5%.

> ***RQ*$_3$ insights:** Tracing the metamodel evolution changes up to the impacted tests allows assessing the co-evolution behavioral correctness, while gaining, on average, a reduction of 88% in the number of tests and 84% in execution time. The reduction gains are similar with no significant difference regardless of whether the tests are manually or automatically written.

5 Threats to Validity

This section discusses threats to validity [46].

5.1 Internal Validity.

To be able to trace the impact of metamodel changes to the tests, we had to have a test suite in the selected projects. However, we observed that the Eclipse projects relying on metamodels do not come with the test suite. This was not only the case of OCL[36], papyrus [38] or Modisco [37], but also other Eclipse languages [47, 48]. Therefore, we were obliged to generate the test suite with a state-of-the-art available tool EvoSuite [16]. Even though, automatic test generation is a best practice, there is the risk of having tests that are different from manually written tests. However, this does not pose a risk to our approach as the main algorithm of our approach is generic

Table 6 Reduction gains of the number of traced tests and their execution time.

Projects co-evolved in response to the evolved metamodels	N^o of tests	N^o of traced tests	Reduction gain in tests	Execution time of tests (s)	Execution time of traced tests (s)	Reduction gain in execution time
$[P1_{V1}]$ ocl.examples.pivot	7322	455	↘ 94%	339.411	48.322	↘ 86%
$[P1_{V2}]$ ocl.examples.pivot	4990	294	↘ 94%	228.88	70.856	↘ 69%
$[P2_{V1}]$ ocl.examples.base	2320	150	↘ 93.5%	123.254	26.518	↘ 79%
$[P2_{V2}]$ ocl.examples.base	2133	6	↘ 99.7%	99.105	5.294	↘ 95%
$[P3_{V1}]$ ocl.pivot	8795	20	↘ 99.7%	859.69	0.497	↘ 99%
$[P3_{V2}]$ ocl.pivot	6396	2	↘ 99.9%	261.792	12.133	↘ 95%
$[P4_{V1}]$ papyrus.infra.extendedtypes	135	13	↘ 90%	16.94	2.924	↘ 83%
$[P4_{V2}]$ papyrus.infra.extendedtypes	248	52	↘ 79%	17.076	2.957	↘ 83%
$[P5_{V1}]$ papyrus.infra.extendedtypes.emf	104	2	↘ 98%	6.912	2.11	↘ 70%
$[P5_{V2}]$ papyrus.infra.extendedtypes.emf	104	3	↘ 97%	7.31	1.802	↘ 75%
$[P6_{V1}]$ papyrus.uml.tools.extendedtypes	75	5	↘ 93%	5.9	1.505	↘ 75%
$[P6_{V2}]$ papyrus.uml.tools.extendedtypes	75	5	↘ 93%	6.246	1.099	↘ 82%
$[P7_{V1}]$ org.eclipse.modisco.infra.discovery.benchmark	524	283	↘ 46%	7.332	2.04	↘ 73%
$[P7_{V2}]$ org.eclipse.modisco.infra.discovery.benchmark	619	118	↘ 81%	14.534	4.107	↘ 72%
$[P8_{V1}]$ org.eclipse.emf.test.core	322	173	↘ 46%	176.372	2.908	↘ 98%
$[P8_{V2}]$ org.eclipse.emf.test.core	322	17	↘ 94%	157.041	0.346	↘ 99%
$[P9_{V1}]$ org.eclipse.emf.test.xml	64	5	↘ 92%	3.764	0.247	↘ 93%
$[P9_{V2}]$ org.eclipse.emf.test.xml	64	5	↘ 92%	3.193	0.257	↘ 92%

and would be able to trace the impact of a metamodel change in the same way for both automatically generated tests as well as manually written tests. This is what we observed with the fourth EMF case study having manually written tests. Indeed, the overall results of tracing tests and reduction gains were not significantly different in all case studies regardless of whether the tests are automatically generated or manually written. The main risk is related to the behavioral correctness of the code co-evolution. To check it, we require unit tests that target single methods. However, automatic test generation can even be more advantageous herein. Indeed, it generates tests for all public methods, whereas developers tend to manually write only few tests for some targeted methods. Thus, if relying only on manually written tests, there is a high risk of not assessing the behavioral correctness of many cases of code co-evolutions that are not covered by test cases. This is mitigated by relying on EvoSuite that generates a full test suite of unit tests with Junit assertions. This tool has shown his efficiency in test generation as a state-of-the-art tool [44, 45]. Indeed, this is observed when computing the coverage metric for each of our considered projects (see Table 4). The highest coverage is on projects with automatically generated tests and the lowest coverage is on the two projects with manually written tests.

Therefore, our approach not only checks the correctness of the code co-evolution with the traced tests, but also favors the best practice of tests generation in each release of a software language after its metamodel evolution.

Finally, as our tracing approach relies on the quality of detected metamodel changes, we analyzed, in our evaluation, each detected change and checked whether it occurred between the original and evolved metamodels. This alleviates the risk of relying on an incorrect metamodel change that would degrade the tracing of the impacted tests by metamodel changes, i.e., not tracing an impacted test by a non-considered metamodel change. In addition, as we did not have the ground truth, we could not report on the precision and recall of our approach. However, our approach uses the Code Call Graph and starts from the generated code elements corresponding to the evolved metamodel elements to recursively trace back any existing tests. Thus, by design we actually detect all the impacted tests that must be traced. To test that our algorithm does not miss any impacted test, and does not trace non-impacted tests, we manually verified the ground truth for our smallest data set projects [$P6$] and [$P9$], because they have fewer tests and are less complex. We checked for every metamodel change all the impacted tests and we found that our approach traces all of them. Further evaluation on a ground truth is left for future work.

5.2 External Validity.

We implemented and evaluated our approach for EMF/Ecore metamodels and Java code with Junit tests. Other languages, such as C# or C++, use a different syntax, but conceptually use the same constructions as in Java. Although we think that the tracing would be applicable for other languages, we cannot generalize our results. Further experimentation on other languages is necessary. However, the only requirement to apply our approach to other languages is to have access to the ASTs of the parsed code and tests, and to adapt our tracing of the tests in the build call graph. Moreover, our evaluation was performed on Eclipse projects from five languages. Thus, we cannot

generalize our findings to all software languages or DSLs. We also cannot generalize our results on manually written tests, in particular the test verdicts of the traced tests, i.e., pass, fail, and error. Further experimentation remains necessary and is left for future work.

5.3 Conclusion Validity.

Our evaluation gave promising results, showing that we could trace the impact of metamodel changes till the tests, and hence, check the behavioral correctness of the code co-evolution in practice for real-world projects. However, even though we evaluated our approach on 18 projects of metamodel evolution and code co-evolution with automatically generated tests and manually written tests, further evaluation is needed on more case studies to have more insights and statistical evidence. Finally, our user study experiment suggesting our approach usefulness needs to be replicated with more participants.

6 Related work

This section discusses the main related work w.r.t. testing the metamodel and code co-evolution.

Extensive literature exists on co-evolution of metamodel and models [49–58], constraints [59–64] and transformations [65–69]. Several other approaches propose to automate the code co-evolution. Henkel et al. [70] proposed an approach that captures refactoring actions and replays them on the code to migrate. They support only the changes renames, moves, and type changes. Nguyen et al. [71] also proposed an approach that guides developers in adapting code by learning adaptation patterns from previously migrated code. Similarly, Dagenais et al. [72] also used a recommendation mechanism of code changes by mining them from previously migrated code. Anderson et al. [73] proposed to migrate drivers in response to evolutions in Linux internal libraries. It identifies common changes made in a set of files to extract a generic patch that can be reused on other code parts. However, all those approaches are not tailored to metamodel and code co-evolution. More importantly, they do not test the behavioral correctness of their co-evolution.

Moreover, a more related approach to metamodel co-evolution, Riedl et al. [7] proposed an approach to detect inconsistencies between UML models and code. Pham et al. [8] proposed an approach to synchronize architectural models and code with bidirectional mappings. Jongeling et al. [9] propose an early approach for the consistency checking between system models and their implementations by focusing on recovering the traceability links between the models and the code. Jongeling et al. [10] later relied on the recovered traces to perform the consistency checking task. Zaheri et al. [11] also proposed to support the checking of the consistency-breaking updates between models and generated artifacts, including the code. Yu et al. [12] proposed to co-evolve the metamodels and the generated API in both directions. Khelladi et al. [13] proposed an approach that propagates the metamodel changes to the code as a co-evolution mechanism. However, all those approaches [7–13] focus on co-evolving the code without checking the behavioral correctness of the co-evolved code.

Our work is also related to regression test selection [74], where different approaches exist based on Genetic algorithms [75], slicing [76], or database safety [77]. However, our goals are different. Our approach aims to trace the impact of metamodel evolution up to the tests to check the behavioral correctness of the code co-evolution rather than fault localization. Furthermore, from the survey of Yoo et al. [78], our approach can be categorized along the incremental test selection approaches, such as Infinitest[11], EKSTAZI [79], and Moose [80]. However, our approach is different from several aspects. Infinitest is a test case selection plugin. It selects only the direct tests of changed methods and not indirectly impacted tests as in our work (cf. Section 3.3). EKSTAZI [79] is a regression test selection tool by dynamically analyzing the code and the tests. EKSTAZI must compile the code including the tests to be able to track the changes of a .class files later and then select the subset of impacted tests. Our work only needs to parse the code and the tests. Moose [80] represents source code entities in a model. This model gathers entities such as packages, classes, methods, and the relations between them. This is the opposite flow of our work because our starting point is a Metamodel. All of Infinitest, EKSTAZI and Moose aim to analyze code changes incrementally to select impacted tests in the evolved version of the code only. As a consequence, the developers can only have the latest states of their selected tests after code changes. They cannot compare them with before and after the applied changes. To do so, developers would have to manually undo the code changes and to manually select and re-run the same tests to be able to compare them before and after manually, which represents a significant burden for the developers. Our approach automates the tests' tracing before and after code co-evolution and gives the output as a visual report to developers for an easier analysis of the impact of each metamodel change and its code co-evolution. If there is an issue, this report would allow the developer to know which metamodel changes are causing this issue, with the tests impacted by these metamodel changes.

Finally, Ge et al. [81] propose to verify the correctness of refactoring with a set of condition checkers that are executed only after the refactoring application. This is rather similar to the intention of our work. However, we rely on a testing technique that is applied to check before and after code co-evolution with the metamodel evolution.

To the best of our knowledge, our work is the first attempt to propose a fully automatic approach for checking the behavioral correctness of the code co-evolution. We leverage the tests in the original and evolved versions and trace the impacted tests before and after co-evolution. Thus, allowing developers to have more confidence in the automatic co-evolution or at least to locate the tests that must be investigated after co-evolution in case of a bug introduction. This avoids to re-run all test suite, which is expensive and time-consuming before manually mapping the tests in both versions, which is tedious and error-prone.

7 Conclusion

This paper proposes an automated tracing of the impacted tests due to metamodel evolution. Thus, by tracing the tests before and after code co-evolution, we check

[11] https://infinitest.github.io/doc/eclipse

its behavioral correctness. Our approach takes as input the metamodel changes and then finds the different pattern usages of each metamodel element in the code. After that, we recursively search for its usages in the code call graph until reaching the tests. Thus, we end up matching metamodel changes with impacted code methods and their corresponding tests. We further implemented our approach in an Eclipse plugin that allows to trace the tests, map them with state-of-the-art solution GumTree and execute them. We then report them back as a diagnostic to the developers for an easier in-depth analysis of the effect of metamodel evolutions rather than re-running and analyzing the whole test suite.

The user study experiment we conducted showed that tracing manually the tests impacted by the evolution of the metamodel is a hard and error-prone task. Not only the participants could not trace all tests, but they even wrongly traced non-impacted tests. We then evaluated our approach on 18 Eclipse projects from OCL, Modisco, Papyrus, and EMF over several evolved versions of metamodels. Four projects had manually written tests and we generate tests for the other 14 projects. The results show that we successfully traced the impacted tests automatically by selecting 1608 out of 34612 tests due to 473 metamodel changes.

When running the traced tests before and after co-evolution, we observed two cases indicating possibly both behaviorally incorrect and correct code co-evolution. Thus, helping the developers to locate the code co-evolution to investigate in more detail. Furthermore, our approach provided gains that represent, on average, a reduction of 88% in number of tests and 84% in execution time. No significant difference was observed between projects with manually written tests and automatically generated ones.

As future work, we first plan to improve the performance of our implementation with optimization of the tests' tracing. We also plan to extend our approach to projects that use an equivalent form of metamodels in other technological space than Eclipse, such as JHipster and OpenAPI that both generate code from a model specification similar to a metamodel. Thus, we can have alternative case studies.

After that, we plan to investigate the techniques of test amplification on the selected tests we traced from the metamodel changes. Indeed, once we select a subset of tests, we could amplify them by generating more similar tests, yet, with different assertions to cover more corner cases. This would amplify the behavioral check of the code co-evolution.

Finally, we will also explore another type of amplification, which is the interchange of the tests between the original and evolved versions. In other words, we aim to co-evolve the tests of the original and evolved versions, respectively forward and backward to the evolved and original versions, while removing duplicates.

Acknowledgment.

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] Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) Integrated Formal Methods, pp. 286–298. Springer, Berlin, Heidelberg (2002)

[2] Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of mde in industry. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 471–480 (2011). ACM

[3] Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 633–642 (2011). ACM

[4] Tolvanen, J.-P., Kelly, S.: Metaedit+: defining and using integrated domain-specific modeling languages. In: The 24th ACM SIGPLAN Conference Companion on OOPSLA, pp. 819–820 (2009)

[5] Cabot, J., Gogolla, M.: In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) Object Constraint Language (OCL): A Definitive Guide, pp. 58–90. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30982-3_3 . https://doi.org/10.1007/978-3-642-30982-3_3

[6] Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: Emf: eclipse modeling framework. pearson education (2008)

[7] Riedl-Ehrenleitner, M., Demuth, A., Egyed, A.: Towards model-and-code consistency checking. In: 2014 IEEE 38th Annual Computer Software and Applications Conference, pp. 85–90 (2014). IEEE

[8] Pham, V.C., Radermacher, A., Gerard, S., Li, S.: Bidirectional mapping between architecture model and code for synchronization. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 239–242 (2017). IEEE

[9] Jongeling, R., Fredriksson, J., Ciccozzi, F., Cicchetti, A., Carlson, J.: Towards consistency checking between a system model and its implementation. In: Int. Conf. on Systems Modelling and Management, pp. 30–39 (2020). Springer

[10] Jongeling, R., Fredriksson, J., Ciccozzi, F., Carlson, J., Cicchetti, A.: Structural consistency between a system model and its implementation: a design science study in industry. In: ECMFA (2022)

[11] Zaheri, M., Famelis, M., Syriani, E.: Towards checking consistency-breaking updates between models and generated artifacts. In: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 400–409 (2021). IEEE

[12] Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., Montrieux, L.: Maintaining invariant traceability through bidirectional transformations. In: 2012 34th International

Conference on Software Engineering (ICSE), pp. 540–550 (2012). IEEE

[13] Khelladi, D.E., Combemale, B., Acher, M., Barais, O., Jézéquel, J.-M.: Co-evolving code with evolving metamodels. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE '20, pp. 1496–1508 (2020)

[14] Khelladi, D.E., Combemale, B., Acher, M., Barais, O.: On the power of abstraction: a model-driven co-evolution approach of software code. In: 2020 IEEE/ACM 42st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) (2020)

[15] Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 313–324 (2014)

[16] Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 416–419 (2011)

[17] Kebaili, Z.K., Khelladi, D.E., Acher, M., Barais, O.: Towards leveraging tests to identify impacts of metamodel and code co-evolution. In: Cabanillas, C., Pérez, F. (eds.) Intelligent Information Systems, pp. 129–137. Springer, Cham (2023)

[18] McMinn, P.: Search-based software test data generation: a survey. Software testing, Verification and reliability **14**(2), 105–156 (2004)

[19] Beyer, D.: Advances in automatic software testing: Test-comp 2022. In: FASE, pp. 321–335 (2022)

[20] Richters, M., Gogolla, M.: Validating uml models and ocl constraints. In: International Conference on the Unified Modeling Language, pp. 265–277 (2000). Springer

[21] Leung, H.K., White, L.: Insights into regression testing (software testing). In: Proceedings. Conference on Software Maintenance-1989, pp. 60–69 (1989). IEEE

[22] Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software testing, verification and reliability **22**(2), 67–120 (2012)

[23] Wong, W.E., Horgan, J.R., London, S., Agrawal, H.: A study of effective regression testing in practice. In: PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering, pp. 264–274 (1997). IEEE

[24] Le Dilavrec, Q., Khelladi, D.E., Blouin, A., Jézéquel, J.-M.: Untangling spaghetti of evolutions in software histories to identify code and test co-evolutions. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 206–216 (2021). IEEE

[25] Zaidman, A., Van Rompaey, B., Van Deursen, A., Demeyer, S.: Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empirical Software Engineering **16**, 325–364 (2011)

[26] Mens, T.: Introduction and Roadmap: History and Challenges of Software Evolution, pp. 1–11. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-76440-3_1 . https://doi.org/10.1007/978-3-540-76440-3_1

[27] Herrmannsdoerfer, M., Vermolen, S.D., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. Lecture Notes in Computer Science **6563 LNCS**, 163–182 (2011)

[28] Vermolen, S.D., Wachsmuth, G., Visser, E.: Reconstructing complex metamodel evolution. In: Sloane, A., Aßmann, U. (eds.) Software Language Engineering, pp. 201–221. Springer, Berlin, Heidelberg (2012)

[29] Khelladi, D.E., Hebig, R., Bendraou, R., Robin, J., Gervais, M.-P.: Detecting complex changes during metamodel evolution. In: CAISE, pp. 263–278 (2015). Springer

[30] Alter, S.: Work system theory: A bridge between business and IT views of systems. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **9097**, 520–521 (2015) https://doi.org/10.1007/978-3-319-19069-3

[31] Williams, J.R., Paige, R.F., Polack, F.A.: Searching for model migration strategies. In: Proceedings of the 6th International Workshop on Models and Evolution, pp. 39–44 (2012). ACM

[32] Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing dependent changes in coupled evolution. In: International Conference on Theory and Practice of Model Transformations, pp. 35–51 (2009). Springer

[33] Langer, P., Wimmer, M., Brosch, P., Herrmannsdörfer, M., Seidl, M., Wieland, K., Kappel, G.: A posteriori operation detection in evolving software models. Journal of Systems and Software **86**(2), 551–566 (2013)

[34] Khelladi, D.E., Hebig, R., Bendraou, R., Robin, J., Gervais, M.P.: Detecting complex changes and refactorings during (Meta)model evolution. Information Systems **62**, 220–241 (2016) https://doi.org/10.1016/j.is.2016.05.002

[35] Iovino, L., Pierantonio, A., Malavolta, I.: On the impact significance of metamodel evolution in mde. Journal of Object Technology **11**(3), 3–1 (2012)

[36] MDT: Model Development Tools. Object Constraints Language (OCL). http://www.eclipse.org/modeling/mdt/?project=ocl (2015)

[37] MDT: Model Development Tools. MoDisco. http://www.eclipse.org/modeling/mdt/?project=modisco (2015)

[38] MDT: Model Development Tools. Papyrus. http://www.eclipse.org/modeling/mdt/?project=papyrus (2015)

[39] EMF, E.: Eclipse Modeling Framework (EMF). https://github.com/eclipse-emf/org.eclipse.emf (2020)

[40] Roslan, M.F., Rojas, J.M., McMinn, P.: An Empirical Comparison of EvoSuite and DSpot for Improving Developer-Written Test Suites with Respect to Mutation Score. In: Papadakis, M., Vergilio, S.R. (eds.) Search-Based Software Engineering, pp. 19–34. Springer, Cham (2022)

[41] Herculano, W.B.R., Alves, E.L.G., Mongiovi, M.: Generated tests in the context of maintenance tasks: A series of empirical studies. IEEE Access **10**, 121418–121443 (2022) https://doi.org/10.1109/ACCESS.2022.3222803

[42] Roziere, B., Zhang, J.M., Charton, F., Harman, M., Synnaeve, G., Lample, G.: Leveraging automated unit tests for unsupervised code translation. arXiv preprint arXiv:2110.06773 (2021)

[43] Gruber, M., Roslan, M.F., Parry, O., Scharnböck, F., McMinn, P., Fraser, G.: Do automatic test generation tools generate flaky tests? (2023)

[44] Danglot, B., Vera-Perez, O., Yu, Z., Zaidman, A., Monperrus, M., Baudry, B.: A snowballing literature study on test amplification. Journal of Systems and Software **157**, 110398 (2019) https://doi.org/10.1016/j.jss.2019.110398

[45] Rojas, J.M., Fraser, G., Arcuri, A.: Seeding strategies in search-based unit test generation. Software Testing, Verification and Reliability **26**(5), 366–401 (2016) https://doi.org/10.1002/stvr.1601 https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1601

[46] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. springer (2012)

[47] OMG: Object Management Group. Unified Modeling Language (UML). http://www.omg.org/spec/UML/ (2015)

[48] OMG: Object Management Group. Business Process Model And Notation (BPMN). https://www.omg.org/spec/BPMN/2.0/About-BPMN/ (2015)

[49] Kessentini, W., Alizadeh, V.: Interactive metamodel/model co-evolution using unsupervised learning and multi-objective search. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 68–78 (2020)

[50] Kessentini, W., Wimmer, M., Sahraoui, H.: 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, pp. 101–111 (2018). ACM

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

[52] Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: 2008 12th International IEEE Enterprise Distributed Object Computing Conference, pp. 222–231 (2008). IEEE

[53] Herrmannsdoerfer, M., Benz, S., Juergens, E.: Cope-automating coupled evolution of metamodels and models. In: ECOOP, vol. 9, pp. 52–76 (2009). Springer

[54] Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Managing model adaptation by precise detection of metamodel changes. In: Model Driven Architecture-Foundations and Applications: 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings 5, pp. 34–49 (2009). Springer

[55] Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: ECOOP, vol. 7, pp. 600–624 (2007). Springer

[56] Paige, R.F., Matragkas, N., Rose, L.M.: Evolving models in model-driven engineering: State-of-the-art and future challenges. Journal of Systems and Software **111**, 272–280 (2016)

[57] Hebig, R., Khelladi, D.E., Bendraou, R.: Approaches to co-evolution of metamodels and models: A survey. IEEE Transactions on Software Engineering **43**(5), 396–414 (2016)

[58] Demuth, A., Riedl-Ehrenleitner, M., Lopez-Herrejon, R.E., Egyed, A.: Co-evolution of metamodels and models through consistent change propagation. Journal of Systems and Software **111**, 281–297 (2016)

[59] Cherfa, E., Mesli-Kesraoui, S., Tibermacine, C., Sadou, S., Fleurquin, R.: Identifying metamodel inaccurate structures during metamodel/constraint co-evolution. In: 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 24–34 (2021). IEEE

[60] Batot, E., Kessentini, W., Sahraoui, H., Famelis, M.: Heuristic-based recommendation for metamodel—ocl coevolution. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 210–220 (2017). IEEE

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

[62] Khelladi, D.E., Bendraou, R., Hebig, R., Gervais, M.-P.: A semi-automatic maintenance and co-evolution of ocl constraints with (meta) model evolution. Journal of Systems and Software **134**, 242–260 (2017)

[63] Correa, A., Werner, C.: Refactoring object constraint language specifications. Software & Systems Modeling **6**(2), 113–138 (2007)

[64] Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schoenboeck, J., Schwinger, W., Wimmer, M.: Systematic co-evolution of ocl expressions. In: 11th APCCM 2015, vol. 27, p. 30 (2015)

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

[66] Khelladi, D.E., Kretschmer, R., Egyed, A.: Change propagation-based and composition-based co-evolution of transformations with evolving metamodels. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 404–414 (2018). ACM

[67] Garcés, K., Vara, J.M., Jouault, F., Marcos, E.: Adapting transformations to metamodel changes via external transformation composition. Software & Systems Modeling **13**(2), 789–806 (2014)

[68] García, J., Diaz, O., Azanza, M.: Model transformation co-evolution: A semi-automatic approach. In: Software Language Engineering: 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers 5, pp. 144–163 (2013). Springer

[69] Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schwinger, W., Schonbock, J.: Consistent co-evolution of models and transformations. In: ACM/IEEE 18th MODELS, pp. 116–125 (2015)

[70] Henkel, J., Diwan, A.: Catchup! capturing and replaying refactorings to support api evolution. In: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005., pp. 274–283 (2005). IEEE

[71] Nguyen, H.A., Nguyen, T.T., Wilson Jr, G., Nguyen, A.T., Kim, M., Nguyen,

T.N.: A graph-based approach to api usage adaptation. ACM Sigplan Notices **45**(10), 302–321 (2010)

[72] Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. ACM Transactions on Software Engineering and Methodology (TOSEM) **20**(4), 19 (2011)

[73] Andersen, J., Lawall, J.L.: Generic patch inference. Automated software engineering **17**(2), 119–148 (2010)

[74] Engström, E., Runeson, P., Skoglund, M.: A systematic review on regression test selection techniques. Information and Software Technology **52**(1), 14–30 (2010)

[75] Mansour, P., El-Fakih, K.: Natural optimization algorithms for optimal regression testing. In: Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97), pp. 511–514 (1997). IEEE

[76] Gupta, R., Harrold, M.J., Soffa, M.L.: An approach to regression testing using slicing. In: ICSM, vol. 92, pp. 299–308 (1992)

[77] Willmor, D., Embury, S.M.: A safe regression test selection technique for database-driven applications. In: 21st IEEE International Conference on Software Maintenance (ICSM'05), pp. 421–430 (2005). IEEE

[78] Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: A survey. Softw. Test. Verif. Reliab. **22**(2), 67–120 (2012) https://doi.org/10.1002/stv.430

[79] Gligoric, M., Eloussi, L., Marinov, D.: Ekstazi: Lightweight test selection. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 713–716 (2015). https://doi.org/10.1109/ICSE.2015.230

[80] Ducasse, S., Lanza, M., Tichelaar, S.: Moose: an extensible language-independent environment for reengineering object-oriented systems. In: Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000), vol. 4 (2000)

[81] Ge, X., Murphy-Hill, E.: Manual refactoring changes with automated refactoring validation. In: Proceedings of the 36th International Conference on Software Engineering, pp. 1095–1105 (2014)