Automated co-evolution of metamodels and code

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 co-evolution of metamodels and code. IEEE Transactions on Software Engineering, 2025, 51 (4), pp.1067-1085. ⟨10.1109/tse.2025.3540545⟩. ⟨hal-04973171⟩

HAL Id: hal-04973171

https://hal.science/hal-04973171v1

Submitted on 2 Mar 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 CC BY 4.0 - Attribution - International License

Automated co-evolution of metamodels and code

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

Abstract—***Context.*** In Software Engineering, Model-Driven Engineering (MDE) is a methodology that considers Metamodels as a cornerstone. As an abstract artifact, a metamodel plays a significant role in the specification of a software language, particularly, in generating other artifacts of lower abstraction level, such as code. Developers then enrich the generated code to build their language services and tooling, e.g., editors, and checkers. ***Problem.*** When a metamodel evolves, the generated code is automatically updated. As a consequence, the developers' additional code is impacted and needs to be co-evolved accordingly. ***Contribution.*** This paper proposes a new fully automatic code co-evolution approach with the evolution of the Ecore metamodel. The approach relies on pattern matching of the additional code errors. This process aims to analyze the abstraction gap between the evolved metamodel elements and the code errors to co-evolve them. ***Evaluation and Results.*** We evaluated our approach on nine Eclipse projects from OCL, Modisco, and Papyrus over several evolved versions of three metamodels. Results show that we automatically co-evolved 771 errors due to metamodel evolution with 631 matched and applied resolutions. Our approach reached an average of 82% of precision and 81% of recall, varying from 48% to 100% for precision and recall respectively. To check the effect of the co-evolution and its behavioral correctness, we rely on generated test cases before and after co-evolution. We observed that the percentage of passing, failing, and erroneous tests remained the same with insignificant variations in some projects. Thus, suggesting the behavioral correctness of the co-evolution Moreover, we conducted a comparison with the use of quick fixes that represent a usual tool for correcting code errors in an IDE. We found that our automatic co-evolution approach outperforms the use of quick fixes that lacked the context of metamodel evolution. Finally, we also compared our approach with the state-of-the-art semi-automatic co-evolution approach. As expected, precision and recall are slightly better with semi-automation, but with the burden of manual intervention, which is alleviated with our automatic co-evolution.

Index Terms—Metamodels, Evolution, Code, Co-evolution, Test.

1 Introduction

The exponential growth of software systems leads to a substantial burden in terms of maintenance, often resulting in a high cost that may surpass the cost of software development itself [1]. Model-Driven Engineering (MDE) helps to ease the increasingly complex development and maintenance of large-systems [2], [3]. In particular, by supporting the task of building Software Languages (SLs) and their tooling that plays a significant role in all phases of development processes [4]. A central artifact in MDE when building languages is the *metamodel*, which defines the aspects of a business domain, i.e. the main concepts, their properties, and relationships between them [5]. A metamodel is the cornerstone to not only model instances, constraints, or transformations, but also to 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 on 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. Developers further enriched this generated code to offer additional functionalities and tools, such as validation, transformation, simulation, or debugging. We refer to this as additional code that developers as on top of the generated code API. A metamodel and its generated code API are, hence, a cornerstone when building a Software Language and its tooling. For instance, UML[1] and BPMN[2] Eclipse implementations rely on the UML and BPMN metamodels to first generate their corresponding code API before building around it all their tooling and services. This set-up of model-based generation of code and code enrichment is also found beyond EMF. For example, the JHipster project [3] proposes to generate, from entity models, the different stacks of modern web applications for both client-side and server-side code. Another popular example is OpenAPI[4], where code is generated from an API specification. Some low-code development platforms rely on metamodels [7] with data models and generators to raise abstraction and hide implementation-level details.

One of the foremost challenges to deal with in *MDE* and low-code platforms, is the impact of the evolution of metamodels on its dependent artifacts. In this paper, we

- *Zohra Kaouter Kebaili, CNRS, Univ. Rennes 1, IRISA, INRIA. E-mail: zohra-kaouter.kebaili@irisa.fr*
- *Djamel Eddine Khelladi, CNRS, Univ. Rennes 1, IRISA, INRIA. E-mail: djamel-eddine.khelladi@irisa.fr*
- *Mathieu Acher, CNRS, INSA Rennes, IUF, IRISA, Inria. E-mail: mathieu.acher@irisa.fr*
- *Olivier Barais, Univ. Rennes 1, IRISA, INRIA. E-mail: olivier.barais@irisa.fr*

Manuscript received February 8, 2023; revised xxxx.

1. https://www.eclipse.org/modeling/mdt/downloads/?project=uml2
2. https://www.eclipse.org/bpmn2-modeler/
3. Adopted in 344 companieshttps://www.jhipster.tech/companies-using-jhipster/
4. https://oai.github.io/Documentation/

focus on the impact of Ecore metamodels' evolution on the code. Indeed, when a metamodel evolves and the core API is regenerated again, the additional code implemented by developers can be impacted. As a consequence, this additional code must be co-evolved accordingly by executing a resolution for each impacted part of the code.

However, manual co-evolution can be tedious, error-prone, and time-consuming. Therefore, it is essential to support an automatic co-evolution of code when metamodels evolve. The co-evolution challenge has been extensively addressed in *MDE*. In particular, the literature has focused on the co-evolution between metamodel and models [8], [9], [10], [11], [12], [13], constraints [14], [15], [16], [17], and transformations [18], [19], [20], [21], [22]. Nonetheless, only a few works addressed the challenge of metamodels and code co-evolution. In particular, [23], [24], [25], [26], [27], [28] focused on consistency checking between models and code, but not its co-evolution. Other works [29], [30], [31] proposed to co-evolve the code. However, the former handles only the generated code API, it does not handle additional code and aims to maintain bidirectional traceability between the model and the code API. The latter supports a semi-automatic co-evolution requiring developers' intervention. Moreover, it does not use any validation process to check the correctness of the co-evolution and with no comparison to a baseline. To the best of our knowledge, no existing approach is aimed at fully automating the co-evolution between metamodels and code.

This paper fills this gap by proposing a new fully automated co-evolution approach of metamodels and code. After the metamodel evolution, we analyzed the resulting errors. We distinguish and handle two types: direct and indirect code errors. Direct errors are impacted code elements that can be matched with the causing change. Whereas indirect errors can't be matched directly with their causing change (more detail in section 3.6). The approach is based on the pattern matching of the direct errors with 1) their cause evolution changes in the metamodel, 2) the pattern of the generated code elements from their metamodel elements (e.g., a setter, a literal, etc.), and 2) the configuration usage of the generated code elements in the additional code (e.g., in a variable declaration, a method parameter, etc.). Then, every direct error is matched with one resolution or more corresponding resolutions if impacted by more than one change. For example, a move property *p* in the metamodel from one class to another through a reference *ref*, will impact its getter *var.getP()*. It can be matched with its associated resolution that will extend its navigation path with *ref* to *var.getRef().getP()*. Moreover, indirect errors that cannot be matched are resolved with the existing quick fixes in the IDE. In addition, to check whether our automatic co-evolution impacts the original code behavior, we rely on generated test cases before and after co-evolution to check the possible effect of the co-evolution as in [32], [33]. Finally, as we fully automate the co-evolution, we further generate a report of the applied resolutions for each error to help developers understand the co-evolution performed. This aims to reduce the trade-off of our automatic co-evolution w.r.t. semi-automatic or manual co-evolution.

We evaluate our approach on three Eclipse EMF Software Language implementations, namely OCL [34], Modisco [35], and Papyrus [36], which have been developed for more than 10 years and have been evolved several times.

We collected projects consisting of both original and manually evolved versions of metamodels and code. Thus, we evaluated to what extent our approach can correctly co-evolve the projects' code in response to the metamodel evolutions. Within three metamodels, the 330 evolution changes caused 837 errors in 9 projects, consisting of 771 direct and 66 indirect errors. Our automatic co-evolution approach was able to match and apply 631 resolutions for the direct errors. All 66 indirect errors were resolved with 5 quick fixes. When comparing our automatic co-evolution to the expected one in the manually evolved versions of the projects, we observed the co-evolution precision and recall varying from 48% to 100%, reaching an average of, respectively, 82% of precision and 81% of recall. Furthermore, with the generated tests before and after co-evolution, we observed that the percentage of passing, failing, and erroneous tests remained stable with insignificant variations in some projects. Thus, suggesting the behavioral correctness of the co-evolution. Finally, we applied the quick fixes of the IDE as a baseline to compare with our approach. We run Eclipse quick fixes automatically on the 837 errors to investigate their results and compare them with our automatic co-evolution results. We found that Eclipse quick fixes eliminate errors improperly, leading to a code that does not conform to the evolved versions of the metamodels. We further compared our approach to the state of the art of metamodel and code co-evolution, namely our previous work [30] that performs a semi-automatic co-evolution. As expected our fully automatic co-evolution has lower precision and recall compared to a semi-automatic co-evolution (81% vs 91%), but removes completely the burden of manual intervention from developers on every code error to co-evolve. We provide a replication package available online[5].

This paper is structured as follows. Section 2 introduces a motivating example. Section 3 presents our overall automatic co-evolution approach. Section 4 details our evaluation results and threats to validity. Section 6 discusses related work and how our work distinguishes from state of the art. Finally, Section 7 concludes the article and reflects on future work.

2 Motivating Example

This section introduces a motivating example to illustrate the challenge of metamodel and code co-evolution. Let us take as an example the Modisco project [35], which has evolved numerous times in the past. Modisco is an academic initiative project implemented in the Eclipse platform to support the development of model-driven tools, reverse engineering, verification, and transformation of existing software systems [37], [38].

Figure 1 shows an excerpt of the "Modisco Discovery Benchmark" metamodel[6] consisting of 10 classes in version 0.9.0. It illustrates some of the domain concepts **Discovery**, **Project**, and **ProjectDiscovery** used for the discovery

5. https://figshare.com/s/8986914e924300be77da
6. 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

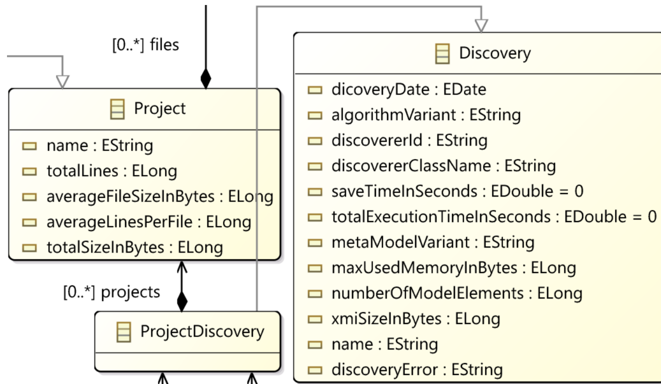

Figure 1: Excerpt of Modisco Benchmark metamodel in version 0.9.0.

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. Listing 1 shows a snippet of the generated Java interfaces and classes from the metamodel in Figure 1.

The generated code API is further enriched by the developers with additional code functionalities in the ”Modisco Discovery Benchmark” project and its dependent projects as well. For instance, by implementing the methods defined in metaclasses and advanced functionalities in new classes. Listing 2 shows the two classes `Report` and `CDOProjectDiscoveryImpl` of the additional code in the same project ”Modisco Dsicovery Benchmark” and in another dependent project, namely the ”Modisco Java Discoverer Benchmark” project. In version 0.11.0, the ”Modisco Discovery Benchmark” metamodel evolved with several significant changes, among which the following impacting changes:

1) Deleting the metaclass `ProjectDiscovery`.
2) Renaming the property *totalExecutionTimeInSeconds* to *discoveryTimeInSeconds* in metaclass `Discovery`.
3) Moving the property *discoveryTimeInSeconds* (after its rename) from metaclass `Discovery` to `DiscoveryIteration`.

After applying these metamodel changes, naturally, the code of Listing 1 is regenerated from the evolved version of the metamodel, which in turn impacts the existing additional code depicted in Listings 2. The resulting errors in the original code in version 0.9.0 are underlined in red in Listing 2. Listing 3 presents the final result of the co-evolution process in version 0.11.0. The co-evolved code is underlined in green. For example, in response to the *delete* of the metaclass `ProjectDiscovery`, its import statement *Line 1* in Listing 2, and any usage of it or its methods are impacted. The import statement is completely removed. The same can be applied to the usages of the class and its methods. Alternatively, they could also be replaced by a default value rather than removing the whole instruction. The intention is to maintain the developers' code with minimal removal co-evolution.

Furthermore, the same changes *rename* and *move* of the property *totalExecutionTimeInSeconds* impact two usages that are co-evolved differently. First, the call of `setTotalExecutionTimeInSeconds` (*Line 4* in Listing 2) that is co-evolved by renaming it to `setDiscoveryTimeInSeconds`, then extending the path with `getIterations()`. The second impact is the use of the generated literal `BenchmarkPackage`.*DISCOVERY__TOTAL_EXECUTION_-TIME_IN_SECONDS*. It is successively co-evolved by renaming it to `BenchmarkPackage`.*DISCOVERY__DISCOVERY_TIME_-IN_SECONDS* before replacing its source class `DISCOVERY` by `DISCOVERY_ITERATION`. Note that when using the IDE quick fixes to co-evolve these errors, it suggests to create the method `setTotalExecutionTimeInSeconds` in the class `Discovery` and the literal *DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS* in the class `BenchmarkPackage`, which does not meet the required co-evolutions shown in Listing 3.

The above examples show the importance of correctly matching the different code usages and patterns of the generated code elements with the metamodel evolution changes to co-evolve them with the appropriate resolutions.

The next section presents our contribution for a fully automatic co-evolution of metamodel and code.

Listing 1: excertprpt of the generated code in org.eclipse.modisco.infra.discovery.benchmark.

```
1  //Discovery Interface
2    public interface Discovery extends EObject {
3    double getTotalExecutionTimeInSeconds();
4    void setTotalExecutionTimeInSeconds(double value);
5          ...
6    }
7  //Project Interface
8    public interface ProjectDiscovery extends Discovery
           {...}
9  //DiscoveryImpl Class
10   public class DiscoveryImpl extends EObjectImpl
          implements Discovery {
11      public double getTotalExecutionTimeInSeconds() {...}
12      public void setTotalExecutionTimeInSeconds(double
            totalExecTime) {...}
13          ...
14    }
```

Listing 2: Excerpt of the additional code V1.

```
1    import org.eclipse.modisco.infra.discovery.benchmark.
          ProjectDiscovery;
2    public class Report {
3     ...
4        discovery.setTotalExecutionTimeInSeconds(...);
5     }
6     ...
7    public class CDOProjectDiscoveryImpl extends
          AbstractCDODiscoveryImpl implements
          CDOProjectDiscovery {
8      ...
9     case JavaBenchmarkPackage.
10    CDO_PROJECT_DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS:
           return BenchmarkPackage.
11     DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS;
12     ...
13    }
```

Listing 3: Excerpt of the additional code V2.

```
1    import org.eclipse.modisco.infra.discovery.benchmark.
          ProjectDiscovery;
2    public class Report {
3    ...
4    discovery.getIterations().
5                  setDiscoveryTimeInSeconds(...);
6    ...
7    }
8    public class CDOProjectDiscoveryImpl extends
          AbstractCDODiscoveryImpl implements
          CDOProjectDiscovery {
9     ...
```

```
10      case JavaBenchmarkPackage.
11      CDO_PROJECT_DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS:
            return BenchmarkPackage.
12      DISCOVERY_ITERATION__DISCOVERY_TIME_IN_SECONDS;
13      ...
14      }
15      ...
16      }
```

3 APPROACH

This section presents the overall approach of our automated co-evolution of code with evolving metamodels, instantiating on the Ecore technological space. First, we give an overview of the approach and specify the metamodel evolution changes we consider. Then, we present how we retrieve the resulting errors due to metamodel evolution, followed by the regeneration of the code API. After that, we present the pattern matching process, which is an important part of our fully automatic co-evolution approach, before discussing the resolutions of the code errors.

3.1 Overview

Figure 2 depicts the overall steps for the automatic co-evolution of the metamodel and code, with horizontally separated parts defining chronological order from the top to the bottom. After the generation step (the upper part of Figure 2), the evolution of the Ecore metamodel will cause errors in the additional Java code that depends on the API of the newly generated code (the middle part of Figure 2). We take as input the evolution changes of the metamodel between the two versions of this metamodel [1]. Then, we parse the additional code [2] to retrieve the list of errors. After that, we get to the bottom part of Figure 2, both the list of metamodel changes and the list of errors are used as inputs for the pattern matching step [3]. It analyzes the structure of the error to match it with its impacting metamodel change and decides which resolution [4] to apply for the error co-evolution [5]. The metamodel changes provide the ingredients and necessary information that are used for the co-evolution. At the end of the automatic co-evolution, we obtain a new co-evolved additional code [6] along a generated report on the applied resolutions. In addition to the automatic co-evolution, we generate test cases before and after co-evolution to highlight its possible effect. In fact, many research papers rely on the use of tests to check the behavior of the code during its evolution. For example, Godefroid et al. [39] uses tests to find regressions in different versions of REST APIs. In particular, Lamothe et al. [32], [33] use tests to validate the evolution of the client code after Android API migration. We apply a similar method to check the effect of the co-evolution. Finally, during the co-evolution process, we generate a report linking the applied resolutions for each code error with its impacting metamodel change. If needed, this can help developers in understanding the performed co-evolution, since we fully automate it.

3.2 Metamodel Evolution Changes

One of the intrinsic properties of software artifacts is its continuous evolution [40]. Metamodels are no different and are meant to evolve. Two types of evolution changes are considered when evolving a metamodel: *atomic* and *complex* changes [41]. Atomic changes are additions, removals, and updates of a metamodel element. Complex changes consist of a sequence of atomic changes combined together [42], [43]. 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 property and add property [41]. Many approaches in the literature [42], [44], [45], [46], [47], [48], [49] exist to detect metamodel changes between two versions. Note that the detected list of complex changes does not include the list of the detected atomic changes, i.e., no overlap in between. Moreover, the detection approaches must order the changes in a consistent way. This is fundamentally a problem that change detection approaches must deal with, and hence, is out of scope for our problem of code co-evolution. Nonetheless, it is important and expect a consistent order of changes to not hinder the quality of the co-evolution.

For the purpose of modularity and extensibility, we use a specification layer for the changes [1] that is simply a connection layer to our co-evolution approach with existing change detection approaches. This connection layer specifies our own representation of a metamodel change that can be mapped later with any change representation. It simply specifies the needed information for each change in the form of its attributes. In the left column of Table 1, we precise the impacting changes that we consider in our work. For each change, we precise in the second column the attributes that represent and compose each change. When using a state-of-the Art detection approach, we analyze in white box the detected changes to extract their attributes and map them to our internal change layer. For example, a rename property change includes information regarding its old name, new name, and its class container. Therefore, in practice, any detection approach [42], [44], [45], [46], [47], [48], [49] can be integrated by bridging its changes' representation to our change layer and the rest of co-evolution can be performed independently. In this paper, we chose to reuse our previous work [48], [50], a heuristic-based approach to actually detect atomic and complex changes between two versions of a metamodel. In the rest of the paper, we focus on the code co-evolution since it is our main contribution.

3.3 Error Retrieval

After the metamodel is evolved and the code API is re-generated, errors will appear in the additional code that must be co-evolved. Unlike code migration context [32], [51], these errors represent the delimited impact of the metamodel evolution. Thus, rather than an impact analysis on the original version to trace the impact of a metamodel change in the code, our approach relies on the compilation result of the code to retrieve its errors. This is necessary and useful in our approach, as we will need to keep updating the list of code errors after co-evolving each given error, hence, iteratively co-evolving the code. We detail this process in the following subsections.

To retrieve those errors, we start by parsing the code of each Java class, called a *compilation unit*, to access the Abstract Syntax Trees (ASTs). An error in a Java code is called a *Marker* that contains the information regarding the detected

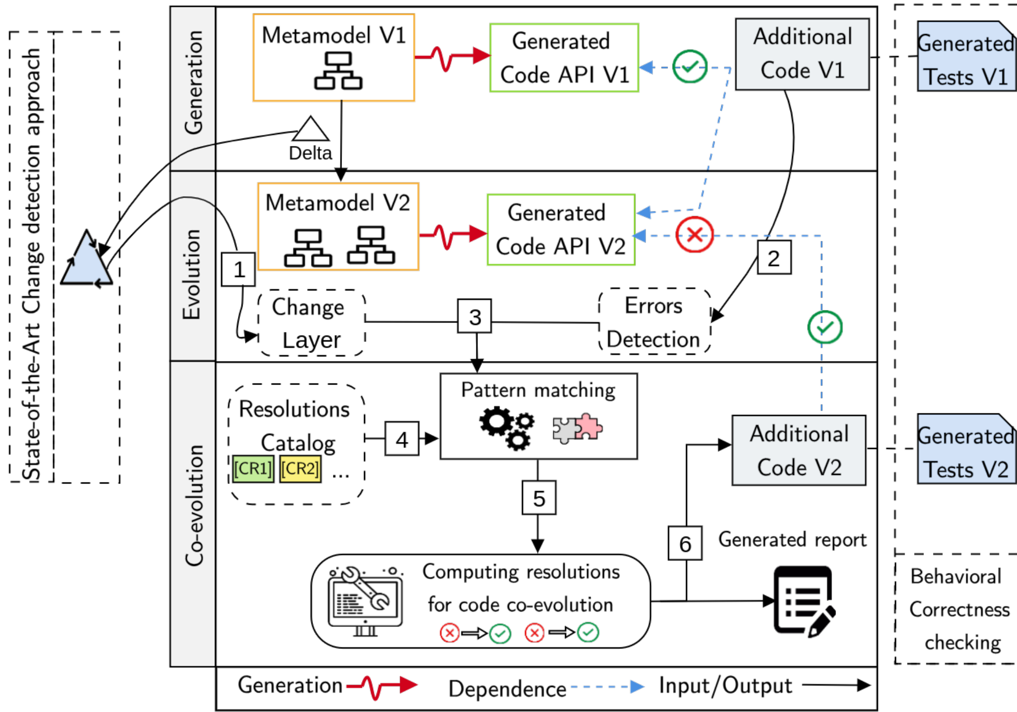

Figure 2: Overall approach for metamodel and code co-evolution

error. It contains the necessary information to locate the exact impacted AST node in the parsed global AST (*i.e.,* char start and end) and to process it (*i.e.,* message). In the remaining part of the paper, instead of Markers, compilation units, and additional code, we respectively refer only to errors, Java classes, and code for the sake of simplicity.

3.4 Resolution Catalog

Now that we have a list of code errors, we need a set of resolutions to co-evolve them. Our co-evolution approach relies on the resolutions shown in Table 1. It depicts the resolutions associated with metamodel changes that are known to have an impact on code [52]. The resolutions are taken from existing co-evolution approaches of various MDE artifacts [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [53], [54], [55], where they showed to be efficient and useful in co-evolving code [30].

For example, resolutions $[CR8, CR9, CR10, CR11]$ aim to co-evolve the different code errors of a move property in the metamodel.

3.5 Pattern Matching for Resolution Selection

As shown in Table 1, alternative resolutions exist per metamodel change. Since we aim to fully automate the co-evolution, we need a mechanism to analyze the code error, the code usage, and its impacting metamodel change to decide which resolution to apply. This section presents the pattern matching process between metamodel changes and code usages for the retrieved code errors to automatically select resolutions for their co-evolution.

Each Metamodel Element ME has corresponding Generated Code Elements $\{GCE_0$, GCE_1, ..., $GCE_n\}$ in the code API. GCE_i have **usages** in the **additional code** as illustrated in Figure 3. Thus, evolving a ME will regenerate $\{GCE_0, GCE_1, ..., GCE_n\}$ which will impact their **usages**. Table 2 classifies the different patterns of the generated code elements GCE_i for each metamodel element ME type, and provides illustrative examples. It shows that various patterns of code elements are generated for each metamodel element type. For example, let us consider the case of a metaclass. EMF generates a corresponding interface and a class implementation, a *createClass()* method in the factory class, three literals (*i.e.,* constants) for the class and an accessor method in the package class, and a corresponding create adapter method. For the attribute case, EMF generates the signature and implementation of a getter and a setter, an accessor method in the package class, and a literal.

This classification is essential to match the different code errors with their corresponding **pattern** of the GCE_i in Table 2. Moreover, each generated code element GCE_i can be used in different **configurations** in the code that must also be considered in the pattern matching process. For example, using a GCE_i as a parameter in a method declaration and in a method invocation, or to initialize a variable declaration, or in an expression call in a statement, etc., are considered as different configurations and can influence which resolution to apply as well. With these ingredients, we can match a resolution for each error.

Algorithm 1 summarizes the pattern matching process. Given a Java class, an error, and a list of metamodel changes, Algorithm 1 first retrieves the error AST node $\boxed{line\ 2}$. After that, it identifies the configuration of the GCE usages $\boxed{Line\ 3}$. Then, for each metamodel change type $\boxed{lines\ 6, 17, 22, 31}$, it identifies the pattern of the corresponding GCE presented in Table 2 $\boxed{line\ 7, 18, 23, 32}$. Depending on the detected configuration, the appropriate pre-selected

Table 1: Catalog of resolutions used for the code co-evolution of direct errors due to the metamodel changes.

Impacting Metamodel Changes	Changes' attributes	Proposed Code Resolutions
⋄ Delete property *p* from class *C*	∘ Property *p* name ∘ Container class *C* name	▷[$CR1$] Remove the direct use of *p* (e.g., label = s.name + s.m1().p.m2() → label = s.name + ((Type_Of_P) s.m1()).m2()) ▷[$CR2$] Remove the statement using *p* (i.e., if, loop, assignment, etc.) ▷[$CR3$] Remove the whole call path of *p* (e.g., label = s.name + s.m1().m2().p → label = s.name) ▷[$CR4$] Replace the whole call path of *p* with a default value (e.g., id = s.id + s.m1().m2().p → id = s.id + 0)
⋄ Delete class `C`	∘ Class *C* name ∘ Container package *Q* name	▷[$CR1$] Remove the direct use of the type `C` (e.g., extending/implementing `C`, in method argument/returned type and not the whole method declaration. Calls to the updated methods are subsequently updated) ▷[$CR2$] Remove the statements using the type `C` (e.g., import, variable declaration, method argument/returned type, method declaration, type instantiation, etc. Calls to the deleted variables and methods are subsequently removed)
⋄ Rename element *e* to *e′*	∘ Element *e* old name ∘ Element *e′* new name ∘ Container package *Q* or class *C* name	▷[$CR5$] Rename *e* in the code
⋄ Generalize multiplicity of property *p* of the class *C* from a single to multiple values	∘ Property *p* name ∘ Container Class *C* ∘ Old multiplicity ∘ New multiplicity	▷[$CR6$] Retrieve the first value of a collection (e.g., value = *lng.p* → value = *lng.p.toArray()[0]* or *lng.p.get(0)*)
⋄ 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*	∘ Property *p* name ∘ Source container class *S* name ∘ Target container class *T* name ∘ Reference *ref* name	▷[$CR7$] Extend navigation path of p_i (e.g., *lng.*p_i → *lng.ref.*p_i) ▷[$CR8$] Extend navigation path of p_i and add a for loop (e.g., *lng.*p_i → *for(v in lng.ref)* {*v.*p_i}) ▷[$CR9$] Reduce navigation path of p_i (e.g., *lng.ref.*p_i → *lng.*p_i) ▷[$CR10$] Replace S by T_REF in Literal values (e.g., *MetamodelPackage.S__*p_i → *MetamodelPackage.T__*p_i)
⋄ Push property *p* from class `Sup` to `Sub`$_1$,...,`Sub`$_n$	∘ Property *p* name ∘ Container class *Sup* name ∘ List of container classes Sub_i names	▷[$CR11$] Introduce a type test with an If statement (e.g., *t.name = s.p.name* → $if(s.p.istypeof(Sub_1))$ {t.name = (Sub$_1$ s).p.name} ... $else\ if(s.p.istypeof(Sub_n)$ {t.name = (Sub$_n$ s).p.name}) ▷[$CR12$] Cast *p* to one specific sub class Sub_i (e.g., *t.name = s.p.name* → *t.name = ((*Sub_i*)s).p.name*) ▷[$CR13$] Duplicate the statement using the literal for each subclass and replace Sup by Sub_i (e.g., *add(Package.Sup__P)* → *add(Package.Sub*$_0$*__P), ... , add(Package.Sub*$_n$*__P)*)
⋄ Pull property *p* from classes `Sub`$_1$,...,`Sub`$_n$ to `Sup`	∘ Property *p* name ∘ List of container classes Sub_i ∘ Container class *Sup* name	▷[$CR14$] Replace Sub_i by Sup in Literal values (e.g., *MetamodelPackage.*Sub_i*__P* → *MetamodelPackage.Sup__P*)
⋄ Inline class `S` to `T` with properties $p_1, ..., p_n$	∘ List of properties p_i ∘ Container Source class *S* name ∘ Container Target class *T* name	▷[$CR9$] Reduce navigation path of p_i (e.g., *lng.ref.*p_i → *lng.*p_i) ▷[$CR15$] Change the class type from `S` to `T` (e.g., List<S> l = ...; → List<T> l = ...;)
⋄ Change property *p* type of the class *C* from `S` to `T`	∘ Property *p* name ∘ Container class *C* name ∘ Old type *S* ∘ New type *T*	▷[$CR16$] Change variable declaration type initialized with *p* from `S` to `T` (e.g., S var = s.p; → T var = s.p;) ▷[$CR17$] Add a cast of *p*

Table 2: Classification of the different patterns of the generated code element from the metamodel elements.

Metamodel element type	**Generated code elements**	**Pattern of the generated code elements**	**Illustrative examples**
Metaclass	Interface	”MetaClassName”	*Constraint*
	createClass() (in metamodelFactory class)	”create”+”MetaClassName”()	*createConstraint()*
	Literals of the class	”META_CLASS_NAME” ”META_CLASS_NAME”+”_”+ ”FEATURE_COUNT” ”META_CLASS_NAME”+ ”_”+”OPERATION_COUNT”	*CONSTRAINT*, *CONSTRAINT_FEATURE_COUNT*, *CONSTRAINT_OPERATION_COUNT*
	Accessor of Meta objects (in metamodelPackage class)	”get”+”MetaClassName”()	*getConstraint()*
	Class implementation	”MetaClassNameImpl”	*ConstraintImpl*
	Adapter	”create”+”MetaClassName”+”Adapter”	*createConstraintAdapter()*
Attribute (same for a reference)	Signature of getters and setters	”get”+”AttributeName”(), ”set”+”AttributeName”()	*getStereotype(), setStereotype()*
	Accessor of Meta objects	”get”+”MetaClassName”+”_”+”AttributeName”()	*getConstraint_Stereotype()*
	Literal	”META_CLASS_NAME”+”__”+”ATTRIBUTE_NAME”	*CONSTRAINT__STEREOTYPE*
	Implementation of getters and setters	”get”+”AttributeName”(), ”set”+”AttributeName”()	getStereotype(), setStereotype()
Method	Declaration of the method	”methodName”()	*UniqueName()*
	Accessor of meta objects	”get”+”MetaClass”+”__”+”MethodName”()	*getCONSTRAINT__UniqueName()*
	Literal	”META_CLASS_NAME”+”__”+”METHOD_NAME”	*CONSTRAINT___UNIQUE_NAME*
	Implementation of the method	”methodName”()	*UniqueName()*

resolution is added to the output set $\boxed{Line\ 10, 13, 19, 26, 35}$. Finally, the selected resolutions set is returned for further processing in the automatic co-evolution $\boxed{Line\ 41}$. Let us take the example of "`Change property type from S to T`" (last change type in Table 1). It has two possible resolutions, CR16 and CR17. Assuming that we have a code error that must be co-evolved. Our pattern matching process first identifies the pattern of the corresponding GCE_i from Table 2 (Line 7), then the configuration of the GCE_i usage (Lines 8). Algorithm 1 starts by parsing the error to find the pattern of the corresponding GCE_i, which will help to find the causing change. The next step is to define the configuration of the GCE_i which means the type of the GCE_i usage. If it is a variable declaration (line 9), the resolution $CR16$ is returned. If any other configuration is detected, the pattern matching process returns $CR17$ as an appropriate resolution.

Note that an error can be matched with more than one resolution because a metamodel element ME can be impacted by more than one change, in other terms interdependent changes. For example, Algorithm 1 allows matching the error in $\boxed{Line 17}$ (rename property) and

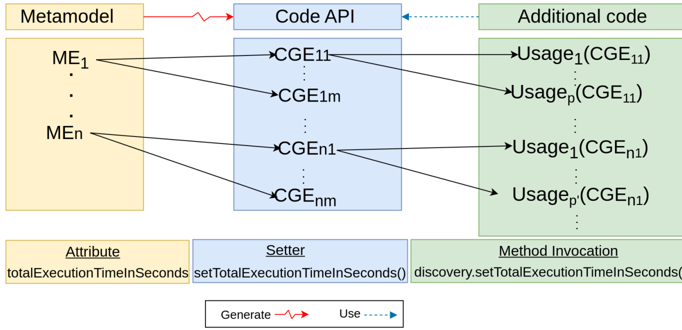

Figure 3: Schema for mapping between the metamodel and code.

$\boxed{Line31}$ (move property) with two metamodel changes of the property *totalExecutionTimeInSeconds* in Listing 2. The generated literal, which is our generated code element GCE_i, is used here in the configuration of a literal static field access *BenchmarkPackage.DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS*. The returned resolutions are $CR5$ and $CR10$ $\boxed{Line19, 35}$, to be executed in the order of detection of their causing metamodel changes.

A similar mechanism is implemented for the rest of metamodel changes to select a resolution based on the pattern of GCE_i and the configuration of its usage. For the sake of readability, Algorithm 1 does not show all possible combinations of $metamodel\ changes \times patterns \times configurations$, but few examples to illustrate its essence. We nonetheless give an extended version in the appendix.

Finally, in our implementation, for the case of a "Delete Class" and "Delete Property", we favor the least deletion when possible. In particular, depending on the configuration usage, we select the resolution that deletes the least possible among $CR1$, $CR2$, $CR3$, $CR4$. For example, for an error in a parameter of a method call, we select the resolution $CR4$ rather than $CR1$ or $CR2$. In Listing 2, Algorithm 1 matches the error in the import declaration (**Configuration**), with the deletion of the metaclass *ProjectDiscovery* (**pattern**), which allows to select the resolution $CR2$.

Algorithm 1: Pattern matching algorithm

```
Data: javaClass, error, changesList
resolution_s ← ϕ
errorNode ← findErrorAstNode(javaClass, error)
configuration ← getConfiguration(javaClass,errornode)
for (change ∈ changesList) do
    switch change do
        case ChangePropertyType do
            if (match(patternGCE,change)) then
                switch configuration do
                    case VariableDeclaration do
                        resolution_s.add("CR16")
                    end
                    otherwise do
                        resolution_s.add("CR17")
                    end
                end
        end
        case RenameProperty do
            if (match(patternGCE,change)) then
                resolution_s.add("CR5")
        end
        case DeleteClass do
            if (match(patternGCE,change)) then
                switch configuration do
                    case ImportDeclaration do
                        resolution_s.add("CR2")
                    end
                    ... /*Other configurations*/
                end
        end
        case MoveProperty do
            if (match(patternGCE,change)) then
                switch configuration do
                    case LiteralStaticField do
                        resolution_s.add("CR10")
                    end
                    ... /*Other configurations*/
                end
        end
        case ... /*Other changes*/ do
            ...
        end
    end
end
return resolution_s
```

3.6 Repair mechanism for the code co-evolution

After the pattern matching step, we can proceed to co-evolve the code. Herein, we distinguish **direct errors** and **indirect errors**. The former are errors that do use a Generated Code Element GCE. They can be matched with one or many metamodel changes to select the appropriate one or many resolution(s). The latter are errors that do not use a generated code element. They are not matched, and hence, cannot be resolved using the pattern matching process.

Algorithm 2 presents the general process of the code co-evolution. It starts by parsing the project Java classes $\boxed{line\ 1}$ to be able to access the AST of the code. Then it browses the parsed classes to retrieve the list of errors $\boxed{line\ 3}$. The next step is to run the pattern matching Algorithm 1 for each error to return the matched resolution(s) if any $\boxed{lines\ 4-15}$. If an error is matched with at least one resolution $\boxed{line\ 7}$, it is a **direct error**. If an error is matched with several changes $\boxed{lines\ 8-10}$, it will be resolved iteratively for each resolution $\boxed{lines\ 9}$.

In the case of an **indirect error**, we cannot apply the pattern matching process, but we still attempt to repair them with the available quick fixes in the IDE. We analyze the error message to match it with one of the proposed Java quick fixes $\boxed{lines\ 11}$. For example, *the type MC must implement the inherited abstract method* is considered as an indirect error. This error can occur when a method is added in a metaclass `MC`. Classes that implement the generated interface generated from `MC` must override the new method. We attempt to repair it with its quick fix *add the unimplemented method* $\boxed{lines\ 12}$.

After applying the resolutions or a quick fix, the Java class has to be refreshed. This is because the modifications of the AST will impact the list of errors and their locations in the Java class $\boxed{lines\ 13-14}$. When refreshing the Java class, the list of errors typically decreases as they are co-evolved, yet, new ones may be introduced. Consequently, they will be handled in the next iterations similarly with our pattern matching and co-evolution algorithm or with

Algorithm 2: Co-evolution of metamodel and code

```
Data: EcoreModelingProject, changesList
javaClasses ← Parse(EcoreModelingProject)
for ( jc ∈ javaClasses) do
    errorsList ← getErrors(jc)
    while (!errorsList.isEmpty()) do
        error ← errorsList.next()
        resolution_s ← patternMatching(jc,error,
         changesList)
        if (! resolution_s.isEmpty()) /*direct errors*/ then
            for (resolution ∈ resolution_s) do
                applyResolution(jc,error, resolution)
            end
        else if error.hasQuickFix() /*indirect errors*/
         then
            useQuickFixes(error)
        refreshJavaClass(jc)
        refreshErrorsList(jc, errorsList)
    end
end
```

the quick fixes.

Finally, this process is repeated until all errors are handled. However, we implemented a stopping criteria to handle the indirect errors with quick fixes. It stops in two cases: 1) when only errors without any quick fix proposal remain, or 2) when the application of a quick fix causes an infinite loop [56], [57], i.e., a cycle of applying a quick fix that causes a previously fixed error (A $\mapsto$ B $\mapsto$ A $\mapsto$ B $\mapsto$ ...). However, as we will see in the evaluation, these cases never happened in our executions.

Note that during the co-evolution process, we log the history of execution: the Java class, the error, its line, the change that provoked it, and the applied resolution(s)/quick fix in a generated report. Thus, we can generate a detailed report that allows the developer to check the modifications applied to the co-evolved code after the co-evolution is finished, i.e, their validation is not mendatory to concretly apply the resolutions. Table 3 shows an example of a report for the corresponding part to our motivation example from Modisco Java Discoverer Benchmark applied co-evolution.

3.7 Prototype Implementation

We implemented our solution as an Eclipse Java plugin handling Ecore/EMF metamodels and their Java code.

The co-evolution process technically consists of the code AST manipulation using the JDT eclipse plugin[7].

- **Error retrieving**: there are many methods to manipulate the compilation errors of the code. We use the marker of IJavaModelMarker.JAVA_MODEL_-PROBLEM_MARKER. Then we filter markers whose severity value equals 2.
- **Change detection layer**: it consists of a set of model classes that specifies the information of each type of atomic and complex changes.
- **Pattern matching**: to match the error AST node with the causing change, we proceed by string formatting between the identifier of the AST node and the relevant information encapsulated in the change. We find the corresponding configuration by looking for the higher levels of the error AST node in the code AST using the package org.eclipse.jdt.core.dom.
- **Resolutions** : The AST nodes of errors are manipulated with edit actions: modification, replacement, or deletion) using the package *org.eclipse.jdt.core.dom.rewrite*, org.eclipse.text.edits.TextEdit, and *org.eclipse.core.filebuffers*.
- **Quick fixes** : we use *org.eclipse.jdt.ui.text.java.IQuickAssistProcessor* and *org.eclipse.jdt.ui.text.java.IJavaCompletionProposal*.

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

4 Evaluation

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

4.1 Evaluation Process

We evaluate our automated co-evolution of code with metamodel evolution by measuring: 1) its ability to co-evolve the errors, 2) its time performance, 3) its correctness by using recall and precision metrics, 4) its behavioral correctness by using test suites before/after the automatic co-evolution, 5) and by comparing it with both quick fixes and our prior work [30].

First, as our approach co-evolves the erroneous code due to metamodel evolution, we need to **provoke the errors in the code**. To do so, we replace the original metamodel with the evolved metamodel. Then, we regenerate the code API with EMF. This will cause errors in the code that our approach must co-evolve (1). We then use the function $System.nanoTime()$ for time measurement (2). After that, we measure the correctness of our code co-evolution (3) by comparing, for the same set of code errors that we automatically co-evolved, how they were manually co-evolved by developers. This allows us to measure the *precision* and *recall* reached by our co-evolution approach. They vary from 0 to 1, i.e., 0% to 100%. They are defined as follows:

$$precision = \frac{AppliedResolutions \cap ExpectedResolutions}{AppliedResolutions}$$

$$recall = \frac{AppliedResolutions \cap ExpectedResolutions}{ExpectedResolutions}$$

The $AppliedResolutions$ are those by our approach (from Table 1) and the $ExpectedResolutions$ are the actual manually performed resolutions by developers.

Moreover, to check if the co-evolution impacts the original code behavior (4), we generate tests for the original and the automatically co-evolved versions. Hence, we observe the behavioral effect of our co-evolution through the tests. To do so, we rely on Evosuite [58], a popular tool for test cases generation, as it is widely used by researchers and developers. Gruber et al. [59] 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

Table 3: Excerpt from the traced report of Modisco Java Discoverer Benchmark project

File	Error	Line	Change	Resolution
CDOProjectDiscoveryImpl.java	ProjectDiscovery	29	Delete class ProjectDiscovery	CR2
Report.java	setTotalExecutionTimeInSeconds	183	Rename property	CR5
Report.java	setDiscoveryTimeInSeconds	183	Moving property	CR7
CDOProjectDiscoveryImpl.java	DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS	799	Rename property	CR5
CDOProjectDiscoveryImpl.java	DISCOVERY__DISCOVERY_TIME_IN_SECONDS	799	Move property	CR10

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.

Furthermore, we compare our approach with the application of quick fixes (5). For each error, we apply the first quick fix in the list of corresponding proposals, then we compare the precision and recall of quick fixes with the precision and recall of our approach. Finally, we compare our approach with our prior work [30] in terms of precision and recall, and general rationale (5).

4.2 Data Set

This section presents the used data set in our evaluation to be found in the attached supplementary material[8]. We chose the EMF case study from the Eclipse platform, which is a popular tool that provides a flexible and extensible platform for creating custom modeling tools and applications. In 2022, Eclipse IDE was downloaded 1 million times per month.

First, we aimed at selecting meaningful evolutions that do not consist of only deleting metamodel elements, but rather include complex evolution changes. This selection criterion resulted in projects that do not contain unit tests, and this is the main reason behind relying on generated tests to check the behavioral correctness of the co-evolution. Moreover, to make sure that the errors are due only to the evolution of a single metamodel at a time, we selected projects that were dependent on one metamodel and not dependent (directly or transitively) on several metamodels that evolved simultaneously. This gives us more confidence in observing the resulting errors in the code due only to metamodel changes. Thus, mitigating the bias related to the ambiguous cases where errors interact and mask each other [60]. Handling the scenario code co-evolution due to multiple metamodels evolution is left for future work.

Therefore, we evaluated our approach on nine 9 Java projects from three case studies of three different language implementations in Eclipse, namely OCL [34], Modisco [35], and Papyrus [36]. 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 the development of model-driven tools, reverse engineering, verification, and transformation of existing software systems. Papyrus is an industrial project led by CEA[9] to support model-based simulation, formal testing, safety analysis, etc. Thus, the three case studies cover standard, academic, and industrial languages that have evolved several times for more than 10 years of continuous development period. In particular, Papyrus and OCL are open-source projects that are actively maintained with frequent releases per year.

8. https://figshare.com/s/8986914e924300be77da
9. http://www-list.cea.fr/en/

Table 4 gives details on the selected case studies, in particular about their metamodels and the changes applied during evolution. The total of applied metamodel changes was 330 atomic changes, including 19 complex changes in the three metamodels. Note that these real-world metamodel evolution changes do not cover all the changes in Table 1. However, we did not force any other missing resolutions to be able to compute recall and precision relatively to the real manual co-evolved code, and to minimize the bias in results. Nonetheless, as a sanity check, we did synthetically try all of them during their implementation before the evaluation. We simulated all metamodel changes with all patterns of generated code elements and configuration usages.

For those three case studies, we collect nine Java projects impacted by those three evolving metamodels and their regenerated code API. We collect the **original** and the **developers' evolved** Java code for those projects. Table 5 gives details on the size of the projects and code of the original versions that we co-evolve. In addition, it gives the number of direct and indirect errors after the metamodel evolution. Finally, Table 8's first line gives the total number of tests in the original and the evolved versions of the projects.

4.3 Research Questions

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

RQ1. *Can our automatic co-evolution approach handle the code errors by resolving them after the metamodel evolution?* This aims to assess the ability and applicability of our automatic approach to co-evolve the code due to evolving metamodels.

RQ2. *To what extent does our automatic approach correctly co-evolve the erroneous code?* This aims to assess the usefulness and measure the precision and recall of our approach when compared to the manually applied co-evolution for direct code errors. It further assesses the behavioral correctness of the co-evolution by observing the tests' execution before and after the co-evolution.

RQ3. *How does our automatic approach for code co-evolution compare to the IDE quick fixes as a baseline?* This aims to assess the ability of the IDE quick fixes to correctly co-evolve the code due to evolving metamodels. We measure

Table 4: Details of the metamodels and their evolutions.

Case study	Evolved metamodels	Versions	Atomic changes in the metamodel	Complex changes in the metamodel
OCL	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
Modisco	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
Papyrus	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

Table 5: Details of the projects and their caused direct and indirect errors by the metamodels evolution.

Evolved metamodels	Projects to co-evolve in response to the evolved metamodels	N^o of packages	N^o of classes	N^o of LOC	N^o of Impacted classes	N^o of total direct errors	N^o of total indirect errors
OCL Pivot.ecore	$[P1]$ ocl.examples.pivot	22	439	74002	56	489	37
	$[P2]$ ocl.examples.xtext.base	12	181	17599	10	27	2
Modisco Benchmark.ecore	$[P3]$ modisco.infra.discovery.benchmark	3	28	2333	1	6	0
	$[P4]$ gmt.modisco.java.discoverer.benchmark	8	21	1947	4	30	0
	$[P5]$ modisco.java.discoverer.benchmark	10	28	2794	9	56	0
	$[P6]$ modisco.java.discoverer.benchmark.javaBenchmark	3	16	1654	9	58	15
Papyrus ExtendedTypes.ecore	$[P7]$ papyrus.infra.extendedtypes	8	37	2057	8	59	0
	$[P8]$ papyrus.infra.extendedtypes.emf	7	12	374	7	23	6
	$[P9]$ papyrus.uml.tools.extendedtypes	7	15	725	7	23	6

and compare the precision and recall of our automatic co-evolution approach to the quick fixes.

RQ4. *How does our automatic approach for code co-evolution compare to the state-of-the-art semi-automatic approach as a baseline?* This aims to assess the ability of the fully automatic co-evolution compared to a semi-automatic co-evolution [30].

4.4 Results

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

4.4.1 RQ1

Following the evaluation protocol and after regenerating the code API from the metamodels, we observed 837 errors, among which 771 (92%) direct and 66 (8%) indirect errors. Regarding the 771 direct errors, a total of 631 resolutions were applied. This shows the applicability of our co-evolution approach which was able to handle 100% of direct errors in the code caused by the metamodel evolution changes. Regarding indirect errors, 5 quick fixes all of *add the unimplemented methods* were applied to repair 100% of the 66 errors. Thus, after automating completely the co-evolution of the code, the developers are always able to consult the generated report and check the history of co-evolution.

Moreover, we observed that the number of applied resolutions is less than the initial number of direct errors in the code. In fact, as code co-evolution advances, some resolutions repair multiple other errors as a side effect [56], [57]. In particular, we observed the case of renaming the type of a declared variable, the resolution $[CR5]$ is applied. As a consequence, all the erroneous usages of this variable were automatically corrected. We also observed the main case of a delete resolution $[CR2]$ of an instruction that contained several errors in its body. For example, if an error occurs in the condition statement of an `IF` and a `For` instructions, the whole `IF`, or a `For` block is deleted. As a result, all errors in their bodies automatically disappear. However, even if we would have co-evolved the inner errors first, we would have ended up by deleting the `IF` and `For` instructions to co-evolve its parent error. Therefore, our automatic co-evolution would have reached the same code state.

Furthermore, Table 6 lists the applied resolutions for each co-evolved project. In total, the 631 applied resolutions represented 11 out of the 17 resolutions from our catalog in Table 1. In particular, $32 \times [CR1], 240 \times [CR2], 73 \times [CR4], 179 \times [CR5], 16 \times [CR7], 20 \times [CR10], 18 \times [CR11], 7 \times [CR13], 2 \times [CR14], 3 \times [CR16], 13 \times [CR17]$. Figure 4 further illustrates the application frequency of each resolution. Some resolutions were never applied, such as $[CR3], [CR6], [CR15]$. This can be explained by two reasons. The first one is that the corresponding change did never occur in our case studies (Table 1) like $[CR15]$. The second reason is that pattern matching favors the least deletion when possible. In particular, $[CR1]$ over $[CR3]$, or $[CR4]$ over $[CR2]$.

Finally, the total co-evolution time per project ranged from a few seconds to almost 10 minutes, respectively, in Modisco *[P3]* and OCL Pivot *[P1]*. On average, the co-evolution per error took less than half a second. The evaluation was run on a Fedora Linux 35 laptop with a Core i9 2.6 GHz and 16 GB RAM.

Table 6: Number of applied resolutions in our code co-evolution for each project and per evolved metamodel.

Evolved metamodels	Co-evolved projects	N^o of patterns	N^o of applied resolutions
OCL Pivot.ecore	$[P1]$	381	$[CR1]:12, [CR2]:176, [CR4]:50,$ $[CR5]:110, [CR11]:13, [CR13]:7,$ $[CR14]:2, [CR16]:2, [CR17]:9$
	$[P2]$	25	$[CR1]:1, [CR2]:7, [CR4]:7,$ $[CR5]:5, [CR11]:4, [CR16]:1$
Modisco Benchmark. ecore	$[P3]$	6	$[CR2]:6$
	$[P4]$	22	$[CR1]:5, [CR2]:13,$ $[CR5]:1, [CR7]:3$
	$[P5]$	50	$[CR1]:6, [CR2]:23, [CR5]:6,$ $[CR7]:13, [CR17]:2$
	$[P6]$	62	$[CR1]:8, [CR2]:14, [CR4]:8$ $[CR5]:12, [CR10]:20, [CR17]:2$
Papyrus ExtendedTypes. ecore	$[P7]$	55	$[CR2]:1, [CR4]:8,$ $[CR5]:45, [CR11]:1$
	$[P8]$	15	$[CR5]:15$
	$[P9]$	15	$[CR5]:15$

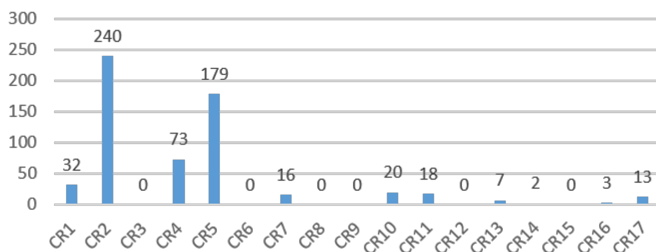

Figure 4: Frequency of applied resolutions overall.

> **RQ_1 insights:**
> We can automatically co-evolve all direct errors in the code after metamodel evolution. Indirect errors are repaired with quick fixes. This allows full automation of the co-evolution with possible manual intervention by checking the generated report.

4.4.2 RQ2

To assess and measure the precision and recall of our automatic co-evolution, we first compared it with the manual co-evolution of the code that developers went through.

Table 7 depicts the reached precision and recall of our automatic co-evolution approach for our nine case studies. We observe that the measured precision and recall varied, respectively from 48% to 100%, and from 51% to 100% reaching an average of, respectively 82% precision and 81% recall. This shows the usefulness of the considered resolutions in Table 1 for the automatic co-evolution of the direct code errors in our case studies.

We further investigated the cause of lowering precision and recall, i.e., the cases where our automatic co-evolution did not match the expected resolutions. The main observation is that several errors that could have been co-evolved by maintaining them were deleted by the developers in the manually evolved version of the code. Rather than to delete the erroneous code, our approach was able to successfully co-evolve and maintain it. For example, in the project *P5*, we observed errors in the code due to moves

Table 7: Measured precision and recall of our projects.

Projects	P1	P2	P3	P4	P5	P6	P7	P8	P9
precision	90%	80%	100%	81%	58%	48%	98%	93%	93%
recall	66%	83%	100%	75%	58%	51%	94%	100%	100%

of the properties *maxUsedMemoryInBytes* and *totalExecutionTimeInSeconds* and its rename to *discoveryTimeInSeconds*. Our approach automatically co-evolved them and maintained the erroneous code, by renaming the setter and extending its path. Whereas, the developers' manual co-evolution consisted of deleting the impacted code. This may hint at a lack of automated co-evolution support that would have easily maintained the code in the new version rather than deleting it. Thus, our co-evolutions here are valid alternatives even if it did not match the correct manual co-evolution.

It is worth noting that the order of the resolutions' application is following the order of the treated metamodel changes. It may affect the number of iterations of Algorithm 2 but it will neither affect the value of recall and precision nor the final result of the co-evolved code.

Finally, to check the behavioral effect of the co-evolution, we observed the tests' execution that we generated for the original and co-evolved code. Table 8 depicts the test execution results. In *[P1, P5, P6, P8, and P9]* similar tests were generated for the original and co-evolved versions. Whereas in *[P2, P3, P7]* there were common tests as well as different generated tests, while we could also not generate tests for *[P4]*. In most cases, we observe that the percentages of passing, failing, and erroneous tests were the same with no significant changes after the co-evolution. Therefore, it suggests that our automatic co-evolution is behaviorally correct by not altering the code behavior of the original projects.

> **RQ_2 insights:**
> Our automatic co-evolution reached an average of an 82% precision and 81% recall. It was able to co-evolve and maintain erroneous code that developers unnecessarily deleted. Generated tests for original and co-evolved code further showed that our co-evolution is not impacting the original code behavior w.r.t. the generated tests.

4.4.3 RQ3

The first baseline we compare to is the quick fixes available in the IDE since they are widely used by developers to repair code errors [61], and more importantly, they have not been compared to code co-evolution approaches before. To compare with the quick fixes as a baseline, we implemented the automatic application of quick fixes as an option in our plugin. It is a variant of our Algorithm 2 that only applies the quick fixes on the errors. The algorithm browses the errors of each compilation unit and applies the first proposed quick fix, since by default Eclipse IDE[10] order the quick fixes by *relevance*. Note that all executions herein terminated normally without any infinite loop.

10. See *org.eclipse.jdt.ui.text.java.CompletionProposalComparator*

Table 8: Observed test before and after co-evolution. [Legend: Before (V1) – After (V2)]

Projects	P1	P2	P3	P5	P6	P7	P8	P9
N^o tests	1987 – 1987	2261 – 2073	475 – 555	67 – 67	427 – 427	142 – 251	105 – 105	75 – 75
N^o pass	826 – 826 (41% – 41%)	1221 – 1161 (54% – 56%)	349 – 417 (73% – 75%)	32 – 32 (47% – 47%)	2 – 2 (0.4% – 0.4%)	18 – 103 (12% – 41%)	16 – 16 (18% – 17%)	14 – 13 (15% – 15%)
N^o fail	14 – 14 (0.7%–0.7%)	36 – 16 (1,6% – 0,8%)	3 – 3 (0.6% – 0,5%)	2 – 2 (2.9% – 2.9%)	0 – 0 (0% – 0%)	3 – 4 (2.1% – 1.5%)	3 – 3 (2.8% – 2.8%)	1 – 1 (1.3% – 1.3%)
N^o error	1147 – 1147 (57%–57%)	1004 – 896 (44% – 43%)	123 – 135 (25.8% – 24.3%)	33 – 33 (49.2% – 49.2%)	425 – 425 (99.5% – 99.5%)	121 – 144 (85.2% – 57.3%)	86 – 86 (81.9% – 81.9%)	60 – 61 (80% – 81.3%)

Table 9: Number of applied Quick Fixes for each project and per evolved metamodel.

Evolved metamodels	Co-evolved projects	% of eliminated errors	N^o of applied Quick Fixes	Total
OCL Pivot.ecore	$[P1]$	82%	[Create Class X]: 3, [Create method m]: 27 , [Change m to m']: 60,[Add cast]: 59, [Change type (of variable or return type of method)]: 17, [Add unimplemented methods]: 43, [Create constants]: 15, [Remove argument]: 1.	225
	$[P2]$	41%	[Create method]: 4,[Change to m']:11, [change type of var or return type]: 1, [Add unimplemented methods]: 2,	18
Modisco Benchmark. ecore	$[P3]$	100%	[Create Class X]: 6, [Add unimplemented methods]: 1	7
	$[P4]$	100%	[Create Class X]: 2,[Create method]: 8, [Change m to m']; 5, [Remove argument]: 1, [Add cast] : 6	22
	$[P5]$	83%	[Create Method]: 17, [Change method m to m']: 16, [Remove argument]: 2, [Change type of var or return type]: 1, [Add Cast]: 16.	52
	$[P6]$	67%	[Change method m to m']: 4, [Add unimplemented methods]: 2, [Create Constant]:9, [change type] : 2, [Add Cast]: 6	23
Papyrus ExtendedTypes. ecore	$[P7]$	69%	[Create Class X]: 3, [Create method m]: 9, [Change method m to m']: 13, [Change type of var or return type]: 5, [Add Cast]: 4, [Add unimplemented methods]: 2.	36
	$[P8]$	93%	[Create Class X]: 1, [Create method] : 14, [Create Const] :3, [Add Cast]: 2, [Change method m to m']:3.	23
	$[P9]$	93%	[Create Class X]: 1, [Create method] : 14, [Create Constant] :3, [Add Cast]: 2, [Change method m to m']:3.	23
Total				429

In Table 9, we present the percentage of errors that quick fixes eliminated for each project, and the frequencies of each type of applied quick fixes.

While the quick fixes eliminated from 41% to 100% errors, we found that the precision and recall of automatic quick fixes are equal to 0, because no correction was applied as expected to the manual developers' resolutions. That is why we refer to the errors are eliminated by the quick fixes and not corrected or co-evolved. For example, concerning errors caused by class or property deletion from the metamodel, renaming a class or an attribute, moving, pushing, or pulling attributes or methods from a class to another, the quick fixes proposed to create them back in their old containers. This is in contradiction of the applied metamodel changes. For errors caused by changing a variable's type, the quick fixes always suggested adding a cast with a wrong type.

Unlike our approach, quick fixes do not take in consideration the context of the impacted code and the information contained in its causing metamodel changes. For example, the quick fix `create the missing method m()` is applied no matter the metamodel change (i.e., deletion, moving, pulling, or pushing changes) or the impacted code location (i.e., statement, variable declaration, parameter, etc.). Our approach takes into account the context of the impacted code and the causing change thanks to the use of pattern matching (see Section 3.5).

For instance, after a move property change, the resolution $[CR7]$ or $[CR8]$ is applied, respectively. Thus, taking into consideration the embedded information in the causing metamodel change, namely the origin class, the target class,

and the multiplicity of the reference between them.

> $\boldsymbol{RQ_3}$ **insights:** The automatic application of quick fixes leads to a faulty evolved code. The same type of quick fixes is applied regardless of the context of the impacted code. With 0 precision and recall, we can conclude the inability of Eclipse quick fixes to co-evolve correctly the code with the evolution of the metamodel.

4.4.4 RQ4

We now compare our approach to the state of the art of metamodel and code co-evolution, namely our previous work [30]. It runs an impact analysis to trace the impacts in the code, while we rely on the compilation errors after regenerating the code API from the evolved metamodel. We also consider an additional change of Pull property and add a new resolution for it ($CR14$). We further distinguish with our previous work by checking behavioral correctness with tests (before/after co-evolution) and by comparing to two baselines. Yet, the main difference is it being semi-automatic requiring intervention to decide between alternative resolutions to co-evolve the code, while we fully automate this decision based on our pattern matching process. Thus, intuitively, it will likely perform better than a fully automatic co-evolution. RQ4 aims to measure this difference.

Table 10 shows precision and recall values of our approach and the semi-automatic co-evolution approach [30]. We notice that the semi-automatic co-evolution approach [30] outperforms our automatic co-evolution approach by a small margin in terms of precision and recall. On average, we had 82% precision and 81% recall in full automatic co-evolution compared to 92% precision and 91% recall in semi-automatic co-evolution. This is due to the fact that our approach favors the least deletion when possible $[CR1]$ over $[CR3]$, or $[CR4]$ over $[CR2]$, while the semi-automatic approach favors integral deletions $[CR3]$ over $[CR1]$, or $[CR2]$ over $[CR4]$. Overall, we observe that the trade-off between full and semi-automatic co-evolution leads to only a reduction of -10% in precision and recall while still maintaining them at a high level ($>$ 81%). This small reduction in performance provides a bigger benefit of speeding up the co-evolution and removing the burden of manual intervention from developers on every code error. Indeed, rather than spending time in guiding the co-evolution at every error, developers can automatically perform it in couples of minutes, in particular, if thousands of errors must be co-evolved after metamodel evolution.

> $\boldsymbol{RQ_4}$ **insights:** Fully automating metamodel and code co-evolution performs less than a semi-automatic co-evolution, but still gives a high precision and recall of ($>$ 81%) while removing the burden of manual intervention from developers on every code error to co-evolve.

4.5 Discussion and limitations

This section discusses the approach and the observed results.

The high-level intuition behind our approach is that we apply a targeted co-evolution by propagating the impacting metamodel changes only to the code errors directly, rather than browsing all code elements statically to fetch the possible impacts. Our automatic co-evolution approach tries to mimic the developers' behavior with known resolutions when co-evolving the code errors in an "iterative way". After the application of each resolution, the list of errors is refreshed to get the list of the new compilation errors for the next iteration.

Moreover, Table 6 shows the distribution of applied resolutions. It is related to the distribution of the impacting metamodel changes (see Table 1) since for a given impacting change, a specific set of resolutions can be applied. However, there is no relation between the distribution of the applied resolution and the recall and precision, which depends more on 'if' the automatic resolutions match "or not" the manual developer's expected resolutions. Moreover, we did not randomly and synthetically choose metamodel changes and their types in the evaluation. Rather, we took the realistic developers' evolutions of the metamodels in our selected real-world software projects. Thus, we did not influence the applied resolutions, which correspond only to the metamodel changes. Of course, having a different distribution of metamodel changes would result in a different distribution of applied resolutions.

Finally, we handled the evolution of one metamodel at once to co-evolve its impacts. Handling multiple metamodels is an interesting case, but is out-of-scope in this paper. Nonetheless, the first favorable scenario we can currently handle is to treat the multiple metamodels in sequence when they are independent of each other. The main limitation is in a second scenario where there are dependencies between the multiple metamodels. For example, a change "set type to T" in a metamodel A referencing a change "add T" in a metamodel B. Herein, the solution would be to find the order of metamodels in which to handle the co-evolution in sequence. This is left as future work.

5 THREATS TO VALIDITY

This section discusses threats to validity [62].

5.1 Internal Validity.

To provoke the code errors, we had to replace the original metamodel evolution and to regenerate the code API. To reduce any bias in the source of errors, we decided to evolve one metamodel at a time. This increases the confidence in the source of the errors, and hence, their co-evolution. Dealing with conflicting errors that can mask each other due to the simultaneous evolution of several metamodels is left for future work. Moreover, to measure the correctness, we analyzed the developers' manual co-evolution. To reduce the risk of misidentifying an expected resolution, for each impacted part of the code, we investigated the entire co-evolved class. If we did not find it, we further searched in other classes in case the original impacted part of the code was moved into another class. Thus, our objective was to reduce the risk of missing any correspondence between an error in the original code and its evolved version. Moreover, as our co-evolution relies on the quality of detected

Table 10: Comparison between our automatic co-evolution approach and the semi-automatic co-evolution approach. [Legend: Precision(P) – Recall(R)]

Projects	P1	P2	P3	P4	P5	P6	P7	P8	P9	Total
Our automatic approach	90%(P) 66%(R)	80%(P) 83%(R)	100%(P) 100%(R)	81%(P) 75%(R)	58%(P) 58%(R)	48%(P) 51%(R)	98%(P) 94%(R)	93%(P) 100%(R)	93%(P) 100%(R)	82%(P) 81%(R)
Semi-automatic approach [30]	92%(P) 92%(R)	93%(P) 86%(R)	100%(P) 100%	100%(P) 100%(R)	100%(P) 100%(R)	48%(P) 48%(R)	100%(P) 100%(R)	100%(P) 100%(R)	100%(P) 100%(R)	92%(P) 91%(R)

metamodel changes, we analyzed 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 pattern matching. Note that the order of changes taken as input does not influence our co-evolution, but the order of errors we treat may affect the distribution of the applied resolutions. However, we took the order in which the errors were detected. Finally, besides the manual checking of the co-evolved code, we used Evosuite as a test suites generation tool that has shown its efficiency in test generation as a state-of-the-art tool [63], [64]. The main reason is that our projects did not have manually written tests. However, automatic test generation can even be more advantageous herein. Indeed, it generates tests for all public methods, whereas developers tend to manually write few tests for only 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. Despite the fact that generated tests can be better for our approach and its results allow us to measure the behavioral correctness of the co-evolution, it still can be combined with manual written tests.

5.2 External Validity.

We implemented and evaluated our approach for EMF and Ecore metamodels and Java code. Although co-evolution could theoretically be applicable for other languages, such as C# or C++, 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 parse the ASTs of the erroneous code and to adapt our resolutions to the new ASTs' structure.

Furthermore, the evaluation was carried out on Eclipse projects in three languages. Thus, we cannot generalize our findings to other software languages and their implementations. However, our approach could be used to co-evolve Java code added on top of a generated code from a domain model, *e.g.,* from a UML class diagram or an entity model. Nonetheless, more evaluations are still necessary.

5.3 Conclusion Validity.

Our evaluation showed promising results with an automatic code co-evolution that is fast and useful, with an average of 82% precision and 81% recall varying from 48% to 100%. The results also showed the usefulness of our approach and the resolutions proposed in our catalog in Table 1. However, even though we evaluated it on nine projects with complex metamodel evolution, we plan to further evaluate on more case studies to have more insights and statistical evidence.

6 Related Work

This section discusses the main related work w.r.t. code co-evolution. We studied this related work and divided it into four (4) main categories : 1) Metamodel and (models/transformations/constraints/ **code**) co-evolution, 2) API and client code co-evolution, 3) Automatic Program repair, and 4) Consistency checking.

6.1 Metamodel and (models/ transformations/ constraints/ code) co-evolution

Many approaches proposed to co-evolve **models** [8], [9], [10], [11], [12], [13], [65], [66], [67], [68], **constraints** [14], [15], [16], [17], [69], [70], **models' transformations** [18], [19], [20], [21], [22], and **code** [25], [26], [27], [28], [29], [30]. Co-evolution of code is distinguished from the co-evolution of other artifacts by the fact that *one* change in a metamodel element will affect n different code elements (see Table 2), in contrast to a *one* to *one* impact relationship between metamodel elements and models, constraints and transformation elements [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22].

Yu et al. [29] proposed to co-evolve the metamodels and the generated API in both directions. However, they do not co-evolve the code on top of it, which our approach does. Khelladi et al. [30] proposed an approach that propagates metamodel changes in the code as a co-evolution mechanism. However, it is based on static analysis to detect the impacts and not on the actual errors that appear from the compilation of the code after the metamodel evolution. It further applies a semi-automatic co-evolution requiring developers' intervention, and without checking behavioral correctness with tests with no comparison to a baseline. Complementary to our work, Kebaili et al. [71], [72] focused on tracing the impacted tests by the metamodel evolution to check among all existing generated or manually written tests. With the emergence of Large Language Models, the problem of metamodel and code co-evolution was investigated in the empirical study of Kebaili et al. [73] using GPT-3.5.

6.2 API and client code co-evolution

Existing approaches for code migration are related to our work. We focus on the main existing approaches to compare them with our approach. Henkel et al. [51] proposed an approach that captures refactoring actions and replays them on the code to migrate. However, they support only the changes renames, moves, and type changes.

Nguyen et al. [74] also proposed an approach that guides developers in adapting code by learning adaptation patterns from previously migrated code. Similarly, Dagenais

et al. [75], [76], [77] also use a recommendation mechanism of code changes by mining them from previously migrated code. Anderson et al. [78] 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. Gerasimou et al. [79] extract a set of mapping rules and apply code-based transformations to update its clients.

Our current work distinguishes from these code migration approaches [51], [74], [75], [78], [79] by considering and reasoning on the changes at the metamodel level to match the different pattern usages of the generated code elements. This is possible thanks to the abstraction offered by the metamodels. Moreover, there are similarities with migration approaches, which is evolving the dependent client code. But these approaches do not handle all the equivalents of the impacting metamodel changes we do (see Table 1), and that occurred in our case studies. They handle only a subset of changes [51], [78]. Other migration approaches [32], [33], [84] rely on pre-collected examples to learn how to evolve the additional client code. Xu et al. [83] instead of learning from code examples, it constructs a database of edits to use during clients' migration. Our approach starts by detecting the changes of the metamodel and then locating the impacted code before co-evolving it without the need to learn from previous examples.

Zhong et al. [82] proposes "LibCatch", a tool to co-evolve client code to APIs evolution by reducing the compilation errors. They do not consider the API changes to correctly propagate them to the code, which may lead to only eliminating the code errors while they could be incorrect resolutions, as shown in our RQ3. They further do not use any mechanism to check the behavioral correctness of the code co-evolution and with not comparison to a ground-truth.

6.3 Automatic Program Repair

In addition to migration approaches, extensive state of the art exists on program repair [85], [86], [87], [88]. However, they do not repair code errors, but rather bugs that are found due to failing tests (e.g., Meng at al. [80], etc.). They could be used as a next step after co-evolution.

Table 11 summarizes closely related work from the metamodel and code co-evolution 6.1, API-client code co-evolution 6.2, and Automatic Program Repair 6.3 categories. Moreover, it compares and highlights the advantages of our approach over those related work. In particular, we compare them with the following criteria:

1) Automation: it indicates whether the approach is automatic, semi-automatic, or manual.
2) "Requiring pre-learning": this feature indicates if a given approach is standalone by immediately co-evolving the code or needs previous external code analysis to learn how to co-evolve client code by synthesising the co-evolution pattern.
3) Changes types: it conveys the changes handled by each approach.
4) Validation: to ensure that the co-evolution did not impact the behavior of the code, a post validation step can be added. This feature indicates if the approach uses any mean of checking behavioral correctness of the code after the co-evolution.

From Table 11, we observe that only three existing approaches are fully automatic and all the rest are semi-automatic. Only four approaches are standalone without requiring a pre-learning phase before the co-evolution. Our approach is fully automatic and standalone. Moreover, several different set of changes are handled by each approach, varying from low AST changes to high level composed (refactoring likes) changes as in Table 1 in our work. Finally, only Fazzini et al. [33] proposed to validate the co-evolved Android Apps with a similar methodology as in our work based on tests' execution.

6.4 Consistency checking

Furthermore, close to code co-evolution, Riedl et al. [23] proposed an approach to detect inconsistencies between UML models and code. Kanakis et al. [24] showed that inconsistency information of model change and code error can help to resolve them in the code, which is equivalent to our matched pattern usages. Pham et al. [25] proposed an approach to synchronize architectural models and code with bidirectional mappings. Jongeling et al. [26] proposed 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. [27] later rely on the recovered traces to perform the consistency checking task. Zaheri et al. [28] also proposed to support the checking of the consistency-breaking updates between models and generated artifacts, including the code. However, [25], [26], [27], [28] do not focus on co-evolving the code to repair the inconsistencies with the models.

To the best of our knowledge, our work is the first attempt to propose a fully automatic co-evolution of the code after its metamodels evolve and to check its behavior with test suites validation step. Thus, removing the burden of manual co-evolution from developers.

7 Conclusion

In this paper, we presented an approach to automatically co-evolve code when metamodels evolve. It relies on a pattern matching algorithm to match the different pattern usages of the metamodel generated code elements in the additional erroneous code. Once a direct error in the code is matched with a pattern usage, we can co-evolve it with its corresponding resolution. For indirect errors that cannot be matched, we apply the available quick fix to repair them. Our code co-evolution was evaluated on nine projects and three metamodels from three Eclipse EMF implementations. After re-generating the code API from the evolved version of the metamodel, hence, causing 837 errors in the additional code. For the 771 direct errors, our automatic co-evolution approach was able to match 631 pattern usages and to apply 631 resolutions. It showed to be efficient and useful in co-evolving all direct errors in the code with an average of 82% precision and 81% recall varying from 48% to 100%. All 66 indirect errors were resolved by calling 5 quick fixes. Moreover, we checked the behavioral

Table 11: Related work comparison

Approaches	Category	Approach	Automation	Requires pre-learning	Change types	Validation
Lamothe et al. [32]			Semi-automatic	Yes ✓	Encapsulate, Move method,Remove parameter, Rename, Consolidate, expose implementation, add contextual data,change type, Replaced by exeternal API	No ×
Fazzini et al. [33]	Android api migration	Identifying migration patterns and rank them to select the most context-similar	Fully-automatic	Yes ✓	Any change in AST level (Insert/Move/Update/Delete)	Yes ✓
Meng et al. [80]	Bug fix context		Semi-automatic	Yes ✓	Any change in AST level (Insert/Move/Update/Delete)	No ×
Wu et al. [81]		Hybrid approach using call dependency graph and textual similarity	Semi-automatic	No ×	change rules : One-to-One, One-to-Many Many-to-One, Simple-Deleted	No ×
Dagenais et al. [75], [76]		Recommendation approach for compilation errors' correction	Semi-automatic	Yes ✓	Deleted or deprecated methods	No ×
Henkel et al. [51]	Java library evolution	Catch refactoring operations during the API revolution the API user can replay these operations later	Semi-automatic	Yes ✓	Refactoring operations: Rename Type, Moving Java Elements, Move static member, Change Method Signature, Rename non-virtual method, Rename non-virtual method, Rename virtual method, change type, rename field, Use super-type where possible, Introduce factory	No ×
N. Guyen et al. [77]		Recommendation approach for API usage adaptation in client	Semi-automatic	Yes ✓	Any change in AST level (Insert/Move/Update/Delete)	No ×
Zhong et al. [82]		Compiler-directed tool for migrating API callsite of client code	Fully-automatic	No ✓	N/a	No ×
Gerasimou et al. [79]	Other library Evolution	Code-based transformation to update client code	Semi-automatic	No ×	A set of mapping rule	No ×
Xu et al. [83]		Mining stored database edits to select applicable edits to be reviewed	Fully-automatic	Yes ✓	Any change in AST level (Insert/Move/Update/Delete) classified into 3 categories : Single statement, Block of statements, MultiBlock of statements	No ×
Khelladi et al. [30]	Model-centric evolution	Impact propagation approach	Semi-automatic	No ×	See Table 1	No ×
Our approach		Code co-evolution guided by Metamodel changes pattern matching	Fully-automatic	No ×	See Table 1	Yes ✓

correctness of our approach by using test suites before and after code co-evolution. We observed that the percentage of passing, failing, and erroneous tests remained stable with insignificant variations in some projects. Thus, suggesting the behavioral correctness of the co-evolution. Finally, we found that the quick fixes are not able to correctly co-evolve the code. This is because they cannot exploit the context of the impacted code and relate it to the changes of the metamodel. We also found that fully automating metamodel and code co-evolution performs less than a semi-automatic co-evolution, but still gives a high precision and recall of ($>81\%$) while removing the burden of manual intervention from developers on every code error to co-evolve.

In summary, our work presents the following contributions : 1/ A fully automatic code co-evolution approach due to Ecore metamodel evolution based on pattern matching. 2/ A prototype implementation of an Eclipse plugin to show the efficiency of our approach. 3/ An evaluation based on four actions: 1) Measuring the co-evolution correctness. 2) Verifying the behavioral correctness using unit tests running before and after automatic code co-evolution. 3) Comparison with the state-of-the art semi-automatic co-evolution approach [30]. 4) Comparison with Quick Fixes popular tool.

As future work, we first plan to explore ordering the errors in the code before co-evolving them. In fact, we co-evolve the errors in the same order they are retrieved by the JDT. Thus, we will investigate whether it could reach better correctness with faster co-evolution or not. Finally, we plan to extend our resolutions with a replace resolution, based on distance metrics to find element replacements to co-evolve the code. After that, we can rely on our approach to extend it with a search-based heuristics, such as genetic algorithms to explore different paths of co-evolution, in contrast to a single path that we compute in this paper. In particular, to explore the different alternative resolutions of $CR1 - CR4$.

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] Y.-T. Chen, C.-Y. Huang, and T.-H. Yang, "Using multi-pattern clustering methods to improve software maintenance quality," *IET Software*, vol. 17, no. 1, pp. 1–22, 2023. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/sfw2.12075

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

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

[4] J.-P. Tolvanen and S. Kelly, "Metaedit+: defining and using integrated domain-specific modeling languages," in *The 24th ACM SIGPLAN conference companion on OOPSLA*, 2009, pp. 819–820.

[5] J. Cabot and M. Gogolla, "Object constraint language (ocl): a definitive guide," in *Formal methods for model-driven engineering*. Springer, 2012, pp. 58–90.

[6] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[7] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer, "Low-code development and model-driven engineering: Two sides of the same coin?" *Software and Systems Modeling*, vol. 21, no. 2, pp. 437–446, 2022.

[8] 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.

[9] 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.

[10] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *2008 12th International IEEE enterprise distributed object computing conference*. IEEE, 2008, pp. 222–231.

[11] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "Cope-automating coupled evolution of metamodels and models." in *ECOOP*, vol. 9. Springer, 2009, pp. 52–76.

[12] 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: 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings 5*. Springer, 2009, pp. 34–49.

[13] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *ECOOP*, vol. 7. Springer, 2007, pp. 600–624.

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

[15] 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," *Journal of Systems and Software*, vol. 134, pp. 242–260, 2017.

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

[17] 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.

[18] 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.

[19] D. E. Khelladi, R. Kretschmer, and A. Egyed, "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*. ACM, 2018, pp. 404–414.

[20] 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.

[21] J. García, O. Diaz, and M. Azanza, "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*. Springer, 2013, pp. 144–163.

[22] 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.

[23] M. Riedl-Ehrenleitner, A. Demuth, and A. Egyed, "Towards model-and-code consistency checking," in *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, 2014, pp. 85–90.

[24] G. Kanakis, D. E. Khelladi, S. Fischer, M. Tröls, and A. Egyed, "An empirical study on the impact of inconsistency feedback during model and code co-changing." *The Journal of Object Technology*, vol. 18, no. 2, pp. 10–1, 2019.

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

[26] R. Jongeling, J. Fredriksson, F. Ciccozzi, A. Cicchetti, and J. Carlson, "Towards consistency checking between a system model and its implementation," in *International Conference on Systems Modelling and Management*. Springer, 2020, pp. 30–39.

[27] R. Jongeling, J. Fredriksson, F. Ciccozzi, J. Carlson, and A. Cicchetti, "Structural consistency between a system model and its implementation: a design science study in industry," in *The European Conference on Modelling Foundations and Applications (ECMFA)*, 2022.

[28] M. Zaheri, M. Famelis, and E. Syriani, "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)*. IEEE, 2021, pp. 400–409.

[29] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux, "Maintaining invariant traceability through bidirectional transformations," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 540–550.

[30] D. E. Khelladi, B. Combemale, M. Acher, O. Barais, and J.-M. Jézéquel, "Co-evolving code with evolving metamodels," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1496–1508. [Online]. Available: https://doi.org/10.1145/3377811.3380324

[31] D. E. Khelladi, B. Combemale, M. Acher, and O. Barais, "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.

[32] M. Lamothe, W. Shang, and T.-H. P. Chen, "A3: Assisting android api migrations using code examples," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 417–431, 2022.

[33] M. Fazzini, Q. Xin, and A. Orso, "Apimigrator: An api-usage migration tool for android apps," in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 77–80. [Online]. Available: https://doi.org/10.1145/3387905.3388608

[34] MDT, "Model development tools. object constraints language (ocl)." http://www.eclipse.org/modeling/mdt/?project=ocl, 2015.

[35] ——, "Model development tools. modisco." http://www.eclipse.org/modeling/mdt/?project=modisco, 2015.

[36] ——, "Model development tools. papyrus." http://www.eclipse.org/modeling/mdt/?project=papyrus, 2015.

[37] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "Modisco: a generic and extensible framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 173–174.

[38] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, "Modisco: A model driven reverse engineering framework," *Information and Software Technology*, vol. 56, no. 8, pp. 1012–1032, 2014.

[39] P. Godefroid, D. Lehmann, and M. Polishchuk, "Differential regression testing for rest apis," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 312–323. [Online]. Available: https://doi.org/10.1145/3395363.3397374

[40] T. Mens, *Introduction and roadmap: History and challenges of software evolution*. Springer, 2008.

[41] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, "An extensive catalog of operators for the coupled evolution of metamodels and models," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6563 LNCS, pp. 163–182, 2011.

[42] S. D. Vermolen, G. Wachsmuth, and E. Visser, "Reconstructing complex metamodel evolution," in *Software Language Engineering*. Springer, 2012, pp. 201–221.

[43] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais, "Detecting complex changes during metamodel evolution," in *CAISE*. Springer, 2015, pp. 263–278.

[44] S. Alter, "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)*, vol. 9097, pp. 520–521, 2015.

[45] J. R. Williams, R. F. Paige, and F. A. Polack, "Searching for model migration strategies," in *Proceedings of the 6th International Workshop on Models and Evolution*. ACM, 2012, pp. 39–44.

[46] A. Cicchetti, D. Di Ruscio, and A. Pierantonio, "Managing dependent changes in coupled evolution," in *Theory and Practice of Model Transformations*. Springer, 2009, pp. 35–51.

[47] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdörfer, M. Seidl, K. Wieland, and G. Kappel, "A posteriori operation detection in

evolving software models," *Journal of Systems and Software*, vol. 86, no. 2, pp. 551–566, 2013.

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

[49] L. Bettini, D. Di Ruscio, L. Iovino, and A. Pierantonio, "An executable metamodel refactoring catalog," *Software and Systems Modeling*, vol. 21, no. 5, pp. 1689–1709, 2022.

[50] D. E. Khelladi, R. Bendraou, and M.-P. Gervais, "Ad-room: a tool for automatic detection of refactorings in object-oriented models," in *ICSE Companion*. ACM, 2016, pp. 617–620.

[51] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE, 2005, pp. 274–283.

[52] L. Iovino, A. Pierantonio, and I. Malavolta, "On the impact significance of metamodel evolution in mde." *Journal of Object Technology*, vol. 11, no. 3, pp. 3–1, 2012.

[53] D. E. Khelladi, H. H. Rodriguez, R. Kretschmer, and A. Egyed, "An exploratory experiment on metamodel-transformation co-evolution," in *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*. IEEE, 2017, pp. 576–581.

[54] R. Hebig, D. E. Khelladi, and R. Bendraou, "Surveying the corpus of model resolution strategies for metamodel evolution," in *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2015, pp. 135–142.

[55] R. Kretschmer, D. E. Khelladi, R. E. Lopez-Herrejon, and A. Egyed, "Consistent change propagation within models," *Software and Systems Modeling*, vol. 20, pp. 539–555, 2021.

[56] J. S. Cuadrado, E. Guerra, and J. de Lara, "Quick fixing atl transformations with speculative analysis," *Software & Systems Modeling*, pp. 1–35, 2018.

[57] D. E. Khelladi, R. Kretschmer, and A. Egyed, "Detecting and exploring side effects when repairing model inconsistencies," in *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, 2019, pp. 113–126.

[58] G. Fraser and A. Arcuri, "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*, 2011, pp. 416–419.

[59] M. Gruber, M. F. Roslan, O. Parry, F. Scharnböck, P. McMinn, and G. Fraser, "Do automatic test generation tools generate flaky tests?" 2023.

[60] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Regression tests to expose change interaction errors," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 334–344.

[61] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis of integrated development environment recommendations," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 669–682. [Online]. Available: https://doi.org/10.1145/2384616.2384665

[62] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[63] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, p. 110398, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219301736

[64] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding strategies in search-based unit test generation," *Software Testing, Verification and Reliability*, vol. 26, no. 5, pp. 366–401, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1601

[65] W. Kessentini and V. Alizadeh, "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*, 2020, pp. 68–78.

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

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

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

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

[70] 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.

[71] Z. K. Kebaili, D. E. Khelladi, M. Acher, and O. Barais, "Towards leveraging tests to identify impacts of metamodel and code co-evolution," in *International Conference on Advanced Information Systems Engineering*. Springer, 2023, pp. 129–137.

[72] ——, "Automated testing of metamodels and code co-evolution," *Software and Systems Modeling*, pp. 1–19, 2024.

[73] ——, "An empirical study on leveraging llms for metamodels and code co-evolution," in *European Conference on Modelling Foundations and Applications (ECMFA 2024)*, vol. 23, no. 3. Journal of Object Technology, 2024, pp. 1–14.

[74] 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.

[75] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, p. 19, 2011.

[76] ——, "Semdiff: Analysis and recommendation support for api evolution," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 599–602.

[77] H. A. Nguyen, T. T. Nguyen, G. Wilson, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," *SIGPLAN Not.*, vol. 45, no. 10, p. 302–321, oct 2010. [Online]. Available: https://doi.org/10.1145/1932682.1869486

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

[79] S. Gerasimou, M. Kechagia, D. Kolovos, R. Paige, and G. Gousios, "On software modernisation due to library obsolescence," in *Proceedings of the 2nd International Workshop on API Usage and Evolution*, ser. WAPI '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 6–9. [Online]. Available: https://doi.org/10.1145/3194793.3194798

[80] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 502–511.

[81] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, 2010, pp. 325–334.

[82] H. Zhong and N. Meng, "Compiler-directed migrating api callsite of client code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639084

[83] S. Xu, Z. Dong, and N. Meng, "Meditor: Inference and application of api migration edits," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 335–346.

[84] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 502–511.

[85] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[86] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 615–627.

[87] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 17, 2018.

[88] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.

8 APPENDIX

Algorithm 1 Pattern matching algorithm (1/2)

```
Data: javaClass, error, changesList
resolution_s ← ϕ
errorNode ← findErrorAstNode(javaClass, error)
configuration ← getConfiguration(javaClass,errornode)
for (change ∈ changesList) do
    switch change do
        case ChangePropertyType do
            if (match(patternGCE,change) then
                switch configuration do
                    case VariableDeclaration do
                        resolution_s.add("CR16")
                    end
                    otherwise do
                        resolution_s.add("CR17")
                    end
                end
        end
        case RenameElement do
            if (match(patternGCE,change) then
                resolution_s.add("CR5")
        end
        case DeleteProperty do
            if (match(patternGCE,change) then
                switch configuration do
                    case ComplexStatement do
                        /*If, loop statements*/
                        resolution_s.add("CR2")
                    end
                    case InvokeSetProperty do
                        resolution_s.add("CR3")
                    end
                    case InvokeGetProperty do
                        resolution_s.add("CR4")
                    end
                    /*Other configurations*/
                end
        end
        case MoveProperty do
            if (match(patternGCE,change) then
                switch configuration do
                    case LiteralStaticField do
                        resolution_s.add("CR10")
                    end
                    case LiteralStaticField do
                        resolution_s.add("CR10")
                    end
                    case MultipleObjectsUpperBound do
                        resolution_s.add("CR8")
                    end
                    case SingleObjectUpperBound do
                        resolution_s.add("CR7")
                    end
                end
        end
```

Algorithm 1 Pattern matching algorithm (2/2)

```
for (change ∈ changesList) do
    ...
    switch change do
        ...
        case DeleteClass do
            if (match(patternGCE,change) then
                switch configuration do
                    case ImportDeclaration do
                        resolution_s.add("CR2")
                    end
                    case ReturnType do
                        resolution_s.add("CR1")
                    end
                    ... /*Other configurations*/
                end
        end
        case GeneralizeMultiplicity do
            if (match(patternGCE,change) then
                resolution_s.add("CR6")
        end
        case PushProperty do
            if (match(patternGCE,change) then
                switch configuration do
                    case LiteralStaticField do
                        resolution_s.add("CR13")
                    end
                    case MethodInvocation do
                        resolution_s.add("CR11")
                    end
                end
        end
        case PullProperty do
            if (match(patternGCE,change) then
                switch configuration do
                    case LiteralStaticField do
                        resolution_s.add("CR14")
                    end
                    ... /*Other configurations*/
                end
        end
        case InlineClass do
            if (match(patternGCE,change) then
                switch configuration do
                    case VariableDeclaration do
                        resolution_s.add("CR15")
                    end
                    .. /*Other configurations*/
                end
        end
    end
    ... /*Other changes*/
end
end
end
return resolution_s
```