Automated co-evolution of metamodels and code

Anonymous Authors

***Abstract*—*Context.* Metamodels are cornerstone in Model Driven Engeneering. They define the different domain concepts and the relations between them. As an abstract artifact, a metamodel plays a significant role in generating other artifacts of lower abstraction level, such as code. The generated code is then enriched by developers to build their language services and tooling, e.g., editors, checkers. *Problem.* As a metamodel evolves, the generated code is automatically updated. As a consequence, the additional developers' code is impacted and requires its errors to be co-evolved accordingly. Moreover, even if an error in the code caused directly by the metamodel evolution is co-evolved, other transitive and indirect errors may appear in the code that need to be part of the co-evolution as well. *Contribution.* This paper proposes a new fully automatic code co-evolution approach of direct and indirect errors due to the metamodel evolution. The approach is based on pattern matching of the direct errors with 1) their cause evolution changes in the metamodel and 2) the pattern usages of the generated code elements from the metamodels. Then, every matched pattern has a corresponding resolution to co-evolve its error. Moreover, indirect errors are nonetheless resolved with the existing quick fixes of the IDE. Finally, we rely on generated test cases before and after co-evolution to check the possible effect of the co-evolution. *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 direct errors due to metamodel evolution with 490 matched patterns and 631 applied resolutions and 66 indirect errors with 5 quick fixes. Our automatic code co-evolution was able to reach an average of 82% of precision and 81% of recall , varying from 48% to 100% for precision and recall respectively. Finally, with the generated tests before and after co-evolution, we observed that the % of passing, failing and erroneous tests stayed the same with insignificant variations in some projects.**

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

I. Introduction

Model-driven engineering (MDE) helps in easing the increasingly complex development and maintenance of large scales systems [1], [2]. Particularly, by supporting the task of building languages and their tooling that play a significant role in all phases of development processes [3]. A central artifact in MDE when building languages is the *metamodel* that defines the aspects of a business domain, i.e. the main concepts, their properties, and relationships between them [4]. 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)* [5] is a prominent example that supports the generation of Java code consisting of a core code API for creating, loading and manipulating the model instances, adapters, serialization facilities, and an editor, all from the metamodel elements. This generated code is further enriched by developers to offer additional functionalities and tooling, such as validation, transformation, simulation, or debuging. A metamodel and its generated code API are, hence, cornerstone when building a language and its tooling. For instance, UML[1] and BPMN[2] Eclipse implementations rely on the UML and BPMN metamodels to first generate their corresponding code API before building around it all their tooling and services.

One of the foremost challenges to deal with in *MDE*, is the impact of the evolution of metamodels on its dependent artifacts. In this paper, we focus on the impacted code. Indeed, when a metamodel evolves and as the core API is re-generated again, the additional code implemented by developers can be impacted. As a consequence, the additional code must be co-evolved accordingly. Moreover, even if an error in the code caused directly by the metamodel evolution is co-evolved, other transitive and indirect errors may appear in the code that need to be part of the co-evolution as well. However, manual co-evolution can be error-prone, tedious 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 [6]–[11], constraints [12]–[15], and transformations [16]–[20]. However, only few works addressed the challenge of metamodels and code co-evolution. In particular, [21]–[26] focused on consistency checking between models and code, but not its co-evolution. Only [27], [28] proposed to co-evolve the code. However, the former does not handle additional code, and the latter support a semi-automatic co-evolution requiring developers intervention. To the best of our knowledge, no existing approach 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. It considers both direct and indirect code errors during co-evolution due to the metamodel evolution. The approach is based on pattern matching of the direct errors with 1) their cause evolution changes in the metamodel and 2) the pattern usages of the generated code elements from the metamodels. Then, every matched pattern has a corresponding resolution to co-evolve its error with knowledge present in the metamodel evolution change. For example, a move property *p* in the metamodel from one class to another through a reference *ref*, will be matched with the code usage of its getter *var.getP()*. Its associated resolution will extend its navigation path with *ref* to *var.getRef().getP()*. Moreover, indirect errors that cannot

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

[2]https://www.eclipse.org/bpmn2-modeler/

be matched are nonetheless resolved with the existing quick fixes in the IDE. Finally, to check whether our automatic co-evolution impact the original code behavior, we rely on generated test cases before and after co-evolution to check the possible effect of the co-evolution. We further generate a report of the applied resolutions for each error to help developers in understanding the co-evolution.

We evaluate our approach on three Eclipse EMF implementations, namely OCL [29], Modisco [30], and Papyrus [31], which have been developed for more than 10 years and have been evolved several times. We collected projects consisting of both original and evolved versions of metamodels and code. Thus, we evaluated to what extent can our approach correctly co-evolve the projects' code in response to the metamodels evolutions. Within three metamodels, the 330 evolution changes caused 837 errors in 9 projects, consisting in 771 direct and 66 indirect errors. Our automatic co-evolution approach were able to match 631 pattern usages and applied 631 resolutions for the direct errors. 63 out of the 66 indirect errors were resolved with 5 quick fixes. When comparing our automatic co-evolution to the expected one in the evolved versions of the projects, we could observe the co-evolution precision and recall varying from 48% to 100%, reaching an average of respectively 82% precision and 81% recall. Finally, with the generated tests before and after co-evolution, we observed that the % of passing, failing and erroneous tests stayed the same with insignificant variations in some projects.

This paper is structured as follows. Section II introduces a motivating example. Section III presents our overall automatic co-evolution approach. Section IV details our evaluation results and threats to validity. Section VI discusses related work and how our work distinguishes from state of the art. Finally, Section VII concludes the paper and reflects on future works.

II. 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 [30], which has evolved numerous times in the past. Modisco is an academic initiative project implemented in the Eclipse platform to support development of model-driven tools, reverse engineering, verification, and transformation of existing software systems [32], [33].

Figure 1 shows an excerpt of the "Modisco Discovery Benchmark" metamodel[3] consisting of 10 classes in the version of 0.9.0. It illustrates some of the domain concepts **Discovery**, **Project** and **ProjectDiscovery** used for the discovery and reverse engineering of an existing software system. From these metaclasses, a first code API is generated, containing Java interfaces and their implementation classes, a factory, a package, etc. 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

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

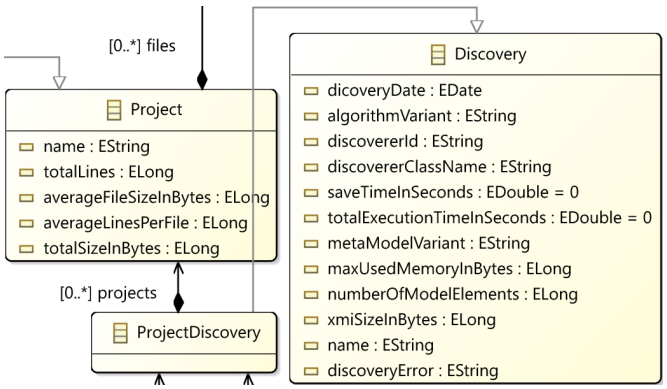

Fig. 1. Excerpt of Modisco Benchmark metamodel in version 0.9.0.

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 an 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 re-generated from the evolved version of the metamodel, which in turn impacts the existing additional code depicted in Listings 2. The resulted 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 in Line 1 in Listing 2 and any usage of it and its methods are impacted. The import 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 maintaining 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` to `DISCOVERY_ITERATION`.

The above examples show the importance of correctly matching the different code usages of the generated code 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
EXCERPT 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   }
```

III. APPROACH

This section presents the overall approach of our automated co-evolution of code with evolving metamodels. It first gives an overview of the approach and specifies the metamodel evolution changes we consider. Then, it presents how we retrieve the caused errors due to metamodel evolution and the regeneration of the code API. After that, it presents the pattern matching process, which is an important part of our automatic co-evolution approach, before to discuss the resolutions of the code errors.

A. Overview

Figure 2 depicts the overall steps for the automatic co-evolution of the metamodel and code. As the evolution of metamodel will cause errors in the additional Java code that depends on the newly generated code API, we take as input the metamodel evolution changes between the two versions of this metamodel [1]. Then, we parse the additional code [2] to retrieve the list of errors. After that, both the list of metamodel changes and the list of errors are used as inputs for the pattern matching step [3]. It analyses the structure of the error to match it with its impacting metamodel change and decides which resolution to apply for the error co-evolution [4]. The metamodel changes provide the ingredients and necessary information that are used for the co-evolution. Finally, we obtain a new co-evolved additional code [5]. In addition, we generate test cases before and after co-evolution to highlight the possible effect of the co-evolution and we generate a report of the applied resolutions for each error to help developers in understanding the co-evolution.

B. Metamodel evolution changes

One of the intrinsic property of software artifacts is its continuous evolution [34]. Metamodels are no different and are meant to evolve. Two types of evolution changes are considered when evolving a metamodel: *atomic* and *complex* changes [35]. Atomic changes are additions, removals, and updates of a metamodel element. Complex changes consist of a sequence of atomic changes combined together [36], [37]. For example, move property is a complex change where a property is moved from a source class to a target class. This is composed of two atomic changes: delete a property and add a property [35]. Many approaches in the literature [36], [38]–[42] exist to detect metamodel changes between two versions. In this work, we use an interface specification of changes [1] that is a connection layer to our co-evolution approach with the existing change detection approaches. Therefore, in practice, any detection approach [36], [38]–[42] can be integrated by bridging its changes to our interface and the rest of co-evolution can be performed independently. In the rest of this work, we focus on code co-evolution. We suppose that we have the metamodel changes and their information as input using the changes detection interface.

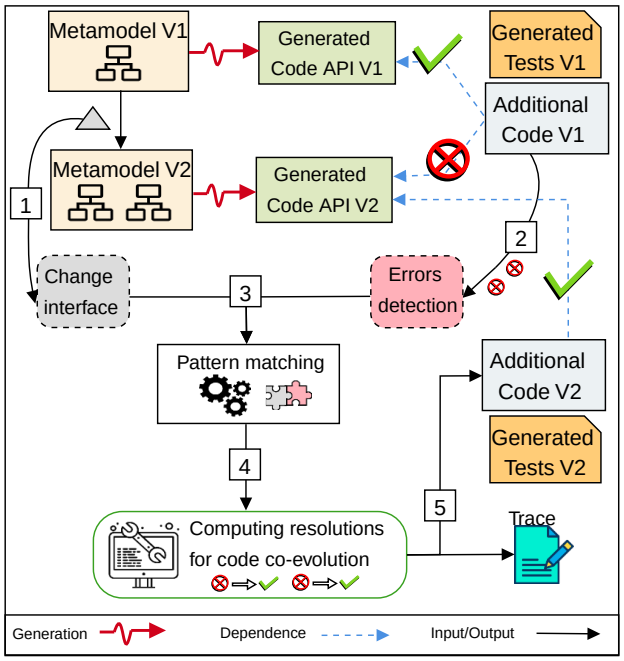

Fig. 2. Overall approach for metamodel and code co-evolution

C. Error retrieving

Our approach does not rely on static impact analysis to detect the impacts of metamodel changes, *i.e.,* by tracing the metamodel elements to the code. Instead, it relies on the compilation result of the code as a more of a dynamic impact analysis to detect the code errors that must be co-evolved. This is necessary and useful in our approach as we will need to keep updating the list of errors after co-evolving each given error. We detail this process in the next 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 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 of the paper, we refer to errors and Java classes for sake of simplicity.

D. Pattern matching

This section introduces the pattern matching between metamodel changes and code usages for the detected errors to prepare their co-evolution. Each metamodel element ME has corresponding generated code elements $\{GCE_0, GCE_1, ..., GCE_n\}$. Thus, evolving ME will impact the usages of $\{GCE_0, GCE_1, ..., GCE_n\}$ in the additional code. Table I classifies the different generated code elements for each metamodel element type and provides illustrative examples. It shows that various code elements are generated for each metamodel element type and with different patterns. Let us take 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 an attribute, EMF generates the signature and implementation of a getter and a setter, an accessor and a literal. This classification is essential to match the different errors with their corresponding pattern usages in the code, before to co-evolve them with the appropriate resolution strategy. Moreover, each generated code element can be used in different configurations in the code that should also be considered in the pattern matching step. For example, using a GCE_i as a parameter for a method declaration and method invocation, or to initialize a variable declaration, etc.

Algorithm 1 summarizes the pattern matching step. Given a Java class, an error, and a list of metamodel changes, Algorithm 1 will return the matched patterns for the erroneous code usage of an evolved metamodel element. Algorithm 1 first retrieves the error AST node $\boxed{line2}$ to identify the different cases of usages in the additional code $\boxed{lines3, 20}$, *e.g., SimpleName*, *QualifiedName*, *SingleVariable*, *MethodDeclaration*, etc. Then, it matches it with the metamodel change causing its error $\boxed{lines6, 12, 15}$ before to cover all different pattern usages of the generated code elements that are presented in Table I. For example, $\boxed{lines7 - 8}$ matches the usage pattern of the getter and setter with a moved property, while $\boxed{lines9 - 10}$ matches the usage pattern of the literal with a moved property. This process is applied for the rest of the metamodel changes. Finally, the code usage, either matched with the appropriate patterns or not, is returned for further processing in the automatic co-evolution $\boxed{line22}$.

Note that several patterns can be matched for a given error. For example, in Listing 2, Algorithm 1 allows to match the error in $\boxed{line6}$ with two patterns caused by the two metamodel changes rename and move of the property *totalExecutionTimeInSeconds*. Both with the code pattern usages of a literal, *i.e.,* `BenchmarkPackage.DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS`. The matched patterns are `RenameProperty_Literal` and `MoveProperty_Literal`. In total, we define 33 patterns that we match with the code usages and metamodel changes. Due to lack of space we attach their list as a supplementary material.

E. 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. They are matched with a pattern and a metamodel evolution change. The latter are errors that do not use a generated code element. They are not matched, and hence, cannot be co-evolved with our approach. However, we still attempt to repair them with the available quick fixes in the IDE. For example, the error *unimplemented method m() of a class C* is indirect if not matched with a metamodel change and is repaired with its quick fix *add the unimplemented method*.

For the direct errors, our co-evolution approach relies on the resolutions shown in Table II. It depicts the resolutions associated for metamodel changes that are known to be

TABLE I
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()*

Algorithm 1: Pattern matching algorithm

```
   Data: javaClass, error, changesList
1  usage ← ϕ
2  errorNode ← findErrorAstNode(javaClass, error)
3  if (errornode.isTypeOf( SimpleName)) then
4      for (change ∈ changesList) do
5          switch change do
6              case MoveProperty do
7                  if (errornode.name="get"+change.name V
                     errornode.name="set"+change.name) then
8                      usage.getPatterns().
                         add(UsagePattern.MoveProperty_GetSet)
9                  else if (errornode.name= change.className
                     +"__" +change.name) then
10                     usage.getPatterns().
                         add(UsagePattern.MoveProperty_Literal)
11             end
12             case RenameProperty do
13                 ...
14             end
15             case DeleteProperty do
16                 ...
17             end
18         end
19     end
20 else if ( errornode.isTypeOf( QualifiedName)) then
21     /*repeat same matching process*/ ...
22 return usage
```

impacting [43]. The resolutions are taken from existing co-evolution approaches of various MDE artifacts [6]–[20], [28], [44], where they showed to be efficient and useful.

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

The selection of the resolution is guaranteed by the pattern detection. For example, for the Move property change, the selection $[CR7]$ is a consequence of the detection of MoveProperty_UpperBoundSingle pattern if the property is move to another class through a reference whose upperbound value equals 1. the selection $[CR8]$ is a consequence of the detection of MoveProperty_UpperBoundMultiple pattern if the property is moved to another class through a reference whose upperbound value equals -1.

Therefore, applying a resolution would only require as input the code AST and the matched usage pattern to produce the co-evolved AST as output. Note that if new errors are introduced after applying a resolution, they will be handled in the next iterations similarly with our pattern matching and co-evolution algorithms.

Algorithm 2 presents the general process of the code co-evolution. It starts by parsing the project Java classes $\boxed{line1}$ to be able to access the AST of the code. Then it browses the parsed classes to retrieve the list of errors $\boxed{line3}$. The next step is to run the pattern matching Algorithm 1 to return the matched pattern usages, if any.

If a usage is matched with at least one pattern $\boxed{line6}$, it is a direct error due to the metamodel evolution. If an error is matched with several patterns $\boxed{lines8}$, it will be resolved with resolutions corresponding to each pattern iteratively. The Algorithm 2 will then select the corresponding resolution for each pattern to correct the error $\boxed{lines9}$. In the case of an indirect error that is not matched with a pattern usage, we analyze the error message to match it with one of the proposed Java quick fixes $\boxed{lines12-13}$. For example, for an unimplemented methods error, we call its quick fix to add them. After applying the resolutions or a quick fix, the Java

Algorithm 2: Co-evolution of metamodel and code

```
   Data: EcoreModelingProject, changesList
1  javaClasses ← Parse(EcoreModelingProject)
2  for ¡ jc ∈ javaClasses¿ do
3      errorsList ← getErrors(jc)
4      while (!errorsList.isEmpty()) do
5          error ← errorsList.next()
6          usage ← matchUsagePattern(jc, error, changesList)
7          if (! usage.getPatterns().isEmpty()) /*direct errors*/
             then
8              for (pattern ∈ usage.getPatterns()) do
9                  resolution ← selectResolution(pattern)
10                 applyResolution(jc, usage.getErrorNode(),
                     resolution)
11             end
12         else if error.hasQuickickFix() /*indirect errors*/ then
13             useQuickFixes(error)
14         refreshJavaClass(jc)
15         refreshErrorsList(jc, errorsList)
16     end
17 end
```

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 $\boxed{lines14-15}$.

Finally, this process is repeated until all errors are corrected. Otherwise, only errors that are not matched with pattern usages and can't be resolved with quick fixes remain.

F. 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 JDT eclipse plugin[4]. The errors' AST nodes are manipulated with edit actions using the package *org.eclipse.jdt.core.dom.rewrite* and *org.eclipse.core.filebuffers*. Moreover, the quick fixes for the indirect errors are called using *org.eclipse.jdt.ui.text.java.IQuickAssistProcessor* and *org.eclipse.jdt.ui.text.java.IJavaCompletionProposal*.

IV. 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.

A. Evaluation Process

We evaluate our automated co-evolution of code by measuring its ability to repair the errors, its time performance and its correctness by using recall and precision metrics.

As our approach co-evolve the erroneous code due to metamodel evolution, we first need to provoke the errors in the code. To do so, we first replace the original metamodel by the evolved metamodel. Then, we regenerate the code API with EMF. This will cause the errors in the additional code that our approach must co-evolve.

Herein, we use the function $System.nanoTime()$ for time measurement. We then measure the correctness of our code co-evolution by comparing for the same set of direct 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 the resolutions applied by our approach (from Table II) and the $ExpectedResolutions$ are the actual manually performed resolutions by developers.

Moreover, to check if the co-evolution impact the original code behavior, we generate tests for the original and co-evolved versions. Hence, we observe the behavioral effect of our co-evolution through the tests. To do so, we rely on Evosuite [45], a popular tool for test cases generation, as it is widely used by research and by developers.

Finally, note that as the metamodel changes from version *1* to version *2* are taken as input of our automatic code co-evolution, we studied the original and evolved versions to confirm the metamodel changes.

B. Data Set

This section presents the used data set in our evaluation to be found in the attached supplementary material. We evaluate on three case studies from three different language implementations in Eclipse, namely OCL [29], Modisco [30], and Papyrus [31]. OCL is a standard language defined by the Object Management Group (OMG) to specify First-order logic constraints. Modisco is an academic initiative to support development of model-driven tools, reverse engineering, verification, and transformation of existing software systems. Papyrus is an industrial project led by CEA[5] 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.

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 caused errors in the code due to only metamodel changes. Thus, mitigating the bias related to the ambiguous cases where errors interact and mask each other [46]. Handling the scenario code co-evolution due to multiple metamodels evolution is left for future work. Moreover, we also aimed at selecting meaningful evolutions that do not consist in only deleting metamodel elements, but rather including complex evolution changes. Table III gives details about the selected case studies, in particular about their metamodels and the applied changes during evolution. The total of applied metamodel changes was

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

[5]http://www-list.cea.fr/en/

TABLE II
CATALOG OF RESOLUTIONS USED FOR THE CODE CO-EVOLUTION OF DIRECT ERRORS DUE TO THE METAMODEL CHANGES.

Impacting Metamodel Changes	Proposed Code Resolutions
◇ Delete property *p*	▷$[CR1]$ 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`	▷$[CR1]$ 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*	▷$[CR5]$ Rename *e* in the code
◇ Generalize multiplicity of property *p* from a single to multiple values	▷$[CR6]$ 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*	▷$[CR7]$ 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 $\texttt{Sub}_1$,...,$\texttt{Sub}_n$	▷$[CR11]$ 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 $\texttt{Sub}_1$,...,$\texttt{Sub}_n$ to `Sup`	▷$[CR14]$ Replace Sub_i by Sup in Literal values (e.g., *MetamodelPackage.*Sub_i*__P* → *MetamodelPackage.Sup__P*)
◇ Inline class `S` to `T` with properties $p_1, ..., p_n$	▷$[CR9]$ 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 from `S` to `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*

330 atomic changes, including 19 complex changes in the three metamodels. For those three case studies, we collected 9 Java projects that were impacted by those three evolving metamodels and their re-generated code API. We collected the original and evolved Java code of those projects. Table IV gives details on the size of the projects and code of the original versions that we co-evolve in addition to the number of direct and indirect errors after the metamodel evolution.

C. 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 after the metamodel evolution?* This aims to assess the ability and applicability of our automatic approach to co-evolve the errors in the code due to evolving metamodels.

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

D. Results

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

1) RQ1: Following the evaluation protocol and after re-generating the code API from the metamodels, we observed 837 errors, among which 771 direct and 66 indirect errors. Regarding the 771 direct errors, a total of 631 patterns were matched and 631 resolutions were applied. This shows the applicability of our co-evolution approach that was able to handle all different 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 the 66 errors, leaving 4 errors that were not matched and with no quick fix. one occurred in Modisco JavaBenchmark Project *[P6]* and 3 in Pivot project *[P1]*. Thus, reducing significantly the burden of manual intervention from developers required only in 2 out 9 projects while covering all co-evolution cases.

Moreover, we observed that the number of detected patterns and the applied resolutions are less than the initial number of direct errors in the code. In fact, as the code co-evolution advances, some resolutions do repair multiple other errors as side effect [47], [48]. In particular,we observed the case a renaming resolution $[CR5]$ of a declared variable, that after being renamed, the errors in its usages were corrected automatically. We observed also the main case of a delete resolution $[CR2]$ of an instruction that contained several errors in its body. For example, an error in the conditions of an `IF` and a `For` instructions that were co-evolved by deleting the whole instruction that contained errors in their bodies. However, even if we would have co-evolved the inner errors first, we would have ended up by deleting the instructions to co-evolve its parent error. Therefore, our automatic co-evolution would have reached the same code state. However, note that the pattern matching favors the least deletion when

TABLE III
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 properies Adds: 8 classes, 9 properties, 8 super types	2 pull property 1 push property 1 extract super class

TABLE IV
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

TABLE V
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	$[CR5]:15$

possible. In particular, $[CR1]$, $[CR3]$, or $[CR4]$ over $[CR2]$.

Furthermore, Table V 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 II, namely $32*[CR1], 240*[CR2], 73*[CR4], 179*[CR5], 16*[CR7], 20*[CR10]$, $18x[CR11], 7*[CR13], 2*[CR14], 3*[CR16], 13*[CR17]$. Finally, the total co-evolution time per project varied from 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 2.6GHz and 16GB RAM.

> **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 reduces significantly the burden of manual intervention from developers required only in 2 out 9 projects.

2) RQ2: To assess and measure precision and recall of our automatic co-evolution, we first compared it with the the manual co-evolution of the code that developers went through.

Table VII 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 II for the automatic co-evolution of the direct code errors in our case studies.

We further investigated the cause of lowering correctness,

TABLE VI
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%

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 evolved version of the code. Rather than to delete the erroneous code, our approach was able to successfully co-evolve and maintain them. For example, in the project P5, we observed errors in the code due to moves of the properties *maxUsedMemoryInBytes* and *totalExecutionTimeInSeconds* and its rename to *discoveryTimeInSeconds*. Our approach automatically co-evolved them and maintain them, by renaming the setter and extending its path. Whereas, the developers' manual co-evolution consisted in deleting them. This may hint on a lack of an automated co-evolution support that would have easily maintained the code in the new version rather than deleting it.

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. In total, respectively for the original and co-evolved versions. Table VII 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 also could not generate tests for [P4]. In most cases we observe that the percentage 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.

V. THREATS TO VALIDITY

This section discusses threats to validity [49].

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 confidence of the source of the errors, and hence, their co-evolution. Dealing with conflicting errors that can mask each other due to simultaneous several metamodel evolution 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. In case 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, we aimed at reducing 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 metamodel changes. We also 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. However, we took the order in which the errors were detected. Besides the manual checking of the co-evolved code, we used Evosuite as a test suites generation tool. Despite the fact that its results allow us to measure the behavioral correctness of the co-evolution, it still not sufficient as it can not replace the manual written tests.

External Validity. We implemented and evaluated our approach for EMF/Ecore metamodels and Java code. Other object-oriented languages use different syntax but conceptually use the same constructions as in Java, such as C# or C++. Although the co-evolution could in theory be applicable for other languages, we cannot generalize our results. Further experimentation on other languages is necessary. However, the only requirement to apply our approach to other languages is to parse the ASTs of the erroneous code and to adapt our resolutions to the new ASTs' structure.

Furthermore, the evaluation was on Eclipse projects from 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, further evaluations remain necessary.

Conclusion Validity. Our evaluation showed promising results with an automatic code co-evolution that is fast and useful, with an average correctness of 81.2%. Results also showed the usefulness of our approach and the proposed resolutions in our catalog in Table II. Nonetheless, even though we evaluated it on 9 projects with complex metamodel evolution, we plan to evaluate further on more case studies to have more insights and statistical evidence.

VI. RELATED WORK

This section discusses the main related work w.r.t. code co-evolution. Many approaches proposed to co-evolve models [6]–[11], [50]–[53], constraints [12]–[15], [54], [55], model transformations [16]–[20], and code [23]–[28]. Co-evolution of code distinguishes with co-evolution of other artifacts by the fact that *one* change in a metamodel element will impact n different code elements (see Table I), in contrast to a *one*

TABLE VII
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%)

to *one* impact relationship between metamodel elements and models, constraints, and transformations elements [6]–[20].

Existing approaches of code migration are related to our work. We focus on the main existing approaches. Henkel et al. [56] proposed an approach that captures refactoring actions and replay them on the code to migrate. However, they support only the changes renames, moves, and type changes.

Nguyen et al. [57] also proposed an approach that guides developers in adapting code by learning adaptation patterns from previously migrated code. Similarly, Dagenais et al. [58] also uses a recommendation mechanism of code changes by mining them from previously migrated code. Anderson et al. [59] 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.

Our current work distinguishes from these code migration approaches [56]–[59] 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. In addition to migration approaches, extensive state of the art exists on program repair [60]–[63]. However, they do not repair code errors but rather bugs that are found due to failing tests. They could be used as a next step after co-evolution.

Furthermore, closely to code co-evolution, Riedl et al. [21] propose an approach to detect inconsistencies between UML models and code. Kanakis et al. [22] showed that inconsistency information of model change and code error can help in resolving them in the code, which is equivalent to our matched pattern usages. Pham et al. [23] propose an approach to synchronize architectural models and code with bidirectional mappings. Jongeling et al. [24] propose an early approach for the consistency checking between system models and their implementations by focusing on recovering the traceability links between the models and the code. Jongeling et al. [25] later rely on the recovered traces to perform the consistency checking task. Zaheri et al. [26] also propose to support the checking of the consistency-breaking updates between models and generated artifacts, including the code. However, [23]–[26] do not focus co-evolving the code to repair the inconsistencies with the models. Yu et al. [27] proposes to co-evolve the metamodels and the generated API in both directions. However, they do not co-evolve the additional code on top of it, which our approach does. Khelladi et al. [28] propose an approach that propagate the metamodel changes to 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. 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. Thus, removing the burden of manual co-evolution from developers.

VII. CONCLUSION

In the 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 could not be matched, we apply the available quick fix to repair it. Our code co-evolution was evaluated on 9 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 an average of 82% precision and 81% recall varying from 48% to 100%. For the 66 indirect errors, 63 were resolved by calling 5 quick fixes.

As future work, we first plan to explore ordering the errors in the code before to co-evolve them. In fact, we co-evolve the error in the same order they are retrieved by the JDT. Thus, will investigate whether it could reach to a better correctness with faster co-evolution or not Finally, we plan to extend our resolutions with a replace resolution that will be 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 genetic algorithm to explore different paths of co-evolution, in contrast to a single path that we compute in this paper.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

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

[6] 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.

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] D. E. Khelladi, B. Combemale, M. Acher, O. Barais, D. E. Khelladi, B. Combemale, M. Acher, O. Barais, J.-m. J. Co, D. E. Khelladi, B. Combemale, and O. Barais, "Co-Evolving Code with Evolving Metamodels To cite this version : HAL Id : hal-03029429 Co-Evolving Code with Evolving Metamodels," 2020.

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

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

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

[32] 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.

[33] 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.

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

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

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

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

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

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

[61] 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.

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

[63] 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.