Co-Evolving Code with Evolving Metamodels

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

Benoit Combemale
Université Toulouse & Inria Rennes
Rennes, France
benoit.combemale@irisa.fr

Mathieu Acher
Univ. Rennes, Inria, IRISA
Rennes, France
mathieu.acher@irisa.fr

Olivier Barais
Univ. Rennes, Inria, IRISA
Rennes, France
olivier.barais@irisa.fr

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

ABSTRACT

Metamodels play a significant role to describe and analyze the relations between domain concepts. They are also cornerstone to build a software language (SL) for a domain and its associated tooling. Metamodel definition generally drives code generation of a core API. The latter is further enriched by developers with additional code implementing advanced functionalities, e.g., checkers, recommenders, etc. When a SL is evolved to the next version, the metamodels are evolved as well before to re-generate the core API code. As a result, the developers added code both in the core API and the SL toolings may be impacted and thus may need to be co-evolved accordingly. Many approaches support the co-evolution of various artifacts when metamodels evolve. However, not the co-evolution of code. This paper fills this gap. We propose a semi-automatic co-evolution approach based on change propagation. The premise is that knowledge of the metamodel evolution changes can be propagated by means of resolutions to drive the code co-evolution. Our approach leverages on the abstraction level of metamodels where a given metamodel element has often different usages in the code. It supports alternative co-evaluations to meet different developers needs. Our work is evaluated on three Eclipse SL implementations, namely OCL, Modisco, and Papyrus over several evolved versions of metamodels and code. In response to five different evolved metamodels, we co-evolved 976 impacts over 18 projects.A comparison of our co-evolved code with the versioned ones shows the usefulness of our approach. Our approach was able to reach a weighted average of 87.4% and 88.9% respectively of precision and recall while supporting useful alternative co-evolution that developers have manually performed.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380324

ACM Reference Format:
Djamel Eddine Khelladi, Benoit Combemale, Mathieu Acher, Olivier Barais, and Jean-Marc Jézéquel. 2020. Co-Evolving Code with Evolving Metamodels. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380324

1 INTRODUCTION

Model-Driven Engineering (MDE) has proven to be effective in the development and maintenance of large scales systems [20, 21]. Notably, by easing the task of building *Software languages (SLs)* and their tooling that can play a significant role in all phases of development processes [46].

When comes the time to specify a SL, the metamodel is pivotal and a central artifact [21]. It introduces the domain concepts, which are later used as a cornerstone to build the expected tooling (*e.g.*, editor, checker, compiler, data access layers, etc.). Regular Expressions, CSS, GraphViz are examples of SLs that use an explicit initial specification to later drive the development of the corresponding tooling [10]. However, it is often overlooked that metamodels are inputs for complex code generators that leverage on the abstractions used in metamodels (*e.g.*, bidirectional references, containments, …) to generate complex and industrial code that may include advanced architecture and different ways to manipulate instances.

As a prominent example, *Eclipse Modeling Framework (EMF)* [45] supports the generation of Java code consisting of a core API (both generic and domain-specific), adapters, serialization facilities, and an editor, all from metamodels. This code can be used to load and manipulate the model instances. More importantly, it is often enriched by developers to offer additional functionalities and on top of which the SLs' toolings are further developed.

Today, many Eclipse projects[1] leverage on the code generated by EMF to support developers in various tasks (e.g., developing advanced editors, specific model transformations, etc.). As a result, many languages have been developed using EMF and its ecosystem, such as BPMN [41] or UML [43]. In all these examples, the initial core API generated with EMF have been complemented, either manually or through other

[1] Cf. https://marketplace.eclipse.org

tools from the EMF ecosystem, to provide advanced functionalities and tooling such as simulation, debuging, etc. A metamodel and its corresponding implementation are hence a corner stone when building a SL and its tooling.

One of the foremost challenges to deal with in *MDE* is the evolution of metamodels and its impact on artifacts that use metamodels as a foundation. As a result of a metamodel evolution, the core API is re-generated and may impact the additional code implemented by developers. Hence, it may need to be co-evolved accordingly. Knowing that other projects/SLs can depend on the impacted code, it is essential to efficiently co-evolve it whenever the metamodel evolves. In the last decade, the co-evolution challenge has been extensively addressed in *MDE*. However, the literature so far has focused on the co-evolution between metamodel and models [5, 11, 18, 24, 25, 49], constraints [2, 6, 27, 33], and transformations [12, 13, 23, 32, 34]. To the best of our knowledge, there exist no work that aims at supporting the co-evolution between metamodels and code.

This paper contributes in filling this gap. We propose a semi-automatic approach to co-evolution of code when the metamodels evolve. The co-evolution propagates the metamodel changes to the code. Change propagation showed to be efficient in many domains, such as in the context of artifact recommendation or co-evolution of models or transformations [7, 9, 32]. This paper leverages on the knowledge provided by the metamodel changes during evolution which are propagated by means of code co-evolution. For example, knowing that a property p in the metamodel is moved from one class to another through a reference *ref*, it is likely that the navigation path to p will also be updated in the code by either extending it with *ref* or reducing it. The current approach first runs an impact analysis to identify all impacts at the code level.

Our approach leverages on the abstraction level of metamodels where a given metamodel element has often a complex implementation in the code with various usages of such a code. For example, for a single metamodel class, an interface and a class are generated in the code, but also a method creation in the factory class, etc. Our impact analysis is designed to identify all impacted usages of a given changed metamodel element. Then, we propose a catalog of resolutions that is used as a basis for our code co-evolution in response to metamodel evolution. Developers can select among our supported alternative resolutions.

Our approach with the proposed resolutions are evaluated on three Eclipse EMF-based SL implementations, namely OCL [37], Modisco [36], and Papyrus [38], 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 our case studies, 477 changes were considered on five metamodels, which caused 976 impacts on 18 projects. When comparing our performed co-evolution to the expected co-evolution from the original to the evolved versions of the projects, our approach showed to support useful alternative co-evolutions that developers have manually performed. Our approach reached a weighted average (by the number of metamodels changes) of 87.4% and 88.9% respectively of precision and recall of co-evolution. Our approach also showed to be useful by maintaining imapacted code while it was unnecessary deleted.

This paper is structured as follows. Section 2 introduces a background and a motivating example. Section 3 presents our overall approach. Section 4 details our evaluation results Section 5 threats to validity. Section 6 discusses related work and how our work distinguishes from code migration approaches. Finally, Section 7 concludes the paper and reflects on future works.

2 BACKGROUND AND MOTIVATING EXAMPLE

This section first gives a background on the role of metamodels when building SLs and their toolings and discusses the scenarios of co-evolution that arises between metamodels and code. After that, it introduces an example taken from the OCL case study in Eclipse of metamodel and code co-evolution to motivate our work.

2.1 Background

Figure 1 depicts an SL structure and usage as in practice, for instance, as in the Eclipse platform. Metamodels are cornerstones when building a SL and its toolings [14, 47]. Metamodels define the aspects of a business domain, i.e. the main concepts, their properties, and relationships between them [3].

Once the metamodels are carefully specified, working piece of code is generated consisting of a core API [45], which is often enriched with additional code to offer more advanced functionalities. For instance, methods defined in classes in the metamodel, are generated without their bodies, which developers must implement. This part consists of enriching the generated core API. Developers also integrate additional classes to implement advanced functionalities (language services, language tooling, ...). This code is built on top of the generated core API.

Take for example the Regular Expression SL. From its metamodel the generated core API allows to parse a given regular expression and to manipulate it problematically (navigate in the AST, modifies it, etc.). However, no advanced functionalities are provided, such as the interpretation and evaluation of the given regular expression. This is the tooling that developers will add on top of the core API. Both depend and uses the generated core API from the metamodel, as shown in Figure 1.

When the metamodel evolves, the same part of the generated code can be re-generated again. As a consequence, the code manually integrated by developers may be impacted and may need to be co-evolved accordingly as well. The next section illustrates these challenges on a real-world use case.

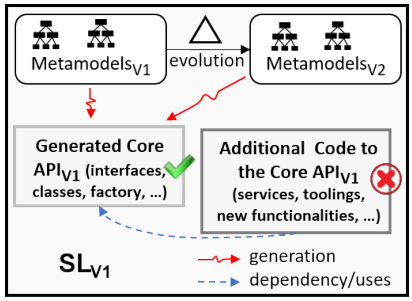

Figure 1: An SL structure and evolution in the Eclipse platform.

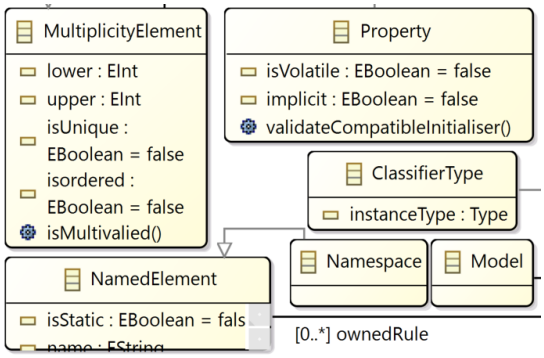

Figure 2: Excerpt of the Pivot metamodel.

2.2 Motivating Example

To motivate our work, this section introduces an example of metamodel and code co-evolution taken from the implementation of the OMG standard *Object Constraint Language (OCL)* [37, 42] in the Eclipse platform, which has evolved several times in the last years. OCL is a declarative language based on first-order logic, to define constraints on models, i.e., conditions that must hold on the models. Part of the OCL implementation is the OCL Pivot which provides the unified merged OCL functionality for the Ecore or UML metamodel instances. The OCL Pivot is implemented as an SL with a metamodel consisting of 111 classes in the version of 3.2.2.

Figure 2 depicts a small excerpt of the Pivot metamodel (6 out of 111 metaclasses). It illustrates the domain concepts `Property`, `MultiplicityElement`, `Model`, `NamedElement`, `Namespace` and `ClassifierType`. From all of these metaclasses, a first code is generated (i.e., core API), consisting of interfaces, classes, a factory, a package, etc. Listing 1 shows a snippet of the generated interfaces and classes for the metamodel excerpt in Figure 2 for the OCL Pivot SL. It is further enriched by the developers with a validation functionality in the new class `PivotValidator`, as shown in Listing 2. Another SL, namely OCL Base, also leverages on the OCL Pivot and its generated interfaces and classes in a conversion functionality from OCL Pivot to OCL Base, as shown in Listing 3.

In version 3.4.4, the OCL Pivot metamodel evolved significantly with various changes, among which, developers: 1) deleted the metaclasses `MultiplicityElement` and `Model` along with their properties, 2) renamed the method *validateCompatibleInitialiser* in the class `Property` to *validateCompatibleDefaultExpression*, 3) renamed the class `ClassifierType` to `MetaClass`, and 4) pushed the property *ownedRule* from `NamedElement` to `Namespace`. Naturally, the Code of Listing 1 was re-generated from the evolved version of the metamodel, which impacted the existing code depicted in Listings 2 and 3. The different caused impacts are underlined in red and their co-evolution is underlined in green in Listings 4 and 5 corresponding to version 3.4.4.

For some impacts, the co-evolution can be unique. For example, the renamed method *validateCompatibleInitialiser* to *validateCompatibleDefaultExpression* or the renamed class `ClassifierType` to `MetaClass` are propagated by developers to the code level as well and renamed in the Java class `PivotValidator` (lines 2 and 6 in Listing 4). However, some other impacts can be co-evolved in many different ways. Indeed, in the class `Pivot2CSConversion`, due to the deleted properties *lower, upper, isOrdered, isUnique* in the class `MultiplicityElement`, two different co-evolutions were applied. The navigation paths to the methods *getLower()* and *getUpper()* were deleted without deleting the whole variable declaration statement (lines 7 and 8 in Listing 3). Whereas, the whole statement of the method call *refreshQualifiers* is deleted rather than deleting the navigation paths to the methods *getIsOrdered()* and *getIsUnique()* (lines 12 and 13 in Listing 3). In the class `PivotValidator` the import and the whole method is deleted due to the delete of `MultiplicityElement` (lines 3 and 12 in Listing 2). Here we already see that a delete change in the metamodel can be co-evolved in different manners to cover different possible co-evolutions to different parts of the code.

Similar situation can even occur for the same impacting change. Take the pushed property *ownedRule* from `NamedElement` to `Namespace`. In the method *refreshClassifier*, the call to *getOwnedRule()* is replaced by *getOwnedInvariant()* (line 17 in Listing 3). Whereas, for the same impacted call to *getOwnedRule()* in the method *refreshTypedElement*, its whole statement of the method call *visitDeclarations* is deleted (line 22 in Listing 3). Other possible co-evolution could also be to introduce a *cast* or a *cast* with an *if* test of the type. This emphasizes the fact that developers should be involved in the co-evolution process to decide which alternatives should be applied.

Listing 1: Excerpt of the generated code in the SL org.eclipse.ocl.examples.pivot.

```
1 //MultiplicityElement Interface
2 public interface MultiplicityElement extends Element
      { ... }
3 //Property Interface
4 public interface Property extends Feature, ... {
5    ...
6    boolean validateCompatibleInitialiser(
         DiagnosticChain diagnostics, Map<Object,
         Object> context);
```

```
7     ...
8 }
9  //PropertyImpl Class
10 public class PropertyImpl implements Property {
11
12     public boolean validateCompatibleInitialiser(
           DiagnosticChain diagnostics, Map<Object,
           Object> context) {
13         try {...} catch (InvalidValueException e)
               {...}
14     }
15 }
```

Listing 2: Excerpt of the additional internal code in the SL org.eclipse.ocl.examples.pivot.

```
1  //Provides validation support
2  import org.eclipse.ocl.examples.pivot.ClassifierType;
3  import org.eclipse.ocl.examples.pivot.MultiplicityElement;
4  public class PivotValidator extends ObjectValidator {
5      ...
6      public boolean validateProperty_
           validateCompatibleInitialiser(...) {
7          return property.validateCompatibleInitialiser(dia,
               con);
8      }
9
10     public boolean validateClassifierType(Classifier
           Type classifierType, ...) { ... }
11
12     public boolean validateMultiplicityElement(Multi
           plicityElement multiplicityElement, ...) { ... }
13 }
```

Listing 3: Excerpt of the additional external code in the SL org.eclipse.ocl.examples.Base.

```
1  //Supports conversion of OCL Pivot to OCL Base
2  public class Pivot2CSConversion extends ... {
3      ...
4      public TypedElementCS
           refreshTypedMultiplicityElement(...) {
5          ...
6          if (csTypeRef != null) {
7             int lower = object.getLower().intValue();
8             int upper = object.getUpper().intValue();
9              ...
10         }
11         ...
12         refreshQualifiers(..., object.getIsOrdered() ?
               Boolean.TRUE : null);
13         refreshQualifiers(..., object.getIsUnique() ?
               null : Boolean.FALSE);
14         return csElement;
15     }
16
17     protected ClassifierCS refreshClassifier(...) {
18         ...
19         refreshList(csElement.getOwnedConstraint(),
               visitDeclarations(ConstraintCS.class,
               object.getOwnedRule(), null));
20         ...
21     }
22
23     public TypedElementCS refreshTypedElement(...) {
24         ...
25         refreshList(csElement.getOwnedConstraint(),
               visitDeclarations(ConstraintCS.class,
               object.getOwnedRule(), null));
26         return csElement;
27     }
28 }
```

Listing 4: Excerpt of the additional internal code in the SL org.eclipse.ocl.examples.pivot.

```
1  //Provides validation support
2  import org.eclipse.ocl.examples.pivot..MetaClass;
3  public class PivotValidator extends ObjectValidator {
4      ...
5      public boolean
           validateProperty_validateCompatibleInitialiser
           (...) {
6          return property.validateCompatibleDefaultExpres
               sion(dia, con);
7      }
8
9      public boolean validateClassifierType(MetaCLass
           metacLass, ...) { ... }
10 }
```

Listing 5: Excerpt of the additional external code in the SL org.eclipse.ocl.examples.Base.

```
1  //Supports conversion of OCL Pivot to OCL Base
2  public class Pivot2CSConversion extends
       AbstractConversion implements PivotConstants {
3      ...
4      public TypedElementCS
           refreshTypedMultiplicityElement(...) {
5          ...
6          if (csTypeRef != null) {
7              int lower;
8              int upper;
9              ...
10         }
11         ...
12         return csElement;
13     }
14
15     protected ClassifierCS refreshClassifier(...) {
16         ...
17         refreshList(csElement.getOwnedConstraint(),
               visitDeclarations(ConstraintCS.class,
               object.getOwnedInvariant(), null));
18         ...
19     }
20
21     public TypedElementCS refreshTypedElement(...) {
22         ...
23         return csElement;
24     }
25 }
```

3 APPROACH

This section presents our approach of code co-evolution with evolving metamodels. Figure 3 shows our overall co-evolution process of code. It first identifies the metamodel changes and then runs an impact analysis on the code [1]. Here, we bridge the gap between the two abstraction levels of metamodels and code. Then a change propagation-based co-evolution is applied [2]. The developer is involved throughout the co-evolution to decide among alternatives, similarly as developers would choose among *quick fixes* in today's IDEs to fix code errors. Finally, the chosen resolutions are applied to co-evolve the code [3].

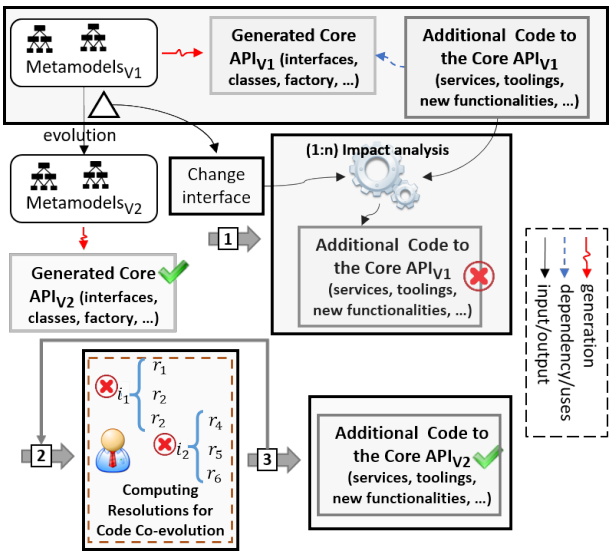

Figure 3: Overall approach [I_n: an impact in the code, R_n: a resolution].

3.1 Change Detection and $1:n$ Impact Analysis

Before we start co-evolving code, we run an impact analysis to detect the code parts that must be co-evolved.

The starting point of our impact analysis is the changes applied during the metamodel evolution. Numerous approaches exists in the literature, such as [4, 29, 30, 35, 48, 50]. To not restrict our approach to a particular detection approach, we define a *Change interface* (see Figure 3) that serves to bridging existing change detection approaches with our internal representation of metamodel changes. Thus, we can replace a given detection approach with another one.

The change interface considers both *atomic* and *complex* changes [19]. *Atomic* changes are additions, removals, and updates of a metamodel element. *Complex* changes consist in a sequence of atomic changes combined together. For example, *push property* is a complex change where a property is moved from a superclass to its subclasses. This is composed of two atomic changes: delete a property and add a property.

With the detected metamodel changes, we run an impact analysis [22] on the code. To this end, we parse the code to access the Abstract Syntax Tree (AST). We then establish a mapping between the metamodel elements $e_1, ..., e_n$ and the code AST nodes corresponding to $e_1, ..., e_n$, as illustrated in Algorithm 1.

In contrast to the co-evolution of metamodel and models/transformations/constraints that considers a $1:1$ relationship between the metamodel elements and their use in the impacted artifacts. Our impact analysis considers a $1:n$ relationship where a metamodel element is used in n different forms at the code level.

Indeed, take for example the class `Property` in the OCL Pivot metamodel. At the code level, an interface `Property` and a class `PropertyImpl` are generated. But also in the classes `PivotFactoryImpl` and `PivotPackageImpl`, respectively the methods *createProperty()* and *getProperty()* are generated. All this n code correspondences of the class `Property` are identified in Algorithm 1 (Lines 5-11). Other metamodel elements also have n code correspondences, such as attributes. For example, for the attribute *implicit* in the class `Property`, a getter and a setter *getImplicit()/setImplicit()* are generated, but also the method *getProperty_Implicit()* is generated in the class `PivotPackageImpl` (handled in Algorithm 1 (Lines 13-20)).

Therefore, one change in the metamodel may impact its n different usages. Our $1:n$ impact analysis leverages on the power of abstraction of the metamodels [28] to identify those n impacts on the code to co-evolve. We leverage on the knowledge of how the core API code is generated to cover all n different usages of a given metamodel element. Finally, the impact analysis consists in accessing for each impacting metamodel change on an element e_i, the set of impacted code parts and their impacted AST nodes.

Algorithm 1 Code impact analysis

```
1: function ConstructMappingTable(Set<MME> mmElements, Set<CE> cElements)
2:     Map<MME, List<CE>> mTable;
3:     for all mme_i ∈ mmElements do
4:         List<CE> cList;
5:         for all ce_i ∈ cElements do
6:             if mme_i.isTypeOf(Class) then
7:                 if (ce_i.name = mme_i.name ∨
8:                     ce_i.name = mme_i.name+"Impl" ∨
9:                     (ce_i.name = "create"+mme_i.name ∧
10:                        ce_i.class.isFactory()) ...) then
11:                    cList.add(ce_i)
12:                end if
13:            else
14:                if mme_i.isTypeOf(Attribute) then
15:                    if ( (ce_i.name=mme_i.name ∨
16:                        ce_i.name="get"+mme_i.name ∨
17:                        ce_i.name='set'+mme_i.name) ∧
18:                        ce_i.class=mme_i.class) ...) then
19:                    end if
20:                    cList.add(ce_i)
21:                else ...//Operations, References, Ennum, etc.
22:                end if
23:            end if
24:            mTable.put(mme_i, cList)
25:        end for
26:    end for
27: end function
```

3.2 Code Co-evolution

Algorithm 2 illustrates the overall co-evolution process. It first retrieves the impacted code parts by a given metamodel change from the mapping table computed by our impact

Algorithm 2 Code co-evolution

```
1: function CODE_CO-EVOLUTION(Set<changes> changes,
   Map<MME, List<CE>> MT, ResolutionCatalog RC)
2:     for all c ∈ changes do
3:         List<CE> codeImpacts ← MT.get(c.element)
4:         for all cImpact ∈ codeImpacts do
5:             resolutions ← RC.getResolutions(cImpact, c)
6:             if resolutions.size > 1 then
7:                 res ← UserDecision(resolutions)
8:             else
9:                 if resolutions.size = 1 then
10:                    res ← resolutions.first()
11:                end if
12:            end if
13:            cImpact.apply(resolution)
14:        end for
15:    end for
16: end function
```

analysis (line 3). Then for each impact, it proposes the appropriate resolutions that can propagate the impacting change (line 5). A developer can choose among the alternative resolutions which one fits her needs. This acts as a user acceptance of the resolution to be applied (line 7). Finally, the chosen resolution is applied on the impacted code part (line 13).

Table 1 depicts our resolutions that propagate to the code metamodel changes that are known to be impacting [22]. The resolutions are inspired from the co-evolution of other artifacts, such as models, constraints, or transformations [2, 5, 6, 11–13, 15, 18, 23–25, 27, 32–34, 49], where they showed to be efficient and useful. Nonetheless, they are further adapted to fit the code co-evolution.

For example, resolutions $CR15, CR16, CR17$ aims to co-evolve the different code impacts of changing a property type in the metamodel. To react to delete changes in the metamodels, we propose to delete different granularities of the impacted code, such as deleting the direct element, its direct parent expression (whole call path), or its instruction. Here we make sure that no error is introduced by further propagating the resolutions when necessary (e.g., to update method calls due to delete of a parameter) and by adding a cast (e.g., to the expression using a deleted property).

Note that, similarly as for code repair approaches [39], we do not claim completeness of our catalog of resolutions. It rather represents meaningful propagations of the metamodel changes at the code level. Enriching our catalog with other resolutions is left for future work. However, as we will show in section 4, the catalog of resolutions in Table 1 is sufficient and useful in practice, and allows to correctly co-evolve impacted code w.r.t. the developers intent.

3.3 Prototype Implementation

Our approach's Java implementation handles Ecore/EMF metamodels and Java code. It interfaces with our previous work of change detection [26] to then run an impact analysis on the code. For each impacted part, we propose resolutions. The resolutions are implemented in Java. Both the impact analysis and the resolutions work on the JDT API[2] to parse and to manipulate the code, application of resolutions.

4 EVALUATION

This section presents the evaluation results of our co-evolution approach. First, we present the dataset and evaluation process. Then, we present the research questions and discuss the obtained results. Time performance of the co-evolution is measured as well.

4.1 Data Set

This section presents the used data set in our evaluation. Our data set covers three case studies covering three different language implementations in Eclipse, namely OCL [37], Modisco [36], and Papyrus [38].

These languages have been developed for more than 10 years and have been evolved several times. 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, verification, and transformation of existing software systems. Papyrus is an industrial project led by CEA[3] to support model-based simulation, formal testing, safety analysis, etc. Thus, our data set covers standard, academic, and industrial languages. We considered original and evolved versions of the case studies, as shown in Table 2.

The three case studies cover code co-evolution in response to five evolved metamodels respectively, two in the OCL language, one in the Modisco Language, and two in the Papyrus language, as shown in Table 2. The total of applied metamodel changes was 477 in the five metamodels.

For those three case studies, we collected 15 Java projects that were impacted by those five evolving metamodels (i.e., projects that are dependent on the metamodels' generated core API). We collected the original and evolved Java code of those projects. Three projects were impacted by two metamodels. Thus, in total we considered 18 projects' co-evolutions due to the five evolved metamodels.

All evolutions in our case studies of the metamodels and code have been performed manually by developers between the different releases that we collected. Thus, we considered the manual code co-evolution as the reference (i.e., ground truth) to which we compare our performed co-evolution.

Table 2 gives details about the selected case studies in particular about their metamodels and the applied changes during evolution. Table 3 further gives details on the size of the projects and code of the original versions that we co-evolve.

[2]Eclipse Java development tools (JDT): https://www.eclipse.org/jdt/core/
[3]http://www-list.cea.fr/en/

Table 1: Catalog of resolutions that propagates 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) ▷$CR5$ Replace p with another property p' (can be given as input by the developer)
◇ 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) ▷$CR5$ Replace `C` with another type `C'`
◇ Rename element e	▷$CR6$ Rename e in the code
◇ Generalize property p multiplicity from a single value to multiple values	▷$CR7$ Introduce a for loop statement to iterate on a collection (e.g., *lng.p.value → for(v in lng.p) {v.val}*) ▷$CR8$ Retrieve the first value of a collection (e.g., value = *lng.p* → value = *lng.p.toArray()[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*	▷$CR9$ Extend navigation path of p_i (e.g., $lng.p_i \rightarrow lng.ref.p_i$) ▷$CR10$ Reduce navigation path of p_i (e.g., $lng.ref.p_i \rightarrow lng.p_i$) ▷$CR11$ Extend navigation path of p_i and add a for loop (e.g., $lng.p_i \rightarrow for(v\ in\ lng.ref)\ \{v.p_i\}$) ▷$CR5$ Replace p_i with another property p'_i
◇ Push property p from class `Sup` to $\texttt{Sub}_1$,...,$\texttt{Sub}_n$	▷$CR12$ Introduce a type test with an If statement (e.g., $t.name = s.p.name \rightarrow if s.p.istypeof Sub_1$) {t.name = ($\text{Sub}_1$ s).p.name} ... $else\ if s.p.istypeof Sub_n$ {t.name = (Sub_n s).p.name}) ▷$CR13$ Cast p to one specific sub class Sub_i (e.g., $t.name = s.p.name \rightarrow t.name = ((Sub_i)s).p.name$) ▷$CR5$ Replace p with another property p'
◇ Inline class `S` to `T` with properties $p_1, ..., p_n$	▷$CR14$ Change the class type from `S` to `T` (e.g., List<S> l = ...; → List<T> l = ...;) ▷$CR10$ Reduce navigation path of p_i (e.g., $lng.ref.p_i \rightarrow lng.p_i$)
◇ Change property p type from `S` to `T`	▷$CR15$ Change variable declaration type initialized with p from `S` to `T` (e.g., S var = s.p; → T var = s.p;) ▷$CR16$ Add a cast of p when used as argument of a method (e.g., mCall(p) → mCall((ExpectedType) p)) ▷$CR17$ Cast the value for the setter of p to `T` (e.g., s.setP(v); → s.setP((T) v);)

Table 2: 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 Total = 218	1 pull property 2 push properties Total = 3
	Base.ecore in project ocl.examples.xtext.base	3.1.0 to 3.4.4	Deletes: 4 classes, 11 properties, 17 super types Renames: 1 property Property changes: 2 type Adds: 10 classes, 38 properties, 19 super types Total = 102	5 moves properties 1 push property 2 extract class 2 extract super class Total = 10
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 Total = 70 changes	4 moves property 6 pull property 1 extract class 1 extract super class Total = 12
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 Total = 42	2 pull property 1 push property 1 extract super class Total = 3
	Configuration.ecore in project papyrus.infra.queries.core.configuration	0.9.0 to 1.1.0	Deletes: 6 classes, 7 properties, 4 super types Total = 17	none

4.2 Evaluation Process

We evaluate our change propagation-based co-evolution of code by measuring the correctness of our co-evolution. We compare for the same set of code how it was manually co-evolved by developers against our proposed co-evolution. In this experiment, we measure the correctness of our approach by using the two metrics *precision* and *recall* that vary from 0 to 1, i.e., 0% to 100%. They are defined as follows:

Table 3: Details of the projects and their impacts caused 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 impacts
OCL Pivot.ecore	$P1$ ocl.examples.pivot	22	439	74002	47	532
	$P2$ ocl.examples.xtext.base	12	181	17599	9	30
OCL Base.ecore	$P3$ ocl.examples.xtext.base	12	181	17599	21	136
	$P4$ ocl.examples.xtext.completeocl	14	83	33807	9	16
	$P5$ ocl.examples.xtext.essentialocl	17	135	30920	15	44
	$P6$ ocl.examples.xtext.oclinecore	15	46	45862	3	10
	$P7$ ocl.examples.xtext.oclinecore.ui	10	25	55648	3	6
	$P8$ ocl.examples.xtext.oclstdlib	14	58	38639	1	2
Modisco Benchmark.ecore	$P9$ modisco.infra.discovery.benchmark	3	28	2333	0	0
	$P10$ gmt.modisco.java.discoverer.benchmark	8	21	1947	3	13
	$P11$ modisco.java.discoverer.benchmark	10	28	2794	5	24
	$P12$ modisco.java.discoverer.benchmark.javaBenchmark	3	16	1654	7	58
Papyrus ExtendedTypes.ecore	$P13$ papyrus.infra.extendedtypes	7	37	2057	8	47
	$P14$ papyrus.infra.extendedtypes.emf	5	12	374	5	14
	$P15$ papyrus.uml.tools.extendedtypes	5	15	725	5	14
Papyrus Configuration.ecore	$P16$ papyrus.infra.queries.core.configuration	4	23	1045	0	0
	$P17$ papyrus.infra.extendedtypes	7	37	2057	4	12
	$P18$ papyrus.infra.extendedtypes.emf	5	12	374	4	18

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

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

The *ProposedResolutions* are the resolutions applied by our approach (from Table 1) and the *ExpectedResolutions* are the actual manually performed resolutions by developers.

Note that we also studied to confirm the metamodel changes from version *1* to version *2*. Based on those changes we run an impact analysis to identify the impacted parts of the code that will be co-evolved.

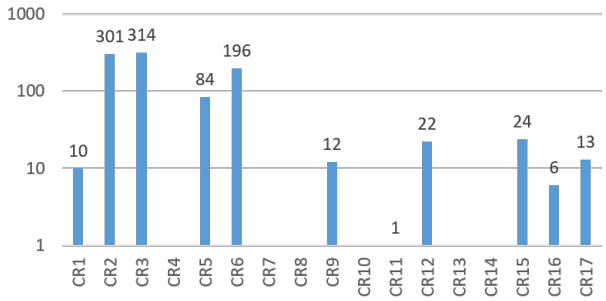

Figure 4: Frequency of the applied resolutions.

4.3 Research Questions

This section defines two research questions (RQs) to assess in particular the applicability, correctness, and the usefulness of our work. The research questions are as follows:

RQ1: To what extent and how fast can our co-evolution approach handle the impacted parts of the code? If we cannot propose a co-evolution for all impacted parts due to evolving metamodels, we can conclude that our change propagation-based co-evolution of code is insufficient. This assesses the overall applicability of our co-evolution approach.

RQ2: To what extent does our approach correctly co-evolve the impacted code? This measures its correctness level with the precision and recall metrics, i.e., to which extent it proposes the expected resolutions from developers. It also allows us to assess its usefulness. For alternative resolutions, we select the one as close as possible to the expected resolution to be able to compare them afterward.

RQ3: Can our approach still propose relevant co-evolution when it fails to propose the expected one by developers? This aims to investigate, when our co-evolution diverges from the expected one, whether our proposed resolutions are still useful to developers.

4.4 Results

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

4.4.1 RQ1. With our impact analysis, we were able to find the impacted code parts by the 477 metamodel changes for which resolutions were proposed by our co-evolution approach. One to five resolutions were proposed for each impact in the code. To co-evolve all impacted parts, a total of 983 resolutions were applied during the change propagation. This shows the applicability of our co-evolution approach that was able to handle all different impacts in the code caused by the metamodel evolution changes. Table 4 gives the list of the applied resolutions for each co-evolved project during co-evolution. In total, the 972 applied resolutions were covered by 11 resolution types from our catalog in Table 1, namely $CR1, CR2, CR3, CR5, CR6, CR6, CR9, CR11, CR12, CR15, CR16, CR17$. These resolutions allowed

Table 4: Number of applied resolutions in our code co-evolution for each project and per evolved metamodel.

Evolved metamodels	Co-evolved projects	N^o of applied resolutions
OCL Pivot.ecore	$P1$	$CR1:3, CR2:57, CR3:298, CR5:19, CR6:108, CR12:18, CR15:15, CR16:3, CR17:11$
	$P2$	$CR2:10, CR3:8, CR5:2, CR6:5, CR12:2, CR15:1 CR16:1, CR17:1$
OCL Base.ecore	$P3$	$CR1:3, CR2:91, CR3:5, CR5:18, CR6:1, CR9:10, CR11:1, CR12:1, CR15:7 CR16:3, CR17:1$
	$P4$	$CR2:10, CR5:6$
	$P5$	$CR1:1, CR2:25, CR5:17, CR9:1, CR11:1, CR12:1$
	$P6$	$CR2:9, CR3:1$
	$P7$	$CR1:1, CR2:5$
	$P8$	$CR5:2$
Modisco Benchmark. ecore	$P9$	*na*
	$P10$	$CR2:13$
	$P11$	$CR2:24$
	$P12$	$CR1:2, CR2:28, CR5:20, CR6:8$
Papyrus ExtendedTypes. ecore	$P13$	$CR2:1, CR6:47$
	$P14$	$CR6:16$
	$P15$	$CR6:14$
Papyrus Configuration. ecore	$P16$	*na*
	$P17$	$CR2:12$
	$P18$	$CR2:18$

us to cover different possible co-evolution for the different impacting metamodel changes. Note that all occurrences of $CR1$ were due to deleted classes (`DC`) in the metamodels that were used in the code by subclasses (i.e., *extends* `DC`). Figure 4 further shows the frequency of each applied resolution. As some impacting change types did not occur in our case studies, their resolutions were never applied, such as $CR7, CR8, CR14$.

The parsing of the project's code took from couple of seconds to a minute for the largest project. For each impacting metamodel change, the detection of the impacted code parts took from milliseconds to few seconds, and the application of each resolution was a matter of less than half a second. The evaluation was run on a Windows 7 PC with a Core i7 3.4GHz and 16GB RAM.

4.4.2 RQ2. The metamodel changes present in our case studies (see Table 2 columns 5 and 6) covered different types of changes and evolution scenarios. In particular the impacting changes for which we proposed resolutions in Table 1. To assess to what extent can our approach correctly co-evolve the impacted code w.r.t. the manual co-evolution of the code that developers went through, we measured precision and recall.

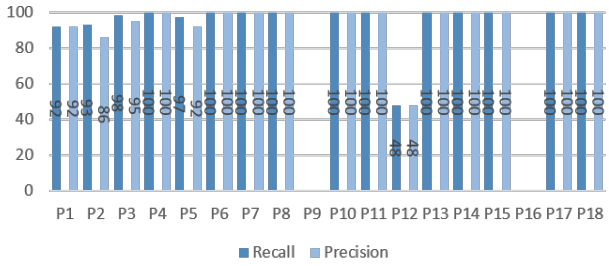

Figure 5: Precision measurement for our code co-evolution.

Figure 5 depict the reached precision and recall in our 18 scenarios of metamodel evolution and code co-evolution. The projects *[P9]* and *[P16]* were not impacted by any metamodel evolution with no performed co-evolution. Thus, we did not measure precision and recall for them. For the rest of the projects, both the measured precision and recall varied from 48% to 100%. Our code co-evolution was able to reach respectively the average of 94,5% and 95,5% of precision and recall. However, as the size of the projects and the metamodels' evolution varied from small to large, we computed a weighted average to better characterize precision and recall. We computed the weighted average by the number of the metamodel changes for each project, resulting in a weighted average of 87.4% and 88.9% of precision and recall. This means that our applied resolutions propagating the impacting metamodel changes covered the expected resolutions by the developers in 88.9% and were correct in 87.4%. This shows that the catalog of resolutions in Table 1 showed to be both sufficient and useful in co-evolving code in our case studies while meeting the developer's co-evolution needs.

4.4.3 RQ3. We further looked into the cases where our proposed resolutions mismatched the expected resolutions, i.e., the cause of lowering precision and recall. We observed that for all those impacts in the code caused by non-delete changes in the metamodels, the expected resolutions applied in the co-evolution by the developers were to delete the impacted code. Rather than to delete them, our co-evolution approach was able to successfully maintain them.

For example, in the project *[P12]*, we observed that 28 impacted parts of the code were due to move and rename changes in the metamodel. Listing 6 shows an excerpt of two impacted part of the code from the project *[P12]* due to: 1) rename *discoveryError* to *discoveryErrors*, and 2) Our co-evolution approach was able to maintain impacted parts of the code, whereas, the developers' manual co-evolution consisted in deleting them as the whole project was deleted in the new version. The applied resolutions were replacing the call to *discoveryDate* to a call from its new class and renaming *discoveryError*, as shown in Listing 8.

Similarly, in the project *[P1]*, we observed the same cases of 41 deleted code impacted by the metamodel changes rename, push property, and type change. Listing 7 shows an excerpt of two impacted part of the code from the project *[P1]* due to: 1) push property *ownedRule* and 2) change type of the property *specification*. The developer manual co-evolutions consisted in deleting the statements where they were used. Instead, we maintained them by changing the type of the variable declaration and by introducing an *if* statement with a cast, as shown in Listing 9.

The above observations show that developers unnecessarily deleted many impacted parts in the code, while our approach instead successfully maintain them during co-evolution. This is the cause of lowering precision and recall in our case studies, e.g., 48% in the project *[P12]*. From our point of view, it is surprising to delete code statements, whereas it would have been possible to maintain them (e.g., with rename, move, or push resolutions). This obviously hints on the lack of co-evolution support. Otherwise, they would have been easily maintained in the new version. Another explanation is that the code functionality (or its requirement) became unnecessary in the new version, which explains its deletion.

Whereas these cases lower precision and recall, we do not consider it as wrong co-evolution. It could actually be preferred over simply deleting the impacted code. Indeed, it is more relevant to propagate a rename of a method than to delete its call elsewhere in the code, which our approach was able to achieve for those cases. Nonetheless, it could be possible that for some case simply deleting the impacted code (by non-delete metamodel changes) is desired. In future work, we could propose delete resolutions as an alternative for any impacted code. That is why it is essential for developers to guide the co-evolution process and decide which resolutions best fit their needs.

Listing 6: Excerpt of the impacted code in the project P12.

```
1 //in the class CDOProjectDiscoveryImpl
2 if (baseClass == Discovery.class) {
3 switch (baseFeatureID) {
4 case BenchmarkPackage.DISCOVERY__DICOVERY_DATE: return
      ...;
5                 ...
6 case BenchmarkPackage.DISCOVERY__DISCOVERY_ERROR: return
      ...;
7 default: return 1;
8}
```

Listing 7: Excerpt of the impacted code in the project P1.

```
1 //in the class InvocationBehavior
2                 ...
3 ValueSpecification valueSpecification = constraint.get
      Specification();
4 if (valueSpecification instanceof ExpressionInOCL) {
      ... }
5                 ...
6 //in the class AbstractDelegatedBehavior
7                 ...
8 for (Constraint constraint : namedElement.getOwne
      dRule()) {
9     ...
10}               ...
```

Listing 8: Excerpt of the co-evolved code in the project P12.

```
1 //in the class CDOProjectDiscoveryImpl
2 if (baseClass == Discovery.class) {
3 switch (baseFeatureID) {
4 case BenchmarkPackage.DISCOVERY_ITERATION__DICOVERY_DATE:
      return ...;
5                 ...
6 case BenchmarkPackage.DISCOVERY__DISCOVERY_ERRORS: return
      ...;
7 default: return 1;
8}
```

Listing 9: Excerpt of the co-evolved code in the project P1.

```
1 //in the class InvocationBehavior
2                 ...
3 OpaqueExpression valueSpecification = constraint.getSpeci
      fication();
4 if (valueSpecification instanceof ExpressionInOCL) {
      ... }
5                 ...
6 //in the class AbstractDelegatedBehavior
7                 ...
8 if(namedElement instanceof Namespace){
9    for (Constraint constraint : (Namespace
       namedElement).getOwnedRule()) {
10       ...
11   }
12}               ...
```

5 THREATS TO VALIDITY

This section discusses threats to validity [51].

5.1 Internal Validity

To measure precision and recall, we had to analyze the developers manual co-evolution to identify the expected resolutions. To reduce the risk of misidentifying an expected resolution, for each impacted part of the code, we investigated the whole 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 the original and evolved impacted parts of the code. Moreover, as our co-evolution relies on the quality of detected metamodel changes. We analyzed each detected change and checked whether it occurred between the original and evolved metamodels to alleviate the risk of relying on an incorrect metamodel change.

Moreover, in the evaluation, when alternative resolutions were proposed during co-evolution, we selected the one as close as possible to the expected resolution to be able to compare them afterward. Our goal in the evaluation was to assess (RQ1) whether we can co-evolve the impacted code and (RQ2) usefulness of our approach in terms of how far can it allow a developer to reach the same manual co-evolution of our case studies. Assessing how different developers would co-evolve the same impacts, possibly diverging with different resolutions, was not the goal of our evaluation. It is left for a future user study where we would also investigate what influences developers to select a given resolution over another

one. Still, We observed a start of an answer in (RQ3) where alternative resolutions were actually relevant.

5.2 External Validity

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

Furthermore, our evaluation was performed on SLs and their toolings in the Eclipse platform. Thus, we cannot generalize our findings to all SLs or DSLs neither domain models for instance. However, our approach could be used to co-evolve Java code built on a generated API from a metamodel or a domain model. This is the case for example for tools such as JHipster that generates a complete and modern Web app or microservice architecture based on a domain model and a set of architectural choices. Further evaluations remain necessary here.

5.3 Conclusion Validity

Our evaluation gave promising results, showing that our code co-evolution is fast and useful with a weighted average precision and recall respectively of 87.4% and 88.9%. The evaluation results also showed the usefulness of our approach and the proposed resolutions in our catalog in Table 1. Even though we evaluated it on 18 scenarios of metamodel evolution and code co-evolution. To have more insights and statistical evidence, further evaluation is needed on more case studies.

6 RELATED WORK

The main idea of change propagation was investigated for different purposes, such as by Cubranic et al. [7] who used it to recommend relevant software development artifacts, or Demuth el al. [9] who used it for models co-evolution. In our work we use change propagation to co-evolve code.

In this section, we present the main related work w.r.t. co-evolution. Many approaches proposed to co-evolve models [5, 11, 16, 18, 24, 25, 44, 49], constraints [2, 6, 27, 31, 33], and model transformations [12, 13, 23, 32, 34]. Our work first distinguishes by co-evolving code. It further distinguishes with related work by performing a $1 : n$ impact analysis. Whereas, a $1 : 1$ impact analysis is performed in the approaches of models/constraints/transformations co-evolution [2, 5, 6, 11–13, 18, 23–25, 27, 32–34, 49].

In this paper, we focus so far the co-evolution on the direction metamodel to code, which is not trivial. Yu et al. [52] 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.

Existing approaches of code migration are related to our work. We focus on the main existing approaches. Henkel et al. [17] 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. [40] also proposed an approach that guides developers in adapting code by learning adaptation patterns from previously migrated code. Similarly, Dagenais et al. [8] also uses a recommendation mechanism of code changes by mining them from previously migrated code. Anderson et al. [1] 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 [1, 8, 17, 40] by considering and reasoning on the changes at the metamodel level and not at the code level. Thus, our approach treats way less changes to identify the impacted parts in the code than code migration approaches [1, 8, 17, 40]. This is possible thanks to the abstraction offered by the metamodels. Whereas our work considers *one* change of a given element, n changes must be considered in order to fully migrate the code. To the best of our knowledge, our work is the first attempt to address the challenge of code co-evolution with evolving metamodels in SLs.

7 CONCLUSION

This paper proposed a code co-evolution approach when metamodels evolve. It supports a change propagation-based co-evolution by leverages on the metamodel changes and propagates them on the code. Our code co-evolution was evaluated on 15 projects and five metamodels from three Eclipse language implementations and it was stress tested on 18 scenarios of metamodel evolution and code co-evolution. It showed to be efficient and useful in co-evolving code with a weighted average precision and recall respectively of 87.4% and 88.9%.

As future work, we first plan to enrich our catalogue of resolutions. Although, we envision and present our resolutions similarly as code quick fixes or repair for developers to choose from, we plan to explore ranking heuristics to ease the task of selecting resolutions for developers. Building on that, we can then propose a full automation based on the highest ranked resolutions to compute co-evolutions plans. Finally, we will evaluate our approach on more case studies other than Eclipse language implementations.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the CNRS PEPS, and from the AIS Rennes Métropole under grant no. 190270.

REFERENCES

[1] Jesper Andersen and Julia L Lawall. 2010. Generic patch inference. *Automated software engineering* 17, 2 (2010), 119–148.

[2] Edouard Batot, Wael Kessentini, Houari Sahraoui, and Michalis Famelis. 2017. Heuristic-Based Recommendation for Metamodel—OCL Coevolution. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 210–220.

[3] Jordi Cabot and Martin Gogolla. 2012. Object constraint language (OCL): a definitive guide. In *Formal methods for model-driven engineering*. Springer, 58–90.

[4] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. 2009. Managing dependent changes in coupled evolution. In *Theory and Practice of Model Transformations*. Springer, 35–51.

[5] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2008. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*. IEEE, 222–231.

[6] Alexandre Correa and Cláudia Werner. 2007. Refactoring object constraint language specifications. *Software & Systems Modeling* 6, 2 (2007), 113–138.

[7] Davor Čubranić and Gail C Murphy. 2003. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th international Conference on Software Engineering*. IEEE Computer Society, 408–418.

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

[9] Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E Lopez-Herrejon, and Alexander Egyed. 2016. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software* 111 (2016), 281–297.

[10] Martin Fowler. 2010. *Domain-specific languages*. Pearson Education.

[11] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. 2009. Managing model adaptation by precise detection of metamodel changes. In *Model Driven Architecture-Foundations and Applications*. Springer, 34–49.

[12] Kelly Garcés, Juan M Vara, Frédéric Jouault, and Esperanza Marcos. 2014. Adapting transformations to metamodel changes via external transformation composition. *Software & Systems Modeling* 13, 2 (2014), 789–806.

[13] Jokin García, Oscar Diaz, and Maider Azanza. 2013. Model transformation co-evolution: A semi-automatic approach. *SLE* 7745 (2013), 144–163.

[14] Richard C Gronback. 2009. *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education.

[15] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. 2015. Surveying the corpus of model resolution strategies for metamodel evolution. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 135–142.

[16] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. 2016. Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering* 43, 5 (2016), 396–414.

[17] Johannes Henkel and Amer Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE, 274–283.

[18] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. 2009. COPE-automating coupled evolution of metamodels and models. In *ECOOP 2009–Object-Oriented Programming*. Springer, 52–76.

[19] Markus Herrmannsdoerfer, Sander D. Vermolen, and Guido Wachsmuth. 2011. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In *Software Language Engineering*, Malloy, Staab, and Brand (Eds.). Springer, 163–182.

[20] John Hutchinson, Mark Rouncefield, and Jon Whittle. 2011. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 633–642.

[21] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 471–480.

[22] Ludovico Iovino, Alfonso Pierantonio, and Ivano Malavolta. 2012. On the Impact Significance of Metamodel Evolution in MDE. *Journal of Object Technology* 11, 3 (2012), 3–1.

[23] Wael Kessentini, Houari Sahraoui, and Manuel Wimmer. 2018. Automated Co-evolution of Metamodels and Transformation Rules: A Search-Based Approach. In *International Symposium on Search Based Software Engineering*. Springer, 229–245.

[24] Wael Kessentini, Houari Sahraoui, and Manuel Wimmer. 2019. Automated metamodel/model co-evolution: A search-based approach. *Information and Software Technology* 106 (2019), 49–67.

[25] Wael Kessentini, Manuel Wimmer, and Houari Sahraoui. 2018. 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, 101–111.

[26] Djamel Eddine Khelladi, Reda Bendraou, and Marie-Pierre Gervais. 2016. Ad-room: a tool for automatic detection of refactorings in object-oriented models. In *ICSE Companion*. ACM, 617–620.

[27] Djamel Eddine Khelladi, Reda Bendraou, Regina Hebig, and Marie-Pierre Gervais. 2017. A semi-automatic maintenance and co-evolution of OCL constraints with (meta) model evolution. *Journal of Systems and Software* 134 (2017), 242–260.

[28] Djamel Eddine Khelladi, Benoit Combemale, Mathieu Acher, and Olivier Barais. 2020. 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)*.

[29] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. 2015. Detecting complex changes during metamodel evolution. In *CAISE*. Springer, 263–278.

[30] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. 2016. Detecting complex changes and refactorings during (meta) model evolution. *Information Systems* (2016).

[31] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. 2016. Metamodel and constraints co-evolution: A semi automatic maintenance of ocl constraints. In *International Conference on Software Reuse*. Springer, 333–349.

[32] Djamel Eddine Khelladi, Roland Kretschmer, and Alexander Egyed. 2018. 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, 404–414.

[33] Angelika Kusel, Juergen Etzlstorfer, Elisabeth Kapsammer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger, and Manuel Wimmer. 2015. Systematic Co-Evolution of OCL Expressions. In *11th APCCM 2015*, Vol. 27. 30.

[34] Angelika Kusel, Jurgen Etzlstorfer, Elisabeth Kapsammer, Werner Retschitzegger, Wieland Schwinger, and Johannes Schonbock. 2015. Consistent co-evolution of models and transformations. In *ACM/IEEE 18th MODELS*. 116–125.

[35] Philip Langer, Manuel Wimmer, Petra Brosch, Markus Herrmannsdörfer, Martina Seidl, Konrad Wieland, and Gerti Kappel. 2013. A posteriori operation detection in evolving software models. *Journal of Systems and Software* 86, 2 (2013), 551–566.

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

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

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

[39] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 17.

[40] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr, Anh Tuan Nguyen, Miryung Kim, and Tien N Nguyen. 2010. A graph-based approach to API usage adaptation. *ACM Sigplan Notices* 45, 10 (2010), 302–321.

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

[42] OMG. 2015. Object Management Group. Object Constraints Language (OCL). http://www.omg.org/spec/OCL/.

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

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

[45] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

[46] Juha-Pekka Tolvanen and Steven Kelly. 2009. MetaEdit+: defining and using integrated domain-specific modeling languages. In *The 24th ACM SIGPLAN conference companion on OOPSLA*. 819–820.

[47] Arie Van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices* 35, 6 (2000), 26–36.

[48] Sander D Vermolen, Guido Wachsmuth, and Eelco Visser. 2012. Reconstructing complex metamodel evolution. In *Software Language Engineering*. Springer, 201–221.

[49] Guido Wachsmuth. 2007. Metamodel adaptation and model co-adaptation. In *ECOOP*. Springer, 600–624.

[50] James R Williams, Richard F Paige, and Fiona AC Polack. 2012. Searching for model migration strategies. In *Proceedings of the 6th International Workshop on Models and Evolution*. ACM, 39–44.

[51] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[52] Yijun Yu, Yu Lin, Zhenjiang Hu, Soichiro Hidaka, Hiroyuki Kato, and Lionel Montrieux. 2012. Maintaining invariant traceability through bidirectional transformations. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 540–550.