Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Detecting Metamodel Evolutions in Repositories of Model-Driven Projects

Lorenzo Bettini[b] Davide Di Ruscio[c] Ludovico Iovino[a]
Alfonso Pierantonio[c]

a. Gran Sasso Science Institute, Italy
b. Università degli Studi di Firenze, Italy
c. Università degli Studi dell'Aquila, Italy

Abstract Model-Driven Engineering (MDE) is a discipline that leverages abstraction and automation in software development. Projects are typically composed of inherently different artifacts, including models, metamodels, model transformations, code generators, and concrete syntax definitions. Despite the increasing availability of reusable projects (e.g., through GitHub), their reuse possibilities depend on the availability of accurate, high-level metadata describing architectural information about the project at hand. In this paper, we enhance an existing approach for extracting relevant architectural information from model-driven projects to detect subsequent metamodel versions in evolution paths. In particular, we are interested in those refactorings that enhance the intrinsic quality of metamodels. The approach has been implemented by extending the existing MDEprofiler tool and has been validated on a dataset consisting of metamodels with different size and covering distinct application domains.

Keywords Model-Driven Engineering, reverse engineering, megamodels, evolution, quality.

1 Introduction

The sheer complexity of software systems requires leveraging abstraction and introducing automated processes in software development. Model-Driven Engineering [Sch06] (MDE) is a discipline that prescribes models as first-class entities to support the development and analysis of complex software systems. Model-driven projects are typically composed of inherently different artifacts, including models, metamodels, model transformations, code generators, and concrete syntax definitions. As in the case of software development that benefits from the availability of source code repositories, having reusable modeling artifacts available can speed-up modeling activities as well as increase the quality of the system being modeled [VSB+13]. Over the last

Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino, Alfonso Pierantonio. *Detecting Metamodel Evolutions in Repositories of Model-Driven Projects.* In Journal of Object Technology, vol. 19, no. 2, 2020, pages 14:1–22.
doi:10.5381/jot.2020.19.2.a14

decade, several both academic and commercial model repositories have been proposed (e.g., [KMB+08, KC13, KH10, HRW09], and the versioning capabilities provided by commercial tools like Enterprise Architect[1], MagicDraw[2], MetaEdit+[3], QualiWare[4], SystemWeaver[5]) so that developers, instructors, and researchers can rely on existing artifacts to support the modeling tasks being undertaken. Despite the increasing availability of reusable projects, their reuse depends on the possibility of retrieval mechanisms based on accurate, high-level metadata describing architectural information about the project at hand.

Unfortunately, understanding how collections of models can be robustly and efficiently managed is still a problem that needs to be better understood and addressed [Ste18]. In particular, as stated in [DRDRH+19] *"the ability of carefully analyzing projects to identify their components and their interrelationships is key to obtain representations at a higher level of abstraction that can support reuse processes"*. In [DRDRH+19], an approach called MDEprofiler is given for the automated retrieval of architectural information about model-driven projects. A megamodel-based representation is used for capturing the different types of artifacts and their interrelationships. Heuristics are used to enable the recovery of specific types of architectural information about the analyzed projects. In particular, current heuristics in MDEprofiler identify a diversity of relationships involving metamodels, including the conformance relation between models and metamodels, and the domain/co-domain conformance between model transformations and metamodels.

In this paper, we extend our approach to the detection of metamodel evolution patterns. In particular, understanding the intents behind modification actions may reveal useful insights about how quality factors are affected by the way a metamodel is modified [BDD+19]. Thus, the discovery capabilities of MDEprofiler are enhanced in order to detect metamodel evolutions in repositories. More specifically, the research problem we want to address in this paper can be summarized as

How to detect quality enhancing metamodel evolutions?

that can be distilled into the following research challenges:

- *RC1: How to automatically detect metamodel evolutions?* Not all metamodel modification can be considered as part of an evolution process [Her11]. In this paper, we consider two subsequent versions of a metamodel being part of an evolution pattern whenever enough information in the final version can be traced back to the initial one. In essence, the two metamodels must share some commonalities. As a consequence, it is necessary to conceive an automated mechanism that is able to automatically detect when a metamodel is the evolution of another one.
- *RC2: How to automatically detect quality improving metamodel evolutions?* Among metamodel evolutions, we want to characterize those that have been operated with the aim to enhance certain quality factors in the original metamodel, e.g., reusability, complexity.

Outline. The structure of the paper is as follows. Section 2 presents a motivational example; whereas the next section illustrates the proposed approach to the detection of

[1] https://sparxsystems.com/
[2] https://www.nomagic.com/products/magicdraw
[3] https://www.metacase.com/
[4] https://www.qualiware.com/
[5] https://www.systemweaver.se/

quality-driven metamodel evolutions. Section 4 illustrates an experimental validation of the proposed techniques. A discussion about the related work is given in Section 5. Finally, Section 6 discusses future work and draws some conclusions.

2 Motivating scenario

As mentioned above, the approach implemented in MDEPROFILER allows to harvest relevant architectural information from projects stored in a modeling repository. Such knowledge can be useful for extending the analyzed repository with elicited structures that can better characterize and leverage the nature of the artifacts and their interrelationships. As an example, consider the project scenario in Fig. 1.a where different artifacts are involved, including EMF-based metamodels, Acceleo[6] model-to-code transformations, and ATL[7] model-to-model transformations. Users who are interested in reusing such a project may have difficulties in understanding its structure and the role of the different artifacts therein. At this stage, MDEPROFILER is able to reverse engineer the project and to represent its architectural structure, as shown in Fig. 1.b. Designers can then analyze the different project components and how they are related to each other, with a consequent increase of reuse possibilities.

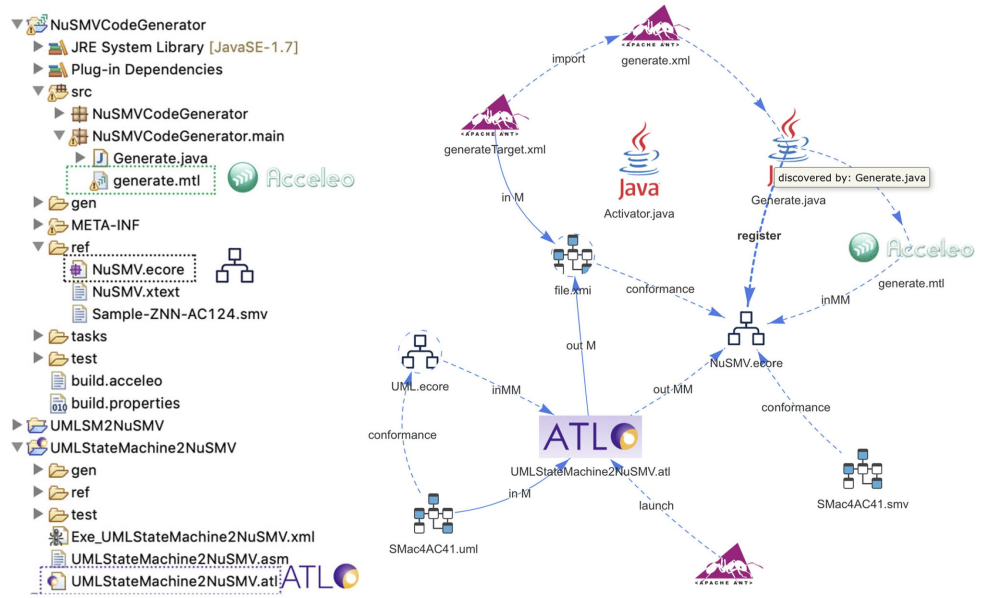

Figure 1 – An explanatory MDE project (borrowed from [DRDRH+19])

MDEPROFILER relies on an extensible library of heuristics, each devoted to discovering specific kinds of artifacts and relationships. The application of each heuristic contributes to the creation of models representing graphs: for each object that can be identified by the heuristics, the recovery approach generates a corresponding node in the model being created. Heuristics are also used to locate artifacts that encode relationships (e.g., build scripts with model transformation applications). MDEPRO-

[6] https://www.eclipse.org/acceleo/
[7] https://www.eclipse.org/atl/

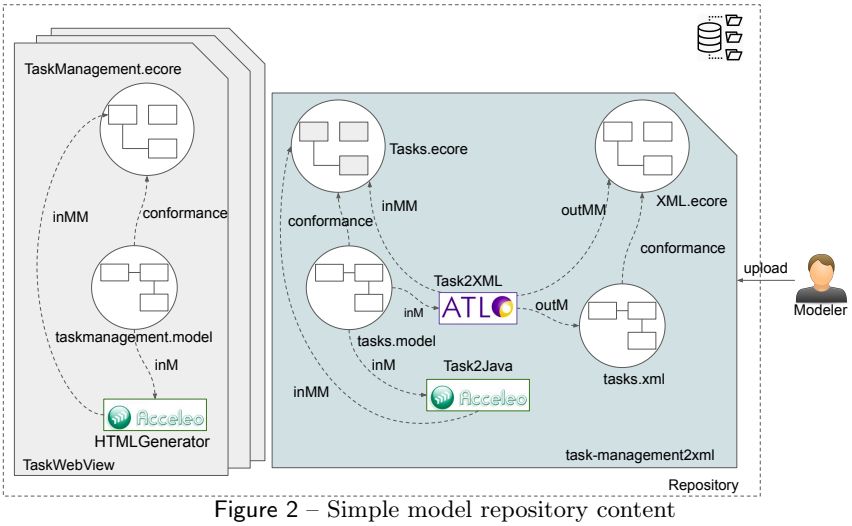

Figure 2 – Simple model repository content

FILER, as defined in [DRDRH+19], is not able to detect relationships related to the evolutionary nature of modeling artifacts. In particular, MDEPROFILER is not able to put in relation metamodels that are evolved versions of already stored ones.

Figure 2 shows a motivating example supporting the need for retrieving relationships among metamodels that might even belong to different projects stored in the same repository. In the initial setting shown in Fig. 2, the explanatory repository contains artifacts that are organized in different projects. On the left-hand side of the figure, the content of the simple TaskWebView project is shown, and it consists of the taskmanagement model conforming to the TaskManagement metamodel, which is shown in Fig. 3. The project includes also the HTMLGenerator Acceleo transformation, which is able to generate target HTML representations from source TaskManagement models.

According to the right-hand side of Fig. 2, the modeler uploads a new project to share it with other users of the repository. The project being uploaded consists of several artifacts including the ATL transformation Task2XML generating XML

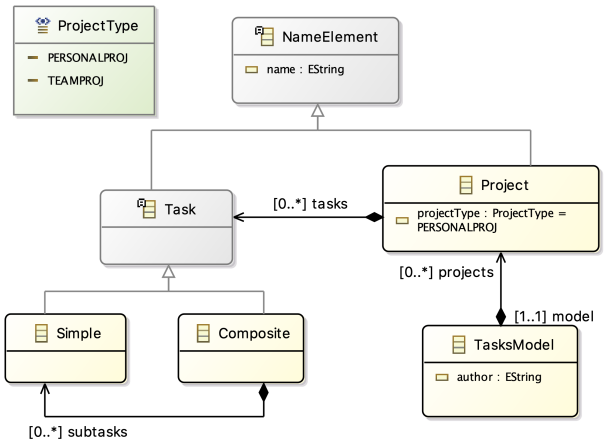

Figure 3 – The *TaskManagement* metamodel

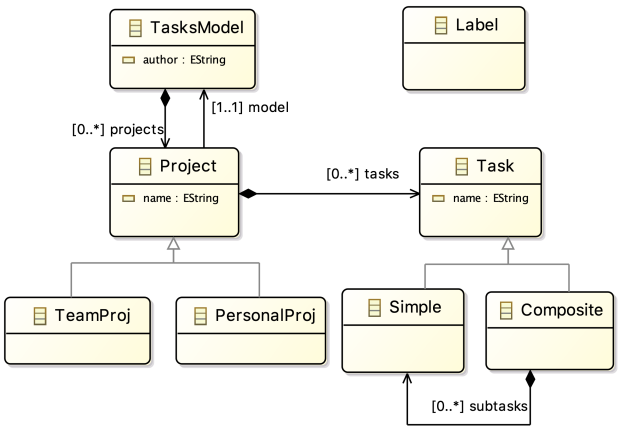

Figure 4 – The *Tasks* metamodel

documents from input models conforming to the Tasks metamodel. The project includes also the models tasks.model and tasks.xml conforming to these metamodels. The Acceleo transformation Task2Java is also part of the project, and it generates Java code for programmatic task definition.

The Tasks metamodel being uploaded in the repository (see Fig. 4) permits modelers to specify projects in terms of composite and straightforward tasks. By analyzing the content of the TaskManagement and Tasks metamodels, it is possible to notice that they are both related to the same application domain (i.e., task management). Moreover, they share several structural similarities: both permit to distinguish projects to be developed in a team or for individual purposes. Moreover, each project can be composed of Tasks, and a single task can contain other tasks (see the relationship subtasks between the Composite and Simple metaclasses).

The example shows a concrete case of two metamodels that are in relationships but they belong to different projects. In the specific case shown in Fig. 2, the two metamodels should be linked using an *evolution* relation to capture the information that one metamodel is an evolved version of the other one.

The benefits related to the availability of such information are many, especially for locating specific artifacts for reuse purposes. In fact, expressive requirements can be formulated to discover metamodels that are, for instance, part of an evolution path or that are endowed with support tools.

3 Understanding Metamodel Evolution

The problem of managing the evolution of modeling artifacts has been largely investigated from different perspectives [PMR16]. Several approaches have been proposed to deal with the coupled evolution problem, especially to migrate existing artifacts (e.g., models, transformations, graphical and textual editors) that have lost conformance after a metamodel underwent modifications [HKB17]. Most of the existing approaches compare two versions of the same metamodel in order to understand how to migrate those artifacts that lost validity with respect to the modified metamodel. Such approaches typically work when the given input metamodels are already known to be related through some evolution.

In [VWV12, KHB+16], the authors deal with the problem of metamodel evolution

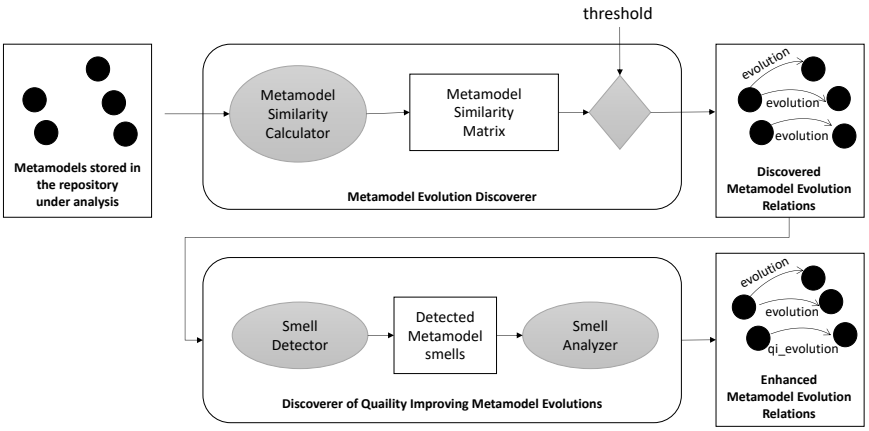

Figure 5 – Overview of the proposed approach

with the main goal of understanding and reproduce the modifications between two versions of the same metamodel. Typically, the detection of the differences between two metamodels can be done through operation- or state-based methods [KHL+10]. In the former ones, the differences are represented in procedural terms using operation recorders, whereas, in the latter ones, metamodels are unrelated, and the differencing is performed by using a matching algorithm [KDRPP09]. However, evolutionary patterns in metamodel repositories are not known beforehand, therefore evolution relationships must be elicited by analyzing and comparing the metamodels with a state-based approach.

Figure 5 shows an overview of the proposed approach, which consists of two main phases: *i) identification of evolution relations* occurring among metamodels stored in a repository (see the evolution labeled relations in Fig. 5); *ii) analysis of the detected evolution relations* with the aim of detecting those that have occurred to improve quality factors of the involved metamodels (labeled with qi_evolution in Fig. 5). The proposed approach is independent from the repository storing the input metamodels. The only requirement that need to be satisfied is that the metamodels under analysis have to be expressed in ECore since both the metamodel evolution and smell detection mechanisms are based on EMF technologies. In the following sections, the two phases of the proposed approach are presented separately.

3.1 Discovering Metamodel Evolutions

The Metamodel Evolution Discoverer component shown on the upper side of Fig. 5 implements the techniques we have conceived for detecting if two metamodels are related through some evolution. The component relies on the application of a similarity technique able to recognize to what extent the given metamodels are similar. Such a similarity measure is expressed in terms of a value ranging from 0 (the analysed metamodels are completely different) to 100 (the metamodels are exactly the same). Deciding which is the most appropriate similarity function is a difficult task that might depend on several factors not excluding semantic considerations about the application domains [CDRDR+20], and the size and quality of the available data sets [NDRRDR20].

Based on previous work [BDRDR+16] on the unsupervised clusterization of repositories, the approach proposed in Fig. 5 places reliance on the possibility of substituting

the employed similarity component depending on the context, and on the possibility of customizing the similarity threshold, which needs to be empirically identified to assess when two similar metamodels are actually one the *evolution* of the other.

In the context of this paper, we employed the *match-based similarity* function [BDRDR+16] that relies on a matching model calculated by means of an algorithm developed in ECL [Kol09]. It is defined as the total number of matched elements identified by the algorithm divided by the total number of elements contained in the analysed pair of metamodels. The implementation of such a similarity function consists of two main components i.e., *i)* the matching algorithm, and *ii)* the calculation of the similarity value depending on the matched elements. Listing 1 shows a fragment of the ECL-based implementation of the matching algorithm, which employs the Levenshtein[8] edit distance [Lev66] when calculating the name similarity of different elements including classes, attributes, and references (see e.g., lines 19,30 of Listing 1).

```
1 pre variables {
2   var simmetrics : new
        ↪ Native('org.epsilon.ecl.tools.textcomparison.simmetrics.SimMetricsTool');
3 }
4 rule EClass
5  match s : Source!EClass
6  with v : Target!EClass {
7  compare {
8   if(s.name.fuzzyMatch(v.name)){
9     return true;
10   }else{
11    return false;
12   }
13  }
14 }
15 rule EAttribute
16  match s : Source!EAttribute
17  with v : Target!EAttribute {
18  compare {
19   if(s.name.fuzzyMatch(v.name) and s.etype.isDefined() and v.etype.isDefined() and
        ↪ s.etype.name.fuzzyMatch(v.etype.name) and
        ↪ s.eContainingClass.name.fuzzyMatch(v.eContainingClass.name)){
20    return true;
21   }else{
22    return false;
23   }
24  }
25 }
26 rule EReference
27  match s : Source!EReference
28  with v : Target!EReference {
29  compare {
30   if(s.name.fuzzyMatch(v.name) and s.etype.name.fuzzyMatch(v.etype.name) and
        ↪ s.eContainingClass.name.fuzzyMatch(v.eContainingClass.name) and
        ↪ s.lowerBound==v.lowerBound and s.upperBound==v.upperBound){
31    return true;
32   }else{
33    return false;
```

[8]This functionality is loaded in the preamble (line 2) as a native Java library

```
34   }
35 }
36} ...
```

Listing 1 – Fragment of the ECL implementation of the matching algorithm

The outcome of such a phase is a matching model given as input to the calculateSimilarity Java method shown in Listing 2. The final similarity index is calculated as shown in Line 29 of Listing 2. It is important to remark that the proposed matching-based similarity mechanism relies on matching rules covering packages, metaclasses, and structural references. OCL constraints (specifying e.g., invariants and complex restrictions of the artefacts that will conform to the metamodel being specified) are not covered yet and their management is planned as future work.

```
1public int calculateSimilarity(IModel modeltocheck, MatchTrace matchModel) {
2
3  int simIndex=0;
4  Double classsim=0.0;
5  Double featuresim=0.0;
6  int nrclasses=0;
7  int nrfeats=0;
8
9  try {
10   nrfeats=modeltocheck.getAllOfKind("EStructuralFeature").size();
11   Collection<EClass> classes;
12   classes = (Collection<EClass>) modeltocheck.getAllOfKind("EClass");
13   nrclasses=classes.size();
14   for (EClass eClass : classes) {
15     if(matchModel.getMatch(eClass)==null) {
16          ...
17     }else {
18       classsim++;
19       for (EStructuralFeature f : eClass.getEStructuralFeatures()) {
20         if(matchModel.getMatch(f)==null) {
21                ...
22         } else {
23           featuresim++;
24         }
25        }
26      }
27   }
28 } catch (EolModelElementTypeNotFoundException e) { ...  }
29 simIndex= (int) ((((classsim*100)/nrclasses)+((featuresim*100)/nrfeats))/2);
30 return simIndex;
31}
```

Listing 2 – Fragment of the Java implementation of the matching-based similarity

In order to identify the threshold value, which permits us to asses when two metamodels are one the evolved version of the other, we considered a data set consisting of 31 metamodels (see Section 4.2). For all of the possible metamodel pairs, we applied the similarity function previously presented and manually checked the corresponding similarity values with respect to the actual content of the analysed metamodels. For instance, Table 1 shows a fragment of the obtained similarity matrix related to 4

Metamodel	(1)	(2)	(3)	(4)
taskmodel.ecore (1)	100	0	0	0
Task.ecore (2)	0	100	**66**	12
TaskManagement.ecore (3)	0	**77**	100	16
task.ecore (4)	0	7	7	100

Table 1 – Metamodel similarity matrix of the considered task management domain

metamodels contained in the considered dataset (including Tasks and TaskMagament of the motivating example), all related to the domain of task management. Table 1 has to be read to check if the metamodels in the columns are evolutions of the metamodels in the rows. The similarity values that are in bold represent metamodel evolution cases that can be considered as such according to a performed manual inspection of the metamodel contents. This means that for such a metamodel cluster, the similarity threshold for identifying evolution cases should be at least 66. More details about the identification of threshold values and the related experiments are given in Section 4.

3.2 Discovering Quality Improving Metamodel Evolutions

The reasons behind the need for metamodel evolutions are many including accommodating new requirements in the considered modeling language or improving the quality of the metamodel at hand. In software development, refactoring operations are also performed to solve *code smells* i.e., structural characteristics of the analyzed source code that might make the system hard to evolve and maintain, and that can be resolved by means of code refactoring [AFBZ12].

In [BDRIP19] we lifted the concept of smell to metamodeling with the aim of identifying metamodel design decisions that might have negative impacts on the quality of the metamodel being developed [BV10]. In particular, we presented an approach able to detect metamodel smells and to properly resolve them by means of a curated catalogue of smell definitions. Metamodel elements that can be affected by smells are structural elements constituting the domain model, e.g., packages, metaclasses, structural features. Among the smells that the approach is able to manage are the following ones:

- *SM1) Duplicated features in metaclasses*: it occurs when a feature with the same name and type is present in different metaclassess. In such a case, duplication of information might be induced by negatively affecting the *maintainability* and *reusability* of the considered metamodel [SHL+16];
- *SM2) Dead metaclass*: it occurs when there are metaclasses completely disconnected from the other modeling elements of the metamodel. According to [SHL+16], this smell can have a negative impact on at least the metamodel *understandability*;
- *SM3) Redundant container relation*: when defining container relations in EMF it is possible to define a corresponding eOpposite reference. If not specified, two unidirectional references are defined without providing the bidirectional navigation of the wanted containment relation. According to [BDRIP19], such a situation can represent a smell with a negative impact on the *complexity* and on the *maintainability* of the considered metamodel;
- *SM4) Classification by enumeration or by hierarchy*: model elements can be classified by means of enumeration or by hierarchies [BDRIP19]. Depending

on the particular case at hand, the *by enumeration* classification can be less appropriate than the *by hierarchy* one, and vice versa. In such cases, the smell can have a negative impact on the *complexity* of the considered metamodels that might contain more metaclasses than those actually needed [BDRIP19].

– *SM5) Concrete Abstract Metaclass*: depending on the particular situation being modeled, it can happen to have the superclass of a given class hierarchy being specified as concrete instead of abstract.

Intuitively, a quality improving evolution occurs when the smells that are identified in the initial version of the analysed metamodel do no longer exist in the evolved one. Consequently, given two subsequent versions of the same metamodel, quality improving evolutions can be detected by identifying and analyzing the smells occurring in the two metamodels. Such an approach is the viable solution when there is not a common agreement about a precise definition of metamodel quality, and consequently the assumption is that less smells occur the better. Thus, by relying on the availability of an extensible catalogue of metamodel smells, the approach shown in the lower side of Fig. 5 permits the analysis of the metamodel evolution relations, which are identified as explained in Section 3.1, and select those occurring because of changes that have been operated to improve quality factors of the initial version of the considered metamodel. By referring to the motivating example and the similarity values shown in Table 1, two possible metamodel evolutions have to be analysed, i.e., from Task.ecore to TaskManagement.ecore and vice-versa. However, for the sake of presentation only the direction from Task.ecore to TaskManagement.ecore is discussed with detail.

3.2.1 Smell detector

The `Smell Detector` shown in the lower side of Fig. 5 is based on Edelta [BDRIP17], a domain-specific language to specify an extensible catalogue of smells and to automatically detect their occurrences. Listing 3 shows a fragment of the Edelta source code implementing the detector for finding occurrences of the *classification by hierarchy* smell (i.e. all the hierarchies that have been used instead of enumerations).

```
def findClassificationByHierarchy(EPackage ePackage) {
  val classification= ePackage.allEClasses
      .filter[
        ESuperTypes.size == 1 &&
        EStructuralFeatures.empty &&
        isNotReferenced
      ]
      .groupBy[ESuperTypes.head].filter[p1, p2| p2.size > 1]
    classification.entrySet.forEach[
      logInfo["Classification by hierarchy: "
        + getEObjectRepr(key) + " - "
        + "subclasses["
        + value.map[getEObjectRepr(it)].join(",")
        + "]"
      ]
    ]
    return classification
}
```

Listing 3 – Edelta-based *classification by hierarchy* smell detector

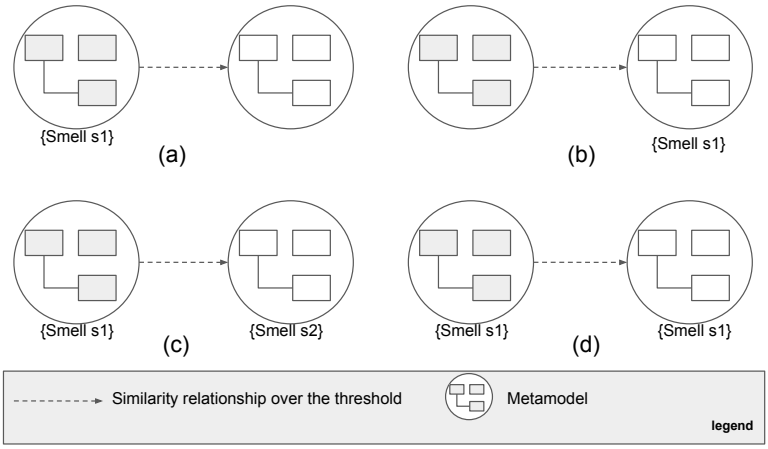

Figure 6 – Representative cases of smell occurrences in metamodel evolutions

The detector collects all the metaclasses of the metamodel and then it selects those with a super type, featureless, and not referenced by any metaclasses (see lines 2–7). Subsequently, it filters this list to get metaclasses having at least two sub-classes satisfying such characteristics. Lines 9–16 produce a log of the matched elements, and returns the found smell together with the meta-elements, which have been matched.

By referring to the Tasks metamodel in Fig. 4, all the smells previously discussed occur. In particular, SM1 occurs because of the attribute name contained in the Task and Project metaclasses. Because of the metaclass Label, the dead metaclass smell also occurs (SM2). The relations TasksModel.projects and Project.model contribute to the occurrence of the *redundant container relation* smell (SM3). The hierarchy of the Project metaclass, which is specialized as TeamProj and PersonalProj, can be considered as an occurrence of the smell *classification by hierarchy* presented above (SM4) and SM5 occurs in the concrete metaclass Task, with two concrete subclasses.

3.2.2 Smell analyzer

In general, the cases that can happen during metamodel evolutions with respect to metamodel smells can be classified as follows:

- *Detected smells have been resolved by the metamodel evolution*: As shown in Fig. 6.a, smells that are detected in the initial version of the considered metamodel have been resolved and do not occur in the new version of the metamodel. In that case, the shown metamodel evolution is considered as a quality improving;
- *New smells are introduced by the metamodel evolution*: Figure 6.b shows the case when new smells are introduced by the operated metamodel evolution. Even though the new version of the metamodel can be still considered as an evolution, different smells are introduced in the new metamodel with a possible reduction of the corresponding quality [BDRIP19];
- *Removal and addition of different smells*: Figure 6.c depicts the situation when existing smells are removed, but different ones get added during the evolution process;
- *Quality idempotent evolutions*: When operating metamodel evolutions, modelers can add new functionalities or in general modify the expressive power of the

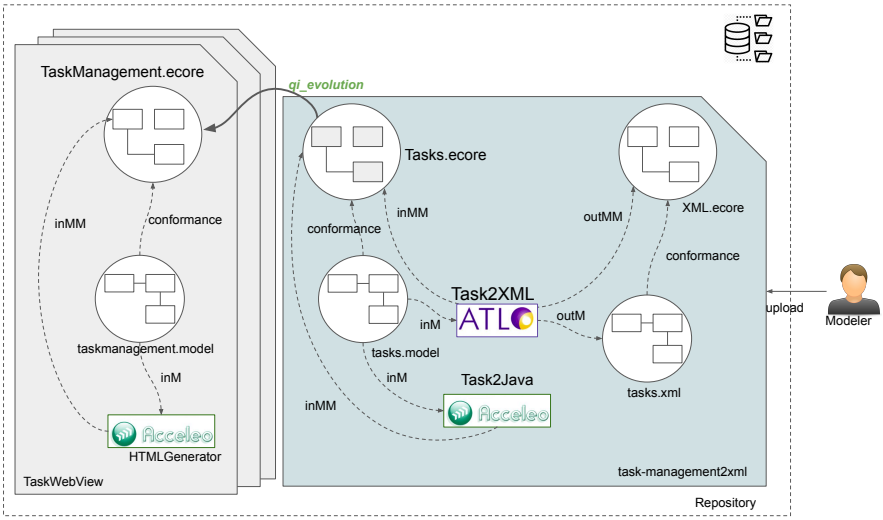

Figure 7 – Repository after the application of the proposed approach

considered metamodel without solving the smells that are in the initial version of the metamodel (see Fig. 6.d).

The `Smell Analyzer` component shown in Fig. 5 is in charge of recognizing such situations and refine the previously identified metamodel evolution relations when the case shown in Fig. 6.a occurs. In such a situation, the previously detected evolution relation is refined as *quality improving metamodel relation*. By referring to the motivating example in Fig. 4, the smells occurring in the Tasks metamodel do not occur in the evolved one as shown in Fig. 3. Thus, the evolution relation between such metamodels identified as presented in the previous section, can be considered as improving the quality. Accordingly, the explanatory repository shown in Fig. 2 has been extended by explicitly adding the quality improving relation, labeled with *qi_evolution*, as shown in Fig. 7.

As previously presented, the detection capabilities of metamodel evolutions and smells rely on the availability of an extensible library implementing the matching mechanisms and the definition of the smells of interest. The discussed cases mainly refer to structural and syntactical features of the considered metamodels. However, modeleres might have the needs of dealing with cases, which can be treated only if some semantic knowledge is also available. Renaming metaclasses with better names to best fit the considered application domain is an example of situations that require the codification of a "goodness function" measuring the appropriateness of the used names with respect to the considered application domain. However, we defer as future work the study of investigating the applicability of the proposed approach when such a kind of semantic knowledge is also needed.

4 Experiments

In this section, we discuss experiments that we have performed to assess the accuracy of the proposed evolution detection mechanisms. The section is organized as follows: Section 4.1 presents the research questions we wanted to answer by means of the performed experiments. Section 4.2 introduces the dataset we curated to serve as

input for the experiments. Section 4.3 and Section 4.4 present the obtained results, whereas Section 4.5 presents the threats to validity of the performed experiments.

4.1 Research Questions

By performing the experiments, we aim at answering the following research questions:

- **RQ1:** *What is the impact of the similarity threshold on the accuracy of the mechanism for detecting metamodel evolutions?* To answer this question, we varied the value of the similarity threshold and calculated the corresponding precision and recall of the detection mechanisms applied on the considered datasets;
- **RQ2:** *What is the accuracy of the approach in detecting quality-improving metamodel evolutions?* The detection of quality improving metamodel evolutions is based on the ability of the approach to detect metamodel smells. To answer this question, we mutated input metamodels and applied the smell detection mechanism;

4.2 Dataset

The performed evaluation is based on a curated set of metamodels consisting of 31 items, which are part of a bigger dataset already considered for evaluating other works of the authors [NDRDR$^+$19, BDRDR$^+$16]. The dataset consists of metamodels covering different application domains and technologies i.e., task management, Petri net modeling, entity-relationships modeling, and different versions of Eclipse GMF. The selection criterion of the considered dataset was to choose metamodels with different size and complexity, and covering different application domains. The considered dataset is available for download[9] and Table 2[10] shows an overview of its contents and how each metamodel has been assigned to a cluster. Specifically in this context, a cluster is a group of metamodels showing similarities [BDRDR$^+$16].

Cluster	#Subjects	Size of the smallest subject	Size of the largest subject
Task management	4	14	111
Petrinet	9	10	42
Entity/Relationship	7	3	64
Eclipse GMF	6	169	190
GWPN	5	12	24

Table 2 – Overview of the considered dataset

4.3 RQ1: Impact of the similarity threshold value on the evolution detection accuracy

To answer RQ1, we applied the evolution detection approach presented in Section 3.1 to all the possible pairs of metamodels belonging to the 5 metamodel clusters of the considered dataset. For each cluster, we considered different thresholds in order to analyse how the accuracy of the evolution detection mechanism is correspondingly affected. For instance, Table 3 shows the produced similarity matrix of the metamodels

[9] https://github.com/gssi/metamodelsdataset-ECMFA2020

[10] The metamodel sizes have been calculated by summing the number of classes, structural features, and packages just to give an example of the considered metamodel complexities.

#	Subject	1	2	3	4	5	6	7	8	9
1	petrinet2.ecore	100	33	33	66	33	50	33	33	50
2	PetriNet.ecore	20	100	64	20	37	30	47	54	27
3	petrinet_extendable.ecore	10	41	100	10	20	20	35	20	15
4	PetriNets.ecore	48	25	25	100	43	37	43	43	25
5	petri_nets.ecore	25	47	37	47	100	50	70	80	35
6	PetrinetDsl.ecore	17	18	18	18	25	100	25	18	17
7	PetriNet_extended.ecore	15	24	26	28	34	15	100	40	15
8	PetriNetModel.ecore	20	52	40	36	58	30	68	100	26
9	petri.ecore	47	40	50	33	40	47	33	40	100

Table 3 – PetriNet cluster similarity matrix

belonging to the PetriNet cluster. By manually analysing all the metamodel pairs therein, the real evolution cases are six. Depending on the selected similarity threshold, it is possible to have the following results: *i)* the analyzed metamodel pair is considered to be linked by an evolution relation and it is confirmed by manual inspection (true positive, TP); *ii)* the metamodel pair at hand is an evolution relation but this is not confirmed by looking at their content (false positive, FP); *iii)* the two metamodels are not in evolution relation but the manual inspection does not confirm such a result (false negative, FN). Then, we compute *precision* (P) as $\frac{\#TP}{\#TP+\#FP}$, and *recall* (R) as $\frac{\#TP}{\#TP+\#FN}$. By considering 60 as a threshold value, 5 evolution cases are detected for the PetriNet cluster, with a consequent precision of 1 and recall 0.83. With lower threshold values the precision decreases, whether the recall decreases with higher values of the threshold.

	T=20		T=40		T=50		T=60		T=70		T=80	
Cluster	**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**	**P**	**R**
Task management	0.5	1	0.5	1	1	1	1	1	1	0.5	0	0
PetriNet	0.1	1	0.22	1	0.5	1	1	0.83	1	0.33	1	0.16
Entity/Relationship	0.14	1	0.16	1	0.3	1	1	0.5	1	0.5	1	0.5
Eclipse GMF	0.65	1	0.65	1	0.65	1	0.65	1	0.65	1	0.65	1
GWPN	0.5	1	0.6	1	0.7	1	0.7	1	0.8	1	1	0.8

Table 4 – Precision and Recall for different thresholds

Table 4 shows the precision and recall for different threshold values applied on the different metamodel clusters. According to the shown values, it is evident that the accuracy of the proposed evolution detection mechanism depends on the selection of the correct threshold value. Moreover, according to Table 4, the similarity threshold is sensible to the application domain and to the available datasets. Thus, in case of repositories consisting of heterogeneous metamodels, it is necessary to select a threshold value that maximizes the average precision and recall. Such an activity would be easier if the managed metamodels belong to the same application domain. However, by relying on our experience on classifying metamodels by means of machine learning (ML) techniques [NDRDR+19], we believe that the overall accuracy of the proposed evolution detection mechanism can be improved by employing machine learning techniques. The architecture of the approach shown in Fig. 5 is open to such improvements. In that particular case, it would mean replacing the Metamodel Similarity Calculator component with a ML-based one. However, this is up to future work.

4.4 RQ2: Accuracy of the detection mechanism for quality improving relationships

To answer RQ2, our evaluation considered the metamodel clusters mentioned in the previous section. For each metamodel in the cluster, we randomly applied metamodel

Kind	Meta-model modification
Additive	Add obligatory / non-obligatory metaclass Add obligatory / non-obligatory metaproperty Generalize metaproperty Pull metaproperty Extract abstract / non-abstract superclass
Subtractive	Eliminate metaclass Eliminate metaproperty Push metaproperty Flatten hierarchy Restrict metaproperty
Updative	Rename metaelement Move metaproperty Extract/inline metaclass

Table 5 – Metamodel change operators used in the evaluation

mutations by systematically applying the metamodel modifications shown in Table 5. Such changes consist of the modifications presented in [CDREP08]. Our aim was to generate many *slightly* different variants of the original metamodel. Finally, we assessed whether a metamodel mutant is a *quality improving* evolution of the initial metamodel.

To determine if a metamodel mutant is of higher quality than the initial version of the analyzed metamodel, we adopted the quality assessment approach presented in [BDD+19] as an oracle: if the measured quality value of the mutant is higher than the initial metamodel, it means that the considered evolution is a quality improving one, not otherwise. Altogether, for each meta-model mutant, we may obtain one of the following results: *i)* the mutant is a quality improving evolution and it is correctly detected by the proposed approach (true positive, TP); *ii)* the mutant is not a quality improving evolution but it is detected as a such (false positive, FP); *iii)* the mutant is a quality improving evolution but the proposed approach considers it as not (false negative, FN). Then, we compute *precision* and *recall* as also done for answering RQ1.

Table 6 shows a fragment of the obtained results by focusing on the metamodels related to *task management* and *Petri net* modeling. For each metamodel, the table shows the smells that occur both on the initial version of it and on the corresponding mutants (see the columns Detected smells). The number of mutants that have been obtained for each subject has been randomly generated in the range $[1, 5]$. The column Quality Variation shows if the quality of the obtained mutant has increased ($\uparrow$), decreased ($\downarrow$), or unchanged ($=$) according to the oracle. The Quality Improvement column shows if, according to the proposed approach, a quality improving relationship can be established between the considered subject and the corresponding mutant. The column Result puts in relation the outcomes of the approach with those of the oracle.

There are some *false positives*, meaning that for some mutants the approach wrongly identified evolution cases. After a manual investigation, we discovered that such cases occurred when the mutations introduced additional occurrences of existing smells by nullifying the effects of having completely removed other smells. For instance, this is what happened for the mutant #2 of the tasks metamodel. For three cases the approach wrongly considered evolutions as not quality improving (*false negatives*) as for instance the mutant #3 of the PetriNets metamodel. According to the proposed detection mechanism, the fact that the operated mutation does not introduce any smells to the initial subject would mean that the evolution is not quality improving. However, in those particular cases, the mutants have been obtained by removing some structural features that induced an enhancement of the metamodel qualities. Nevertheless, the overall accuracy of the approach is still good (see the precision and recall values in Table 6) even though there is room for improvement as discussed in the next section.

Subject						Mutants						Quality Variation (Oracle)	Quality Improvement (Approach)	Result
Metamodel	Detected smells					id	Detected smells							
	S 1	S 2	S 3	S 4	S 5		S 1	S 2	S 3	S 4	S 5			
taskmodel	√	√			√	1	√	√				↑	Yes	TP
						2	√	√				↓	Yes	FP
						3	√	√			√	=	No	TN
						4	√	√			√	=	No	TN
						5						↑	Yes	TP
tasks	√	√	√	√	√	1	√	√	√	√	√	=	No	TN
						2	√	√	√		√	=	Yes	FP
						3	√	√	√		√	↓	Yes	FP
						4						↑	Yes	TP
TaskManagement						1	√					↓	No	TN
						2						=	No	TN
Tasks	√			√	√	1	√			√	√	=	No	TN
						2	√				√	↓	Yes	FP
						4	√			√	√	=	No	TN
						5						↑	Yes	TP
petrinet2	√					1						↑	Yes	TP
						2	√	√				↓	No	TN
						3						↑	Yes	TP
PetriNet	√					1	√			√		↓	No	TN
						2	√			√		↓	No	TN
						3						↑	Yes	TP
						4	√					↓	No	TN
Petrinet_extendable	√					1	√	√				↓	No	TN
						2	√					↓	No	TN
						3						↑	Yes	TP
						4	√					=	No	TN
PetriNets						1						↓	No	TN
						2						=	No	TN
						3						↑	No	FN
petri_nets						1		√				↓	No	TN
						2	√					↓	No	TN
						3						=	No	TN
						4	√	√				↓	No	TN
PetrinetDsl	√					1	√					↓	No	TN
						2	√	√				↓	No	TN
						3	√					=	No	TN
						4	√	√				↓	No	TN
						5						↑	Yes	TP
petrinet_extended						1						↓	No	TN
						2						=	No	TN
						3						↓	No	TN
PetrinetModel						1						↑	No	FN
						2			√			↓	No	TN
						3						=	No	TN
petri	√					1	√					↓	No	TN
						2	√					↓	No	TN
						3	√					=	No	TN
						4						↑	Yes	TP
# True Positive (TP)														10
# False Positive (FP)														4
# False Negative (FN)														3
Precision														0,7143
Recall														0,7692

Table 6 – Fragment of the experiment results

4.5 Threats to validity

In this section, potential threats to validity associated with the experimental validation are discussed. In particular, we distinguish threats among internal and external validity, as follows.

Internal validity. Such threats are the internal factors that could have influenced the final outcomes of the preformed experiments. We have used a relatively low number of metamodels. The reason is that we wanted to operate manual checks on the obtained

results. However, we considered different sets of metamodels with the aim of covering different domains and metamodels with different structural characteristics. Another threat to internal validity is that the evaluation for RQ2 has used the quality assessment approach in [BDD+19] as oracle. To avoid distortions on the evaluation results due to possible errors in our oracle, we have manually checked dubious cases and have not found any incorrect result. We believe that by increasing the number of detectable smells, the accuracy of the approach can increase even though providing evidence of that is up to future work.

External validity. The main threat to external validity concerns the generalizability of our findings, i.e., whether they would still be valid outside the scope of this study. We attempted to moderate the threat by considering different kinds of metamodels that are of different size and cover different domains. However, it is essential to evaluate the approach by considering a bigger dataset. Also, this task is considered as a future work. Moreover, the metamodel mutations that have been operated for the presented evaluation might not reflect all the evolution situations that can occur in practice. Unfortunately, differently from source code development that can rely on advanced platforms like GitHub, we do not have the availability of similar open infrastructures to store and share with the community the history of modeling artifacts including metamodels. To the best of our knowledge, model mutation is a technique, which is commonly used to artificially create artifacts that are needed for performing experiments like those presented in this paper.

5 Related Work

This section discusses relevant works that are related to *megamodels and recovery approaches*, *understanding metamodel evolutions*, and *metamodel matching and artifact similarity*.

Megamodels and recovery approaches The adoption of megamodeling was initially promoted in [BJV04] to support the management of different kinds of modeling artifacts. Megamodels are also used to orchestrate the subsequent applications of modeling tasks e.g., transformations, querying, merging and constraint checking (see e.g., [BJRV05]). In the context of model repositories, in [KJW+12] the authors propose the adoption of a layer on top of heterogeneous repositories to get a homogenous and uniform way to access the system by means of model operations specified in a domain-specific language. Megamodeling is discussed for MDE technologies (including EMF, ATL, and Xtext) in [HHH+17], but reverse engineering is not leveraged, despite being stated as a direction for future work. In [Ste20] authors propose the adoption of build systems in combination with a megamodel to restore desired consistency relationships between models (in case of modification). Further than our previous work [DRDRH+19], to the best of our knowledge there is no related work on megamodels that employ heuristics to identify model elements and to recover relationships among them.

Understanding metamodel evolutions Over the last decade, undertanding metamodel evolutions has been the subject of intense research especially to deal with the coupled evolution problem of metamodels and related modeling artifacts including models, model tranformations, and code generators [DRIP11]. In [VWV12, KHB+16] the authors deal with the problem of metamodel evolutions with the main goal of reconstructing complex evolution between two versions of the same metamodel. Similarly, in [KHB+16] the authors propose an approach to detect complex changes from

sequences of atomic ones operated by users. Differently from the work proposed in this paper, such approaches take as input two versions of the same metamodel that are already known to be one the evolved version of the other. In this paper, we focus on understanding if given any pair of metamodels, a quality improving evolution relationship can be established between them.

Metamodel matching and artifact similarity Metamodel matching and similarity underpin different approaches that aim at automating the migration of models in response to metamodel evolution. In particular, matching approaches try to automatically derive a model migration from the difference between two metamodel versions. A detailed survey of available model migration approaches is presented in [HKB17]. Different techniques and similarity functions are available for calculating the similarity value between two input metamodels [FHLN08, MGMR02]. In [FHLN08] the authors propose a technique that converts the input metamodels into graphs and apply on them the similarity flooding algorithm [MGMR02]. As already mentioned, identifying the right similarity function for the problem at hand is a difficult task. In this paper, we relied on that successfully employed in the context of model repositories and in particular for supporting the clusterization of metamodels [BDRDR$^+$16]. Calculating the similarity of software artifacts is of interest also in the domain of software development in general. Recently, in [NDRRDR20, NRRD20] the authors proposed an approach to classify open source projects (OSS) that are stored in software repositories like GitHub and Maven. The used similarity approach is based on a graph-based representation of the considered OSS ecosystem, which is then used to feed a collaborative-filtering algorithm to provide developers with useful recommendations. We see promising applications of similar techniques to support the detection of evolving metamodels. We intend to pursue this as a future work.

6 Conclusions and Future Work

Model-driven projects are typically stored in model repositories without any high-level description that might help modelers understand the structure and the roles of the contained artifacts. Thus, given a project of interest, modelers are supposed to manually explore it and to figure out the kinds of contained artifacts and how they are interrelated. Consequently, reuse possibilities of already developed model-driven projects are largely limited. MDEprofiler is an approach based on megamodels to enable the recovery of the structure of model-driven projects, which are thus represented as typed nodes and relationships among them. However, understanding the evolution of the stored metamodels is not supported by MDEprofiler as presented in [DRDRH$^+$19]. In this paper, we increased the discovery capabilities of MDEprofiler by means of an automated approach able to detect metamodel evolutions by distinguishing those that have been operated to improve the quality of the initial version of the considered metamodels. The approach has been validated by means of a data set consisting of 31 metamodels. As future work, we plan to apply the approach on a bigger dataset and to other kinds of modeling artifacts including models, transformations, and code generators. Furthermore, we plan to investigate the application of machine learning techniques to increase the accuracy of the proposed approach, as already done by the authors to manage similar problems (e.g., see [NRRD20, NDRDR$^+$19, NDRRDR20]).

References

[AFBZ12] F. Arcelli Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *The Journal of Object Technology*, 11(2):5:1, 2012.

[BDD+19] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. A Tool-Supported Approach for Assessing the Quality of Modeling Artifacts. *Journal of Visual Languages and Computing*, 51:173–192, 2019.

[BDRDR+16] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. Automated clustering of metamodel repositories. In *Advanced information systems engineering*, pages 342–358. Springer Int. Publishing, 2016.

[BDRIP17] L. Bettini, D. Di Ruscio, L. Iovino, and A. Pierantonio. Edelta: An approach for defining and applying reusable metamodel refactorings. In *MODELS (satellite events)*, pages 71–80, 2017.

[BDRIP19] L. Bettini, D. Di Ruscio, L. Iovino, and A. Pierantonio. Quality-driven detection and resolution of metamodel smells. *IEEE Access*, 7:16364–16376, 2019.

[BJRV05] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *European MDA workshops MDAFA 2003 and MDAFA 2004, revised selected papers*, volume 3599 of *LNCS*, pages 33–46. Springer, 2005.

[BJV04] J. Bézivin, F. Jouault, and P. Valduriez. On the need for megamodels. In *Procs. of 19th annual ACM Conf. on object-oriented programming, systems, languages, and applications*, pages 1–9, 2004.

[BV10] Manuel F. Bertoa and Antonio Vallecillo. Quality attributes for software metamodels. In *QAOOSE'10 workshop proceedings*, 2010.

[CDRDR+20] A. Capiluppi, D. Di Ruscio, J. Di Rocco, P. T. Nguyen, and N. Ajienka. Detecting Java Software Similarities by using Different Clustering Techniques. *Information and Software Technology*, February 2020.

[CDREP08] A Cicchetti, D. Di Ruscio, R Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *12th Int. IEEE enterprise distributed object computing Conf., EDOC 2008*, pages 222–231. IEEE Computer Society, 2008.

[DRDRH+19] J. Di Rocco, D. Di Ruscio, J. Haertel, L. Iovino, R. Lämmel, and A. Pierantonio. Understanding MDE projects: megamodels to the rescue for architecture recovery. *Software & Systems Modeling*, July 2019.

[DRIP11] D. Di Ruscio, L. Iovino, and A. Pierantonio. What is needed for managing co-evolution in MDE? In *Procs. of the 2nd Int. workshop on model comparison in practice*, IWMCP '11, pages 30–38. ACM, 2011.

[FHLN08] J.-Rémy Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel matching for automatic model transformation generation. In

Model driven engineering languages and systems, volume 5301 of *LNCS*, pages 326–340. Springer, 2008.

[Her11] M. Herrmannsdoerfer. *Evolutionary Metamodeling*. PhD thesis, 2011.

[HHH+17] J. Härtel, L. Härtel, M. Heinz, R. Lämmel, and A. Varanovich. Interconnected linguistic architecture. *The Art, Science, and Engineering of Programming Journal*, 1(1), 2017.

[HKB17] R. Hebig, D. E. Khelladi, and R. Bendraou. Approaches to Co-Evolution of Metamodels and Models: A Survey. *IEEE Transactions on Software Engineering*, 43(5):396–414, May 2017.

[HRW09] Christian Hein, Tom Ritter, and Michael Wagner. Model-driven tool integration with modelbus. In *Workshop future trends of model-driven development*, pages 50–52, 2009.

[KC13] B. Karasneh and M. RV Chaudron. Online Img2UML repository: An online repository for UML models. In *EESSMOD@ MoDELS*, pages 61–66, 2013.

[KDRPP09] D. S Kolovos, D. Di Ruscio, A. Pierantonio, and R. F Paige. Different models for model matching: An analysis of approaches to support model differencing. In *2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6. IEEE, 2009.

[KH10] M. Koegel and J. Helming. EMFStore: a model repository for EMF models. In *2010 ACM/IEEE 32nd Int. Conf. on software engineering*, volume 2, pages 307–308, 2010.

[KHB+16] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M-P Gervais. Detecting complex changes and refactorings during (Meta)model evolution. *Information Systems*, 62:220 – 241, 2016.

[KHL+10] M. Koegel, M. Herrmannsdoerfer, Y. Li, J. Helming, and J. David. Comparing state-and operation-based change tracking on models. In *2010 14th IEEE Int. Enterprise Distributed Object Computing Conf.*, pages 163–172. IEEE, 2010.

[KJW+12] W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot. MoScript: A DSL for querying and manipulating model repositories. In *Proc. SLE 2011*, volume 6940 of *LNCS*, pages 180–200. Springer, 2012.

[KMB+08] R. Kutsche, N. Milanovic, G. Bauhoff, T. Baum, M. Cartsburg, D. Kumpe, and J. Widiker. Bizycle: Model-based interoperability platform for software and data integration. *Procs. of the MDTPI at ECMDA*, 430, 2008.

[Kol09] D. S. Kolovos. Establishing correspondences between models with the epsilon comparison language. In *Model driven architecture - foundations and applications*, pages 146–157. Springer, 2009.

[Lev66] VI Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, 1966.

[MGMR02] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In *Procs. 18th Int. Conf. on data engineering, 2002*, pages 117–128, 2002.

[NDRDR+19] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, A. Pierantonio, and L. Iovino. Automated Classification of Metamodel Repositories: A Machine Learning Approach. In *IEEE / ACM 22nd Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, page 12, 2019.

[NDRRDR20] P. T. Nguyen, J. Di Rocco, R. Rubei, and D. Di Ruscio. An automated approach to assess the similarity of GitHub repositories. *Software Quality Journal*, February 2020.

[NRRD20] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, and M. Di Penta. CrossRec: Supporting software developers by recommending third-party libraries. *Journal of Systems and Software*, 161:110460, March 2020.

[PMR16] 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*, pages 272–280, January 2016.

[Sch06] D. C. Schmidt. Guest noeditor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, February 2006.

[SHL+16] M. Strittmatter, G. Hinkel, M. Langhammer, R. Jung, and R. Heinrich. Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel. In *CEUR workshop proceedings*, volume 1706, pages 30–39. CEUR, 2016.

[Ste18] P. Stevens. Towards sound, optimal, and flexible building from megamodels. In *Procs. of the 21th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems*, pages 301–311, 2018.

[Ste20] P. Stevens. Maintaining consistency in networks of models: bidirectional transformations in the large. *Software and Systems Modeling*, 19(1):39–65, 2020.

[VSB+13] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen. *Model-driven software development: technology, engineering, management.* John Wiley & Sons, 2013.

[VWV12] S. D. Vermolen, G. Wachsmuth, and E. Visser. Reconstructing complex metamodel evolution. In *Software language engineering*, pages 201–221. Springer, 2012.

About the authors

Lorenzo Bettini is an Associate Professor in Computer Science at DISIA Dipartimento di Statistica, Informatica, Applicazioni 'Giuseppe Parenti', Università di Firenze, Italy, since February 2016. Previously, he was an Assistant Professor (Researcher) in Computer Science at Dipartimento di Informatica, Università di Torino, Italy. His research interests cover design, theory and implementation of programming languages (in particular Object-Oriented languages and Network aware languages). Contact him at `lorenzo.bettini@unifi.it`, or visit `http://www.lorenzobettini.it`.

Davide Di Ruscio is Associate Professor at the University of L'Aquila. His main research interests include software engineering, and several aspects of Model Driven Engineering including domain-specific languages, model transformations, and model evolution. He has published more than 100 papers in various journals, conferences and workshops on such topics. Over the last decade he has worked on several European projects by contributing the application of MDE in different application domains like service-based software systems, autonomous systems, and open source software (OSS). Contact him at `davide.diruscio@univaq.it`, or visit `http://people.disim.univaq.it/~diruscio/`.

Ludovico Iovino is Assistant Professor at the GSSI – Gran Sasso Science Institute, L'Aquila - in the Computer Science department. His interests include Model Driven Engineering (MDE), Model Transformations, Metamodel Evolution, code generation and software quality evaluation. Currently he is working on model-based artifacts and issues related to the metamodel evolution problem. He has been included in program commitees of numerous conferences and in the local organisation of the STAF 2015 and iCities 2018 conferences, he organised also the models and evolution workshop at MODELS 2018. He is part of different academic projects related to Model Repositories, model migration tools and Eclipse Plugins. Contact him at `ludovico.iovino@gssi.it`, or visit `http://www.ludovicoiovino.com`.

Alfonso Pierantonio is Professor at the University of L'Aquila, Italy. His interests include Model-Driven Engineering with a specific emphasis on co-evolution problems, bidirectionality, and megamodeling. He has chaired a number of international conferences and organized numerous scientific events (including ICMT and STAF). He is in the editorial board of several scientific journals (including SoSyM and JOT). Contact him at `alfonso.pierantonio@univaq.it`, or visit `http://pieranton.io`.