International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-022-00646-2

GENERAL

Regular

Supporting safe metamodel evolution with edelta

Lorenzo Bettini[1] · Davide Di Ruscio[2] · Ludovico Iovino[3] · Alfonso Pierantonio[2]

Accepted: 6 January 2022
© The Author(s) 2022

Abstract

Metamodels play a crucial role in any model-based application. They underpin the definition of models and tools, and the development of model management operations, including model transformations and analysis. Like any software artifacts, metamodels are subject to evolution to improve their quality or implement unforeseen requirements. Metamodels can be defined in terms of existing ones to increase the separation of concerns and foster reuse. However, the induced coupling can give additional evolution complexity, and dedicated support is needed to avoid breaking metamodels defined in terms of those being changed. This paper presents a tool-supported approach that can automatically analyze the available metamodels and alert modelers in case of change operations that can give place to invalid situations like dangling references. The approach has been implemented in the Edelta development environment and successfully applied to metamodels retrieved from a publicly available Ecore models dataset.

Keywords Model-driven engineering · Metamodel evolution · Parallel evolution · Safe evolution

1 Introduction

In Model-Driven Engineering (MDE), metamodels are typically used to encode the knowledge of analyzed domains, which get formalized in terms of identified concepts, relationships, and constraints. Metamodels are created by capturing the concepts and structures of a particular domain to construct models of that domain [41]. UML is an example of metamodel for general modeling languages. Still, we can also find examples of domain-specific modeling languages created to address specific software engineering domains, e.g., Autosar for automotive [14], WebRatio [1] for web applications development. The definition of metamodels is preparatory to the development of inherently different artifacts, including models, metamodels, model transformations, and code generators. All of them contribute to the definition of modeling ecosystems [23]. To foster separation of concerns and reuse, metamodels can be defined in terms of existing ones. By including elements, which are encoded in already defined metamodels, it is possible to avoid duplicating the same elements across different metamodels [31]. Even though such reuse practices can speed up the definition of new metamodels, they can give place to complex dependencies, with subsequent increases of metamodel couplings and negative impacts on usage flexibility.

Like any other software artifacts, metamodels are subject to evolutive operations because of various reasons [19] such as addressing unforeseen requirements [32], implementing design improvements [20], or removing bad smells [6,36]. Examples of metamodel changes include renaming a concept, moving a property from a metaclass to another, and redirecting a relation between two metamodel elements. Moreover, atomic changes can compose more refined and complex patterns, giving place to what we call metamodel evolution. Over the last years, several approaches have been conceived to define and apply metamodel changes (see, e.g., [5,40]). In the Eclipse Modeling Framework (EMF)

✉ Ludovico Iovino
ludovico.iovino@gssi.it

Lorenzo Bettini
lorenzo.bettini@unifi.it

Davide Di Ruscio
davide.diruscio@univaq.it

Alfonso Pierantonio
alfonso.pierantonio@univaq.it

[1] DiSIA, University of Florence, Florence, Italy

[2] Information Engineering, Computer Science and Mathematics Department, University of L'Aquila, L'Aquila, Italy

[3] Gran Sasso Science Institute, L'Aquila, Italy

Published online: 03 February 2022

[35], metamodels are specified with a dedicated language called Ecore. In EMF, predefined APIs can programmatically manipulate Ecore models through Java code. Even though such techniques permit modelers to specify the evolution of individual metamodels, they do not provide first-class mechanisms to manage *metamodel dependencies* during the specification and application of evolution operators. Consequently, by evolving a given metamodel, e.g., by removing a metaclass `MC`, it can happen that the validity of other metamodels is broken, e.g., those that refer to the removed `MC`. Existing approaches deal with this kind of inconsistencies by performing *model repair*, i.e., the activity of restoring the validity of the corrupted models. Nevertheless, in this case, if the inconsistency is created by evolving or refactoring a metamodel used as an external resource, it could be problematic to restore the validity of the affected artifacts [4]. This problem is common in different domains, as in the case of software package dependencies [12] or model transformation dependency analysis [30].

This paper proposes a tool-supported approach to define and apply *safe metamodel evolutions*, which do not break the validity of metamodels that depend on the elements being changed. In case of possibly unsafe changes, modelers get early alerts from the development environment. The approach has been implemented in Edelta [7], which is a dedicated framework to evolve Ecore models. The framework comes with a dedicated DSL providing developers with constructs to define complex evolution operators invoked on a subject metamodel. Until now, Edelta has been used on a single subject metamodel. In this paper, we show the extended version of Edelta that supports the safe co-evolution of multiple dependent metamodels simultaneously. This paper extends the Edelta framework [5,7] from both a methodological and technological point of views. In particular, the Edelta framework presented in this paper includes a new component for dependencies analysis of metamodel repositories, a graphical view to represent the result of the analysis in terms of a graph-based representation, and an Edelta template generator. The modeler can use generated Edelta specifications to start a new Edelta program, correctly including the involved metamodels. To show the effectiveness of the proposed approach, we have used Edelta to apply representative evolutions on a dataset of metamodels retrieved from GitHub. Edelta is an open-source project available at https://github.com/LorenzoBettini/edelta.

Structure of the paper. The paper is structured as follows: Section 2 shows two motivating examples in which dependant metamodels are shown. Section 3 makes an overview of Edelta and highlights its limitations to avoid unsafe metamodel evolution. Such limitations are addressed by the approach proposed in Sect. 4. Section 5 presents the experiments that have been performed to assess the effectiveness of the proposed approach. Section 6 describes the related work, whereas Sect. 7 concludes the paper with some future directions.

2 Background and motivating examples

In MDE, metamodels formalize concepts and relationships of a given application domain and underpin the definition of modeling languages and model management operations. Metamodels define the abstract syntax of domain-specific modeling languages as shown in Fig. 1b, which depicts the modeling constructs representing persons and associated credit cards.

The definition of metamodels can be performed by importing existing ones as shown in Figure 1a, which represents the dependency occurring between the metamodels `WebApp` and `Persons`. The relation occurs because some metaclasses in `WebApp` refer to some elements in `Persons`. In such cases, the application of refactoring operations on used metamodels has to be carefully performed to avoid breaking the consistency of the whole ecosystem. For example, the `Persons` metamodel contains an instance of the *Dead Classifier* smell due to the metaclass `NameElement` (see Figure 1b), which is completely disconnected from the other elements of the metamodel [6]. In object-oriented design, similar situations are referred to as dead code or oxbow code [10].

The resolution of this smell is usually faced up with the removal of the indicted metaclass. Since the `Persons` metamodel does not use `NameElement`, this could be considered as a dead class, and it could be removed. However, this metaclass is being used by the metamodel `WebApp` as supertype for the classifiers with the `name` attribute, i.e., `WebApp` and `Service`. Thus, even though the considered smell resolution can be beneficial for the `Persons` metamodel, it is not safe concerning the whole ecosystem: it will lead to metaclasses with `null` supertype in the `WebApp` metamodel, raising a validation error as shown in the left-hand side of Fig. 1b.

Figure 2 shows an example involving the `Persons` and `WebApp` metamodels in a *cross-referencing* [9] scenario. In particular, the metamodel on the left is linked to the one on the right by using a *weaving model* [11]. In the weaving model, the relation is expressed by using the metamodel `Subscription` in the center. In this metamodel, we model the subscriptions by referencing a person and a credit card to a specific account. Moreover, on the top center, a constraint on the weaving metamodel is defined. This constraint checks that for the subscriptions defined in the model, if an account activated services of the web app, then at least a credit card has to be defined in the weaving. (Here, we could also check that the service is a paid service, but we use a more straightforward constraint for simplicity.) Cross-referencing implicitly involves several artifacts during the metamodel evolution

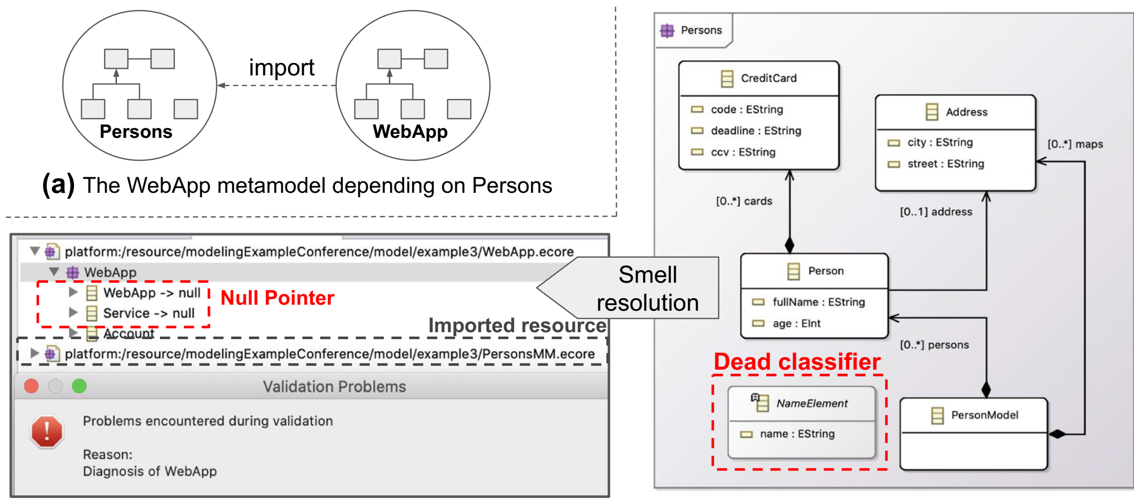


Fig. 1 `Persons` and `WebApp` metamodels: removing smell creates supertype inconsistency

activity. Indeed, if we inlined the metaclass `CreditCard` into `Person`, for creating an evolved metamodel as shown at the bottom of Figure 2, we would create an inconsistency in the weaving metamodel in the center.

Inlining a class is a well-known refactoring [33] in which a referenced metaclass is deleted, and the contained properties are moved to the old source of the relation. If we observe the `Subscription` metamodel in the center, the metaclass `Subscription` has three cross-references, one of which has type `CreditCard`, which will be a dangling reference in the evolved metamodel after the inlining. Moreover, the validation constraint defined on the weaving metamodel is also corrupted since it predicates over the credit card defined as a reference in the weaving metamodel. Also, in this example, an evolution considered on a single artifact created inconsistencies. In fact, the subjects of the evolution should be multiple to perform the changes correctly on various points, e.g., by first removing the future dangling reference before the inlining. The validation constraints should be adapted as well, but this regards the problem of co-evolution of constraints [18], which is out of the scope of this paper.

To summarize, Fig. 1a represents the *import* dependency, where metamodel `WebApp` imports the metamodel `Persons` as an external resource. In this case, the dependency exists because `WebApp` uses concepts of `Persons`, so `WebApp` may be corrupted if `Persons` is modified. Figure 2 represents the cross-reference dependency. The dependency is unidirectional from the weaving metamodel in the center to the linked metamodels. For this reason, if we modify `Persons` and/or `WebApp` we may create inconsistencies in the `Subscription` metamodel. One way to manage metamodel dependencies could be to use the EMF API [35] programmatically, but this presents some limitations. In fact, the EMF Java APIs do not check for inconsistencies during the execution, making it easy to create invalid resources. For instance, even if all the related metamodels are loaded in the same resource set, EMF does not check automatically that modifying a metaclass does not introduce a dangling reference. Such validity constraints on the Ecore models are checked only when the resources are saved to disk, which might be too late. Another difficulty dimension is related to the size and complexity of the metamodel repository being used. For instance, the dataset considered in Sect. 5 consists of approximately 2’400 metamodels, and it contains elements reused by more than 200 metamodels. Managing dependencies in such complex configurations is error-prone, and it demands dedicated support. In the following sections, we present an approach based on Edelta to address such issues.

3 Evolving metamodels with Edelta

In this section, we recall the main features of Edelta that we rely on to implement the approach presented in this paper. We also highlight the new features that were added to Edelta to support such an approach. Edelta [7] is a framework for refactorings and evolutions of EMF metamodels. Edelta consists of a runtime library and a DSL. It aims at providing EMF modelers with linguistic constructs for specifying basic metamodel changes (i.e., additions, deletions, and a few basic changes applied on meta-elements), and complex reusable metamodel changes by properly aggregating already declared ones in libraries (e.g., defining an operation for extracting a metaclass given a set of references).

Fig. 2 Evolution creating a pending reference type error

The Java API of Edelta is built on top of the standard EMF API, but it aims at providing a more statically safe set of operations that can be easily chained in a "fluent" style.

The Edelta DSL provides a syntax that is similar to Java but removes much "syntactic" noise. For example, terminating semicolons are optional and the parenthesis can be omitted in a method call expression with no arguments. The `return` keyword is also optional: the last expression will be returned. Edelta provides syntactic sugar for getters and setters: one can simply write `o.name` instead of `o.getName()` and `o.name = "..."` instead of `o.setName("...")`.

The Edelta DSL is statically typed, relying on type inference so that most types can be omitted in declarations. In particular, the type system of Edelta is completely compliant and interoperable with the Java type system, so that from an Edelta program we can access any Java type. This means that an Edelta program can seamlessly use any existing Java code and Java libraries. The Edelta compiler will translate Edelta programs into standard Java code, which uses the Edelta runtime library. In Edelta *lambda expressions* have the shape:

```
[ param1, param2, ... | body ].
```

When a lambda is the last argument of a method call, it can be moved out of the parenthesis; for example, instead of writing `m(..., [...])`, one can write `m(...)[...]`. When a lambda is expected to have a single parameter, the parameter can be omitted and it will be automatically available with the name `it`. In general, the symbol `it` acts as an implicit receiver, so, just like `this`, it can be omitted in method invocations.

Edelta provides a specific syntax to refer to Ecore elements in a statically typed way, `ecoreref(...)`. Indeed, Edelta programs refer directly to the classes inside of an Ecore model. Note that this approach works even in situations where the EMF Java model has not been generated at all. References to Ecore elements, such as packages, classes, data types, features, and enumerations, can be specified by their fully qualified name in an `ecoreref` expression using the standard dot notation, or by their simple name if there are no ambiguities (possible ambiguities are checked by the compiler).

All these features make Edelta programs much more compact than Java programs and much easier to read and maintain. The syntax of Edelta should be easily understood by Java programmers.

An Edelta program consists of a few parts, besides Java-like `import` statements (for importing Java types) and a Java-like `package` declaration (used for the generated Java code). First, existing EMF metamodels are imported using the syntax `metamodel` followed by the `EPackage`'s name. (The Ecore files are searched for in the classpath of the current project.) Then, existing Edelta libraries can be imported with the syntax `use ... as ...`. Such libraries can then be used in the current program just like standard Java objects (e.g., for method invocation). Some reusable functions can be defined with a syntax similar to Java methods (starting with the keyword `def`; the return type can be omitted and it is inferred from the operation's body). Such functions can be used in the same program or imported in other Edelta programs with the above-mentioned `use` syntax. Finally, actual evolution operations on a specific imported `EPackage` are specified with the syntax `modifyEcore`. For further details on the Edelta syntax, we refer the interested reader to [7].

The Edelta DSL is embedded in an Eclipse-based IDE, with all the typical IDE mechanisms, such as syntax highlighting, content assist, code navigation, quick-fixes, incremental building, error reporting, and also debugging. In particular, the Edelta editor provides a "live" development environment for evolving metamodels. This feature is particularly useful for the modelers who will receive immediate feedback on the evolved version of the metamodels in the IDE. Moreover, Edelta performs many static checks, also employing an interpreter that keeps track on-the-fly of the evolved metamodel, enforcing the correctness of the evolution right in the IDE, based on the flow of the execution of the evolution operations specified by the user. The Edelta Outline view shows the preview of the evolved metamodels, which is the result of the interpretation of the Edelta program. This way, modelers can immediately inspect the evolved metamodels before applying the actual evolutions. The interpretation is performed on an in-memory copy of the original metamodels, so modelers are free to experiment without affecting the original metamodels. These mechanisms allow for very fast development cycles since the "live" preview is available even without saving the program in the editor.

Finally, Edelta allows the users to easily introduce additional validation checks in their Edelta programs, which are taken into consideration by the Edelta compiler and the IDE. The Edelta refactoring library heavily relies on this feature, so that we can provide error and warning feedback directly from our Edelta reusable operations, without having to modify the Edelta compiler.

Edelta can be used in different ways, e.g., to directly apply metamodel evolutions or to programmatically exploit bad smells resolutions. These two applications have been explored in [5,7] and in [6], respectively. The above-mentioned Edelta refactoring library includes the bad smells resolutions.

In previous works [5,7] Edelta was used to evolve only a single metamodel. For the approach presented in this paper, we extended Edelta so that it uses all the imported metamodels when performing static checks (see next section).

For smell resolution, Edelta provides a mechanism based on three different components that work in synergy: the bad

smell finder (in charge of matching a bad smell in metamodels), the refactoring library (which includes, for example, the above-mentioned `inlineClass`), and the resolver. This last component associates a bad smell with an operation, automatically matching and resolving the found smell. All these components are implemented with the Edelta DSL.

As an explanatory example, Listing 1 shows the operations detecting the bad smell *Dead classifier* shown in Fig. 1. The implementation should be easy to read: we find the classifiers that do not refer to other classifiers (i.e., including data types) and that are not used by other classes. Note that, as said above, Edelta can access any existing Java types. In this case, we rely on the EMF `EcoreUtil` for finding cross-references and usage cross-references.

Listing 1 The functions for finding dead classifiers

```
1   def findDeadClassifiers(EPackage ePackage)
        {
2       ePackage.EClassifiers
3       .filter[isDeadClassifier]
4       .toList
5   }
6
7   def isDeadClassifier(EClassifier cl) {
8       cl.doesNotReferToClasses &&
9           cl.isNotReferredByClassifiers
10  }
11
12  def doesNotReferToClasses(EClassifier c) {
13      EcoreUtil.CrossReferencer.find(
            newArrayList(c))
14      .keySet
15      .filter(EClass)
16      .empty
17  }
18
19  def isNotReferredByClassifiers(EClassifier
        cl) {
20      EcoreUtil.UsageCrossReferencer
21      .find(cl, packagesToInspect(cl))
22      .empty
23  }
```

The Edelta utility function `packagesToInspect` retrieves all the `EPackages` in the current resource set so that we can inspect all the imported metamodels when this bad smell finder is used from within an Edelta program. This function is part of the extension of Edelta that is required for the goals of this paper. When all the dependant metamodels are correctly imported in an Edelta program, Edelta will be able to inspect them all when performing static checks. In Listing 2 we report the resolver associated with the dead classifier smell. This metamodel change simply removes the indicted metaclass by using the `EcoreUtil.remove` method. We observe that this resolver is particularly trivial since the smell is associated with an atomic operation and not with a complex evolution pattern.

Listing 2 The resolver for the bad smell *Dead classifier*

```
1   ...
2   def resolveDeadClassifiers(EPackage
        ePackage) {
3       finder.findDeadClassifiers(ePackage)
4       .forEach[EcoreUtil.remove(it)]
5   }
6   ...
```

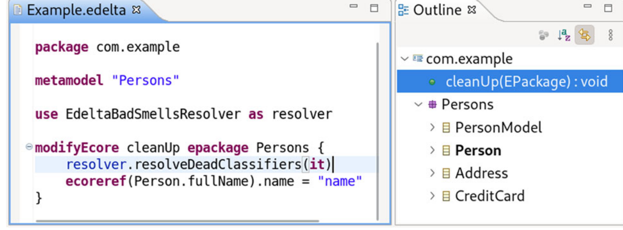


Fig. 3 The resolution for the smell dead classifier is matched since we imported only `Persons`

In Fig. 3 we show an Edelta program that executes the above-mentioned resolver for dead classifiers on the single imported metamodel `Persons` (see Fig. 1). The symbol `it` refers to the `EPackage` specified in the `modifyEcore`. (For demonstration, we also perform another basic operation, i.e., the rename of a feature.)

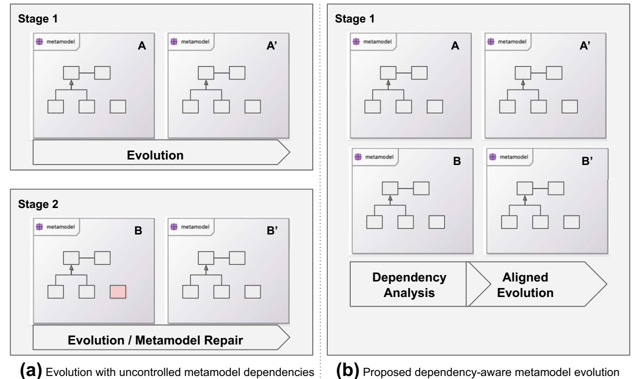
In this case, the dead classifier is matched, as can be seen from the Outline where the class `NameElement` is not present anymore. Indeed, since we imported only the metamodel `Persons`, Edelta correctly detects that `NameElement` is a dead classifier. However, as stressed in Sect. 2, the `WebApp` metamodel would then be invalid. The modeler should be aware of `WebApp` depending on the currently evolved imported metamodel `Persons`. If the modeler imported also `WebApp` in the Edelta program, then Edelta, extended with the new features like the above-mentioned `packagesToInspect`, would be able to avoid such problems right away. As shown later in Sect. 4.2, Edelta would not consider `NameElement` as a dead classifier if also `WebApp` was imported.

For these reasons, and to avoid such problems, we propose an approach based on aligned metamodel evolutions supported by an extension of the Edelta tool. We show that co-evolving metamodels in a dependency-aware manner is safer than evolving them as single units.

4 Safe metamodel evolutions with Edelta

In this section, we present an approach to deal with the issues discussed in the previous sections and that are due to metamodel dependencies, which are not managed during evolutions. In particular, as shown in the explanatory example shown in Fig. 4.a, two possibly related metamodels are singularly evolved in separated stages. If the maintenance of these metamodels is conducted in a way that *Stage 1* precedes *Stage 2* and the evolution of the metamodel in *Stage 1* touches elements cross-referenced in the metamodel in *Stage 2*, then the evolution might give place to inconsistencies.

Figure 4 b shows the way the proposed approach works: it supports the evolution of all the depending metamodels in a



(a) Evolution with uncontrolled metamodel dependencies (b) Proposed dependency-aware metamodel evolution

Fig. 4 Metamodel evolution phases

single stage. In particular, the application of metamodel operators is restricted depending on the occurring dependencies to reduce the risk of creating inconsistencies.

In the next sections, we describe the two main phases of the proposed tool-supported approach, i.e., *dependency analysis*, and *aligned evolution*. The former is in charge of automatically deriving a graph encoding the dependencies among all the metamodels under the availability of the user. The latter employs the created dependency graph to guide users while specifying metamodel evolutions with Edelta. Early alerts are raised in case of evolutions that might produce inconsistencies.

The original Edelta framework [5,7] has been extended for this application to include: *i)* a new component for dependency analysis of metamodel repositories, *ii)* a graphical view to represent the result of the analysis, *iii)* and an Edelta template generator. This generated Edelta specifications can be used to compose a new Edelta program, correctly including the involved metamodels, calculated by the analysis phase. The Edelta plugin has been extended to include these new components.

4.1 Dependency analysis

In this phase, the metamodel being evolved is analyzed with the goal of searching for cross-references or references to external resources. To this aim, the whole available metamodel repository is analyzed, and a model conforming to the metamodel shown in Fig. 5 is generated. It is inspired by the one presented in [13] and it allows us to represent model repositories[1] as graphs. In particular, a `Repository` can be represented as a graph that is composed of `Nodes` and `Edges`. `Nodes` can be model-based artifacts, e.g., models or metamodels. The attribute `highlighted` is used for visualization purposes, e.g., highlighting the node in the representation, as shown later. `Edges` are relationships between the artifacts. Possible specializations of the `Edge` metaclass are `Dependency` and `Conformance`. The former represents metamodel dependencies, whereas the latter enables the specification of conformance relations of models with the corresponding metamodels. Dependency edges are established between source (`src`) and a target (`trg`) elements and can be `bidirectional`.

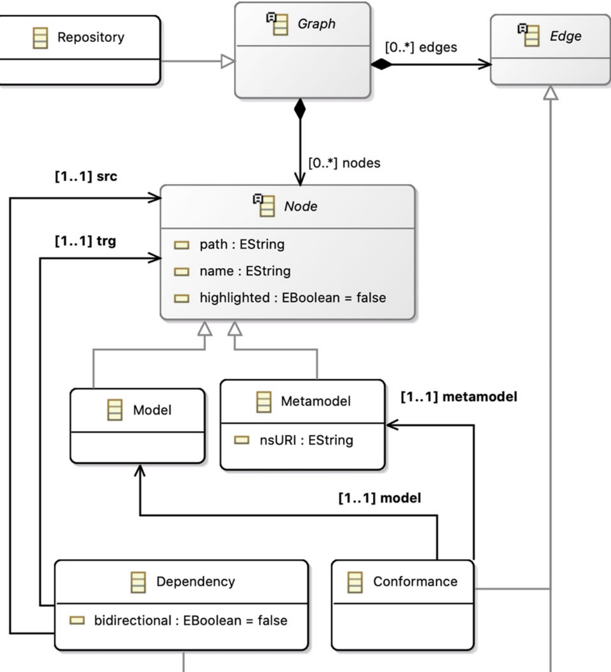


Fig. 5 Graph-based metamodel for dependency recovery

The analysis mechanism that generates dependency models conforming to the metamodel in Fig. 5 starts from the package of the metamodel that is the subject of evolution and analyses all the model elements to get possible references to other packages. The interesting parts of this phase are shown in Listing 3. This way, the modeler has a double help: *i)* the evolution program already includes the dependant metamodels (by means of the Edelta `metamodel` statements, Sect. 3), so that Edelta can perform its static checks on the evolved metamodels; *ii)* the modeler has an immediate and graphical feedback of the dependencies.

Listing 3 Part of dependency analyzer code

```
1  public static Collection<EPackage> usedPackages
       (EPackage ePackage) {
2      // keys: EObjects used by elements of this
          package
3      var map = EcoreUtil.CrossReferencer
4          .find(List.of(ePackage));
5      return
6          // only the used EClassifiers...
7          filterByType(map.keySet().stream(),
8                  EClassifier.class)
9          // ...to get their packages
10         .map(EClassifier::getEPackage)
```

[1] For simplicity, in this paper, repositories are considered as local projects (stored in workspaces) instead of online resources.

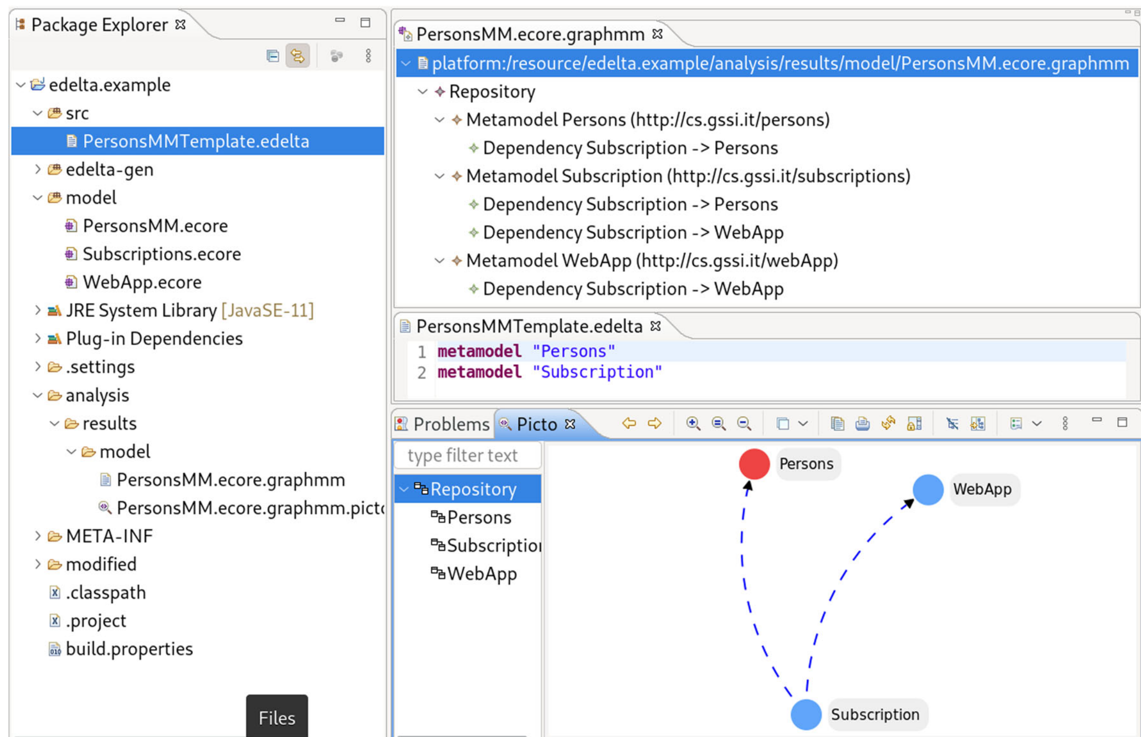


Fig. 6 The developed approach at work

```
11          .filter(Objects::nonNull) // safety
               condition
12          .filter(notEcore()) // skip the Ecore.
               ecore
13          // different from our package
14          .filter(p -> !Objects.equals(ePackage,
               p))
15          // just one occurrence of each used
               package
16          .collect(Collectors.toSet());
17 }
18
19 private static Predicate<? super EPackage>
20          notEcore() {
21     return p ->
22          !EcorePackage.eNS_URI.equals(p.getNsURI
               ());
23 }
24
25 private static <T, R> Stream<R>
26          filterByType(Stream<T> stream,
27                  Class<R> desiredType) {
28     return stream
29              .filter(desiredType::isInstance)
30              .map(desiredType::cast);
31 }
```

Then, by using the code of Listing 3, the procedure is iterated over all the packages of the repository, recursively, avoiding possible cycles. This way, we compute the closure of dependencies, both the outgoing and the incoming dependencies. During this procedure, we also build the model conforming to the metamodel in Fig. 5.

The dependency analysis process has been implemented in an Eclipse plugin, part of the Edelta distribution, as shown in Fig. 6. In particular, a contextual menu, enabled on the Ecore files, is provided, which invokes the above-mentioned analysis process on the selected Ecore file and on the other Ecore files in the same directory. The menu generates a model conforming to the metamodel in Fig. 5 in an output directory (`analysis/results`). The generated graph model is also coupled with a generated model to text transformation (the file with extension `picto`), which we do not show here. Such a transformation uses the Picto [24] view for rendering the graph of dependencies. This view represents the local repository in which the nodes are the metamodels and the edges are their dependencies. The subject metamodel is depicted in red (by using the `highlighted` attribute of the node), representing the metamodel of interest to the modeler. The view can be filtered, e.g., by selecting a class only and the metamodels connected to it will be shown.[2] The contents of the view can be easily navigated, rotated, and zoomed. The context menu automatically opens the generated graph model and the Picto view. Another context menu is provided to generate an initial Edelta template file (also shown in Fig. 6). This file imports the metamodels to be included in the evolution program, due to occurring dependencies. In the next section, we show the use of the generated templates to support aligned metamodel evolutions by considering the explanatory examples shown in Sect. 2.

[2] The view renders the HTML based on D3.js, https://d3js.org, Bootstrap, https://getbootstrap.com and other HTML and JavaScript technologies, which are taken directly from remote URLs. This means that an Internet connection is required for the view to be rendered correctly.

In Fig. 6 we show the example in Fig. 2 actualized in the tool. The two context menus described above have been executed on the Ecore file corresponding to the metamodel `Persons` (`PersonsMM.ecore`). In fact, its node is highlighted in red. The `Subscription` metamodel has two dependency links to the metamodels `Persons` and `WebApp`. This way, the view offers an immediate feedback w.r.t. the metamodel of interest in the repository, which in this case, for demonstrative purposes is quite simple, but in general it can include a large set of models, nodes, and dependencies.

The dependency analysis generates the graph model once, and it can be used as a cached representation until the repository is untouched. For a medium project like the one used in Sect. 5 consisting of ≈2’400 metamodels, the dependency analysis takes ≈700ms.

4.2 Use of generated edelta templates to evolve dependant metamodels

In this section, we show how Edelta can be used together with the dependency analysis tool, introduced in Section 4.1, to implement the proposed approach and achieve safe evolutions of interrelated metamodels.

In Fig. 7, we report a part of the Edelta program to evolve the metamodels `Persons` and `Subscription` of the example shown in Fig. 2. The `metamodel` imports are automatically generated in the template file, by using the contextual menu on the `PersonsMM.ecore` file (as described in Sect. 4.1, see also Fig. 6). The file has then been renamed to “Example.edelta.”

The modelers can then specify the evolution as in the remaining of the screenshot for instance. The metamodels must be imported together so that the scenario in Fig. 2 can be evolved avoiding the dangling cross-reference. Indeed, as mentioned in Sect. 3, we extended Edelta so that it uses the entire resource set, which contains the imported metamodels. This way, when performing validation checks, Edelta can immediately detect problems such as the mentioned dangling cross-references. For example, in Fig. 7, as soon as the modeler specifies the `inlineClass` refactoring, an error pops up: such a refactoring cannot be applied since it requires a single usage of the class to inline. Since Edelta has both metamodels in the resource set, it can detect such an ambiguity, avoiding a possible dangling reference if the class was inlined in the class `Person`. The refactoring `inlineClass`, which is part of the Edelta refactoring library, uses the mechanism for participating in the validation of Edelta programs mentioned in Sect. 3.

On the contrary, if we had not imported `Subscription`, the refactoring would succeed, as shown in Fig. 8. However, while the evolved metamodel `Persons` would still be valid,

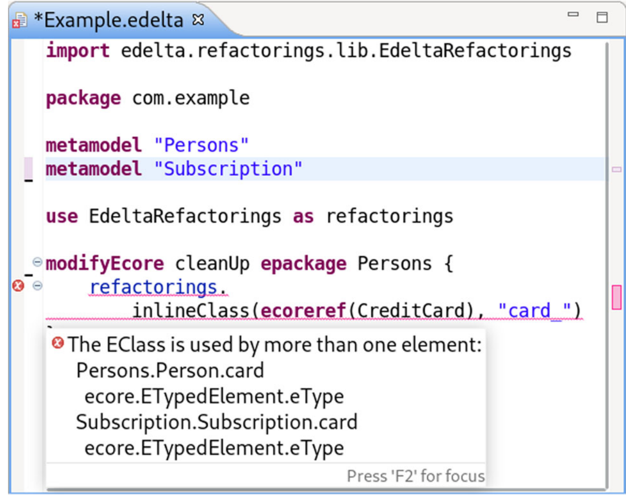


Fig. 7 The `inlineClass` refactoring shows an error since we imported also the `Subscriptions` metamodel that refers to `CreditCard`

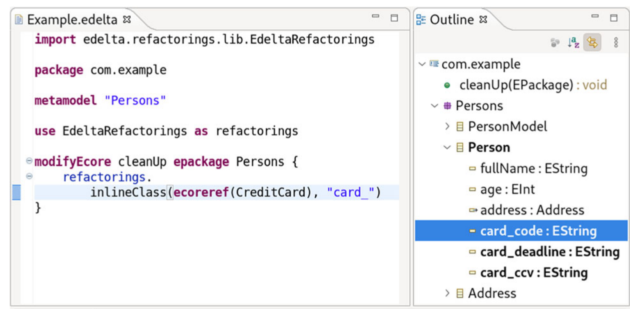


Fig. 8 The `inlineClass` refactoring succeeds since we imported only the `Persons` metamodel

the dependant metamodel `Subscription` would be corrupted by a dangling reference, as anticipated in Sect. 2.

Once the modeler is notified by the system about the problem she can decide how to fix it. For example, before applying `inlineClass`, the reference `card` to the class `CreditCard` (in the other metamodel) can be removed. Figure 9 shows such a situation. Note that we use the fully qualified name of the reference in the `ecoreref` to avoid the ambiguity with the homonymous reference in `Person`. Consequently, the `inlineClass` can be safely performed. Recall that Edelta interprets the current program on the fly, keeping the order of the statements into consideration. Note that the Outline view of the Eclipse IDE shows the preview of the evolved metamodels, where the elements that were modified are highlighted in bold: we can see that the class `CreditCard` disappeared, its features have been inlined in `Person` (with the specified prefix), and that the reference `card` has been removed from `Subscription`.

In Fig 10, we show an Edelta program that tries to apply the resolver for dead classifiers on the metamodel `Persons` (see Fig. 1) in a program where also the dependant metamodel

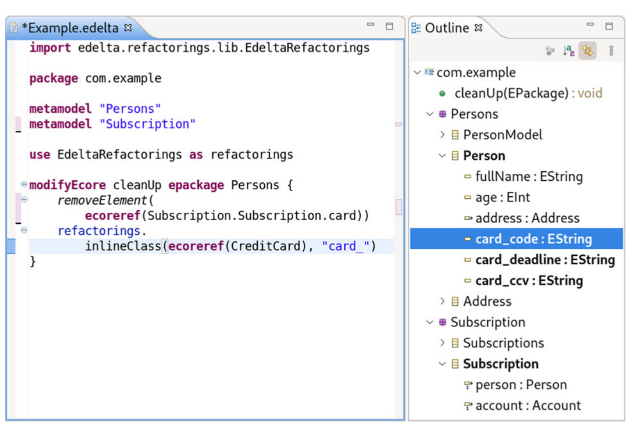


Fig. 9 The `inlineClass` refactoring succeeds since we first remove from `Subscription` the reference to `CreditCard`

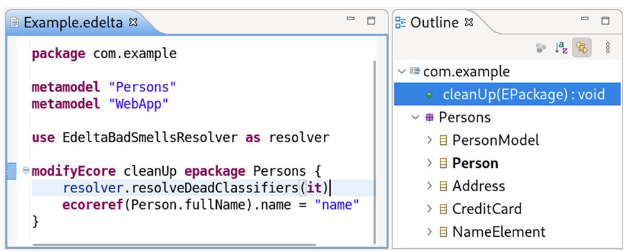


Fig. 10 The resolution for the smell dead classifier is NOT matched since we imported also `WebApp`: the `NameElement` is still there

`WebApp` is imported. Also in this example, the `metamodel` imports are automatically generated by clicking on the contextual menu enabled on the subject metamodel. Differently from what we shown in Fig. 3 (Section 3), the bad smell finder for the dead classifiers is not matched when the dependant metamodel `WebApp` is also imported: `NameElement` is used as a supertype in the dependant metamodel. Indeed, in the Outline, the `NameElement` is still present. Of course, in this case, no error is shown: the bad smell resolver simply did not detect any dead classifier.

The approach described in this paper is based on the abstract architecture reported in Fig. 11. Basically the developed tool is based on the Eclipse Modeling Framework as core for manipulating models. In particular we rely on Epsilon [25] for the visualization part of the dependency models. Epsilon is a family of languages for automating common model-based software engineering tasks, such as code generation, model-to-model transformation. We have used the Eclipse UI extension points to create the contextual menus described in Sect. 4.1.

The contextual menus impose the selection of a metamodel as subject from which the analysis is performed. The result of the Edelta template generator can be further refined and extended by using the Edelta editor for producing the evolved metamodel result of the evolution. The obtained model will be stored in the initial repository. The examples shown in this paper can be found at https://github.com/LorenzoBettini/edelta-safe-metamodel-evolution-examples.

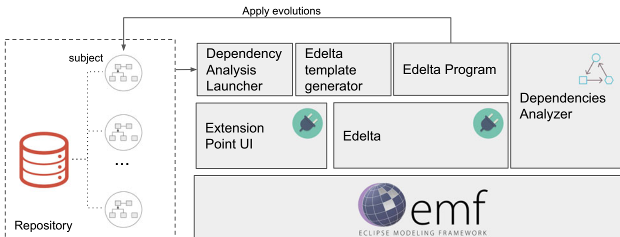


Fig. 11 Abstract architecture of the developed toolkit

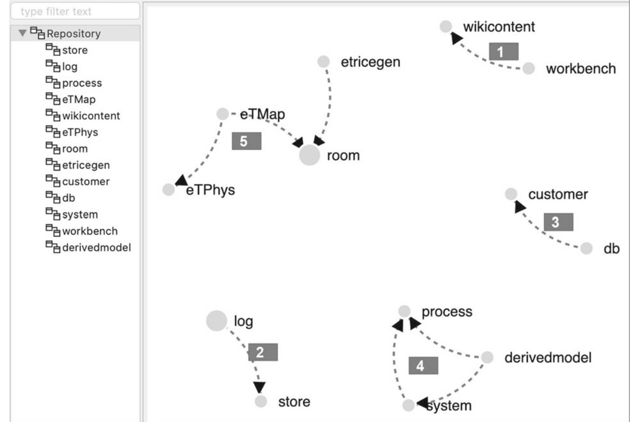


Fig. 12 Sample of metamodel dependencies

5 Experiments

In this section we discuss the experiments that have been performed to assess the effectiveness of the proposed tool-supported approach with the aim of answering the following research question:

> ***RQ:*** *Given a metamodel to be refactored, does the proposed approach correctly generate Edelta templates so to correctly raise errors in case of unsafe refactorings?*

In the following subsections, we first explain the setup of the experiments (Sec. 5.1), and then we discuss the obtained results (Sec. 5.2). Threats to validity are discussed in Sec. 5.3, by distinguishing them in internal and external.

5.1 Experiment setup

For our experiments, we selected a dataset of metamodels publicly available and presented in [3]. This dataset contains 2'417 metamodels collected by crawling online GitHub repositories.

By exploiting the dependency analysis approach discussed in Sec. 4.1, we took as input the whole dataset and generated

a dependency graph conforming to the metamodel shown in Fig. 5. The graphical representation of the whole graph is publicly available online[3]. We randomly selected existing dependencies and the corresponding metamodels as subjects of evolution operations. By applying our approach, we have generated the Edelta specifications to manage the involved subject metamodels. For explanatory reasons, some of the selected metamodel dependencies are represented in Fig. 12, which are also shown in column *Subgraph* of Table 1. To check the correctness of the generated templates, and thus of the import statements that are needed to possibly detect unsafe evolutions, for each subgraph we have performed mutations consisting of two actions: *i)* keep the generated import statements untouched (marked with the symbol '=' in Tab. 1) or remove some of them as represented by the symbol '–' (e.g., concerning subgraph 1, the removal of the `Workbench` import has been operated for three mutations out of five), *ii)* application of metamodel change to elements of the target metamodel of the considered dependency (e.g., the element `store::Checkout` has been removed for one of the mutations of subgraph 2[4]).

5.2 Results

All the mutations shown in Tab. 1 have been manually analyzed to check if unsafe evolutions are correctly detected by the approach. We can have the following cases:
The metamodel import statements are untouched (=): in this case the expected results can be as follows: if the meta-elements affected by the metamodel mutation (e.g., removal of `wikicontent::Wiki` in the fourth mutation of subgraph 1) are part of the dependant elements (e.g., the element was used by the dependant metamodel, i.e., `Workbench`), then the proposed approach is effective if Edelta shows an error due changes on the metamodel `wikicontent` and does not permit to produce inconsistent states because the metamodel `workbench` depends on `wikicontent` (e.g., see the *Expected* value marked as *x* for the fourth mutation of the subgraph 1). Moreover, we can have the case in which the mutated meta-element is not used by the dependant metamodel, and in this case Edelta does not show any error because the removal can be operated safely, even if the metamodels are dependant (see the output of the first 1 mutation).
The dependant metamodel import statements are dropped (–): we can have two cases: *i)* the mutation applied on the meta-element affects a dependant metamodel (e.g., the metaclass `customer::CustomerType` in the first mutation of subgraph 3); *ii)* the mutation does not affect a dependant meta-element (e.g., `customer::AddressType` of 3). In both cases, a metamodel evolution should be allowed by the tool, without raising any errors even though in the first case we will have an invalid metamodel, which is not recognized by the tool due to the removal of the import statement.

As shown in Tab. 1, the outputs produced by the unsafe evolution detection mechanism are always as expected. Thus, this supports that the approach correctly raises errors when needed by forcing the modeler to maintain the interrelated metamodels in a valid state. When the mutation removes the required imports, the metamodels are posed in an invalid state if the operated metamodel mutation affects dependant elements. This confirms the effectiveness of the approach in response to RQ .

5.3 Threats to validity

We distinguish the threats in internal and external validity of the performed experiments, and in the following we discuss the most relevant ones.

5.3.1 Internal validity

Internal validity threats are the internal factors that may influence the outcomes of the experiment. We have used a relatively small number of metamodels for the experiment. The reason is that first, we wanted to manually check the obtained results and second, the Edelta specification has to be inspected to check the found and not found inconsistencies. However, we considered random subgraphs of metamodels from the extracted repository to cover different domains and metamodels. The precision of the dependency analysis seems to be reliable, from the manual sample inspection. This can be considered as a threat since the algorithm could identify not existing dependencies or miss existing ones for a different pattern used for referring to the external resource. This has been mitigated by exploring samples and by implementing the algorithm with the available data. We plan to refine it by importing further Ecore models and manually inspecting a larger sample.

5.3.2 External validity

The main threat in this category regards the generalizability of our findings, i.e., whether they would still be valid outside the scope of this paper. We considered different kinds of metamodels belonging to different domains. However, we plan to evaluate the approach by considering a bigger dataset, covering more subgraphs of the extracted repository as future work. Moreover, the metamodel mutations that have been used for the evaluation might not reflect all the possible evolutions that can be applied to metamodels. Indeed, only removal

[3] https://gssi.github.io/MetamodelDependenciesAnalyzer/

[4] Metamodel changes consisting of removals of dependant meta-elements are shown in light red cells in Tab. 1

Table 1 Experiment results

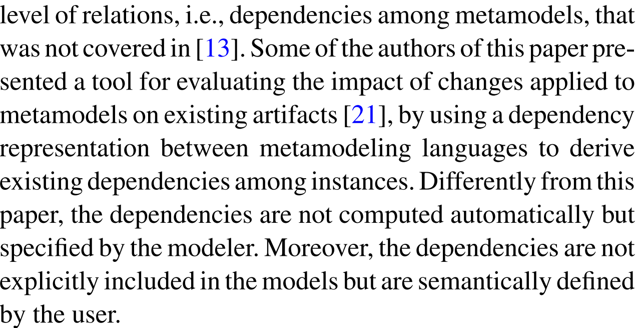
Subgraph	Mutations		Output of unsafe evolutions detection	
	Import	Meta-element subject	Expected	Produced
1 workbench → wikicontent	=	wikicontent::Row	○	○
	Workbench (-)	wikicontent::Attachment	○	○
	Workbench (-)	wikicontent::Wiki	○	○
	=	wikicontent::Wiki	x	x
	Workbench (-)	wikicontent::TextElement	○	○
2 log → store	log (-)	store::Project	○	○
	log (-)	store::Revision	○	○
	=	store::Checkout	x	x
	log (-)	store::UserSession	○	○
	log (-)	store::PluginDescriptor	○	○
3 db → customer	db (-)	customer::CustomerType	○	○
	db (-)	customer::AddressType	○	○
	=	customer::CreditInfo	○	○
	=	customer::CustomerType	x	x
	=	customer::USAddr	○	○
4 process ← system ← derivedmodel → process	derivedModel (-)	system::Job	○	○
	=	process::Task	x	x
	=	process::Gateway	○	○
	system (-)	process::Process	○	○
	derivedModel (-)	system::Data	○	○
5 eTPhys ← eTMap → room ← etricegen	=	room::RoomModel	x	x
	etMap (-)	room::ExternalPort	○	○
	etricegen (-)	room::ActorClass	○	○
	etMap (-)	room::LogicalSystem	○	○
	=	ETPhys::PhysicalThread	x	x

has been used, but also other complex evolutions, e.g., moving meta-elements would lead to inconsistencies. To the best of our knowledge, the model mutation is a technique that is commonly used to artificially create artifacts that are needed for performing this kind of experiments. However, we will further extend the evolution operators applied as mutants to consider other possible corrupting instructions.

6 Related work

The section has been organized to explore refactoring approaches, automatic detection of evolutions, and dependency analysis tools.

Dependency Analysis. Various approaches work in the direction of dependency analysis in multiple domains, e.g., package dependency in OS [39] or source code analysis [37]. We limit this discussion to model-based artifacts dependency analysis, as for instance the work in [15]. This work presents an automated approach to generating and validating trace dependencies among software development artifacts, such as model descriptions, diagrammatic languages, abstract (formal) specifications, and source code. Some of the authors of this paper previously presented an approach for reconstructing relationships among model-based artifacts in repositories. The work in [13] shares many similarities with the proposed approach and it has been used as the main source of inspiration, but it works in a different level of relations, i.e., dependencies among metamodels, that was not covered in [13]. Some of the authors of this paper presented a tool for evaluating the impact of changes applied to metamodels on existing artifacts [21], by using a dependency representation between metamodeling languages to derive existing dependencies among instances. Differently from this paper, the dependencies are not computed automatically but specified by the modeler. Moreover, the dependencies are not explicitly included in the models but are semantically defined by the user.



In [31] the authors present two strategies to describe relationships between metamodels. The first one is based on the definition of explicit dependencies between concrete metamodels. The second one is based on the description of contracts for metamodel entities. This last strategy introduces a new level of indirection in the definition of the dependencies that specifies the name of methods and events used to bound elements. The goal of that work is to propose new types of relations between metamodels, models, and model instances specifically in the Cumbia platform, and it is not in the direction of discovering metamodel dependencies as presented in our work.

Refactoring Approaches. The concept of model refactoring has been explored using a UML class diagram in [28] and applied to Ecore models in [33]. The authors of these works show how graph transformations are applied for supporting model refactoring. Indeed, every refactoring is expressed as a graph production. On the contrary, in Edelta the refactoring is

directly translated into Java code, and in the Edelta editor the refactoring is applied on the fly on the subject metamodels to perform static checks, giving immediate feedback to the modelers.

A research on refactoring tools is reported in [29], where the need to address the refactoring process in a more consistent, generic, and scalable way is strongly highlighted. The authors in [8] present a metamodel for specifying atomic operations. A single change is seen as an atomic transformation and the metamodel used in that approach is similar to the one at the base of Edelta.

In [2] a tool called EMFRefactor is presented with the intent of specifying and applying refactorings on models. This tool uses Henshin's model transformation engine for executing refactorings. The main difference with Edelta is that this tool implements the refactorings by implementing Java methods and coupling them with the UI. Edelta provides a DSL that is more extensible w.r.t. new refactorings, which in the other approach can only be implemented with more coding.

Concerning Edelta applications, in [5,6] a library of reusable metamodel refactorings has been used by following the formal definitions at https://www.metamodelrefactoring.org, previously inspired to Fowler [17]. Another work that has been part of the inspirational examples for building the Edelta refactoring catalog has been presented in [34]. This work presents a catalog of nine co-evolution operation specifications for automating the migration of ArchiMate models when the ArchiMate language is evolved. A set of refactorings preserving the behavior of UML models is also presented in [38].

Automatic detection of changes. Automatic detection of code refactoring is the topic of [27]. The authors present approach that takes as input an external library containing a list of possible refactorings, a set of structural metrics, and the initial and revised versions of the source code. As output, it generates a sequence of detected refactorings from the input by using a search-based process. Edelta collects the refactorings in libraries that can be reused in the entire process by modelers and developers. A different approach in [42] proposes a detection mechanism for identifying refactorings by analyzing the system evolution at the design level. Also the work in [16] detects high-level model changes. The authors in [26] search for occurrences of complex refactorings within a set of detected atomic ones in a post-processing approach. Another detection mechanism is proposed in [22] where the detection of complex changes applied to metamodel evolutions is presented. In these cases, the main difference with Edelta is that our DSL works in a programmatic application of the defined changes to produce the evolved model, whereas the above approaches already compute two versions of models and source code.

7 Conclusion and future work

In this paper, we presented an extension of the Edelta framework for supporting safe metamodel evolutions. Metamodels are not often used in isolation; for this reason, when languages are interrelated, e.g., in cross-referencing, evolving them in a standalone stage can create inconsistencies. The proposed approach consists of a two-factor process in which first the dependencies of the given repository of metamodels are analyzed by also considering the metamodel to be evolved. This analysis helps modelers in multiple ways. Moreover, the proposed approach generates Edelta templates, including the necessary import statements, to recognize possible unsafe evolution patterns. This way, all the dependant metamodels will be loaded in the same resource set. Thus, by exploiting its live evolution environment, the Edelta framework will consider all the dependant metamodels. Future directions are manifold: *i)* enriching the visual representation of the repository; *ii)* including a quality evaluation mechanism by considering all the dependant metamodels, which are subject to the evolution; *iii)* evaluate the approach with bigger datasets. Moreover, we plan to investigate the usage of Edelta to co-evolve different metamodels linked by non-physical dependencies (e.g., metamodels that underpin the definition of the same model transformations). In such cases, using a single Edelta specification to co-evolve related metamodels may result efficient and less verbose compared to multi-stage evolution mechanisms.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Acerbis, R., Bongio, A., Brambilla, M., Butti, S., Ceri, S., Fraternali, P.: Web applications design and development with webml and webratio 5.0. In: International conference on objects, components, models and patterns, pp. 392–411. Springer (2008)
2. Arendt, T., Taentzer, G.: A tool environment for quality assurance based on the eclipse modeling framework. Automat. Soft. Eng. **20**(2), 141–184 (2013)
3. Barriga, A., Di Ruscio, D., Iovino, L., Nguyen, P.T., Pierantonio, A.: An extensible tool-chain for analyzing datasets of metamodels.

In: Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems: companion proceedings, pp. 1–8 (2020)
4. Barriga, A., Rutle, A., Rogardt, H.: Improving model repair through experience sharing. J Object Tech **19**(2), 13 (2020)
5. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Edelta: An approach for defining and applying reusable metamodel refactorings. In: Procs of MODELS 2017 satellite event, pp. 71–80 (2017)
6. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Quality-driven detection and resolution of metamodel smells. IEEE Access **7**, 16364–16376 (2019). Publisher: IEEE
7. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Edelta 2.0: Supporting live metamodel evolutions. In: Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems: companion proceedings, MODELS '20. Association for computing machinery (2020)
8. Burger, E., Gruschko, B.: A change metamodel for the evolution of MOF-based metamodels. Modellierung **161**, 285–300 (2010)
9. De Lara, J., Guerra, E., Kienzle, J., Hattab, Y.: Facet-oriented modelling: open objects for model-driven engineering. In: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, pp. 147–159 (2018)
10. Debray, S.K., Evans, W., Muth, R., De Sutter, B.: Compiler techniques for code compaction. ACM Trans Programm Languag Sys (TOPLAS) **22**(2), 378–415 (2000)
11. Del Fabro, M.D., Bézivin, J., Valduriez, P.: Weaving models with the eclipse amw plugin. Eclipse Model Symposium, Eclipse Summit Europe **2006**, 37–44 (2006)
12. Di Cosmo, R., Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Supporting software evolution in component-based foss systems. Sci Comput Programm **76**(12), 1144–1160 (2011)
13. Di Rocco, J., Di Ruscio, D., Härtel, J., Iovino, L., Lämmel, R., Pierantonio, A.: Understanding mde projects: megamodels to the rescue for architecture recovery. Soft Sys Model **19**(2), 401–423 (2020)
14. Durisic, D., Staron, M., Tichy, M., Hansson, J.: Evolution of long-term industrial meta-models–an automotive case study of autosar. In: 2014 40th EUROMICRO conference on software engineering and advanced applications, pp. 141–148. IEEE (2014)
15. Egyed, A.: A scenario-driven approach to trace dependency analysis. IEEE Transact Soft Eng **29**(2), 116–132 (2003). https://doi.org/10.1109/TSE.2003.1178051
16. Fadhel, A., Kessentini, M., Langer, P., Wimmer, M.: Search-based detection of high-level model changes. In: ICSM, pp. 212–221. IEEE Computer Society (2012)
17. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: improving the design of existing code. Addison-Wesley (1999)
18. Hassam, K., Sadou, S., Le Gloahec, V., Fleurquin, R.: Assistance system for ocl constraints adaptation during metamodel evolution. In: 2011 15th European conference on software maintenance and reengineering, pp. 151–160. IEEE (2011)
19. Hebig, R., Khelladi, D.E., Bendraou, R.: Approaches to co-evolution of metamodels and models: a survey. IEEE Transact Soft Eng **43**(5), 396–414 (2016)
20. Hinkel, G., Kramer, M., Burger, E., Strittmatter, M., Happe, L.: An empirical study on the perception of metamodel quality. In: 2016 4th International conference on model-driven engineering and software development (MODELSWARD), pp. 145–152. IEEE (2016)
21. Iovino, L., Pierantonio, A., Malavolta, I :On the impact significance of metamodel evolution in MDE. J Object Tech, **11**(3) (2012)
22. Khelladi, D.E., Hebig, R., Bendraou, R., Robin, J., Gervais, M.P.: Detecting complex changes and refactorings during (Meta)model evolution. Inf. Syst **62**, 220–241 (2016)
23. Kling, W., Jouault, F., Wagelaar, D., Brambilla, M., Cabot, J.: Moscript: A dsl for querying and manipulating model repositories. In: International conference on software language engineering, pp. 180–200. Springer (2011)
24. Kolovos, D., de la Vega, A., Cooper, J.: Efficient generation of graphical model views via lazy model-to-text transformation. In: Proceedings of the 23rd ACM/IEEE International conference on model driven engineering languages and systems, MODELS '20, p. 12-23. Association for Computing Machinery (2020)
25. Kolovos, D.S., Paige, R.F., Polack, F.A.: The epsilon object language (eol). In: European conference on model driven architecture-foundations and applications, pp. 128–142. Springer (2006)
26. Langer, P., Wimmer, M., Brosch, P., Herrmannsdörfer, M., Seidl, M., Wieland, K., Kappel, G.: A posteriori operation detection in evolving software models. J. Syst. Softw **86**(2), 551–566 (2013)
27. Mahouachi, R., Kessentini, M., Cinnéide, M.Ó.: Search-based refactoring detection using software metrics variation, pp. 126–140. Springer, Berlin, Heidelberg (2013)
28. Mens, T.: On the use of graph transformations for model refactoring, pp. 219–257. Springer, Berlin, Heidelberg (2006)
29. Mens, T., Demeyer, S., Bois, B.D., Stenten, H., Gorp, P.V.: Refactoring: current research and future trends. Electron Note Theoret Comput Sci **82**(3), 483–499 (2003)
30. Mens, T., Kniesel, G., Runge, O.: Transformation dependency analysis-a comparison of two approaches. In: LMO, pp. 167–184 (2006)
31. Rodríguez, C., Sánchez, M., Villalobos, J.: Metamodel Dependencies for Executable Models. In: Bishop, J., Vallecillo, A. (eds.) Objects, Models, Components, Patterns, vol. 6705, pp. 83–98. Springer, Berlin, Heidelberg (2011)
32. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.: An analysis of approaches to model migration. In: Proc. Joint MoDSE-MCCM Workshop, pp. 6–15 (2009)
33. Rutle, A., Iovino, L., König, H., Diskin, Z.: A query-retyping approach to model transformation co-evolution. Softw Sys Model **19**, 1107–1138 (2020)
34. Silva, N., Sousa, P., da Silva, M.M.: Evolution of archimate and archimate models: An operations catalogue for automating the migration of archimate models. In: New perspectives on information systems modeling and design, pp. 1–19. IGI Global (2019)
35. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse modeling framework 2.0, 2nd edn. Addison-wesley professional (2009)
36. Strittmatter, M., Hinkel, G., Langhammer, M., Jung, R., Heinrich, R.: Challenges in the evolution of metamodels: smells and anti-patterns of a historically-grown metamodel. In: ME@MODELS, pp. 30–39. CEUR (2016)
37. Sugawara, N., Yamamoto, T.: Call graph dependency extraction by static source code analysis (2013). US Patent 8,347,272
38. Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.M.: Refactoring uml models. The unified modeling language. Model Lang, Concepts, Tools LNCS **2185**, 134–148 (2001)
39. Treinen, R., Zacchiroli, S.: Solving package dependencies: from edos to mancoosi. arXiv preprint arXiv:0811.3620 (2008)
40. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: European conference on object-oriented programming, pp. 600–624. Springer (2007)
41. Williams, J.R., Zolotas, A., Matragkas, N.D., Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.: What do metamodels really look like? Eessmod@ Model **1078**, 55–60 (2013)
42. Xing, Z., Stroulia, E.: Refactoring detection based on UMLDiff change-facts queries. In: WCRE, pp. 263–274. IEEE computer society (2006)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.