Edelta: an approach for defining and applying reusable metamodel refactorings

Lorenzo Bettini
DiSIA - University of Florence, Italy
lorenzo.bettini@unifi.it

Davide Di Ruscio
DISIM - University of L'Aquila, Italy
davide.diruscio@univaq.it

Ludovico Iovino
Gran Sasso Science Institute - L'Aquila, Italy
ludovico.iovino@gssi.it

Alfonso Pierantonio
DISIM - University of L'Aquila, Italy
alfonso.pierantonio@univaq.it

***Abstract*—Metamodels can be considered one of the key artifacts of any model-based project. Similarly to other software artifacts, metamodels are expected to evolve during their life-cycle and consequently it is crucial to develop approaches and tools supporting the definition and re-use of metamodel refactorings in a disciplined way.**

This paper proposes Edelta, a domain specific language for specifying reusable libraries of metamodel refactorings. The language allows both atomic and complex changes and it is supported by an Eclipse-based IDE. The developed supporting environment allows the developer to apply refactorings both in a *batch manner* and in a *step-by-step fashion*, which provides developers with an immediate view of the evolving Ecore model before actually changing it.

I. Introduction

Refactoring can be defined as the process of changing a software system to improve its design, readability, and to reduce bugs [1]. Metamodels play a key role in any model-based approach and, similarly to any software artifact, they are subject of evolutionary pressures that can arise for several reasons. In particular, metamodels can be changed for perfective, corrective, preventive and adaptive maintenance goals [2].

Metamodel refactorings can be expressed as a sequence of additions, deletions and changes of elements [3]. Additionally, metamodel refactorings can involve also complex operations, which are obtained by mixing atomic changes [4], [5], [6], [2], [7]. Existing approaches supporting the evolution of modeling artifacts are mainly based on tedious and error-prone manual activities [8]. Moreover, the possibility to organize and reuse already defined refactorings is also limited in currently available approaches [9].

In this paper we present a metamodel refactoring approach based on the proposed Edelta domain specific language. Edelta offers the basic mechanisms to express atomic metamodel changes i.e., addition, deletion and changes. Moreover, Edelta provides developers with an extension mechanism enabling the use of already developed refactorings, which can be organized in reusable libraries. The language is endowed with an Eclipse-based IDE providing features that are typically expected from mature development environments, i.e., syntax highlighting, code completion, error reporting and incremental building, not to mention debugging. Moreover, the supporting environment allows the developer to estimate and evaluate the impact of the refactorings being specified before actually changing the evolving metamodel.

The paper is organized as follows: Section II introduces the problem of metamodel refactoring and discusses requirements that a metamodel refactoring approach should satisfy. Section III presents the Edelta language, its implementation, and the constructs for defining both atomic and complex changes. Section IV makes an overview of the features provided the the Eclipse-based IDE supporting the specification and execution of Edelta specifications. Section V discusses the advantages of the proposed approach with respect to the requirements presented in Section II. Related works are discussed in Section VI. Section VII concludes the paper and outlines some future plans.

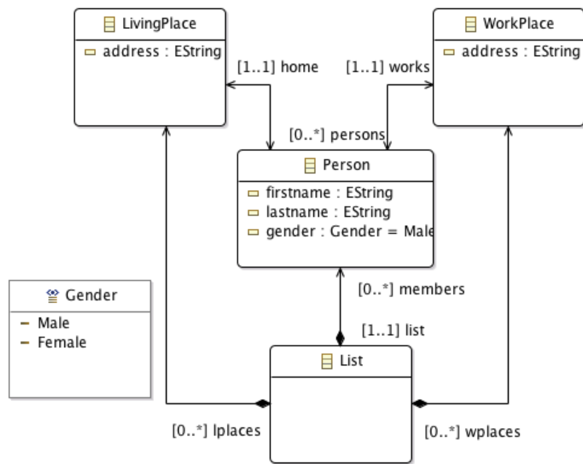
II. Mastering Metamodel Refactorings

In order to satisfy unforeseen requirements or to better represent the considered application domain, metamodels can be subject to changes as for instance in the case of the simple `PersonList` metamodel shown in Fig. 1.a. The metamodel represents list of persons with the corresponding places where they work or live. The metamodel includes the root metaclass `List` that has a containment reference of type `Person`, which in turn is characterized by the attributes `firstname`, `lastname`, and `gender`. In particular, the `gender` attribute is specified with an enumeration enabling the specification of the literals `Male` and `Female`. A person is related to a `WorkPlace` and a `LivingPlace`, both characterized by the attribute `address`.

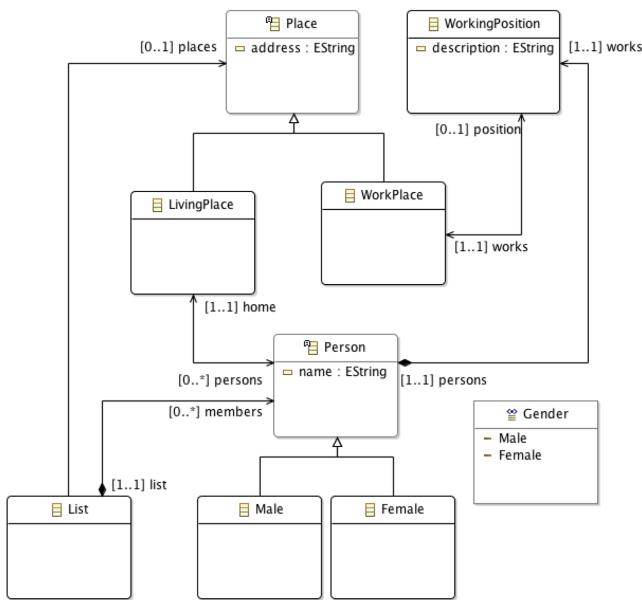
The new version of the metamodel shown in Fig. 1.b has been modified by means of the following changes:

C1. the attributes `firstname` and `lastname` of the metaclass `Person` have been merged into the new `name` attribute;

C2. the metaclass `Place` has been added as an abstract super class of `WorkPlace` and `LivingPlace`. Moreover, their `address` attribute has been pulled-up in the new metaclass;



(a) Initial version



(b) Evolved version

Fig. 1. Evolution of the simple PersonList metamodel

C3. the `gender` attribute of the metaclass `Person` has been replaced by the new sub-classes `Male` and Female;

C4. the reference `works` of the metaclass `Person` has been replaced by the new metaclass `WorkingPosition` between the new version of `Person` and `WorkPlace`. Note that `works` and `persons` are still bidirectional in the refactored metamodel, as they used to be in the original metamodel.

Even though such changes might appear specific for the particular metamodel evolution at-hand, modelers might want to define them so to similarly evolve other metamodels in the future. In this respect, by borrowing concepts from object-oriented programming [10] several works [11], [12], [13], [14] have defined catalogs of possible metamodel refactorings with the aim of dealing with specific issues, like the coupled evolution of metamodels and models. Examples of recurrent metamodel refactorings are `ExtractSuperClass` and `IntroduceSubclass` [12]. Concrete instances of such refactorings are changes *C2* and *C3* above, respectively.

Operating metamodel refactorings without a dedicated support can be an error-prone task. In particular, there is the need for languages and tools supporting the specification and application of metamodel refactorings in a disciplined way. To this end, by taking inspiration from [15], we defined a set of requirements that a language for specifying reusable metamodel refactorings should implement as presented in the following.

a) Conciseness and comprehensibility: Metamodel refactorings should be expressed as concisely and comprehensibly as possible for two main reasons: *i)* unnecessarily voluminous code increases development costs and this is particularly the case of metamodel manipulations implemented with GPLs like Java; *ii)* the quality of the implemented solutions can decrease in case of long specifications since errors might not be easily detected. Contrariwise, if the considered metamodel refactoring language offers adequate constructs for specifying solutions with terms that are closer to the problem at-hand, domain experts can be involved in the development. In this way, the number of bugs and errors introduced with traditional manipulation techniques can be reduced.

b) Integration: It should be possible to integrate the metamodel refactoring language with other tools e.g., for dealing with bigger problems like the coupled evolution of metamodels and models.

c) Static checks: The language should be statically typed. In the considered context the elements on which refactorings are applied are those of the meta-metamodeling language. Thus, implementing a statically typed DSL in this context means that all references to metaclasses, features of the metaclasses, etc. have to be statically checked by the DSL compiler and assignments between such elements have to be checked for type conformance as well.

d) Refactorings Composability: It is important to support the composition of refactorings into more complex ones. Some of the composing refactorings might depend on other ones and consequently the supporting tools should be able to analyze the specified refactorings to determine which are mutually independent, and which refactorings have to be applied sequentially [9].

e) IDE support: Nowadays a DSL should come with a supporting tool in order to effectively use the language in practice in a productive way. The IDE should integrate all the useful features for the modeled domain, in this case the refactoring of metamodels.

III. The Edelta Language

This section presents Edelta, the domain specific language we have defined for specifying and applying metamodel refactorings. Edelta provides modelers with constructs for specifying atomic refactoring i.e., additions, deletions and

changes, and to define complex reusable metamodel changes by properly composing already defined ones.

Fig. 2. A simplified version of the Edelta metamodel

An overview of the concepts implemented by the language are shown in Fig. 2. An Edelta specification (represented by the `EdeltaModule` in the metamodel) can declare operation definitions (which can be reused both by the same program and by other Edelta program with the clause `use ...as ...` as shown later) and sequences of basic refactoring operations (to add, remove, and change metaclasses and structural features therein). Such built-in operations are induced by the Ecore metamodel [16] according to the approach defined in [12].

In the next section, the Xtext-based implementation of the Edelta language is presented. Details about how atomic and complex refactorings can be specified in Edelta are also given in Section III-B and Section III-C, respectively.

A. Xtext-based implementation of Edelta

The Edelta DSL has been implemented with Xtext [17], a popular Eclipse framework for the development of programming languages and domain-specific languages. Given a language grammar definition, Xtext generates the infrastructure for the compiler and a complete IDE support based on Eclipse (e.g., editor with syntax highlighting, code completion, error reporting and incremental building).

Figure 3 shows the artifacts that are involved when evolving a metamodel by means of Edelta. The Ecore Metamodel subject of the evolution is imported in the Edelta program being specified. The Edelta program contains all the Edelta instructions describing the refactorings to be operated by re-using operations already defined in available libraries. When the Edelta program is saved and no errors are found by the compiler, corresponding Java code is generated for manipulating the input metamodel with the refactorings expressed in the Edelta program. The generated Java code uses the Edelta Java runtime APIs, and it can be called from any Java program. The Edelta project wizard creates a Java file with a main that calls the generated Java code; this main file, which can be seen as a starting point, is shown in listing 1. When running

Fig. 3. Overview of the artifacts involved in Edelta-based metamodel refactorings

the main file a new metamodel is generated in the *modified* folder of the project.

```
1 ...
2 public static void main(String[] args) throws Exception {
3  // Create an instance of the generated Java class
4  AbstractEdelta edelta = new Example();
5  // Make sure you load all the used Ecores
6  edelta.loadEcoreFile("model/My.ecore");
7  // Execute the actual transformations defined in the DSL
8  edelta.execute();
9  // Save the modified Ecore model into a new path
10 edelta.saveModifiedEcores("modified");
11 }
```

Listing 1. Generated main file from Edelta projects

The generated Java class (`Example` in listing 1) extends the Edelta Java runtime library base class `AbstractEdelta`, which provides many methods for configuring reading and writing refactored Ecore models.

The Edelta project is available at
https://github.com/LorenzoBettini/edelta.

Edelta uses Xbase [18] to provide a rich Java-like syntax for its expressions. In order to make code snippets presented in the paper comprehensible, in the following we sketch the main features of Xbase. Afterwards, the typical structure of Edelta programs is discussed.

1) Xbase in a nutshell: Xbase is an extensible and reusable expression language that is completely interoperable with Java and its type system, and is meant to be embedded in an Xtext DSL. Xbase is Java-like, avoiding much of the "syntactic noise" verbosity that is typical of Java. Xbase should be easily understood by Java programmers. The complete interoperability with Java implies that our language reuses the

Java type system, including generics. All existing Java types can be referred and used from within an Xbase expression, that is, all existing Java libraries can be seamlessly reused in Edelta. This also means that if one has existing refactoring implementations for Ecore models written in Java, then this existing code can be reused in an Edelta program out of the box. This interoperability of Xbase DSLs with Java types follows the standard Java classpath mechanisms, meaning that all the Java libraries that are part of the current classpath can be used seamlessly in Edelta programs. Also Edelta reusable refactoring libraries are handled according to the same classpath mechanism. Furthermore, Xbase handles the classpath of standard Eclipse projects, including the Eclipse project created by the Edelta project wizard itself. This means that in order to reuse a refactoring library, which can be imported in an Edelta program with the clause `use ...as ...` as shown later, that library must be specified as a dependency of the Eclipse project; no further custom registries for Java refactorings needs to be implemented.

Terminating semicolons are optional in Xbase. Variable declarations in Xbase start with `val` or `var`, for final and non final variables, respectively, and do not require to specify the type if it can be inferred from the initialization expression. A cast expression in Xbase is written using the infix keyword `as`.

Xbase *extension methods* are a syntactic sugar mechanism to simulate adding new methods to existing types without modifying them. For example, if `m(Entity)` is an extension method, and `e` is of type `Entity`, you can write `e.m()` instead of `m(e)`, even though `m` is not a method defined in `Entity`.

Xbase *lambda expressions* have a simpler shape than Java lambda expressions: `[ param1, param2, ... | body ]`. The types of parameters can be omitted if they can be inferred from the context. 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(...)[...]`.

Xbase has special syntax for collections: `#[e1, ..., en]` allows the programmer to easily specify a list with initial contents. Operators such as `+`, `-`, `+=`, `-=` are available in Xbase also on collections.

Xbase provides syntactic sugar for getters and setters: instead of writing `o.getName()`, one can simply write `o.name`; similarly, instead of writing `o.setName("...")`, one can write `o.name = "..."`.

Besides `this`, Xbase has another additional special variable, `it`. Just like `this`, which has the same semantics as in Java, `it` can be omitted as object receiver of method call and member access expressions. Differently from `this`, the programmer is allowed to declare any variable or method parameter with the name `it`. Thus, the programmer can define a custom implicit object receiver in any scope of the program. Furthermore, 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`.

Xbase lambda expressions, thanks to all the syntactic sugar mechanisms just mentioned, allow the programmer to easily write statements and expressions that are compact and more readable than in Java. For instance, assuming the method `newEClass` takes a string and a lambda and returns an `EClass`, one can simply write

```
1 newEClass("aclass") [
2   name = "MyEClass"
3 ]
```

instead of

```
1 newEClass("aclass",
2   [ e | e.setName("MyEClass") ])
```

2) Specifying and compiling Edelta programs: An Edelta program consists of a bunch of function definitions, which can be reused across Edelta programs (with the clause `use ...as ...`, as shown later in the examples), and a list of Ecore refactoring instructions, which represent the actual refactoring implemented by the Edelta program.

The EPackages that are used in the Edelta program must be explicitly specified by their name, using the clauses `metamodel "..."` at the beginning of the Edelta program (these include both the EPackages under refactoring and the EPackages that are simply referred to in the program, e.g., the Ecore metamodel, typically used to specify data types such as EString, EInt, etc.). Content assist is provided by the Edelta editor, proposing all the Ecore files that are available in the current project's classpath (using the standard plug-in dependencies of Eclipse projects).

As mentioned above, the expression syntax of Edelta programs is based upon Xbase, thus Edelta provides the same expressive power of Java, though in a "cleaner" and less verbose way. Besides that, special Edelta expression syntax has been introduced to make refactoring easier and statically type-checked.

Edelta is based on the Java APIs that we implemented as part of the Edelta runtime library. These APIs implement the actual refactoring operations that are applied on an Ecore model loaded in memory and also take care of saving the changed Ecore model into a new file. The Edelta compiler generates Java code that uses the Edelta runtime Java APIs. Note that the generated Java code has no further dependency on the Edelta compiler, nor on the Xtext framework itself: it depends only on the Edelta runtime library, which weights only a few mega bytes. This means that the generated Java code that performs actual refactoring on Ecore files can be executed independently from Eclipse and the whole Xtext infrastructure.

Edelta programs refer directly to Ecore model classes, thus you do not need to access the Java code generated from an Ecore model. This also means that our approach works even in situations where the EMF Java model has not been generated at all (e.g., in those contexts where EMF models are created in a completely reflective way). References to Ecore elements, such as packages, classes, data types, features and enumerations, can be specified by their fully qualified name in Edelta using the standard dot notation, or by their simple name if there are no ambiguities (possible ambiguities are checked

by the compiler). Such Ecore model references can be used directly from within an Edelta refactoring instruction (such as, `createEClass` and `changeEClass`, explained later). On the contrary, when used from within an Xbase expression, they would conflict with Java type references and fully qualified Java references (e.g., field and method calls). To avoid such ambiguities, in Edelta Xbase expressions, references to Ecore model elements are wrapped inside an `ecoreref(...)` specification[1].

In order to show the differences between Java type references and Ecore model elements in Edelta Xbase expressions we use the following artificial example of variable declarations, where we use both references to the Java types of the Ecore Java model classes and references to Ecore model elements[2]:

```
1 val org.eclipse.emf.ecore.EDataType d = ecoreref(ecore.
     EString)
2 val org.eclipse.emf.ecore.EClass c = ecoreref(ecore.
     EDataType)
3
4 // Type mismatch: cannot convert from EClass to EDataType
5 val org.eclipse.emf.ecore.EDataType d2 = ecoreref(ecore.
     EDataType)
```

Here, `org.eclipse.emf.ecore.EDataType` is the reference to the Java interface `EDataType` defined in the Java package `org.eclipse.emf.ecore`, while `ecoreref(ecore.EDataType)` is the reference to the Ecore model element `EDataType`, defined in the `EPackage` with name `ecore`. Everything is statically type-checked in Edelta, according to the Java type system. Thus, the first two assignments are correct, since the Ecore model element `EString` is a Java `EDataType`; similarly, the Ecore model element `EDataType` is a Java `EClass`. For this reason, the third assignment is rejected, since the Java interface `EClass` (i.e., the type of the Ecore model element `EDataType`) is not a subtype of the Java interface `EDataType`.

B. Built-in metamodel refactoring operations

Edelta provides a set of basic refactoring operations acting on EClassifiers and on the features of the EClassifiers. For example, one can introduce a new EClass in the Ecore package that is being refactored with the following instruction:

```
1 createEClass <name> in <package_reference> {
2   ... initialization block
3 }
```

When creating a new EClass, one can specify also the supertypes with the `extends` clause. In the initialization block one can set the properties of the newly created EClass and add features with similar instructions (e.g., `createEAttribute`, `createEReference`, etc.).

An existing EClass of the package that is being refactored can be modified with the following instruction:

[1]We could have used a different separator for qualified references to Ecore model elements, but the ambiguity would still be there for simple names. Moreover, the use of `ecoreref` explicit specification allows us to provide much better tooling support, especially content assist proposals.

[2]Fully qualified names for Java types would not be necessary here, since Edelta supports Java-like imports; we use fully qualified names only for the sake of the explanation.

```
1 changeEClass <package_reference>.<class_refererence> {
2   ... modification block
3 }
```

The modification block can be used to modify properties, adding features (with the above mentioned instructions) or changing existing features (e.g., `changeEAttribute`, `changeEReference`, etc.).

In the initialization and modification blocks, the reference to the created or changed Ecore element is automatically available through the special variable `it`, previously explained. Together with the above mentioned syntactic sugar mechanisms of Xbase, one can write compact refactoring expressions such as

```
1 createEClass MyNewClass in mypackage {
2   abstract = true // short for it.setAbstract(true)
3 }
```

One of the main features of Edelta is that it automatically keeps track of the refactoring operations that are specified in the program and applies them to an in-memory loaded Ecore model. This way, while the developer is editing an Edelta refactoring specification, the new and modified elements are immediately available and can be used for static type checking. For example, let consider the following two refactoring instructions:

```
1 createEClass MyNewClass in mypackage extends AnotherClass {
2   ... initialization block
3 }
4
5 changeEClass mypackage.AnExistingClass {
6   name = "AnotherClass"
7   createEReference aNewReference type MyNewClass { ... }
8 }
```

The new EClass extends `AnotherClass`, which is actually an existing class that is being renamed in the second instruction; the new reference created in the changed EClass refers to the new EClass created in the first instruction. All these references are type checked by the Edelta compiler, by keeping track of the refactorings that are being specified in the program. Note that the generated Java code that performs the actual refactoring applies the refactorings in such a way that mutual dependencies do not generate problems.

The above described mechanism is implemented by using the interpreter provided by Xbase. In fact, Xbase provides an interpreter that is able to interpret both the Xbase expressions of your DSL programs (even if the corresponding Java code has not been generated yet) and the possible calls to external Java classes (both field accesses and method calls) by Java reflection.[3]

The Edelta compiler invokes the interpreter on a copy, which is kept in memory, of the Ecore files used in the Edelta program. The validation and type checking of the Edelta program (which also includes binding references to Ecore elements) is implemented using the Ecore models kept in memory, on which the interpretation of the refactorings

[3]The DSL implementor has to customize this interpreter concerning the DSL's specific additional expressions, like, in our case, `createEClass`, `changeEClass`, etc.

is being applied. This allows the compiler to implement the following features:

- The program can refer to Ecore model elements (EClasses, EStructuralFeatures, etc.) that are created by the refactoring operations specified in the program itself. This includes also possible modifications to the Ecore elements. For example, the Edelta compiler's type checking is able to know in a given part of the program if a reference has been turned into an attribute or if a feature has been renamed.
- Since the interpretation is applied sequentially on all the refactoring operations, the Edelta compiler is able to detect possible errors due to the order of the refactorings, such as, e.g., referring to an Ecore element, in a part of the program, which has been removed by a refactoring operation specified before that part of the program. These errors are detected by the compiler, thus the programmer has an immediate feedback about the correctness of the specified refactorings, so that when the actual refactoring is applied to actual Ecore models, no bad surprises will happen (and the refactored models will still be valid Ecore models).
- The interpreter will rely on the same Edelta Java APIs that are used by the generated Java code. This means that the semantics of the interpretation will be the same as the semantics of the Java code generated by the Edelta compiler. Thus, if no validation errors are raised by the Edelta compiler by relying on the interpretation, then the Ecore models after the refactorings performed by the generated code will still be valid Ecore models (see also the previous item).

For example, in the previous code snippet, `AnExistingClass` is renamed to `AnotherClass` by calling the standard Ecore Java API[4]. Since the Edelta compiler interprets on the fly the body of the `changeEClass` operation, the `AnotherClass` can be immediately used in the program (e.g., it is the base class of `MyNewClass`). Let us stress that all of this is statically checked.[5]

The interpreter uses a configurable timeout (which defaults to 2 seconds) when interpreting each refactoring Edelta operation. If the interpretation does not end within the timeout, the interpreter is interrupted and a warning is issued by the Edelta compiler (and a corresponding warning marker is put on the Eclipse editor and Problems view). This avoids the interpreter to block the IDE in case of possible endless loops in the refactoring operations or in case of long running operations[6]. This way, termination of the interpreter is always guaranteed, either successfully or with a timeout warning reported on the operation that caused the timeout. It is then up to the developer to either increase the timeout or fix the parts that do not terminate. It would be interesting to keep track of the applied refactorings and detect these cases; this is the subject of future work.

C. Example of Edelta applied to the PersonList metamodel

In this section we show how complex metamodel refactorings can be specified in Edelta. The metamodel evolution shown in Fig. 1 is considered throughout the section.

Listing 2 shows the specification of all the refactorings applied on the initial version of the simple `PersonList` metamodel. Line 4 of Listing 2 specifies the package name for the evolving metamodel, i.e., `PersonList`. Line 7 defines the usage of an external Edelta specification called `refactoringslib`, containing all the defined refactorings previously identified in [19]. A fragment of the *refactoringslib* library is shown in Listing 3.

At line 9 of Listing 2 the first refactoring is performed and the metaclass `Person` is changed: `introduceSubclasses` is called by passing as parameters the type of the attribute `gender`, and the containing class `Person` (specified using `it`, which refers to the metaclass currently being refactored). Note the use of `ecoreref` (Section II) to access metamodel elements of the EPackages specified by the `metamodel` clauses at lines 4-5. The definition of the refactoring `introduceSubclasses` is in Listing 3: for each literal of the enumeration used as attribute type a new EClass is introduced as subtype of the containing class, e.g., in the example `Male` and `Female` as subclasses of `Person`. At the end the attribute is removed from the containing class and the EClass becomes abstract. The effect of this first refactoring is shown in fig. 1.b, where the `Person` metaclass is abstract and it has two subclasses `Male` and `Female`.

```
1 ...
2 package gssi.personexample
3
4 metamodel "PersonList"
5 metamodel "ecore"
6
7 use MMrefactorings as refactoringslib
8
9 changeEClass PersonList.Person {
10  refactorings.introduceSubclasses(
11   ecoreref(Person.gender),
12   ecoreref(Person.gender).EAttributeType as EEnum,
13   it);
14  EStructuralFeatures+=
15   refactorings.mergeAttributes("name",
16    ecoreref(Person.firstname).EType,
17    #[ecoreref(Person.firstname),ecoreref(Person.lastname)]
18   );
19 }
20
21 createEClass Place in PersonList {
22  abstract = true
23  refactorings.extractSuperclass(it,
24   #[ecoreref(LivingPlace.address), ecoreref(WorkPlace.
        address)]);
25 }
26
27 createEClass WorkingPosition in PersonList {
28  createEAttribute description type EString {}
29  refactorings.extractMetaClass(it,
30    ecoreref(Person.works),"position","works");
31 }
```

[4]Recall that `name = ...` is equivalent to `it.setName(...)`

[5]Further details about the interplay between the interpreter and the compiler require a deep knowledge of a few advanced mechanisms of the Xtext/Xbase framework. For this reason, the full description of the static checks algorithm will be provided in future works.

[6]In our experience, 2 seconds are more than enough to execute even complex refactoring operations on Ecore models.

```
32
33 changeEClass PersonList.List {
34  EStructuralFeatures+=
35   refactorings.mergeReferences("places",
36    ecoreref(Place),
37    #[ecoreref(List.wplaces),ecoreref(List.lplaces)]
38   );
39 }
```

Listing 2. Snippet of the Edelta program for PersonList metamodel example

Line 14 of Listing 2 enriches the structural features of `Person` with the result of the `mergeAttributes` refactoring defined in Listing 3. Such a refactoring is responsible of merging a list of attributes into a single one, if they have the same type. In this example the attributes `firstname` and `lastname` are merged into the new one `name`.

The refactorings specified at lines 21-25 of Listing 2 are responsible of the extraction of a superclass `Place` from the attributes `address` of `LivingPlace` and `WorkPlace`, by calling `extractSuperclass`, defined in Listing 3. This refactoring is called when there are two metaclasses with similar features: a new super-metaclass is created and the common features are moved to the superclass [10]. The newly created metaclass `Place` is passed as implicit parameter `it` to the method definition call at line 23. The effect of this refactoring is shown in Fig. 1.b, where a new abstract metaclass `Place` is introduced and the `address` attribute is moved up in the hierarchy.

Line 29 of Listing 2 invokes the extract metaclass refactoring on a newly created metaclass `WorkingPosition`. This refactoring creates a new metaclass representing a feature of the owner metaclass. In the example the reference `works` is a simple association between the metaclass `Person` and the `WorkPlace`. We also pass as parameters the `in` and `out` references from and to this new metaclass from the old one (in this example, `position` and `works`, respectively). The result of this refactoring is shown in Fig. 1.b where the `WorkingPosition` metaclass is linked to `Workplace` and `Person` with the references `position` and `works`. A new attribute `description` of the job position that will be used to characterize the person, is also created at line 28.

Since the metaclass `List` has two different references for the metaclasses `LivingPlace` and `WorkPlace`, having now a common superclass, they can be merged into a single reference having the same common supertype. This is the case of the `mergeReferences` refactoring specified at lines 33-38 of Listing 2.

```
1 ...
2 def introduceSubclasses(EAttribute attr,EEnum attr_type,
      EClass containingclass){
3  containingclass.abstract = true;
4  for(subc : attr_type.ELiterals){
5   containingclass.EPackage.EClassifiers+=newEClass(subc.
        literal)[
6    ESuperTypes+=containingclass;
7   ];
8   containingclass.EStructuralFeatures-=attr;
9  }
10 }
11 ...
12 def mergeAttributes(String newAttrName, EClassifier etype,
      List<EAttribute> attrs): EAttribute{
13  val newAttr=newEAttribute(newAttrName)[
14   EType=etype;
15  ]
16  for(a:attrs){
17   a.EContainingClass.EStructuralFeatures-=a;
18  }
19  return newAttr;
20 }
21 ...
22 def mergeReferences(String newAttrName, EClassifier etype,
      List<EReference> refs): EReference{
23  val newRef=newEReference(newAttrName)[
24   EType=etype;
25  ]
26  for(r:refs){
27   r.EContainingClass.EStructuralFeatures-=r;
28  }
29  return newRef;
30 }
31 ...
32 def extractSuperclass(EClass superclass, List<EAttribute>
      attrs){
33  val extracted_attr=attrs.head;
34  for(attr: attrs){
35   attr.EContainingClass.ESuperTypes+=superclass;
36   attr.EContainingClass.EStructuralFeatures-=attr
37  }
38  superclass.EStructuralFeatures+=extracted_attr;
39 }
40 ...
41 def extractMetaClass(EClass extracted_class, EReference f,
      String _in, String _out) : void {
42  val ref_in=newEReference(_in)[
43   EType=extracted_class;
44   lowerBound=f.EOpposite.lowerBound;
45   upperBound=1;
46  ];
47  val old_ref=newEReference(f.name)[
48   lowerBound=1;
49   upperBound=1;
50   EType=f.EType;
51   EOpposite=ref_in;
52  ];
53  extracted_class.EStructuralFeatures+=old_ref;
54  ref_in.EOpposite=old_ref;
55  f.EOpposite.lowerBound=1;
56  f.EOpposite.upperBound=1;
57  extracted_class.EStructuralFeatures+=f.EOpposite;
58  f.EReferenceType.EStructuralFeatures+=ref_in;
59  f.EType=extracted_class;
60  f.containment=true;
61  f.name=_out;
62 }
63 ...
```

Listing 3. Fragment of the existing *refactoringlib* library

It is worth noting that the same refactoring example can be applied to other metamodels presenting the same structural elements, obtaining the same improvements of this running example. This is possible thanks to the reusable common refactoring definitions that can be used in Edelta.

IV. The Edelta editor

The Edelta editor provides typical Eclipse tooling mechanisms, including content assist and navigation to definitions. This is implemented both for Java types and for Ecore model elements. For example, in Fig. 4 we show the content assist for references to EClasses of the imported Ecore models. Navigating to such an element automatically opens the standard EMF Ecore tree editor.

Besides allowing our compiler to perform static type checking, keeping the in-memory Ecore model continuously updated by interpreting the specified refactoring operations also allows us to provide the developer with a immediate view of the modified Ecore model in the Outline view. This is shown in

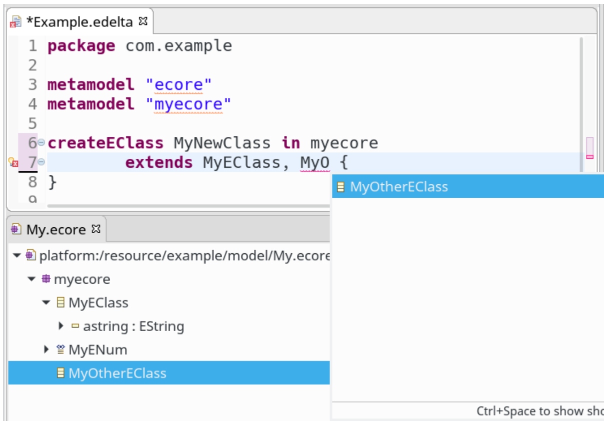


Fig. 4. Content assist for Ecore model elements.

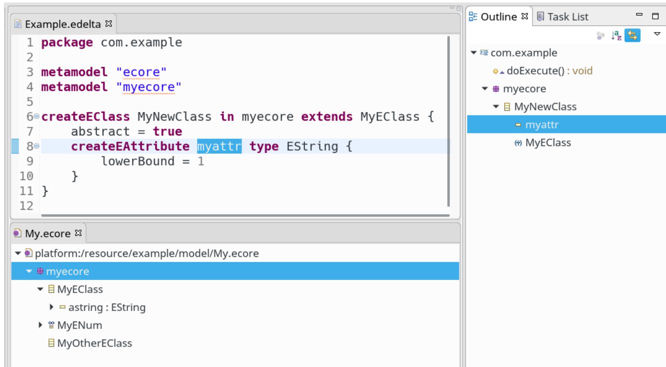


Fig. 5. The Edelta DSL editor, showing the synchronized Outline with the Ecore model being refactored and the original Ecore model that can be navigated from the Edelta DSL editor.

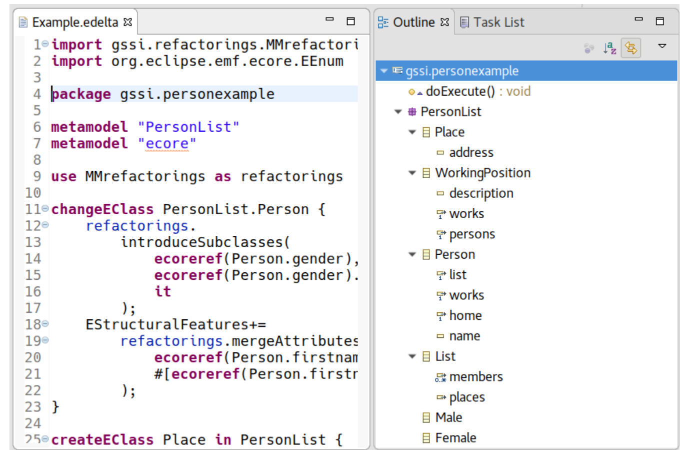


Fig. 6. Evolution result of the interpretation of the Edelta program in the Outline view

Fig. 5: in the program we created a new EClass and a new EAttribute and this is immediately reflected in the Outline view. New elements created by the refactoring operations and elements modified by the refactoring operations are also immediately available in the content assist. By referring to the running example, the Outline view shown in Fig. 6 represents the modified metamodel, which is computed by the Edelta compiler on-the-fly through the interpretation of the Edelta specification given in Listing 2.

Finally, one of the advantages of using Xbase is that Edelta specifications can be directly debugged in Eclipse: when running the generated Java code in the Eclipse debugger, the original Edelta source code can be debugged (all the standard Eclipse debugging features are available, i.e., variables view and breakpoints).

V. Discussion

In this section we will stress the advantages of having a metamodel refactoring tool instead of using traditional code based techniques, e.g., Java code for metamodel evolution. In the following we briefly recall the requirements identified in Section II and discuss how the proposed approach addresses them.

a) Conciseness and comprehensibility: If we compare the Edelta specification of the running example with the generated Edelta Java code in terms of lines of code, the result would be 42 vs 225 (81+144). We only considered the Edelta program since the refactoring library is defined once and then imported in the Edelta program. The mechanism of importing already defined refactorings increases the understandability of the Edelta program if we compare this with reading the full Java code. At the same level a comparison with other model transformation frameworks used as a metamodel refactoring mechanism can be proposed and will be part of future plans.

b) Integration: An important application of this requirement concerns the integration of the proposed approach with tools supporting the coupled evolution of metamodels and models. We designed Edelta in order to enable its integration with EMFMigrate [20]. Edelta will be used to specify the patterns that have to be matched to trigger model migrations [21].

c) Static checks: The proposed language refers to actual meta-elements of the Ecore metamodel. This means that Edelta is able to catch possible refactoring errors at compilation time, so that the generated Java code implementing the actual refactor will not misbehave when executed on the Ecore model being refactored. In order to reduce possible problems when executing the generated Java code, the Edelta compiler also performs an on-the-fly interpretation of the refactoring and uses such information to perform further static checks.

d) Refactorings Composability: Such a requirement is satisfied by the possibility of defining reusable refactoring libraries that can be included when specifying Edelta programs.

e) IDE support: Edelta is distributed with a complete Eclipse IDE tooling support, as shown in Section IV, by making the metamodel refactoring a supported methodology with all the related features.

VI. Related Work

The analysis of the existing refactoring tool made in [9] highlights that there are still a lot of open issues that remain to be solved. They conclude that there is the need for formalisms,

processes, methods and tools that address refactoring in a more consistent, generic, scalable and flexible way. In [22] authors explored the concept of model refactoring using a UML class diagram as metamodel. Some concrete experiments have been used to show how graph transformations can be used to support model refactoring. The proposed approach is implemented in the Fijaba tool. This paper shows in concrete experiments how graph transformation technology can be used to support model refactoring. A refactoring plug-in can be developed to refactor UML class diagrams, where each refactoring is expressed as a graph production. The similarity with the presented approach is based on the fact that class diagrams are customary used as simplified representation for metamodel languages. The approach in [22] uses graph transformations to apply the refactoring and in Edelta the refactoring is directly translated into Java code. Model refactoring, with a very similar intent, is also explored in [23], where an Eclipse incubation project providing specification and application of refactorings on models is proposed.

In [24] refactoring is described as one of the most important and commonly used techniques of transforming a piece of software in order to improve its quality. They analyzed source code version control system logs of popular open source software systems to detect changes marked as refactorings and examine how the software metrics are affected by this process, in order to evaluate whether refactoring is effectively used as a means to improve software quality. This kind of applications are possible future improvements that can be introduced with the complete Edelta environment, since the refactoring can be simulated in order to understand if metamodeling quality attributes can be improved applying it.

In [4] authors developed an approach to automate the detection of source code refactorings using structural information extracted from the source code. This approach takes as input a list of possible refactorings as an external library, a set of structural metrics and the initial and revised versions of the source code. It generates as output a sequence of detected changes expressed as refactorings. This refactoring sequence is determined by a search-based process that minimizes the metrics variation between the revised version of the software and the version yielded by the application of the refactoring sequence to the initial version of the software. This work is very close to our final aim, i.e., use the language we presented as automatic refactorings detection and application, where metrics calculation returns an improvement in metamodel quality [25].

Concerning the catalog of refactorings in object oriented programming presented by Fowler [10] there is a strong correlation with the library of refactoring we defined by using the proposed Edelta language. For the running example in section III-C a library of reusable metamodel refactorings has been implemented following the definitions in http://www.metamodelrefactoring.org/. The proposed metamodel refactoring catalog presents many similarities with the one maintained by Fowler for the reasons already exposed above.

The work in [3] presents a metamodel called *Change Metamodel* for specifying atomic operations specification with a related classification. A metamodel change is seen as a transformation of one version of a metamodel to another. The changes are then classified by their impact on the compatibility to existing model data and this classification is formalized using OCL constraints. The metamodel presented in that paper is quite similar to the Edelta metamodel, even if the implementation is not cited in [3]. The derived classification can be in the future included in Edelta with the intent to estimate the impact of metamodel refactorings on existing artifacts, as we already proposed in [26].

The *Wodel* domain specific language is presented in [27] with the intent of facilitating the specification and creation of model mutations by means of a meta-model independent approach. The language offers primitives for model mutations (e.g., creation, deletion, reference reversal), item selection strategies, and composition of mutations. As in Edelta, Wodel specifications are compiled into Java, and can be extended with post-processor steps for particular applications. The differences with Edelta are in the abstraction layers in which the refactoring / mutation is applied, but Edelta can be easily adapted and extended in order to support also instances refactorings, and this is part of the future plans.

VII. Conclusions

In Model Driven Engineering metamodels play a key role since they underpin the definition of different kinds of artifacts including models, transformations, and code generators. Similarly to any kind of software artifacts, metamodels can evolve under evolutionary pressure that arises when clients and users express the need for enhancements. Metamodel refactorings are typically operated in ad-hoc manners and by means of manual and error-prone editing processes.

In this paper we proposed Edelta, a domain specific language for specifying and applying reusable metamodel refactorings. The language allows the developer to specify both atomic and complex changes. The implementation of Edelta is based on Xtext and the language is endowed with an Eclipse-based development environment providing also early evaluation of the refactoring being applied. As future work we plan to extend the language and the supporting environment by introducing the notion of quality models. Essentially, modelers will have the possibility to assess the refactoring impact on the quality attributes that might be defined and associated to the metamodel being changed.

References

[1] K. O. Elish and M. Alshayeb, "A classification of refactoring methods based on software quality attributes," *Arabian Journal for Science and Engineering*, vol. 36, no. 7, pp. 1253–1267, Nov 2011. [Online]. Available: http://dx.doi.org/10.1007/s13369-011-0117-x

[2] M. Herrmannsdoerfer, "COPE - A workbench for the coupled evolution of metamodels and models," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6563 LNCS, pp. 286–295, 2011.

[3] E. Burger and B. Gruschko, "A Change Metamodel for the Evolution of MOF-Based Metamodels." *Modellierung*, vol. 161, pp. 285–300, 2010.

[4] R. Mahouachi, M. Kessentini, and M. Ó. Cinnéide, *Search-Based Refactoring Detection Using Software Metrics Variation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 126–140. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39742-4_11

[5] L. M. Rose, D. S. Kolovos, R. F. Paige, F. A. C. Polack, and S. Poulding, "Epsilon Flock: a model migration language," *Software and Systems Modeling*, pp. 1–21, 2012.

[6] B. Gruschko, "Towards synchronizing models with evolving metamodels," in *In Proc. Int. Workshop on Model-Driven Software Evolution held with the ECSMR*, 2007.

[7] A. Narayanan, T. Levendovszky, D. Balasubramanian, and G. Karsai, "Automatic domain model migration to manage metamodel evolution," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5795 LNCS, pp. 706–711, 2009.

[8] B. Meyers and H. Vangheluwe, "A framework for evolution of modelling languages," *Science of Computer Programming*, vol. 76, no. 12, pp. 1223–1246, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2011.01.002

[9] T. Mens, S. Demeyer, B. D. Bois, H. Stenten, and P. V. Gorp, "Refactoring: Current research and future trends," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 3, pp. 483 – 499, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066105826246

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[11] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "Cope - automating coupled evolution of metamodels and models," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 52–76. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_4

[12] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *12th International IEEE ECOC 2008, 15-19 September 2008, Munich, Germany*. IEEE Computer Society, 2008, pp. 222–231.

[13] A. Cicchetti, D. Di Ruscio, and A. Pierantonio, "Managing Dependent Changes in Coupled Evolution," in *ICMT*, ser. LNCS. Springer, 2009, vol. 5563, pp. 35–51.

[14] D. Di Ruscio, L. Iovino, and A. Pierantonio, "Evolutionary Togetherness: How to Manage Coupled Evolution in Metamodeling Ecosystems," in *ICGT*, vol. 7562. Springer, 2012, pp. 20–37.

[15] B. G. Humm and R. S. Engelschall, "Language-oriented programming via dsl stacking," in *Proceedings of the 5th International Conference on Software and Data Technologies*, 2010, pp. 279–287.

[16] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[17] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*, 2nd ed. Packt Publishing, 2016.

[18] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, W. Hasselbring, and R. von Massow, "Xbase: Implementing Domain-Specific Languages for Java," in *GPCE*. ACM, 2012, pp. 112–121.

[19] M. D. Univaq, "Metamodel refactorings catalog," http://www.metamodelrefactoring.org, 2011, accessed: 2016-09-30.

[20] D. Di Ruscio, L. Iovino, and A. Pierantonio, "What is needed for managing co-evolution in mde?" in *Proceedings of the 2nd IWMCP'11*. ACM, 2011, pp. 30–38.

[21] D. Wagelaar, L. Iovino, D. Di Ruscio, and A. Pierantonio, "Translational semantics of a co-evolution specific language with the EMF transformation virtual machine," in *ICMT*, ser. LNCS. Springer, 2012, vol. 7303, pp. 192–207.

[22] T. Mens, *On the Use of Graph Transformations for Model Refactoring*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 219–257. [Online]. Available: http://dx.doi.org/10.1007/11877028_7

[23] T. Arendt, F. Mantz, and G. Taentzer, "Emf refactor: specification and application of model refactorings within the eclipse modeling framework," in *Proceedings of the BENEVOL workshop*, 2010.

[24] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Proceedings of the 5th International Workshop on Software Quality*, ser. WoSQ '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 10–. [Online]. Available: http://dx.doi.org/10.1109/WOSQ.2007.11

[25] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Mining metrics for understanding metamodel characteristics," in *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, ser. MiSE 2014. New York, NY, USA: ACM, 2014, pp. 55–60. [Online]. Available: http://doi.acm.org/10.1145/2593770.2593774

[26] L. Iovino, A. Pierantonio, and I. Malavolta, "On the Impact Significance of Metamodel Evolution in MDE," *JoT*, vol. 11, no. 3, pp. 3:1–33, Oct. 2012.

[27] P. Gómez-abajo, E. Guerra, J. D. Lara, P. Gomeza, and E. Guerra, "Wodel : A Domain-Specific Language for Model Mutation," pp. 1–6, 2016.