Edelta 2.0: Supporting Live Metamodel Evolutions

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

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

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

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

ABSTRACT

Evolving metamodels is a delicate task, both from the programming effort's point of view and, more importantly, from the correctness point of view: the evolved version of a metamodel must be correct and must not contain invalid elements (e.g., dangling references). In this paper we present the new version of Edelta, which provides EMF modelers with linguistic constructs for specifying both basic and complex refactorings. Edelta 2.0 is supported by an Eclipse-based IDE, which provides in this new version a "live" development environment for evolving metamodels. The modelers receive an immediate feedback of the evolved versions of the metamodels in the IDE. Moreover, Edelta performs many static checks, also by means of 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 refactoring operations specified by the user. Finally, Edelta 2.0 allows the users to easily introduce additional validation checks in their own Edelta programs, which are taken into consideration by the Edelta compiler and the IDE.

CCS CONCEPTS

• **Software and its engineering** → **Model-driven software engineering**; **Software system models**; **Software system structures**; *Software organization and properties.*

KEYWORDS

Metamodels, Evolution, Edelta, Refactoring

ACM Reference Format:
Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2020. Edelta 2.0: Supporting Live Metamodel Evolutions. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3417990.3419501

MODELS '20 Companion, October 18–23, 2020, Virtual Event, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8135-2/20/10…$15.00
https://doi.org/10.1145/3417990.3419501

1 INTRODUCTION

Metamodels are critical components for the definition of modeling artifacts e.g., models, transformations, editors, and code generators. Metamodels are subject to evolutionary pressure, similarly to any other software artifact, and this activity is named evolution or refactoring, depending on the nature of the applied changes [9]. In this paper, we treat the two categories in the same manner, and we use the two terms interchangeably. A metamodel refactoring can be expressed as an atomic operation, e.g., addition, deletion, and change of meta-elements or as a sequence of them. However, atomic changes are not enough for managing complex evolution scenarions, which require the adoption of complex operations, which are defined in terms of aggregations of atomic changes [8, 10, 14, 15, 33]. Various approaches supporting the evolution [26] and coupled evolution [17, 18] of modeling artifacts have been presented even though the provided aggregation mechanisms supporting the reuse of already defined refactorings are still limited. In [4] we proposed Edelta, a domain-specific language and supporting tools for specifying and applying reusable metamodel refactorings. However, the facilities provided by the initial versions of Edelta were still limited, especially concerning the supporting tools.

In this paper, we present a new version of Edelta that improves the original one under different dimensions: a new safe and fluent runtime API has been implemented; the refactoring IDE permits to execute refactorings in a complete live manner; a continuously up-to-date view of the metamodels under modification has been integrated into the IDE, even in case of errors in the program; statically detected informational errors are contextual to the refactoring flow and are shown by the IDE; a completely new quick-fixes mechanism has been integrated as well as a smart content assist. Improved navigation to the metamodel elements is now available in the IDE, and ambiguous metamodel references are identified.

The paper is organized as follows: Section 2 describes the improvements and the new features of Edelta 2.0, with a particular focus on the runtime library and the DSL. Section 3 offers a discussion of the new characteristics of the compiler and the interpreter, whereas the renewed development environment is described in Section 4. Related works are discussed in Section 5 and Section 6 draws some conclusions and future lines of work.

2 EDELTA 2.0

This section describes the new version of Edelta, a language and supporting framework, which provides EMF modelers with linguistic constructs for specifying basic metamodel refactorings (i.e., additions, deletions and a few basic changes), and complex reusable metamodel changes by properly composing already existing ones. We use the term "Ecore models" and "metamodels" to refer to the same concepts in the remainder of this paper.

Edelta is an open source project available at https://github.com/LorenzoBettini/edelta and we provide also an Eclipse update site and a complete Eclipse distribution with Edelta installed. It is important to mention that we provide also a Maven plugin to compile Edelta programs in a headless way, e.g., in a Continuous Integration server. The above git repository also contains the project `edelta.examples` with a few examples and case studies of metamodel evolutions with Edelta.

To better show the novelties of Edelta, Fig. 1 shows the evolution process supported by the new version of Edelta consisting of the following phases:

(1) In this phase, the modeler inspects the metamodels to find inconsistencies or concepts not up to date with the engineered domain.
(2) It is an optional step towards the application of possible refactorings, and it consists of the definition or the import of existing general refactorings organized in libraries (e.g., extract metaclass, merge attributes, etc.). An example of Edelta library has been defined in [4] and a refactoring catalog has been previously organized in [23].
(3) This phase consists of an analysis process on the candidate metamodels in order to understand what types of refactorings should be applied. Automatic bad smell matching [5] or anti-patterns [30] can be applied in order to identify candidate elements to be refactored.
(4) In this phase, the modifications are eventually applied in a dedicated Edelta program. With a reuse mechanism, it is possible to import existing libraries of refactorings and invoke them on a specific subject metamodel. As we describe later (Section 3 and 4), the new Edelta environment provides immediate feedback, giving place to the live effect that the modeler can see if the refactorings are valid; alternatively, errors will be immediately raised in case the validation is not completed positively in phase **5**.
(5) In this phase, the effect is immediately seen using the live evolution editor, and in case of inconsistent refactorings, feedback will be provided.
(6) It is an optional activity that is only considered if the effect of the selected refactorings does not reflect what the modeler wanted to achieve: the process can be iterated many times until the modeler obtains the desired artifact. As a last resort, a rollback can be invoked to restore the initial version of the metamodel and to restart again with this process.

This process is fully supported by the new Edelta implementation and by its IDE as described in the next sections. As detailed in the rest of the paper, the modeler can always have an immediate view of the metamodels with the applied modifications. However, the original metamodels are never touched and all the modifications are applied to a copy of the metamodels handled by Edelta automatically.

The overall implementation of Edelta basically consists of two collaborating parts: i) a Java runtime library, which implements the basic operations for manipulating an Ecore model; ii) a DSL that is compiled into Java code that uses the Edelta runtime library. The Edelta DSL focuses on many static checks in order to catch most problems during the compilation of the program, so that the generated Java code, when executed, does not create an invalid metamodel (we explain this aspect throughout the rest of the paper). Moreover, it also aims at providing a powerful Eclipse IDE support, implementing a live view on the evolution of the metamodel while the user is writing an Edelta program in the Eclipse Edelta editor. Concerning the DSL, an Edelta program can contain:

- definitions of "reusable operations", which are common general refactorings and are meant to be reused both in the same program and in other Edelta programs;
- modifications (evolutions and refactorings) for specific metamodels, which are not meant to be reused outside that Edelta program.

In this paper we concentrate on the latter, since the shape and features of reusable operations have not changed since the first presentation of Edelta [4]. On the contrary, the new version of Edelta that we present in this paper has completely re-implemented the second kind of linguistic mechanisms.

ID	**Features**	Edelta [4]	**Edelta 2.0**
1	Basic Refactoring	✓	✓+
2	Fluent-style DSL	✓	✓+
3	Safe API (w.r.t. EMF)	-	✓
4	Reusable operations	✓	✓
5	Readable syntax	✓	✓+
6	On-the-fly interpretation	✓	✓+
7	Execution flow-aware	-	✓
8	Quickfixes support	-	✓
9	Static safety	-	✓
10	Error recovery support	-	✓
11	Up-to-date content assist	-	✓
12	Advanced navigability	-	✓
13	Dynamic extensibility	-	✓

Table 1: New and improved features of Edelta 2.0 w.r.t. the previous version presented in [4]

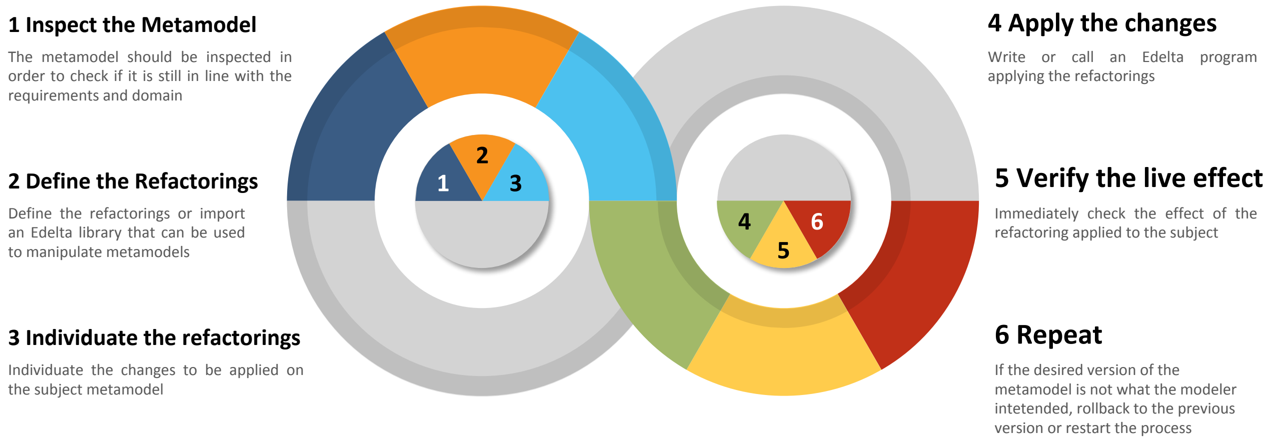
In Table 1 we highlight the new features implemented in the new version of Edelta w.r.t. the old version presented in [4]. We use - to represent a feature or characteristic not available in the previous version, with ✓an available feature, and with ✓+ a feature or characteristic that has been improved since the first version. All these features will be discussed in detail in the remaining sections referring them with the corresponding label id.

In the following subsections we present first the *runtime library* and then the *DSL*.

2.1 Runtime library

The Edelta runtime library provides a set of Java APIs that implement the main basic refactoring operations for manipulating an Ecore model loaded in memory (1). The runtime library also provides mechanisms for saving the changed Ecore model into a new file. In this new version of Edelta we provide a new API, which aims

Figure 1: Continuous Metamodel Evolution Process supported by Edelta


1 Inspect the Metamodel
The metamodel should be inspected in order to check if it is still in line with the requirements and domain
2 Define the Refactorings
Define the refactorings or import an Edelta library that can be used to manipulate metamodels
3 Individuate the refactorings
Individuate the changes to be applied on the subject metamodel
1
2
3
4
5
6
4 Apply the changes
Write or call an Edelta program applying the refactorings
5 Verify the live effect
Immediately check the effect of the refactoring applied to the subject
6 Repeat
If the desired version of the metamodel is not what the modeler intetended, rollback to the previous version or restart the process

at being safe (as detailed in the following) and at providing a *fluent* API [12] (②). In particular, the new runtime library replaces the syntactic forms for basic refactoring operations that were part of the previous Edelta DSL (more details will be provided in Section 3). Our API leverages the standard EMF runtime API [29] but aims at improving it under many respects. In particular, our API for creating Ecore elements is based on required features, so that the developer does not risk to create new Ecore elements or change existing ones that make the Ecore model invalid. For example, an EAttribute requires its feature eType to be set as an EDataType (though the type of the corresponding Java field, inherited from ETypedElement, is of supertype EClassifier). By using the standard EMF API it is easy to create an invalid model by forgetting to call setEType or by passing an EClassifier that is not an EDataType. Similarly, it is easy to forget to add the created elements into the corresponding containers, leading to a model with dangling references.

On the contrary, our Java API minimizes the chances to create invalid Ecore models, because of the *Safe API* (③). For example, this is our method for creating a new EAttribute and for adding it into an EClass:

```
public EAttribute addNewEAttribute(EClass eClass, String name,
  EDataType dataType, Consumer<EAttribute> initializer)
```

The method has a parameter for the required feature with the proper Java type. The methods in our Edelta runtime Java library have this shape, returning the element that has been created and added to a container. This allows the client code, that is, the Edelta code, to use method chaining for easily creating complex models, using the lambda passed as the last argument to further initialize the created element, as we will show in the next sections. Such a lambda is executed only after the element has been added to the container.

The Edelta runtime consists of several methods of the above shape, including methods for adding a superclass, for removing elements from a container, etc. As we will see in Section 4, Edelta provides content assist, which can be used to easily find and select the appropriate method of our Java API.

Besides this runtime library, which is strictly required by the code generated by the Edelta compiler (shown in the next section), Edelta also provides an additional library with some standard refactorings and *code smell* detectors and solvers. This can be optionally used in Edelta programs. Such a library has been partly presented in [5].

2.2 The DSL

The Edelta DSL has been implemented with Xtext [3], the most popular Eclipse framework for the development of programming languages and DSLs. Xtext, besides generating the infrastructure for the compiler, also generates a complete IDE support based on Eclipse. This includes an editor with syntax highlighting and code completion and typical IDE mechanisms like immediate error reporting, incremental building, navigation to definitions and outline.

Given an Edelta program, the Edelta compiler generates a Java file, which relies on the Edelta Java runtime library (Section 2.1). The generated Java code has no further dependency on the Edelta compiler, nor on the Xtext framework itself.

In the rest of this section we briefly describe the syntax and informally the semantics of Edelta programs.

First of all, the EPackages that are used in the Edelta program must be explicitly specified by their name, using the instruction metamodel "..." at the beginning of the Edelta program. These include both the EPackages under modifications and the EPackages that are simply referred to in the program. An example of the latter is the Ecore metamodel (Ecore.ecore), which can be 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.

A reusable operation ④ is defined in Edelta with the following syntax:

```
def <name>(<... parameters ...>) : <returntype> {
  <body>
}
```

Parameter declarations have the same syntax as in Java. The return type can be omitted and will be inferred by the compiler.

A modification operation for an imported EPackage is defined in this new version of Edelta with the following syntax:

```
modifyEcore <name> epackage <EPackage name> {
  <body>
}
```

The bodies of reusable operations and of `modifyEcore` specifications consist of Xbase [11] expressions. In order to make code snippets presented in the paper comprehensible, we also briefly describe the features of Xbase that are used by Edelta.

Xbase, which is part of the Xtext framework, is an extensible and reusable expression language that is meant to be embedded in an Xtext DSL. By embedding Xbase in an Xtext language like Edelta, one gets for free a rich Java-like expression language, including its type system, which is the same as Java, and Java code generation. In particular, Xbase is completely interoperable with Java: all existing Java types and Java libraries can be seamlessly reused in an Edelta program.[1]

Because of its compact syntax and of its powerful type inference, Xbase expressions have the short and readable shape that are typical of a dynamically (i.e., untyped) language, while retaining all the benefits of a statically typed language. The syntax of Xbase is Java-like, thus, it should be easily understood by Java programmers. It includes Java-like OOP constructs such as `new`, `instanceof`, casts and method invocations. However, Xbase aims at avoiding much of the "syntactic noise" verbosity that is typical of Java. For instance, in Xbase, terminating semicolons are optional. 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.

Variable declarations in Xbase start with `val` or `var`, for final and non final variables. The type can be omitted if it can be inferred from the initialization expression. Operators such as `+`, `-`, `+=`, `-=` are available in Xbase also on collections, with the expected semantics. Moreover, Xbase also provides syntactic sugar for getters and setters: one can simply write `o.name` instead of `o.getName()`; similarly, `o.name = "..."` corresponds `o.setName("...")`. For example, if `p` is an `EPackage` and `e` an `EClassifier`,
`p.EClassifiers -= e`
corresponds to
`p.getEClassifiers().remove(e)`.

Xbase *extension methods* are a syntactic sugar mechanism to simulate adding new methods to existing types without modifying them. The first argument of a method invocation can be used as its receiver, as if the method was one of the argument type's members. Extension methods allow for a more readable fluent style based on method chaining. We will use existing methods from our runtime library (Section 2.1) and from external libraries as extension methods in Edelta programs.

Besides `this`, Xbase has another additional special variable, `it`. Similarly to `this`, `it` can be omitted as object receiver of method call and member access expressions. The programmer can declare any variable or parameter with the name `it`. This way, a custom implicit object receiver can be defined in any scope of the program.

Xbase *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`.

[1]Java types must be explicitly imported in an Edelta program, using the same "import" syntax as in Java. The Edelta Eclipse editor provides code completion for imports and the typical Eclipse "Organize Imports" mechanism.

Because of all the linguistic mechanisms just mentioned, the programmer can easily write statements and expressions that are compact and more readable than in Java (5). For example, by leveraging the Edelta Java API described in Section 2.1, we can create an `EClass` with a required `EAttribute` like this in Edelta:

```
ePackage.addNewEClass("MyClass") [
  addNewEAttribute("myAttr", ecoreref(EString)) [
    lowerBound = 1
  ]
]
```

We use the `addNewXXX` methods from our API. Note that `addNewEClass` is used as an extension method: it is not a method of the type `EPackage`, but the `EPackage` instance is used as the receiver instead of as the first argument. The implicit receiver of the method `addNewEAttribute` is the implicit lambda parameter `it`; again, we use it as an extension method, since `addNewEAttribute` takes as the first argument an `EClass`. The type of `it` in the outer lambda is inferred as `EClass`. The type of the implicit parameter `it` in the inner lambda is inferred as `EAttribute`, so everything is statically well-typed.

The `ecoreref(EString)` expression is part of our customization of the Xbase syntax. This specific syntax allows the programmer to refer to Ecore elements in a statically typed way. In fact, Edelta programs refer directly to the model classes of an Ecore, 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. References to Ecore elements, such as packages, classes, data types, features and enumerations, can be specified by their fully qualified name in an `ecoreref` using the standard dot notation, or by their simple name if there are no ambiguities (possible ambiguities are checked by the compiler, as explained later). For example, we can refer to the Ecore element `EString` by `ecoreref(EString)` or by `ecoreref(ecore.EString)`. Let us stress again that the argument of `ecoreref` is an Ecore element, not a Java type.

In a `modifyEcore` specification's block, Edelta automatically declares the variable `it` of type `EPackage`, which refers to the `EPackage` specified in the `epackage` clause. For example, let us consider this code snippet:

```
metamodel "ecore"
metamodel "myecore"

modifyEcore someChanges epackage myecore {
  // 'it' in this example refers to the EPackage myecore
  addNewEClass("MyClass") [
    addNewEAttribute("myAttr", ecoreref(EString)) [
      lowerBound = 1
    ]
  ]
  ...
}
```

Here we use `addNewEClass` as an extension method without specifying the receiver, which is the implicit receiver `it`.

Finally, we just mention here that the reusable operations specified in Edelta programs can be imported in other Edelta programs with the clause `use ... as ...`. The referred code must be specified as a dependency of the project. This works both for your own reusable operations and the ones that we provide together with Edelta, described in [5].

When an Edelta program is saved and no errors are found by the compiler, the corresponding Java code is generated for manipulating the input metamodels as specified in the Edelta program. In particular, the Java method `execute` is also generated: this method calls all the Java methods corresponding to the `modifyEcore` operations in the order they are defined in the original Edelta program. We provide an Eclipse wizard for creating an Edelta project with the required dependencies, an example Edelta program and an example Ecore file, which is evolved in the example program. Moreover, a Java file with a `main` is also generated that calls the generated Java code. This file can be seen as a starting point for the programmer.

3 THE COMPILER AND THE INTERPRETER

In this section we describe the main features of the Edelta compiler and its interaction with the Edelta interpreter, which provides a live and up-to-date view of the metamodels being modified in the program in each specific part of the program. In particular, we will stress the differences with respect to the previous version of Edelta.

One of the main and distinguishing features of the Edelta compiler is that it automatically keeps track of the metamodels modified by the program according to the operations that are specified in the `modifyEcore` operations of the program. During parsing and validation, the Edelta compiler interprets on-the-fly such operations (6). During the interpretation the evolution operations are applied to a copy of the Ecore models imported in the program. This way, while the developer is writing an Edelta program in the Eclipse Edelta editor, the new and modified elements are immediately available: they can be referred using `ecoreref` and they are used for static type checking. This is what we call *Live Evolution*.

For example, let us consider this snippet:

```
modifyEcore ... {
  addNewEClass("ANewClass")
  val aNewClass = EcoreFactory.eINSTANCE.createEClass
  aNewClass.name = "AnotherNewClass"
  EClassifiers += aNewClass
  // correctly resolved and type-checked
  ecoreref(AnotherNewClass).ESuperTypes += ecoreref(ANewClass)
}
```

Since the compiler interprets the operations during validation and type checking, it can correctly resolve the `ecoreref` references and it is also able to correctly verify that the last expression is correct, since it knows that `ecoreref(ANewClass)` and `ecoreref(AnotherNewClass)` are of type `EClass`. We stress that in the above example we created the new `EClasses` both by using our runtime API (`addNewEClass`) and the standard EMF API (i.e., by first creating a new `EClass` with the standard `EcoreFactory`, by setting its name and by finally adding the created `EClass` to the `EPackage`'s classifiers). The interpreter does not make any distinction: it interprets the operations and updates the in-memory modified metamodel.

This was not the case in the previous implementation of Edelta: we were updating the in-memory model only when the program was using our own specific syntactic expressions for creating and modifying model elements. Indeed, while the interpretation during the compilation was a feature present in the first version of Edelta presented in [4], in the new version of Edelta that we are presenting in this paper, such a mechanism has been completely rewritten and made much more powerful. The original implementation was a proof of concept for the feasibility of the interaction between the compiler and the on-the-fly interpretation. However, it did not use any flow of control information. This means, for example, that the original implementation did not guarantee that the generated Java code did not encounter a run-time error due to a wrong reference to an element. Moreover, the generated Java code could create an invalid Ecore model during the execution of evolution operations. In the current new implementation all of these problems are not there anymore, as we describe in this section.

The switch to a complete new syntax for specifying specific metamodel evolutions (i.e., `modifyEcore`) instead of the previous dedicated syntactic forms, allowed us to implement a more robust interpretation during the compilation. Besides this technical motivation, we also think that a generic `modifyEcore` operation definition acting on a specific `EPackage` is also more readable and more maintainable for the programmer than the previous single syntactic forms. In fact, in the original Edelta DSL, one had to use specific syntactic expressions `createEClass`, `modifyEClass`, `createEAttribute`, etc. Each of such expressions was part of the grammar definition of Edelta, making it hard to introduce new features to the language. Since now we switched to a complete runtime API approach (apart from the simple syntax for `modifyEcore`), extending the language basically means extend the run-time API or inject custom operations, which can be easily done also by external contributors by simply providing reusable operations, without changing the grammar of Edelta. In particular, the users can participate to the validation phase as detailed in Section 4.1.

The on-the-fly interpretation is applied sequentially on all expressions of the `modifyEcore` operations. This way, the Edelta compiler is able to keep track of the current state of the Ecore models being modified in each specific context of the program. The compiler can then detect possible errors due to references to stale elements in a specific part of the program: in that context such elements have been already removed or renamed. This mechanism, introduced in the new version of Edelta, can be defined as *execution flow-awareness* (7). This way, we can provide meaningful and useful comments, as shown in this example:

```
modifyEcore ... {
  removeEClassifier(ecoreref(MyExistingClass))
  ecoreref(MyOtherExistingClass).name = "ANewName"
  ...
  // ERROR: MyExistingClass is not available anymore in this
       context
  ecoreref(MyExistingClass)
  // ERROR: MyOtherExistingClass in now available with name
       ANewName in this context
  ecoreref(MyOtherExistingClass)
}
```

Being able to detect meaningful errors (instead of generic "unresolved reference" errors) in the compiler also means that in the IDE we can provide useful quickfixes for such error situations. The *quickfix support* (8) will be shown in Section 4 (Fig. 4). All of these mechanisms were not present in the previous version of Edelta.

Our interpreter is built on top of the `XbaseInterpreter`. By default, the `XbaseInterpreter`, once configured, is able to interpret all the standard Xbase expressions.[2] Moreover, it also interprets any Java code (even in binary-only format) by relying on standard Java reflection. We customized such an interpreter to keep into consideration also our `ecoreref` expressions and our reusable functions. 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 modifications performed by the generated code will still be valid Ecore models. We defined this aspect as *static safety* (9).

One important aspect to keep into consideration, especially in the context of an IDE, is that for most of the time, while the programmer is writing code in the editor, the program is invalid. An editor in an IDE continuously validates the program while the programmer is typing in the editor, thus, the editor (i.e., the compiler and, in our case, also the interpreter) must be able to deal with error recovery. This means that a validation error in a part of the program must not lead to many cascading errors in the rest of the program. Typical examples are syntax errors, leading to a malformed AST node or symbol resolution errors, like the call to a non existing method or a reference to a non existing Ecore element. Indeed, errors must be placed only in the important parts of the program, with meaningful and helpful error messages. On the contrary, subsequent cascading errors would only distract the programmer, making it hard to understand the real cause of the problem. In our context, we have to keep error recovery into consideration also when interpreting the program on the fly. If the interpreter is evaluating an expression that contains an error, an exception will be thrown during such an evaluation. The Edelta interpreter takes care of handling such exceptions gracefully, and tries to go on interpreting as much as possible. Thus, concerning the *error recovery support* (10), expressions that change the Ecore under evolution will be executed also in the presence of errors in the program and references to new or modified elements in the rest of the program will be resolved.

For example, in the following snippet, we have a syntax error in the first line, but the interpreter recovers from such an error and keeps on interpreting the rest of the program. In particular, the created `EClass` can be correctly referred in the last line and the only error reported is on the syntax error.

```
modifyEcore someChanges epackage myecore {
  ecoreref MyClass).abstract = false // syntax error: missing (
  addNewEClass("MyNewClass") // this is interpreted anyway
  ...
  ecoreref(MyNewClass).abstract = true // this can be resolved
}
```

The interpreter in the previous implementation of Edelta did not enjoy such error recovery mechanisms and in many cases this led to too many errors reported in the program, making it difficult to understand the real cause of the problems. For example, in the above snippet, it would have marked also the last line with errors: “Unresolved reference `MyNewClass`” and “Unresolved symbol `abstract`”.

[2]The details of the configuration of the `XbaseInterpreter` is out of the scope of the present paper. However, such a configuration is not well documented in Xtext and in future work we might describe the technical details of our use of the interpreter in Edelta, so that the same technique could be reused also in other language implementations.

Other errors detected during compilation that we did not handle in the previous implementation of Edelta are shown in Section 4, where we also show the quickfixes proposed by the Edelta editor.

4 THE DEVELOPMENT ENVIRONMENT

The Edelta editor provides typical Eclipse IDE features, like content assist (a few mechanisms of the content assist have already been described in the previous sections), navigation to definitions and quick-fixes. These mechanisms are available both for Java types and for Ecore model elements.

Because of the continuous interpretation described in Section 3, the content assist also takes into consideration elements added or renamed in the program, in a specific program context. For example, in Fig. 2 the content assist also proposes the `EClass` that has been added (by means of an `addNewEClass("MyNewEClass")`), the one that has been renamed and the ones present in the imported metamodel that are not modified. Indeed, the original `MyEClass` is not proposed in that program context, while its renamed version is proposed, `MyRenamedEClass`. This is defined as *Up-to-date content assist* (11).

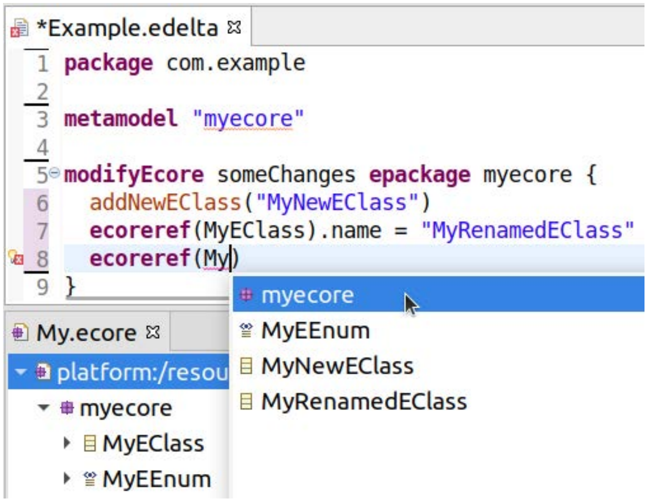
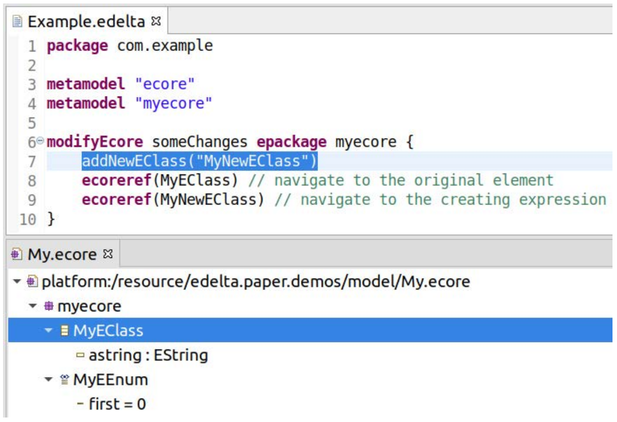

*Example.edelta
```
package com.example

metamodel "myecore"

modifyEcore someChanges epackage myecore {
  addNewEClass("MyNewEClass")
  ecoreref(MyEClass).name = "MyRenamedEClass"
  ecoreref(My)
}
```

Figure 2: Content assist for Ecore model elements added/-modified in the program.

For this reason, when using the Edelta content assist for specifying references to Ecore elements (i.e., `ecoreref`) in any program context, the programmer does not risk to introduce errors (e.g., access to renamed or removed elements, Section 3). In fact, the Edelta content assist only proposes completions that are valid in that specific program context. Of course, for this to work correctly, it is crucial that our interpreter is able to recover from possible errors in the program, as explained in Section 3.

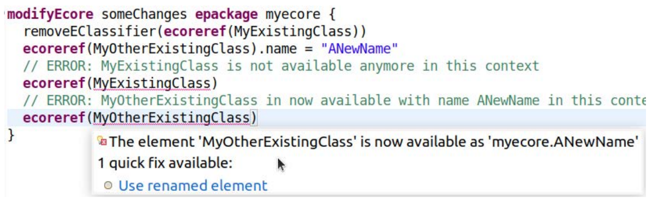
Navigating to Ecore model elements that are present in the original imported metamodel automatically opens the standard EMF Ecore tree editor. However, if we try to navigate from a reference that refers to an element added or renamed in the program, the destination will be the expression in the program that added or renamed that element. For example, in Fig. 3 from the `ecoreref(MyEClass)` we navigate to the element in the original Ecore file (note the highlighted element in the automatically opened EMF Ecore tree editor), while from `ecoreref(MyNewEClass)` we navigate to the expression that created that element (note the highlighted expression `addNewEClass("MyNewEClass")`). This form of *advanced navigability* (12) has been introduced with this new version.



```
package com.example

metamodel "ecore"
metamodel "myecore"

modifyEcore someChanges epackage myecore {
    addNewEClass("MyNewEClass")
    ecoreref(MyEClass) // navigate to the original element
    ecoreref(MyNewEClass) // navigate to the creating expression
}
```

Figure 3: Navigation to Ecore model elements.

As we said in Section 3, when interpreting the program we collect information about possible errors due to accessing an element that is not available anymore in a specific program context. In case the program is trying to access an element that has been renamed, the editor provides a useful quickfix to correct the reference by using the renamed element, as shown in Fig. 4.



```
modifyEcore someChanges epackage myecore {
  removeEClassifier(ecoreref(MyExistingClass))
  ecoreref(MyOtherExistingClass).name = "ANewName"
  // ERROR: MyExistingClass is not available anymore in this context
  ecoreref(MyExistingClass)
  // ERROR: MyOtherExistingClass in now available with name ANewName in this conte
  ecoreref(MyOtherExistingClass)
}
```

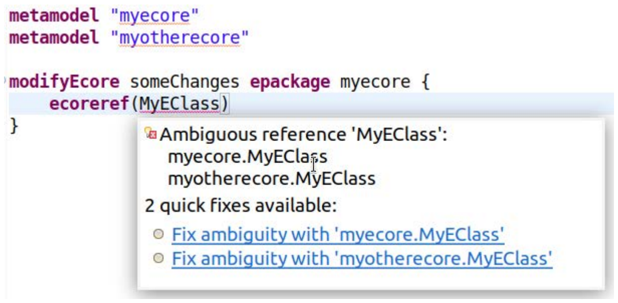
The element 'MyOtherExistingClass' is now available as 'myecore.ANewName'
1 quick fix available:
Use renamed element

Figure 4: Quickfix for using the renamed element.

The new version of the Edelta compiler also checks possible ambiguous reference to Ecore elements. In case of such an ambiguity, the Edelta editor shows some quickfixes to help the programmer solve the ambiguity by proposing a fully qualified reference, as shown in Fig. 5.

Besides allowing our compiler to perform static type checking, keeping the in-memory Ecore model continuously updated by interpreting the specified evolving operations also allows us to provide the developer with an immediate view of the modified Ecore model in the Outline view. New elements created in the program or modified in the program appear in the Outline view. Clicking on such an element in the Outline view tree highlights in the editor the expression responsible of the creation or modification of that element.

Finally, one of the benefits of using Xbase is that we can debug Edelta programs while running the generated Java code in the Eclipse debugger: all the standard Eclipse debugging features, e.g.,



```
metamodel "myecore"
metamodel "myotherecore"

modifyEcore someChanges epackage myecore {
    ecoreref(MyEClass)
}
```

Ambiguous reference 'MyEClass':
myecore.MyEClass
myotherecore.MyEClass
2 quick fixes available:
Fix ambiguity with 'myecore.MyEClass'
Fix ambiguity with 'myotherecore.MyEClass'

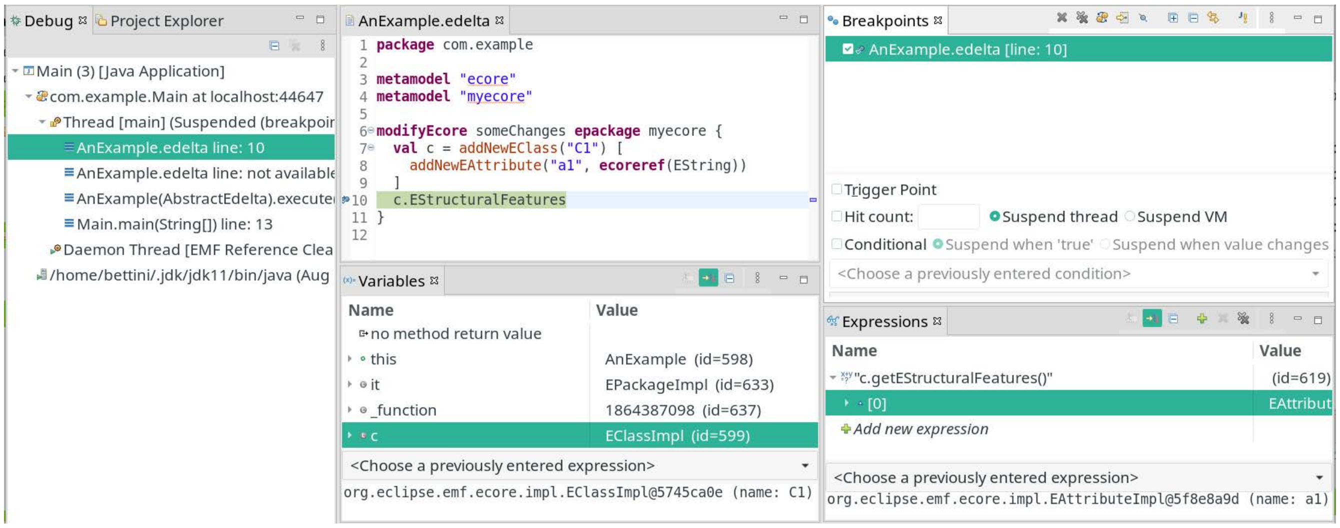
Figure 5: Quickfix in case of an ambiguous reference.

the views "Variables", "Breakpoints" and "Expressions", are available directly on the Edelta program source code. An example of a debugging session is shown in Figure 6. Indeed, while debugging the generated Java code is still possible, we believe that being able to debug the original Edelta source code is crucial for productivity.

For lack of space, we only mention some other new features in the version of Edelta presented in this paper. Subpackages are now handled by Edelta, including spotting possible cycles in the subpackage relation. Possible cycles are also detected in the supertype relation in `EClasses`. These new features are an improvement to the basic refactoring operations with respect to the previous version of Edelta. Moreover, Edelta is now able to handle evolutions and refactorings spanning several (possibly mutually dependent) metamodels.

4.1 Dynamic Extensibility

Besides the advantages that we showed in the previous sections of the re-implementation of the Edelta DSL/compiler/interpreter infrastructure, one of the main goals of such a new version of Edelta was also its *dynamic extensibility* (13) not only from our developers' perspective but also from the users' perspective. We introduced the ability to participate to the validation phase of the compiler and interpreter, without having to customize the Edelta implementation itself. In fact, the standard Edelta runtime library now provides two specific methods `showError` and `showWarning` that the Edelta users can call in their own Edelta programs to "show" errors and warnings, respectively, specifying also the cause of the issue (typically an Ecore model element). The standard Edelta interpreter catches such method calls and uses the arguments of such method calls to create errors and warnings, respectively, on the program. We demonstrate this feature in Fig. 7. The program defines a reusable function, `checkEClassifierNames`, which checks that all `EClassifiers` in a given `EPackage` have a name that start with a capital letter (according to the standard Java convention). If an `EClassifier` does not satisfy such a property, it calls `showWarning` passing the element to blame and a message. In the `modifyEcore` the program adds a few `EClassifiers` and as a last expression it calls the reusable function (as an extension method where the implicit parameter `it` is the `EPackage` under modification). You can see that the interpreter effectively calls `checkEClassifierNames` and the editor shows a warning marker on the expression that created the element with the name starting with a lowercase, the `addNewEClass`. On the contrary, the `addNewEDataType` creates an



```
Main (3) [Java Application]
  com.example.Main at localhost:44647
    Thread [main] (Suspended (breakpoir
      AnExample.edelta line: 10
      AnExample.edelta line: not available
      AnExample(AbstractEdelta).execute
      Main.main(String[]) line: 13
    Daemon Thread [EMF Reference Clea
  /home/bettini/.jdk/jdk11/bin/java (Aug
```

```
 1 package com.example
 2
 3 metamodel "ecore"
 4 metamodel "myecore"
 5
 6 modifyEcore someChanges epackage myecore {
 7   val c = addNewEClass("C1") [
 8     addNewEAttribute("a1", ecoreref(EString))
 9   ]
10   c.EStructuralFeatures
11 }
12
```

AnExample.edelta [line: 10]

Trigger Point
Hit count: [✓ Suspend thread] Suspend VM
Conditional [✓ Suspend when 'true'] Suspend when value changes
<Choose a previously entered condition>

Name	Value
no method return value	
this	AnExample (id=598)
it	EPackageImpl (id=633)
_function	1864387098 (id=637)
c	EClassImpl (id=599)

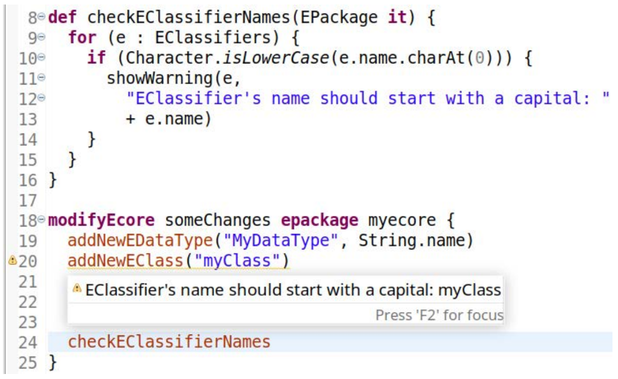
<Choose a previously entered expression>
`org.eclipse.emf.ecore.impl.EClassImpl@5745ca0e (name: C1)`

Name	Value
"c.getEStructuralFeatures()"	(id=619)
[0]	EAttribut
Add new expression	

<Choose a previously entered expression>
`org.eclipse.emf.ecore.impl.EAttributeImpl@5f8e8a9d (name: a1)`

Figure 6: Debugging an Edelta program.

EClassifier with the name starting with a capital, so no warning is shown on that expression. Note that this mechanism implicitly makes use of the information for each interpreted expression collected during the interpretation (described in Section 3).



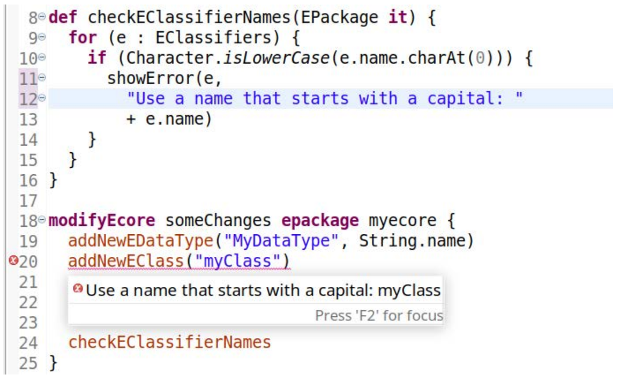
```
def checkEClassifierNames(EPackage it) {
  for (e : EClassifiers) {
    if (Character.isLowerCase(e.name.charAt(0))) {
      showWarning(e,
        "EClassifier's name should start with a capital: "
        + e.name)
    }
  }
}

modifyEcore someChanges epackage myecore {
  addNewEDataType("MyDataType", String.name)
  addNewEClass("myClass")
      EClassifier's name should start with a capital: myClass
                                          Press 'F2' for focus

  checkEClassifierNames
}
```

Figure 7: Warning generated by the program.

In order to show that such a mechanism is effectively dynamic, in Fig. 8 we modify the body of checkEClassifierNames so that it uses showError (instead of showWarning) and passes a different message. You can see that even without saving the file[3] the editor now shows an error marker, instead of a warning marker, on addNewEClass and also the message has changed.

Thus, this mechanism allows the programmer to extend the validation aspect of Edelta on-the-fly, turning the Edelta IDE into a live development environment. Indeed, the user can easily experiment with such validation mechanisms even without saving the file.

Besides letting the users extend the Edelta validations, the mechanism described above also allows us to easily extend Edelta without hardcoding checks in the compiler. Of course, the compiler and the interpreter still have some validations hardcoded in their implementation, but these are the general ones, which make sense in all Edelta programs.



```
 8 def checkEClassifierNames(EPackage it) {
 9   for (e : EClassifiers) {
10     if (Character.isLowerCase(e.name.charAt(0))) {
11       showError(e,
12         "Use a name that starts with a capital: "
13         + e.name)
14     }
15   }
16 }
17
18 modifyEcore someChanges epackage myecore {
19   addNewEDataType("MyDataType", String.name)
20   addNewEClass("myClass")
21     Use a name that starts with a capital: myClass
22                                Press 'F2' for focus
23
24   checkEClassifierNames
25 }
```

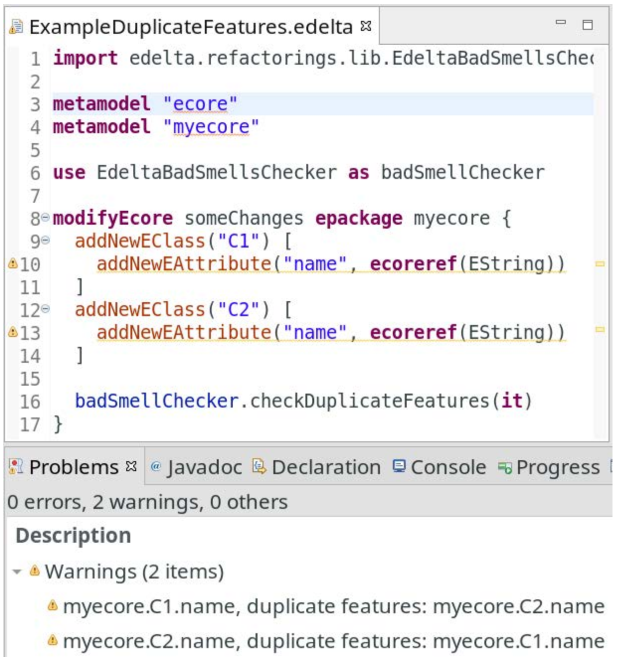
Figure 8: Error generated by the program.

For example, in [5] we presented our Edelta reusable library of metamodel bad smell finders and the corresponding refactorings. The bad smell finders have been modified so that they use showWarning when they spot a bad smell.[4] Thus, the users can call our bad smell finders from within their Edelta programs and can see live whether their metamodels under modifications have some bad smells. Moreover, and more importantly, they can make sure that, while they evolve their models in the current program, they do not introduce other bad smells. A very simple example is shown in Figure 9: we are using one of our bad smell library function (that checks whether two features in an EPackage are duplicates), imported with use ... as, and we call it at the end of our evolutions.

[3]The unsaved Eclipse editor's left ruler highlights the two modified lines 11 and 12.

[4]We believe it is better to only show a warning instead of an error in case of a bad smell, even because the Edelta compiler does not generate Java code if a program has an error, while it still generates code in case of warnings.

Two warnings are shown on the operations that created the two duplicate features.



```
import edelta.refactorings.lib.EdeltaBadSmellsChec

metamodel "ecore"
metamodel "myecore"

use EdeltaBadSmellsChecker as badSmellChecker

modifyEcore someChanges epackage myecore {
  addNewEClass("C1") [
    addNewEAttribute("name", ecoreref(EString))
  ]
  addNewEClass("C2") [
    addNewEAttribute("name", ecoreref(EString))
  ]

  badSmellChecker.checkDuplicateFeatures(it)
}
```

```
0 errors, 2 warnings, 0 others
Description
Warnings (2 items)
  myecore.C1.name, duplicate features: myecore.C2.name
  myecore.C2.name, duplicate features: myecore.C1.name
```

Figure 9: Warnings generated by the bad smell library.

5 RELATED WORK

The analysis conducted in [25] concerning the refactoring tools highlights that we still need to address the refactoring process in a more consistent, generic and scalable way. A first experiment has been conducted in [24] where the concept of model refactoring has been explored using a UML class diagram. It shows how graph transformation technology can be used to support model refactoring, where every refactoring is expressed as a graph production. This approach is based on a tool using graph transformations to apply the refactoring, whereas 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 metamodel to perform static checks.

In [31] the authors analyzed source code version control system logs of popular open source software systems to detect refactoring applications and see how software metrics are affected by these refactorings. We applied Edelta to a similar process in [5] where the refactoring is applied in order to understand if correlated metamodeling quality attributes are improved.

An approach to automatically detect source code refactorings using structural information has been presented in [22]. 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 identifies a sequence of detected refactorings from the input using 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. The similarity with our work relies in having a set of refactorings organized in libraries that can be used in the entire process.

A state-based differencing approach has also been used in the automatic detection and application of changes in a metamodel quality improvement process in [7].

A very similar intent is also explored in [1], where an Eclipse project called EMFRefactor provides a way to specify and apply refactorings on models. This tool uses Henshin's model transformation engine for executing the refactoring.

A metamodel called *Change Metamodel* for specifying atomic operations is proposed in [6]. A metamodel change is seen as an atomic transformation of one version of a metamodel to another. The metamodel used in this approach shares some similarities with the Edelta metamodel. 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 investigated in [19].

Fowler [13] presented a famous catalog of refactorings in object oriented programming that inspired our library of refactoring defined by using the proposed Edelta language. For the running examples in [4, 5] a library of reusable metamodel refactorings has been implemented following the formal definitions at http://www.metamodelrefactoring.org/.

A DSL called *Wodel* is presented in [16], allowing the creation of model mutations by means of a metamodel independent specifications. The primitives for model mutations (e.g., creation, deletion, reference reversal), item selection strategies, and composition of mutations are similar to the basic Edelta operations. Also in Wodel the specifications are compiled into Java but the differences with Edelta are in the abstraction layers in which the refactoring / mutation is applied.

The authors in [28] present a catalog of nine co-evolution operation specifications for automating the migration of ArchiMate models when the ArchiMate language is evolved. The organization of evolution scenarios in a catalog is a similar characteristic in our language, that can be used as matching mechanism for triggering co-evolution actions.

A set of refactorings preserving the behavior of UML models is presented in [32], where some examples are also illustrated. These refactorings are in part similar to what we applied to metamodels and in particular with the library offered by Edelta. A different approach in [34] presents a mechanism for detecting refactorings by analyzing the system evolution at the design level. In the same direction also the work in [2] detects high-level model changes in terms of refactoring and generates as output a list of detected changes representing the sequence of found refactorings. The authors in [27] conducted an investigation using open source Java applications to determine how precisely it is possible to refactor program source code to a desired design. Langer et al. [21] propose an approach searching for occurrences of composite operations within a set of detected atomic ones in a post-processing manner. In [20], another detection mechanism is proposed for the detection of complex changes applied to metamodel evolutions. All these approaches work in the opposite direction of Edelta, since Edelta applies evolution programmatically, reflecting modeler's intention, whereas these approaches work for detecting already applied evolutions.

6 CONCLUSIONS

In this paper we presented the new version of Edelta. Such a new version includes several improvements and new features in the compiler and in the IDE, thus providing a development environment for live and safe metamodel evolutions. These features contribute to make the tool more robust in producing metamodels without any form of inconsistencies, that can be for example introduced by using the standard EMF API directly in a Java program. Moreover the IDE has been improved in supporting the modeler with many helpful features like content assist, navigation and quickfixes. Other extensions of the new version of Edelta have not been presented in detail due to lack of space and we are already working on an extension showing the applicability of the approach on multiple (possibly mutual dependent) metamodels in parallel. We also already started to design an Edelta mechanism in order to support also refactorings of metamodel instances. This way, Edelta will allow the modeler to perform safe metamodel and model evolution, possibly in combination with the bad smells library. Future work related to benchmarking Edelta and the bad smell library are under evaluation.

REFERENCES

[1] Thorsten Arendt, Florian Mantz, and Gabriele Taentzer. 2010. EMF Refactor: Specification and Application of Model Refactorings within the Eclipse Modeling Framework. In *BElgian-NEtherlands software eVOLution seminar (BENEVOL)*.

[2] Ameni ben Fadhel, Marouane Kessentini, Philip Langer, and Manuel Wimmer. 2012. Search-based detection of high-level model changes. In *ICSM*. IEEE Computer Society, 212–221.

[3] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend* (2nd ed.). Packt Publishing.

[4] Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2017. Edelta: An Approach for Defining and Applying Reusable Metamodel Refactorings. In *Procs of MODELS 2017 Satellite Event*. 71–80.

[5] Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2019. Quality-Driven detection and resolution of metamodel smells. *IEEE Access* 7 (2019), 16364–16376. Publisher: IEEE.

[6] Erik Burger and Boris Gruschko. 2010. A Change Metamodel for the Evolution of MOF-Based Metamodels. *Modellierung* 161 (2010), 285–300.

[7] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2014. Mining Metrics for Understanding Metamodel Characteristics. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering* (Hyderabad, India) *(MiSE 2014)*. ACM, New York, NY, USA, 55–60. https://doi.org/10.1145/2593770.2593774

[8] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2013. Managing the coupled evolution of metamodels and textual concrete syntax specifications. In *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 114–121.

[9] Davide Di Ruscio, Ludovico Iovino, Alfonso Pierantonio, and Lorenzo Bettini. 2020. Detecting metamodel evolutions in repositories of MDE projects. In *Modelling Foundations and Applications*. Springer International Publishing. to appear.

[10] Davide Di Ruscio, Ralf Lämmel, and Alfonso Pierantonio. 2010. Automated co-evolution of GMF editor models. In *International Conference on Software Language Engineering*. Springer, 143–162.

[11] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Wilhelm Hasselbring, and Robert von Massow. 2012. Xbase: Implementing Domain-Specific Languages for Java. In *GPCE*. ACM, 112–121.

[12] Martin Fowler. 2005. FluentInterface. https://www.martinfowler.com/bliki/FluentInterface.html.

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

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

[15] Jokin García, Oscar Diaz, and Maider Azanza. 2012. Model transformation co-evolution: A semi-automatic approach. In *International Conference on Software Language Engineering*. Springer, 144–163.

[16] Pablo Gómez-abajo, Esther Guerra, Juan De Lara, Pablo Gomeza, and Esther Guerra. 2016. Wodel : A Domain-Specific Language for Model Mutation. (2016), 1–6.

[17] Markus Herrmannsdoerfer. 2010. COPE - A Workbench for the Coupled Evolution of Metamodels and Models. In *SLE (LNCS, Vol. 6563)*. Springer, 286–295.

[18] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Jürgens. 2009. COPE - Automating Coupled Evolution of Metamodels and Models. In *ECOOP (LNCS, Vol. 5653)*. Springer, 52–76.

[19] Ludovico Iovino, Alfonso Pierantonio, and Ivano Malavolta. 2012. On the Impact Significance of Metamodel Evolution in MDE. *JoT* 11, 3 (Oct. 2012), 3:1–33.

[20] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. 2016. Detecting complex changes and refactorings during (Meta)model evolution. *Inf. Syst* 62 (2016), 220–241.

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

[22] Rim Mahouachi, Marouane Kessentini, and Mel Ó Cinnéide. 2013. *Search-Based Refactoring Detection Using Software Metrics Variation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 126–140. https://doi.org/10.1007/978-3-642-39742-4_11

[23] MDE Research Group. [n.d.]. The Metamodel Refactorings Catalog. http://www.metamodelrefactoring.org. University of L'Aquila.

[24] Tom Mens. 2006. *On the Use of Graph Transformations for Model Refactoring*. Springer Berlin Heidelberg, Berlin, Heidelberg, 219–257. https://doi.org/10.1007/11877028_7

[25] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. 2003. Refactoring: Current Research and Future Trends. *Electronic Notes in Theoretical Computer Science* 82, 3 (2003), 483 – 499. https://doi.org/10.1016/S1571-0661(05)82624-6

[26] Bart Meyers and Hans Vangheluwe. 2011. A framework for evolution of modelling languages. *Science of Computer Programming* 76, 12 (2011), 1223–1246.

[27] Iman Hemati Moghadam and Mel Ó Cinnéide. 2012. Automated Refactoring Using Design Differencing. In *CSMR*. IEEE Computer Society, 43–52.

[28] Nuno Silva, Pedro Sousa, and Miguel Mira da Silva. 2019. 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*. IGI Global, 1–19.

[29] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.

[30] Misha Strittmatter, Georg Hinkel, Michael Langhammer, Reiner Jung, and Robert Heinrich. 2016. Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel. (2016).

[31] Konstantinos Stroggylos and Diomidis Spinellis. 2007. Refactoring–Does It Improve Software Quality?. In *Proceedings of the 5th International Workshop on Software Quality (WoSQ '07)*. IEEE Computer Society, Washington, DC, USA, 10–. https://doi.org/10.1109/WOSQ.2007.11

[32] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. 2001. Refactoring UML Models. In *The Unified Modeling Language. Modeling Languages, Concepts, and Tools (LNCS, Vol. 2185)*. Springer, 134–148.

[33] Sander D Vermolen, Guido Wachsmuth, and Eelco Visser. 2011. Reconstructing complex metamodel evolution. In *International Conference on Software Language Engineering*. Springer, 201–221.

[34] Zhenchang Xing and Eleni Stroulia. 2006. Refactoring Detection based on UMLDiff Change-Facts Queries. In *WCRE*. IEEE Computer Society, 263–274.