IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received November 19, 2018, accepted December 22, 2018, date of publication February 1, 2019, date of current version February 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2891357

Quality-Driven Detection and Resolution of Metamodel Smells

LORENZO BETTINI[1], DAVIDE DI RUSCIO[2], LUDOVICO IOVINO[3], AND ALFONSO PIERANTONIO[2]

[1]DiSIA, University of Florence, 50121 Florence, Italy
[2]Information Engineering, Computer Science and Mathematics Department, University of L'Aquila, 67100 L'Aquila, Italy
[3]Gran Sasso Science Institute, 67100 L'Aquila, Italy

Corresponding author: Ludovico Iovino (ludovico.iovino@gssi.it)

The research described has been carried out as part of the CROSSMINER Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant 732223.

ABSTRACT In model-driven engineering, analogously to any software development practice, metamodel design must be accurate and performed by considering relevant quality factors, including maintainability, reusability, and understandability. The quality of metamodels might be compromised by the introduction of smells that can be the result of inappropriate design decisions. Detecting and resolving metamodel smells are a complex task. The existing approaches deal with this problem by supporting the identification and resolution of smells without providing the means to explicitly trace them with the quality attributes that can be potentially affected. In this paper, we present an approach to defining extensible catalogues of metamodel smells. Each smell can be linked to the corresponding quality attributes. Such links are exploited to automatically select only those smells that have to be necessarily resolved for enhancing the quality factors that are of interest for the modeler. The implementation of the approach is based on the Edelta language, and it has been validated on a corpus of metamodels retrieved from a publicly available repository.

INDEX TERMS Domain-specific languages, model-driven engineering, software quality engineering.

I. INTRODUCTION

Bad smells are symptoms that something may be wrong in the system design or code [1]. A prominent example of bad smell is code duplication, a common problem that severely complicates the maintenance and evolution of large software systems. There are many bad smells defined in the literature and detecting them is far from trivial. Therefore, several tools have been proposed to automate bad smell detection aiming to improve software maintainability [2].

Model-driven engineering community has made considerable progress in the last decade as regards developing software systems with enhanced productivity, quality, and platform independence. Metamodels are central assets that permit designers to analyze and formalize application domains and to achieve (by means of related transformation techniques) superior automation, whether it be refactoring, simulation, or code generation. Development methods were devised to take advantage of these opportunities, and the accompanying methodologies have matured to the point where they are generally useful.

Like any long-living software artifact, metamodels are prone to changes. Evolutionary pressure to accommodate new requirements in the modeling language and insights emerging from the domain is more the rule than the exception. Unfortunately, subsequent evolutionary iterations may deteriorate the metamodel intrinsic consistency and uniformity, potentially putting the overall metamodel quality in jeopardy. Thus, lifting the concept of bad smell to metamodeling can improve the detection of bad modeling practices that might have a negative impact on the quality of the modeling artifact defined upon the metamodels. As a consequence, advanced techniques to detect metamodel smells and to properly resolve them are strongly needed [3], [4].

However, refactoring metamodels to remove bad smells does not come without a price: whenever a metamodel undergoes modification, the modeling ecosystem defined upon it might be not valid any longer [5]. While a number of approaches have been introduced for the co-evolution of different kinds of artifacts, including models, transformations and diagrammatic editors, the problem of keeping the corresponding ecosystem consistent is far from being definitively resolved [6]. Because of such intrinsic difficulties, quality assurance processes are applied in order to assure that only bad smells affecting specific quality attributes are resolved.

In this paper, we present a quality-based approach to bad smell detection and resolution that follows a least-change

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

principle in order to mitigate the impact of the changes on the modeling ecosystem. The domain-specific language named Edelta [7] is exploited to specify the detection and resolution of bad smells, which are in turn aligned as proposed in this paper with the potential impact on quality attributes. Thus, the contribution of this paper consists of techniques and tools allowing modelers to resolve only those bad smells affecting certain quality factors. The alignment among bad smell resolutions and quality attributes is given by means of weaving models. Moreover, the proposed approach is also able to produce an automated assessment of the quality attributes before and after the bad smell resolution. The obtained results in the experimental evaluation suggest that the proposed approach is promising and can support the selection of potential bad smells that should be removed for improving specific metamodel quality attributes.

The paper is organized as follows: Section II discusses representative bad metamodel smells and describes them by means of a running example. Section III explores the existing approaches introducing some qualities that drove our proposal. Section IV details the proposed process and supporting tools for dealing with bad smells and metamodel refactorings based on weaving models and Edelta operations. Section V presents an evaluation of the approach performed on a dataset consisting of 10 metamodels, nine of them extracted from the ATL Zoo [8]. Section VI concludes with the summary of the paper and the future work.

II. METAMODEL QUALITY ASSURANCE

In software development, code smells are defined as structural characteristics that may indicate problems in the code, making the system hard to evolve and maintain, and that can be resolved by means of code refactoring [9]. A refactoring is defined as a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [10].

The concept of *smell* can be lifted to the context of metamodeling since bad metamodel design decisions might have negative impacts on the quality of the modeling artifacts being developed [11]. For this reason, advanced techniques to detect metamodel smells and to properly resolve them are strongly needed [3], [4]. Smells can be categorized as automatically detectable ones and as those that can only be reliably detected manually. By smells that cannot be automatically detected we mean that it is not possible to conceive automated procedures that are able to identify metamodel parts, which might be source of quality issues [12]. According to [13], model quality assurance processes are typically based on the three main activities shown in Fig. 1. In particular, the models at hand are *analyzed* by means of specific metrics that once evaluated might help modelers to *identify* bad model smells. Appropriate refactoring steps are then performed to *resolve* the identified smells and consequently to enhance the quality of the analyzed artifacts. The process shown in Fig. 1 can be instantiated to manage the quality of any modeling artifacts, including metamodels as done in [13].

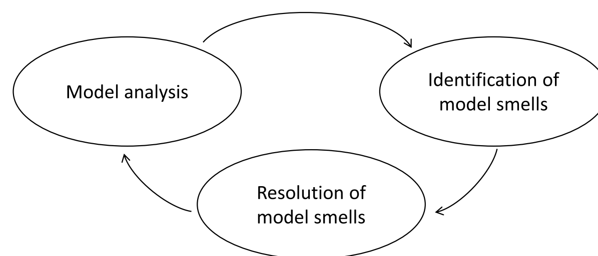

FIGURE 1. High-level view of model quality assurance processes.

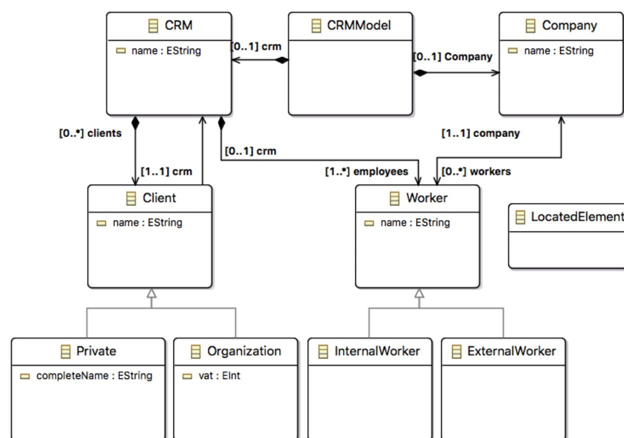

FIGURE 2. An explanatory metamodel with smells.

By focusing on the management of metamodel quality, we discuss the sample metamodel shown in Fig. 2 and some representative bad smells contained therein in the remaining of this section. The metamodel has been designed to support the specification of CRM (Customer Relationship Management) applications. A `CRMModel` is composed of a `CRM` and a `Company` on which the CRM operates on. CRM systems can contain `Clients` and `Workers`, whereas `Company` refers to its own employees, which can be `InternalWorker` or `ExternalWorker` instances. `Client` elements can be distinguished between `Private` and `Organization` (in case of companies).

According to the smells presented in [12], the simple CRM metamodel is the result of poor design decisions due to the following smells that have been introduced during the development of the metamodel in Fig. 2: *Duplicated features in metaclasses*, *Dead metaclass*, *Redundant container relation*, *Classification by enumeration or by hierarchy*, and *Concrete abstract metaclass*. Such metamodel smells have an impact on the overall quality of the metamodel. In particular, each smell affects specific quality attributes [12], [14] as shown in Table 1.[1] The bad smells that are in the simple CRM metamodel and the corresponding affected quality attributes are detailed in the following.

[1]The goal of Table 1 is to summarize some of the impacts retrieved from [12] that are used in this paper to describe the proposed approach. An exhaustive discussion on metamodel smells and corresponding quality attributes is beyond the scope of this work.

TABLE 1. Impact of metamodel smells on quality attributes.

Bad smell \ Quality attribute	Maintainability	Complexity	Understandability	Reusability
(BS1) Duplicated features in metaclasses	✓			✓
(BS2) Dead metaclass			✓	
(BS3) Redundant container relation	✓	✓		
(BS4) Classification by enumeration or by hierarchy		✓		
(BS5) Concrete abstract metaclass			✓	

Duplicated features in metaclasses (BS1): when a feature (attribute or reference) is present in different metaclasses (with the same type and properties), duplication of information might be induced. The resolution for this smell can be managed by introducing a hierarchy, and moving the shared feature up to the newly created super-metaclass. According to [12] redundant feature declarations can affect the *maintainability* and the *reusability* of the considered metamodel, as they must be consistently maintained in all the metaclasses. An example of duplicated feature, and in particular of a duplicated attribute, is shown in Fig. 2, where the `name` attribute of type `String` is present in four different metaclasses, i.e., `Company`, `CRM`, `Client`, and `Worker`. Note that, on the contrary, `crm` in `CRM`, `Client`, and `Worker` should not be considered as duplicated feature, since, apart from the name, the other properties of the `crm` feature are different.

Dead metaclass (BS2): depending on the goal of the metamodel under analysis, it can happen to have metaclasses completely disconnected from the other elements of the metamodel. In object-oriented design, similar situations are referred as *dead code* or *oxbow code* [15]. It is possible to statically detect the existence of metaclasses that are not connected with any other elements of the considered metamodel. However, a manual assessment is then required to decide whether the identified class should be considered as dead. For instance, it is viable (though not obligatory) to consider root container metaclasses as uncontainable. One of the possible refactoring operations that can be applied in these cases is the removal of the dead metaclass (even though such an action should be confirmed by the modeler). According to [12], this smell can have a negative impact on at least the *understandability* of the affected metamodels. An example of dead class is `LocatedElement` in the metamodel shown in Fig. 2. This metaclass is usually used to instantiate shapes or elements in graphical editors or to maintain the current position in textual editors. Thus, depending on the final use of the metamodel, modelers can decide what to do with the `LocatedElement` metaclass.

Redundant container relation (BS3): containment relations represent compositions of target elements from source ones. When navigating model elements (e.g., to analyze the considered model or to evaluate a query over it), it can be necessary to traverse containment relations from the contained elements to the containing ones. In EMF, the generic and implicit `eContainer` reference is available and it is also possible to define an explicit container reference using the concept of `eOpposite` references. If `eOpposite` is not set for a given containment relation, a metamodel smell can occur since two unidirectional references are defined instead, without providing the bidirectional navigation of the wanted containment relation. The presence of such a smell has several negative implications. In particular, it introduces redundancy, since the implicit `eContainer` reference is always present, and the addition of an explicit container reference represents a conceptual duplication. Thus, such a smell can increase at least the *complexity* and the *maintainability* of the considered metamodel. Figure 2 contains an example of this smell where the containment reference `clients` of the `CRM` metaclass is not set as opposite of the reference `crm` of `Client`. This smell does not make the navigation of such elements easy.

Classification by enumeration or by hierarchy (BS4): as discussed in [12] model elements can be classified by means of enumeration or by hierarchies. For instance, according to Fig. 2, workers can be classified as internal and external ones by means of two specializations of the metaclass `Worker`. Alternatively, an enumeration with two different literals might be used in the `Worker` metaclass. Depending on the particular case at hand, the *by enumeration* classification can be less appropriate than the *by hierarchy* one, and vice versa. For instance, modelers can consider the situation like the one in Fig. 2 a bad smell, since the `InternalWorker` and `ExternalWorker` metaclasses do not add any additional features to that of the superclass, and consequently a classification by enumeration would have been preferred. In such cases, the smell can have a negative impact on the *complexity* of the considered metamodels that, e.g., might contain more metaclasses than those actually needed.

Concrete abstract metaclass (BS5): depending on the particular situation being modeled, it can happen to have the superclass of a given class hierarchy being specified as concrete instead of abstract. If such a smell occurs, the corresponding resolution consists of changing the concrete metaclass into an abstract one. A metaclass that should be abstract and that is specified concrete can have negative impacts on the *understandability* of the considered metamodel. In particular, due to the fact that a metaclass of the considered metamodel

can be directly instantiated, might give place to erroneous situations. An example of such a situation is reported in Fig. 2 where the metaclass `Client`, being concrete, can be instantiated. However, in that specific case this should not be allowed since only the creation of `Private` and `Organization` instances should be enabled. The dual smell of this one is the *abstract concrete metaclass*.

Currently available model assurance processes (like the one presented in [13]) provide modelers with the means to manually select the smells to be removed from the analyzed metamodel and support the application of corresponding refactorings. However, managing metamodel changes is a challenging task especially because of the ripple effects that metamodel changes can have on the other depending artifacts like models and transformations. Consequently, it is of crucial importance to have the possibility of specifying the quality attributes that modelers would like to improve and consequently be supported in the selection and application of *only* those refactorings that are *actually needed* to enhance the selected quality attributes. Section IV will present the approach we have conceived to support such a quality-driven refactoring of metamodels.

III. EXISTING APPROACHES FOR MANAGING METAMODEL QUALITY AND REFACTORING

Bertoa and Vallecillo [11] identified a set of quality attributes for metamodels, and compose them in a model to represent their characteristics. This approach provides all the information required to evaluate metamodels according to different criteria. A language called *mmSpec* has been presented in [16], and it allows the specification of properties to be checked on meta-models. In this work they also present *metaBest*, a tool to visualize and report the problematic elements.

The work that is closest to ours is the one presented in [13] and [17]. It consists of the EMF Refactor tool that permits users to perform quantitative analysis of models, to implement resolutions of model smells, and apply them. EMF Refactor provides modelers with an extensible infrastructure that exploits the Eclipse extension mechanisms for adding new metrics, new smells detectors, and new model refactorings.

Strittmatter *et al.* [12] present a list of metamodel smells found in the Palladio Component Model [18]. They identified ten different types of smells, some of them inspired by the object-oriented programming paradigm. For each smell authors discuss the observed negative effects and further consequences that are expected. We relied on the discussions presented in [12] to link bad smells and quality attributes as detailed in the next section.

Another related work is presented in [19]. It consists of an approach for modeling refactorings for different modeling and meta-modeling languages. It is an extensible refactoring framework based on EMF. According to the work in [19], structural requirements for refactorings can be formalized in terms of role models. Then, by means of a mapping specification, such role models can be related to specific modeling languages. This mapping defines which elements of a language play which role in the context of a refactoring. Based on the mapping, generic transformation specifications are executed to restructure models. Thus, generic refactorings can be reused for different languages only by providing a mapping. This part of the approach is related to Edelta constructs that can be defined and extended by modelers using a concrete textual syntax that is more similar to programming languages in which usually refactorings are expressed. However, the explicit management of links among quality attributes and corresponding bad smells as proposed in this paper is not available in the approach presented in [19].

With the aim of addressing the need of having quality assessment stages as systematic part of the development process, several quality models have been proposed over the last decades (see [14], [20]–[24] just to mention a few). The quality attributes considered in this paper borrow their definitions from existing work in the area of software quality engineering [20], and consequently should not be considered as part of the novel contributions of this paper, which instead presents the means to link existing quality attributes definitions with bad smells of interest. A quality evaluation approach in metamodeling has been proposed in [14], where an infrastructure for defining customizable quality definition for different types of model-based artifacts has been conceived. The approach is based on a DSL for defining the customized quality model, which can be evaluated on the artifact subject of the evaluation. The result is the same quality model instantiated with the calculated quality attribute values. The concept of bad smells can be applied also for managing architectural aspects of software systems as proposed in [25]. The paper focuses on bad design smells that can have non-obvious and significant detrimental effects on the analyzed systems. Some of the identified bad smell presented in [25] can resemble modeling bad smells since software architectures can be specified by means of domain specific modeling languages.

According to the approaches previously summarized and to the requirements defined in [13], any approach able to identify and resolve bad metamodel smells, should implement the following features:

- *Quality attribute specification and evaluation*: The modeler should be provided with the means to specify quality attributes to be used for assessing the quality of the metamodels at hand. Ideally, also the evaluation of the defined attributes should be tool supported.
- *Bad smell specification and detection*: Modelers should be provided with dedicated languages and tools for specifying recurrent metamodel bad smells. Such specifications should be automatically manageable in order to enable the detection of the defined bad smells on concrete metamodels.
- *Refactoring specification and application*: Dedicated support is needed to specify and execute metamodel refactorings in a reusable manner.

TABLE 2. Approaches for managing the metamodel quality and refactoring.

Approach	Quality Attribute		Bad Smell		Refactoring		Quality attributes and Bad Smells linking	Bad Smells and Refactorings linking	Extension mechanism	Quality-driven resolution of metamodel bad smells
	Specification	*Evaluation*	*Specification*	*Detection*	*Specification*	*Application*				
Bertoa et al. [11]	DSL-based	-	-	-	-	-	-	-	-	-
Reiman et al. [19]	-	-	-	-	DSL-based	√	DSL-based	-	DSL-based	-
Basciani et al. [20]	DSL-based	√	-	-	-	-	-	-	DSL-based	-
Garcia et al. [26]	Textual (description)	-	-	-	-	-	-	-	-	-
EMF Refactor [17] [12]	GPL-based (Eclipse extension mechanisms)	√	GPL-based (Eclipse extension mechanisms)	√	GPL-based (Eclipse extension mechanisms)	√	GUI	GUI	GPL-based (Eclipse extension mechanisms)	-
Presented Edelta-based Approach	DSL-based	√	DSL-based	√	DSL-based	√	Based on Weaving Models	Based on Weaving Models	DSL-based	√

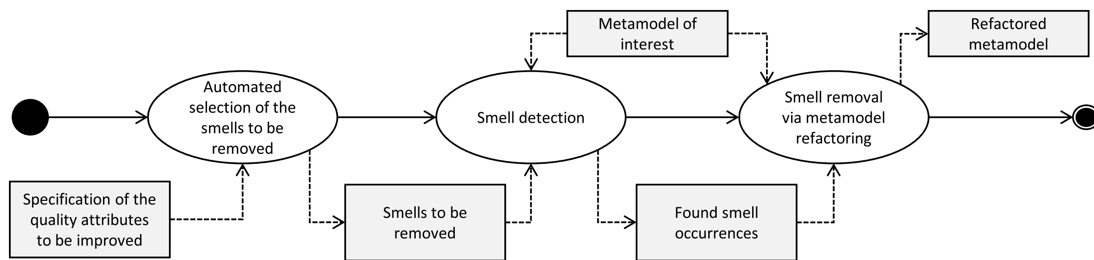

FIGURE 3. The proposed process for quality-driven detection and resolution of metamodel smells.

– *Quality attributes and bad smells linking*: It should be possible to link quality attributes with bad smells, which (if occurring) can reduce the quality of the metamodel at hand and that consequently, need to be resolved by applying appropriate metamodel refactoring operations.
– *Bad smells and refactorings linking*: It should be possible to link bad smells with metamodel refactorings, which might be suitable to resolve all the occurrences of the linked bad smells.
– *Extension mechanism*: The mechanisms that modelers can exploit to specify quality attributes, bad smells, and metamodel refactorings can be based on general purpose or domain specific languages.
– *Quality-driven resolution of metamodel bad smells*: modelers should be able to check the existence of bad smells that negatively affect the quality attributes that are of interest for the user.

Table 2 shows the approaches previously summarized with respect to the features presented above. Interestingly, the EMF Refactor approach [13], [17] permits users to perform quantitative analysis of models, to implement resolutions of model smells, and to apply them. However, modelers that want to add new metrics, new smells detectors, and new model refactorings have to implement them in Java by following the constraints of the extension mechanisms imposed by Eclipse platform. Moreover, EMF Refactor does not support quality-driven resolutions of metamodel bad smells as shown in the last column of Table 2. Such limitations are overcome by the approach proposed in this paper as described in next section.

IV. PROPOSED APPROACH

The last row of Table 2 is related to the approach presented in this section. In particular, it is based on Edelta [7], a DSL for easily defining metamodel evolutions and refactorings. The core language Edelta and the constructs of the domain specific language have been previously presented in [7]. In this work we rely on Edelta to define bad smells, which are resolved by means of the proposed quality driven process.

Figure 3 shows the high level view of the process underpinning the proposed approach, and consisting of both control and data flows. In particular, the process starts with the selection of the quality attributes that the modeler would like to improve in the metamodel at hand. Such quality attributes are used to select from a library of known metamodel smells only those that should be potentially removed. Thus, all the occurrences of such automatically selected smells are resolved by means of metamodel refactoring actions.

It is important to remark that detecting and resolving smells are very complex tasks since they might be related to design flows that cannot be always detected in automated manners [26]. Similarly, smell resolutions might need the involvement of humans that can be required to take decisions about how to solve some semantic flows that cannot be encoded in automated procedures. Consequently, the process shown in Fig. 3 has been conceived to support the

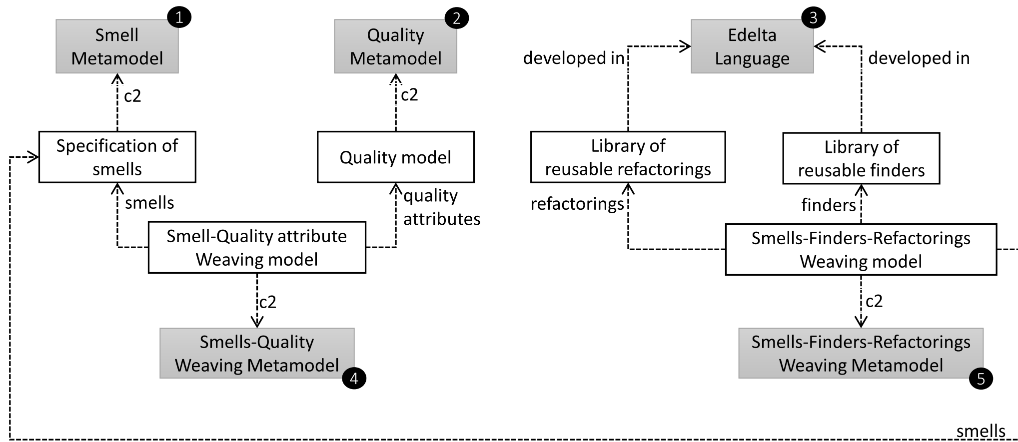

FIGURE 4. Technical artifacts supporting the proposed smell detection and resolution process.

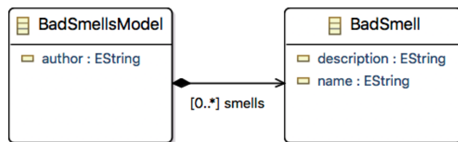

FIGURE 5. Metamodel for representing smells.

management of only those smells whose identification and resolution can be done algorithmically.

To support the process shown in Fig. 3 different components have been conceived as shown in Fig. 4 and described in the following.

Metamodel for specifying metamodel smells (❶): it permits the modeler to define libraries of smell names. The metamodel is simple and consists of two metaclasses as shown in Fig. 5.

Metamodel for specifying metamodel quality attributes (❷): it permits the modeler to specify the quality attributes of interest. A fragment of the metamodel is shown in Fig. 6. It is a simplification of the one proposed in [14] and it mainly consists of the `QualityAttribute` and `MetricProvider` constructs. A `QualityAttribute` represents a quality aspect of interest like maintainability, understandability, reusability, etc. A quality attribute can be aligned with other attributes, e.g., maintainability can be defined in terms of changeability and modularity. Thus, each quality attribute specifies how to combine the contained attributes in order to provide an overall quality value. The value definition of a quality attribute is computed by the application of a given `MetricProvider`, which refers to the software component able to calculate a specific metric.

Domain specific language for developing libraries of smell finders (❸): the Edelta language [7] is used for such a purpose. In particular, Edelta operations are developed to specify queries to be evaluated on the metamodel under analysis as detailed in Sec. IV-B.

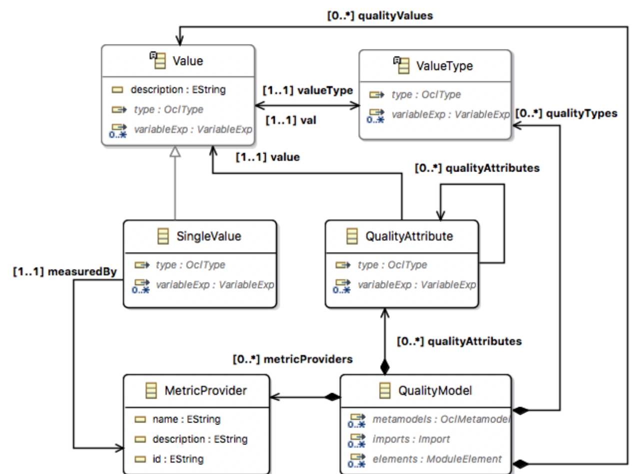

FIGURE 6. Fragment of the metamodel for specifying quality attributes [14].

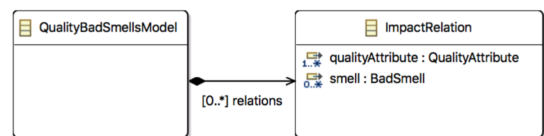

FIGURE 7. Metamodel for defining links among smells and quality attributes.

Domain specific language for developing libraries of reusable metamodel refactorings (❸): the Edelta language is used also to specify metamodel refactorings as detailed in Sec. IV-C.

Metamodel for linking the quality attributes that are impacted by each metamodel smell (❹): model weaving [27] is used for establishing links between model elements. This task is supported by the weaving metamodel shown in Fig. 7 which permits the modeler to define impact relation elements. In fact, weaving models are used when distinct operations have to be executed with respect to the semantics of the

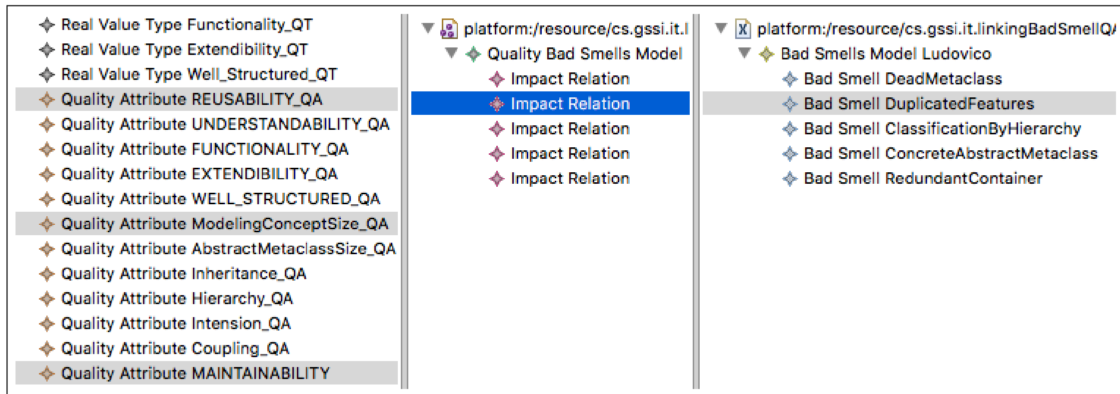

FIGURE 8. Sample model defining links among smells and quality attributes.

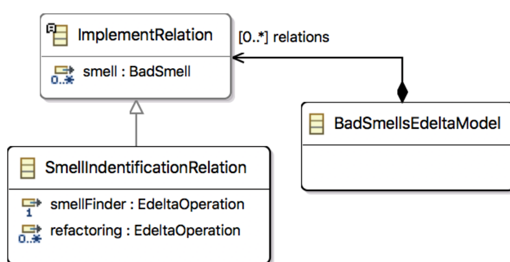

FIGURE 9. Metamodel for specifying links among smells, finders, and refactorings.

specified weaving links [27]. Each `ImpactRelation` element specifies how `QualityAttributes` can be impacted negatively by metamodel `Smells`. Figure 8 shows a model conforming to the metamodel in Fig.7. According to the shown example, three different quality attributes including *reusability* and *maintainability* are negatively impacted by the smell *DuplicatedFeatures*.

Metamodel for specifying links between smells, corresponding finders, and metamodel refactorings (❺): a fragment of this metamodel is shown in Fig. 9 and it permits the modeler to specify models like the one in Fig. 10. According to such a specification, the *DuplicatedFeatures* smell can be detected by means of the finder *findDuplicateFeatures* available in a reusable library implemented with the Edelta language. The resolution is also specified, it is the Edelta operation named *extractSuperclass* available from an available library of refactorings as specified in the property view at the bottom of Fig. 10.

Details about the proposed quality-driven selection of metamodel smells are given in Sec. IV-A. Edelta-based specifications of smell finders are given in Sec. IV-B, whereas corresponding metamodel refactorings are explained in Sec. IV-C. The tools discussed in this section are available for download (together with the data used for the evaluation presented in the next section) at https://github.com/gssi/Edelta_bad_smells/.

A. QUALITY-DRIVEN SELECTION OF METAMODEL SMELLS

As previously mentioned, the proposed approach enables the selection of quality attributes that modelers want to improve in the metamodel under analysis, and the automated identification of the corresponding smells that need to be detected and resolved. To this end, specific queries have been developed in order to trace elements specified in *Smell-Finders-Refactorings* models with those in *Smell-Quality attribute* models with respect to the quality attributes of interest for the user. For instance, according to the models discussed in the previous section, in order to increase the *maintainability* of the metamodel under analysis, all the occurrences of the *DuplicatedFeatures* smell should be removed (see Fig. 8). To this end, all the occurrences of such a smell can be detected by means of the finder *findDuplicateFeatures*; each of them can be resolved by means of the refactoring *extractSuperclass* (see Fig. 10). The navigation of the weaving models is performed by means of OCL queries like the one shown below:

```
1 self.relations->select(r|r.quality
              Attribute->
2   collect(n|n.varName)->includes
('MAINTAINABILITY\_QA'))->
3   collect(bs|bs.smell)
```

If the OCL query would be evaluated on the model shown in Fig. 8 (which has been defined to represent the impacts shown in Table 1) a collection would be given as result consisting of the *DuplicatedFeatures* and *RedundantContainer* smells.

B. FINDING METAMODEL SMELLS WITH EDELTA

In this section we make an overview of the Edelta language already proposed in [7]. Edelta is at the core of the overall process for detecting metamodel smells and for resolving them by means of reusable metamodel refactorings.

Edelta is integrated in Eclipse with a fully-fledged Eclipse editor with all the typical IDE mechanisms (from code completion to debugging). Our DSL is also completely

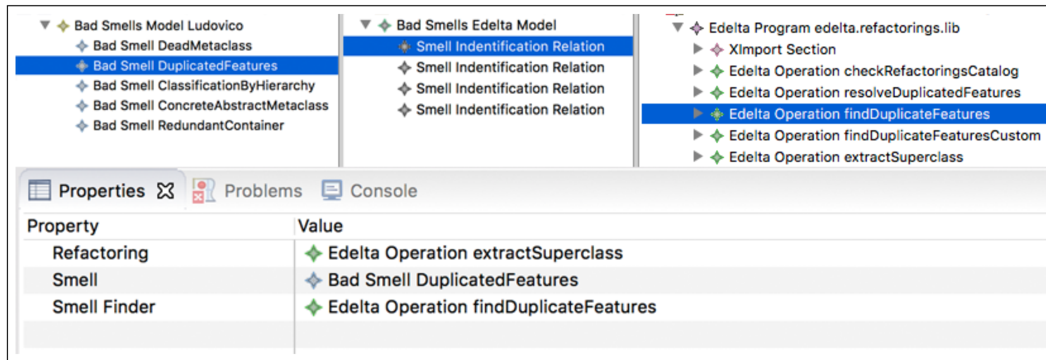

FIGURE 10. Linking the *DuplicatedFeatures* smell with the corresponding Edelta finder and refactoring operation.

interoperable with Java and its type system, meaning that any existing Java code can be seamlessly used from within an Edelta program. Thus, our approach allows the developer to use a compact syntax for specifying metamodel analysis and evolution, without forcing the developer to use Edelta for everything: Java code can also used only if desired. This allows for easy extension mechanisms that do not rely on cumbersome plug-ins. Moreover, the Edelta compiler and Eclipse editor also interprets the current Edelta specification on-the-fly, giving the developer an immediate feedback on the resulting modified metamodel without rebooting the IDE (more details can be found in [7]). Finally, since an Edelta program is represented in memory as an EMF model, existing EMF tools can be used to manipulate an Edelta program. For example, we can use the standard EMF tree editor to link smells, finders and refactoring operations (see Figure 10).

Edelta is publicly available as an open source project.[2] In order to enable the adoption of the approach we provide both the link for installing the Eclipse plugin in existing distributions, and a complete Eclipse distribution with Edelta already installed and ready to use.

It is important to remark that definitions of smells can be very subjective and informal [26]. Consequently, we decided to provide modelers with a language enabling the specification of custom smell finders and metamodel refactorings, which can be properly organized in reusable libraries. In this paper, we focus on automatically detectable smells and in this section we show explanatory examples of definitions falling in this category. Concerning, metamodel changes the language allows developers to specify both atomic and complex ones. Moreover, the implementation of Edelta is based on Xtext [28] and the language is endowed with an Eclipse-based development environment providing also early evaluation of the refactoring being applied and the ability to debug Edelta programs. Edelta programs can contain Edelta instructions, called *Operations*, by reusing other operations already defined in available *libraries*.

Edelta uses Xbase [29] to provide a rich Java-like syntax for its expressions. In order to make code snippets presented in the paper comprehensible, we will briefly sketch the main features of Xbase; for a deeper description of the Edelta Xbase expression language, we refer the reader to [7]. Xbase should be easily understood by Java programmers. Since Xbase is completely interoperable with Java, this implies that Edelta reuses the Java type system, including generics, and existing Java libraries can be used in Edelta. This also means that existing refactoring implementations for Ecore models written in Java, can be seamlessly reused in an Edelta program.

Xbase removes much of the "syntactic noise" verbosity that is typical of Java. Terminating semicolons are optional in Xbase. Xbase comes with a powerful type inference mechanism that allows the programmer to avoid specifying types in declarations when they can be inferred from the context (e.g., in method signatures and variable declarations). This makes Edelta expressions compact and readable like in scripting and untyped languages, while still enjoying the static type safety, not to mention rich content assist in Eclipse, based on the inferred types.

Xbase *extension methods* are a syntactic sugar mechanism to simulate adding new methods to existing types without modifying them. Using extension methods results in a more readable code, since method calls are chained, e.g., `o.foo().bar()` rather than nested, e.g., `bar(foo(o))`. Syntactic sugar for getters and setters is also provided: one can simply write `o.name` and `o.name = "..."`, instead of `o.getName()` and `o.setName("...")`, respectively.

Xbase *lambda expressions* have the shape:

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

The types of parameters can be omitted if they can be inferred from the context. Xbase has another additional special variable, `it`. Similar to `this`, `it` can be omitted as object receiver of method call and member access expressions. The programmer is allowed to declare any variable or method parameter with the name `it`, thus a custom implicit object receiver can be declared in any scope of the program. When a lambda has a single parameter, the parameter can be omitted and it will be automatically `it`.

[2]https://github.com/LorenzoBettini/edelta

```
1// inferred return type: Iterable<EStructuralFeature>
2def allEStructuralFeatures(EPackage epackage) {
3 epackage.allEClasses.
4  map[EStructuralFeatures].flatten
5}
6
7def allEClasses(EPackage epackage) {
8 // EClassifiers instead of getEClassifiers()
9 epackage.EClassifiers.filter(EClass)
10 // no need to append .class to refer to a Java type
11}
```

Listing 1. Edelta example of reusable functions.

Thanks to all these linguistic mechanisms, Edelta allows the programmer to easily write reusable functions in a compact and readable way, like the ones shown in Listing 1 (note the use of syntactic sugar for getters, `allEClasses` used as an extension method, and the type inference in function return types).

Listing 2 reports an extract of an Edelta library able to find and match the bad smells previously defined. It is important to understand that the library is independent from the specific case and it contains all the bad smell finder instructions covered by the various modelers.

In the specific case of lines 4-11, the `findDuplicateFeatures` operation is defined. Note that this operation calls another operation, which actually performs the search for duplicate features, `findDuplicateFeaturesCustom`. This operation takes as argument an additional lambda that is responsible of deciding whether two features should be considered equal in two different EClasses. `findDuplicateFeatures` calls this operation with a lambda that relies on our default implementation of equality detection for features, which scans all the properties of two given features. The modeler could have a different strategy for deciding whether two features have to be considered equal, that is why we also provide `findDuplicateFeaturesCustom`. In particular, we follow the same pattern also for other bad smell finders in our library.

This operation returns a map whose values are lists of duplicated features in any EClass of the specified EPackage. Instead of flattening the lists of duplicated features, we keep them separated so that, later, we can extract a superclass for each duplicated feature in a straightforward way, without having to visit again the model. We follow the same pattern in the other bad smell finders: the information returned by the finders contains all the elements to implement the corresponding refactoring.

C. REMOVING METAMODEL SMELLS WITH EDELTA

When a bad smell is found by Edelta operations, the *refactoring* specified in the considered weaving model like the one shown in Fig. 10 can be applied in order to resolve it. In this specific case the bad smell *duplicated features* can be automatically resolved by applying the refactoring `extractSuperClass` as defined by the modeler in Listing 3. The listed Edelta operation is part of the refactoring

```
1...
2package edelta.badsmells.finder.lib
3
4def findDuplicateFeatures(EPackage epackage) {
5 return findDuplicateFeaturesCustom(
6  epackage,
7  [existing, current|
8   new EstructuralFeatureEqualityHelper()
9    .equals(existing, current)
10  ]
11 )
12}
13def findDuplicateFeaturesCustom(EPackage epackage,
14   BiPredicate<EStructuralFeature, EStructuralFeature>
        matcher) {
15 val allFeatures = epackage.allEStructuralFeatures
16 val map =
17  <EStructuralFeature, List<EStructuralFeature>>
        newLinkedHashMap
18 for (f : allFeatures) {
19  val found = map.entrySet.findFirst[matcher.test(it.key, f
        )]
20  if (found !== null) {
21   found.value += f
22  } else {
23   map.put(f, newArrayList(f))
24  }
25 }
26 // only entries with a mapped list of size > 1 are
        duplicates
27 return map.filter[p1, p2| p2.size > 1]
28}
29...
```

Listing 2. Edelta snippet of the bad smell identification library.

```
1...
2package edelta.refactorings.lib
3...
4def extractSuperclass(List<? extends EStructuralFeature>
      duplicates){
5 val feature = duplicates.head;
6 val containingEPackage = feature.EContainingClass.EPackage
7 val superClassName =
8  ensureEClassNameIsUnique(containingEPackage,
9   feature.name.toFirstUpper + "Element"
10  )
11 val superclass = newEClass(superClassName) [
12  abstract = true
13  EStructuralFeatures += EcoreUtil.copy(feature)
14 ]
15 containingEPackage.EClassifiers += superclass
16
17 for (duplicate : duplicates) {
18  val eContainingClass = duplicate.EContainingClass
19  // set supertype of the EClass of the attribute
20  eContainingClass.ESuperTypes += superclass;
21  // and then remove the attribute from the original EClass
22  eContainingClass.EStructuralFeatures -= duplicate
23 }
24}
25...
```

Listing 3. Edelta snippet of the refactorings library.

library containing all the possible refactorings we defined to cover the catalog published at [3].

In this refactoring the argument is the list of found duplicated features, that will be removed from the metaclasses and moved up (in a single feature) to the hierarchy. The extracted metaclass created will have the name of the feature concatenated with the postfix `Element` (plus a possible suffix to guarantee that such a name is not already used within the same EPackage). Then, for all the metaclasses containing the duplicated features, the supertype will be set to the newly created metaclass, and the original feature is removed from the initial metaclass.

By applying the proposed approach on the metamodel shown in Fig. 2, with the aim of improving the

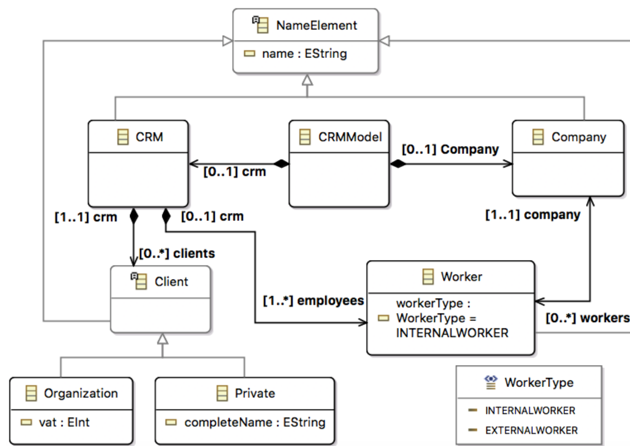

FIGURE 11. The original metamodel after the refactorings.

maintainability, complexity, understandability, and reuse quality attributes, the metamodel shown in Fig. 11 can be obtained. In particular,

i) the *duplicated features* smell, due to the `name` attribute occurring in four metaclasses, has been resolved by means of the extract metaclass refactoring, which added the new `NameElement` metaclass with the `name` attribute pulled in;

ii) the *redundant containment relation* between the `CRM` and `Client` metaclasses has been removed by properly setting the `eOpposite` property of the kept relation between such metaclasses;

iii) the *dead metaclass* smell that was found on the `LocatedElement` metaclass has been resolved with the removal of the dead metaclass, after the user interaction confirmation (as detailed below);

iv) the *bad classification* smell, previously identified in the sub-classes of `Worker`, has been resolved by introducing the enumeration `WorkerType`, and the introduction of the attribute `workerType` in the relative metaclass;

v) finally, the erroneously concrete metaclass identified in `Client` has been fixed by setting it as abstract.

The console shown in Fig. 12 reports the smells identified and resolved automatically and also the detection of those that required user intervention. For instance, concerning the found *dead metaclass*, the user was asked to confirm the refactoring chosen by the Edelta program i.e., the removal of the `LocatedElement` metaclass.

V. EVALUATION

In this section we discuss the evaluation we have performed with the aim of answering the following research question: *is the proposed approach able to refactor the metamodel under analysis by affecting the quality attributes as selected by the modeler?*

```
Console ☒
Main [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Hom
0    [main] INFO    edelta.lib.EdeltaEPackageManager  - Loading model/
My.ecore (URI: file:/Users/administrator/git/edelta_improving_mm/
testIteration/model/My.ecore)
221  [main] INFO  adsmells.finder.lib.BadSmellsFinder  - Duplicate
features: CRM:Client:name, CRM:Worker:name, CRM:CRM:name,
CRM:Company:name
237  [main] INFO  adsmells.finder.lib.BadSmellsFinder  - Dead
classifier: CRM:LocatedElement
Should I remove dead classifier CRM:LocatedElement? (y/n)
```

FIGURE 12. Edelta console asking confirmation to apply the refactoring to resolve the found *dead metaclass* smell.

A. EXPERIMENT SETUP

In order to validate the approach, Table 1 has been specified in terms of the required (weaving) models as presented in the previous section. A data set consisting of 10 metamodels (i.e., the `CRM` metamodel presented in the previous section and additional 9 metamodels retrieved from the ATL Zoo [8]) has been considered. The metamodels have been selected among the group of packages provided in the ATL Zoo where no errors have been observed in the execution without manual inspection. Then, similarly to mutation testing, for each metamodel *MM* in the dataset the following process has been applied:

1) for each quality attribute (*qa*), *MM* has been manually mutated by introducing the bad smells (bs) affecting *qa* according to Table 1. For instance, concerning the maintainability attribute, two different mutated metamodels have been obtained from *MM*: one has been obtained by introducing an instance of the *duplicated features* smell, and another one by adding an instance of the *redundant container* smell;
2) the proposed approach has been applied on each mutated metamodel obtained in the previous step. In particular, the tool has been asked to select the bad smells that were needed to be resolved for improving each quality attribute on each mutated metamodel. Then, each quality attribute has been evaluated on each mutated metamodel, before and after the refactorings selected by the tool according to the given weaving specifications. If the value of the considered quality attribute gets improved after the application of the selected refactoring, then it is confirmed that the approach is able to select and identify the right bad smells, and to finally resolve them.

B. DEFINITION OF THE CONSIDERED QUALITY ATTRIBUTES

For measuring the quality attributes during the experiment, the quality assessment tool presented in [14] has been used. It permits to define quality attributes, which can be hierarchically organized. Each quality attribute consists of an expression defining how the values of sub-attributes or metrics have to be combined.

The *maintainability* quality attribute considered in this paper has been defined according to the definition given

TABLE 3. Excerpt of the metrics considered in the evaluation.

Metric	Acronym
Number of MetaClass	NC
Number of TotalReference	NR
Number of Opposite Reference	NOPR
Number of TotalReference containment	NCR
Number of TotalAttribute	NA
Number of Unidirectional reference	NUR
Max generalization hierarchical level	DITmax
Max Reference Sibling (max fan Out)	FANOUTmax
Number of TotalFeatures	NTF
Sum of inherited structural features	INHF
Attribute inheritance factor	AIF
Number of predecessor in hierarchy	PRED

in [30] and that is based on some of the metrics shown in Table 3 as follows:

$$Maintainability = \left(\frac{NC + NA + NR + DIT_{Max} + Fanout_{Max}}{5}\right) \quad (1)$$

According to the considered definition of *maintainability* the lower values the better.

The definitions of the *Understandability* and *Complexity* quality attributes have been taken from [31]. In particular, understandability can be defined as follows

$$Understandability = \left(\frac{\sum_{k=1}^{NC} PRED + 1}{NC}\right) \quad (2)$$

where PRED regards the predecessors of each metaclass, since, in order to understand a metaclass, we have to understand all of the ancestor metaclasses that affect the class as well as the metaclass itself. According to such a definition the higher values for the understandability quality attribute the better.

Complexity can be defined in terms of the number of static relationships between the metaclasses of the considered metamodel (i.e., number of references). The complexity of the association and aggregation relationships is counted as the number of direct connections, whereas the generalization relationship is counted as the number of all the ancestor and descendant metaclasses. Thus, the complexity quality attribute can be defined as follows:

$$Complexity = (NR - NUR + NOPR + UND + (NR - NCR)) \quad (3)$$

where NUR is the number of unidirectional references calculated as the difference between bidirectional and total reference number, and UND is the understandability value calculated as defined in Def. 2. According to the given definition, the lower values for the complexity attribute the better.

The *reusability* of a given metamodel can be calculated in different ways. One among the proposed ones is to use the attribute inheritance factor *AIF* as proposed in [32] where it is stated that a higher value indicates a higher level of reuse. As presented in [33], AIF can be defined as follows:

$$AIF = \left(\frac{INHF}{NTF}\right) \quad (4)$$

where *INHF* is the sum of the inherited features in all metaclasses, and *NTF* is the total number of available features.

C. RESULTS

The results of the performed experiment are shown in Table 4. Each quality attribute consists of different columns. For instance, concerning the quality attribute *Complexity*, the columns $BS3_{pm}$, $BS3_{pr}$, $BS4_{pm}$, and $BS4_{pr}$ are shown. The subscripts *pm* and *pr* are used to distinguish the values of the quality attribute evaluated after the application of the mutations (to add the bad smells BS3 and BS4), and after the execution of the metamodel refactorings (to remove the bad smells BS3 and BS4 from the mutated models), respectively. For example, the refactoring operated by the proposed approach to reduce the complexity of the metamodel *Persons*, obtained after the mutation to add the smell BS4, has been effective. In fact, the value of $BS4_{pr}$ for the metamodel *Persons* is lower than $BS4_{pm}$ for the same metamodel.

The row *Expected delta* in Table 4 shows if the difference between one value under the considered column *pr* and that in the cell under the corresponding *pr* column is expected to be lower or greater than 0. The last row of Table 4 shows the number of cases when the expected delta occurred. The obtained results suggest that the proposed approach is promising and can be used to get support for selecting potential bad smells that should be removed for improving specific metamodel quality attributes.

D. THREATS TO VALIDITY

In this section, potential threats to validity associated with the experimental validation are discussed, by distinguishing construct, internal, and external validity.

Construct validity concerns any factor that can compromise the validity of the experiment and of the resulting observations. A potential threat to construct validity is related to the specification of the weaving models that might link erroneously quality attributes, smells, and corresponding metamodel refactorings. We did our best to specify the used weaving models in a consistent way with the considered literature. Moreover, we did several manual assessments to check if suggested bad smells were actually consistent with the quality attributes to be improved.

Internal validity concerns any confounding factor that could influence our results. We attempted to avoid any bias in the definition of the weaving models and in the selection of the quality attributes: *i)* by considering already existing quality attributes as they were presented in their original papers; *ii)* by completely delegating the evaluation of the considered quality attributes to external assessment tools (in particular the one presented in [14]). Indeed, the implemented tools could be defective. To contrast and mitigate

TABLE 4. Quality attribute values calculated post-mutations (*pm*) and post-refactorings (*pr*).

Metamodel	Maintainability				Complexity				Understandability				Reusability	
	$BS1_{pm}$	$BS1_{pr}$	$BS3_{pm}$	$BS3_{pr}$	$BS3_{pm}$	$BS3_{pr}$	$BS4_{pm}$	$BS4_{pr}$	$BS2_{pr}$	$BS2_{pr}$	$BS5_{pm}$	$BS5_{pr}$	$BS1_{pm}$	$BS1_{pr}$
CRM	22,17	19,33	22,17	20,17	11,40	10,40	11,40	11,25	1,40	1,44	1,40	1,47	0,71	1,27
Families	9,00	8,17	4,23	6,23	3,17	4,18	9,50	9,00	1,00	1,05	1,33	1,44	0,00	0,20
Persons	7,17	6,17	7,17	5,17	4,58	3,58	3,67	3,17	1,53	1,58	1,50	1,58	0,80	1,50
Grafcet	19,50	16,50	16,50	14,50	20,26	19,26	19,57	19,26	3,03	3,26	3,22	3,26	0,86	1,21
PNML	21,50	20,50	22,50	20,50	19,67	18,67	18,67	18,42	2,55	2,67	2,64	2,67	1,37	1,56
PathExp	13,17	12,17	10,17	8,17	11,58	10,58	10,81	10,73	1,47	1,58	1,50	1,58	1,00	1,30
Petrinet	16,17	13,33	14,17	12,17	18,76	17,76	17,79	17,76	1,67	1,76	1,76	1,81	0,24	0,57
TypeB	9,17	8,33	9,17	7,17	5,58	4,58	4,87	4,58	1,39	1,58	1,87	1,93	0,33	0,80
SimpleClass	16,17	14,33	18,17	16,17	12,40	11,40	11,40	11,25	1,33	1,40	1,40	1,47	0,18	0,56
SimpleRDBMS	16,00	14,17	16,00	14,00	9,00	8,00	9,25	9,00	1,00	1,07	1,25	1,33	0,00	0,50
Expected delta	< 0		< 0		< 0		< 0		> 0		> 0		> 0	
Confirmed expected delta	10		9		9		10		10		10		10	

this threat, we have run several manual assessments and counter-checks.

External validity refers to the extent to which the results of our study can be generalized. The metamodels we have investigated come from a publicly available repository already used in other experiments discussed in literature. However, we cannot claim that the results of our experimental evaluation are generalizable, even though the performed validation provides us with an acceptable confidence about the effectiveness of the approach on a well-known and recognized data set.

VI. CONCLUSIONS AND FUTURE WORK

We have presented an approach to support the selection of smells that should be potentially removed from the metamodel under developer in order to enhance the particular quality attribute that the modeler is interested in. The approach is essentially based on weaving specifications able to link smells, Edelta operations able to find them, and corresponding metamodel refactorings also specified in Edelta. We have presented an evaluation of the approach by considering a set of metamodels retrieved from the ATL Zoo. The obtained results suggest that the proposed approach is promising to deal with the problem of metamodel bad smell detection and resolution.

We plan to extend the approach in different directions. Currently, the proposed methodology works properly under the conditions that the considered bad smells do not depend on other ones, and consequently that the weaving models are specified so that the proposed solution for enhancing the quality attribute of interest is unique. However, different kinds of bad smells are not independent of each other, and the resolution of one kind of bad smells may influence the resolution of other bad smells [34]. Thus, we plan to extend the bad smell metamodel allowing modelers to specify also the relation of containment for bad smells, e.g., a bad smell containing a simpler bad smell, and the order of resolution and influence. The proposed approach allows modelers to define automatically detectable smells and we plan to define a custom library of finders based on bad smell definitions and examples available in the literature, also inspired by code smells. However, this does not exclude the possibility to define additional domain-specific smell finders, since the Edelta language supports custom definitions organized in different libraries as described in Section IV-B. Since Edelta acts on Ecore models through the standard EMF API, we plan to integrate Edelta with existing EMF frameworks, such as, e.g., Edapt [35] for the migration of existing models and Refacola [36] for the existing generated code.

REFERENCES

[1] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, 2014, Art. no. 33.

[2] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proc. ACM EASE*, New York, NY, USA, 2016, pp. 18-1–18-12.

[3] MDE Research Group, University of L'Aquila. *The Metamodel Refactorings Catalog*. Accessed: Jan. 15, 2018. [Online]. Available: http://www.metamodelrefactoring.org

[4] R. Hebig, D. E. Khelladi, and R. Bendraou, "Approaches to co-evolution of metamodels and models: A survey," *IEEE Trans. Softw. Eng.*, vol. 43, no. 5, pp. 396–414, May 2017.

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

[6] D. Di Ruscio, L. Iovino, and A. Pierantonio, "Coupled evolution in model-driven engineering," *IEEE Softw.*, vol. 29, no. 6, pp. 78–84, Nov./Dec. 2012.

[7] L. Bettini, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Edelta: An approach for defining and applying reusable metamodel refactorings," in *Proc. MODELS Satellite Event*, 2017, pp. 71–80.

[8] Eclipse. (2012). *ATL Transformations Zoo*. [Online]. Available: https://www.eclipse.org/atl/atlTransformations/

[9] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *J. Object Technol.*, vol. 11, no. 2, pp. 1–5, 2012.

[10] G. B. Regulwar and R. M. Tugnayat, "Detection of bad smell code for software refactoring," in *Innovations in Computer Science and Engineering*, H. S. Saini, R. Sayal, A. Govardhan, and R. Buyya, Eds. Singapore: Springer, 2019, pp. 143–152.

[11] M. F. Bertoa and A. Vallecillo, "Quality attributes for software metamodels," Univ. Málaga, Málaga, Spain, Tech. Rep., 2010.

[12] M. Strittmatter, G. Hinkel, M. Langhammer, R. Jung, and R. Heinrich, "Challenges in the evolution of metamodels: Smells and anti-patterns of a historically-grown metamodel," in *Proc. CEUR Workshop*, vol. 1706, 2016, pp. 30–39.

[13] T. Arendt and G. Taentzer, "A tool environment for quality assurance based on the eclipse modeling framework," *Automat. Softw. Eng.*, vol. 20, no. 2, pp. 141–184, Jun. 2013.

[14] F. Basciani, J. di Rocco, D. di Ruscio, L. Iovino, and A. Pierantonio, ''A customizable approach for the automated quality assessment of modelling artifacts,'' in *Proc. 10th Int. Conf. Qual. Inf. Commun. Technol. (QUATIC)*, Sep. 2016, pp. 88–93.

[15] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, ''Compiler techniques for code compaction,'' *ACM Trans. Programm. Lang. Syst.*, vol. 22, no. 2, pp. 378–415, Mar. 2000.

[16] Z. Ma, X. He, and C. Liu, ''Assessing the quality of metamodels,'' *Frontiers Comput. Sci.*, vol. 7, no. 4, pp. 558–570, Aug. 2013, doi: 10.1007/s11704-013-1151-5.

[17] T. Arendt and G. Taentzer, ''Integration of smells and refactorings within the Eclipse modeling framework,'' in *Proc. ACM 5th Workshop Refactoring Tools (WRT)*, New York, NY, USA, 2012, pp. 8–15.

[18] R. H. Reussner *et al.*, *Modeling and Simulating Software Architectures: The Palladio Approach*. Cambridge, MA, USA: MIT Press, 2016.

[19] J. Reimann, M. Seifert, and U. Aßmann, ''Role-based generic model refactoring,'' in *Model Driven Engineering Languages and Systems*, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds. Berlin, Germany: Springer, 2010, pp. 78–92.

[20] S. H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2002.

[21] P. Berander *et al.*, ''Software quality attributes and trade-offs,'' *Blekinge Inst. Technol.*, 2005.

[22] B. Boehm, J. Brown, J. Kaspar, and M. Lipow, *Characteristics of Software Quality*. Amsterdam, The Netherlands: North Holland, 1978.

[23] R. G. Dromey, ''A model for software product quality,'' *IEEE Trans. Softw. Eng.*, vol. 21, no. 2, pp. 146–162, Feb. 1995.

[24] R. B. Grady and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*. Upper Saddle River, NJ, USA: Prentice-Hall, 1987.

[25] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, ''Identifying architectural bad smells,'' in *Proc. 13th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2009, pp. 255–258.

[26] F. A. Fontana, P. Braione, and M. Zanoni, ''Automatic detection of bad smells in code: An experimental assessment,'' *J. Object Technol.*, vol. 11, no. 2, pp. 5-1–5-38, Aug. 2012.

[27] M. D. Del Fabro, J. Bézivin, and P. Valduriez, ''Weaving models with the eclipse AMW plugin,'' in *Proc. Eclipse Modeling Symp., Eclipse Summit Europe*, 2006, pp. 37–44.

[28] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*, 2nd ed. Birmingham, U.K.: Packt Publishing, 2016.

[29] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, W. Hasselbring, and R. von Massow, ''Xbase: Implementing domain-specific languages for Java,'' in *Proc. ACM GPCE*, 2012, pp. 112–121.

[30] M. Genero and M. Piattini, ''Empirical validation of measures for class diagram structural complexity through controlled experiments,'' in *Proc. QAOOSE@ECOOP*, 2001, pp. 87–95.

[31] F. T. Sheldon and H. Chung, ''Measuring the complexity of class diagrams in reverse engineering,'' *J. Softw. Maintenance Evolution, Res. Pract.*, vol. 18, no. 5, pp. 333–350, 2006.

[32] T. Arendt, F. Mantz, and G. Taentzer, ''UML model quality assurance techniques,'' Philipps-Univ. Marburg, Marburg, Germany, Tech. Rep., Oct. 2009.

[33] J. Al-Ja'Afer, K. Eddin, and M. Sabri, ''Metrics for object oriented design (MOOD) to assess java programs,'' Univ. Jordan, Amman, Jordan, Tech. Rep., 2007.

[34] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao, ''Facilitating software refactoring with appropriate resolution order of bad smells,'' in *Proc. ACM ESEC/FSE*, New York, NY, USA, 2009, pp. 265–268.

[35] Eclipse. (2018). *Edapt—Migrating EMF Models*. [Online]. Available: https://www.eclipse.org/edapt/

[36] J. von Pilgrim, B. Ulke, A. Thies, and F. Steimann, ''Model/code co-refactoring: An MDE approach,'' in *Proc. 28th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, New York, NY, USA, Nov. 2013, pp. 682–687.

LORENZO BETTINI was an Assistant Professor (Researcher) of computer science with the Dipartimento di Informatica, Università di Torino, Italy. He has been an Associate Professor of computer science with the DISIA Dipartimento di Statistica, Informatica, Applicazioni Giuseppe Parenti, Università di Firenze, Italy, since 2016. His research interests include design, theory, and implementation of programming languages (in particular object-oriented languages and network aware languages).

DAVIDE DI RUSCIO is currently an Assistant Professor with the University of L'Aquila. Over the last decade, he has worked on several European projects by contributing the application of model-driven engineering in different application domains, such as service-based software systems, autonomous systems, and open source software. His main research interests include software engineering and several aspects of model-driven engineering, including domain-specific languages, model transformations, and model evolution. He has published more than 100 papers in various journals, conferences, and workshops on these topics.

LUDOVICO IOVINO is currently an Assistant Professor with the Computer Science department, GSSI Gran Sasso Science Institute, L'Aquila. His research interests include model-driven engineering, model transformations, metamodel evolution, code generation, and software quality evaluation. He has been working on the model-based artifacts and issues related to the metamodel evolution problem. He is a part of different academic projects related to model repositories, model migration tools, and eclipse plugins. He was included in the program committees of numerous conferences and in the local organization of the STAF 2015 and the iCities 2018 conferences. He has organized the models and evolution workshop at the MODELS 2018.

ALFONSO PIERANTONIO is currently an Associate Professor with the University of L'Aquila, Italy. His research interests include model-driven engineering with a specific emphasis on co-evolution problems, bidirectionality, and megamodeling. He has chaired a number of international conferences and has organized over 20 workshops (including ICMT and STAF). He is on the Editorial Board of several scientific journals (including SoSyM and JOT).