REGULAR PAPER

MUPPIT: a method for using proper patterns in model transformations

Mahsa Panahandeh[1] · Mohammad Hamdaqa[2,3] · Bahman Zamani[4] · Abdelwahab Hamou-Lhadj[5]

Abstract
Model transformation plays an important role in developing software systems using the model-driven engineering paradigm. Examples of applications of model transformation include forward engineering, reverse engineering of code into models, and refactoring. Poor-quality model transformation code is costly and hard to maintain. There is a need to develop techniques and tools that can support transformation engineers in designing high-quality model transformations. The goal of this paper is to present a process, called MUPPIT (method for using proper patterns in model transformations), which can be used by transformation engineers to improve the quality of model transformations by detecting anti-patterns in the transformations and automatically applying pattern solutions. MUPPIT consists of four phases: (1) identifying a transformation anti-pattern, (2) proposing a pattern-solution, (3) applying the pattern-solution, and (4) evaluating the transformation model. MUPPIT takes a transformation design model (TDM), which is a representation of the given transformation, to search for the presence of an anti-pattern of interest. If found, MUPPIT proposes a pattern solution from a catalogue of patterns to the transformation engineer. The application of the pattern solution results in the restructuring of the TDM. While MUPPIT, as a process, is independent of any transformation language and transformation engineering framework, we have implemented an instance of it as a tool using transML and MeTAGeM, which support exogenous transformations using rule-based transformation and OCL-based languages such as ATL and ETL. We evaluate MUPPIT through a number of case studies in which we show how MUPPIT can detect four anti-patterns and propose the corresponding pattern solutions. We also evaluate MUPPIT by collecting a number of metrics to assess the quality of the resulting transformations. The results show that MUPPIT optimizes the transformations by improving reusability, modularity, simplicity, and maintainability, as well as decreasing the complexity. MUPPIT can help transformation engineers to produce high-quality transformations using a pattern-based approach. An immediate future direction would be to experiment with more anti-patterns and pattern solutions. Moreover, we need to implement MUPPIT using other transformation engineering frameworks.

Keywords Transformation pattern · Transformation anti-pattern · Model-driven engineering · Transformation engineering

1 Introduction

Models play an important role in specifying, understanding, analyzing, and visualizing software systems [1]. In model-driven engineering (MDE), model transformation, converting a model from one domain into another, is as essential as the models themselves [2]. Examples of model transformations include generating executable code from

✉ Bahman Zamani
  zamani@eng.ui.ac.ir

  Mahsa Panahandeh
  panahand@ualberta.ca

  Mohammad Hamdaqa
  mhamdaqa@ru.is

  Abdelwahab Hamou-Lhadj
  wahab.hamou-lhadj@concordia.ca

1 Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada

2 Department of Computer Engineering and Software Engineering, Polytechnique Montreal, Montreal, Canada

3 School of Computer Science, Reykjavik University, Reykjavík, Iceland

4 MDSE Research Group, Faculty of Computer Engineering, University of Isfahan, Isfahan, Iran

5 Department of Electrical and Computer Engineering, Concordia University, Montreal, QC, Canada

models (forward engineering), reverse engineering code to models (backward engineering), refactoring, and migration between different platforms [3].

Similar to other software artifacts, the quality of model transformations can be improved by applying engineering principles [3–8]. Several "transformation engineering" frameworks have been proposed to generate and manage models and facilitate transformations. In recent years, a considerable effort has been devoted to the definition of transformation patterns to assist software developers in developing effective transformation models [9–13], similar to the way design patterns are used in software development [14,15]. An important aspect of pattern application is the ability to identify opportunities when a specific pattern is needed and to apply the pattern correctly on the transformation. To this end, many studies have been proposed, ranging from the development of metrics for the evaluation of transformation models according to predefined patterns [13,16,17] to the automatic application of patterns on a transformation design model (TDM), which is a representation of a transformation [7,18–20]. Although these approaches have been shown to be useful, they only provide a partial solution to the broader problem of automatic application of transformation patterns. In addition, they almost always require from developers to manually (or semi-automatically) examine the transformation structure in order to recognize situations where the application of pattern solutions is needed [13].

In this paper, we propose a process, called MUPPIT (method for using proper patterns in model transformation), which consists of four steps: (I) identifying the transformation anti-patterns, (II) proposing transformation pattern solutions, (III) applying the pattern solutions, and (IV) evaluating the resulting transformation design model and providing feedback. MUPPIT takes a TDM as input and generates a new pattern-based model as output, which can then be evaluated to show the benefits of using the transformation pattern solution.

MUPPIT is implemented as an Eclipse plug-in, and one of its distinctive features over existing frameworks is that it enables the definition and generation of TDMs using high-level abstraction models as opposed to formal specification languages. The current implementation of MUPPIT relies on transML [3,21] and MeTAGeM [6,22] specifications, which support exogenous transformations using mapping (rule)-based transformation and OCL-based languages such as ATL and ETL. Although MUPPIT uses transML and MeTAGeM, we believe that it is readily extensible to other frameworks such as TROPIC [5], UMLRSDS [23], and the framework proposed by Didonet Del Fabro [24].

The benefits of the MUPPIT approach are demonstrated using three case studies. Firstly, we perform a walk-through of the MUPPIT process in a case study to verify the flow and logic of the MUPPIT steps in details. Secondly, MUPPIT per-

formance is assessed for all case studies using a quantitative evaluation in which several metrics such as syntactic complexity and modularity are measured on the transformations before and after using MUPPIT. These metrics can be used to evaluate a TDM against many indicators of inefficiencies and poor quality, i.e., bad smells. Moreover, we use these metrics in a feedback loop to further suggest new patterns that can enhance the quality of the generated transformation model.

MUPPIT is built on our previous study [17], where we showed how two specific model transformation patterns, namely the *Phased Construction* and the *Auxiliary Model Patterns*, can be recommended to transformation engineers on the basis of analyzing TDMs. MUPPIT is a major extension to the work presented in [17]. More precisely, this paper makes the following new contributions:

– Proposing an end-to-end pattern-based transformation process that enables transformation engineers to automatically identify anti-patterns and apply the corresponding pattern solutions.
– Defining several bad smells that may indicate the presence of anti-patterns by analyzing TDMs.
– Implementing the MUPPIT process as an Eclipse plug-in using Epsilon family of languages (e.g., EOL, ETL, EPL) [25] and Java.
– Applying MUPPIT to three case studies to detect four predefined anti-patterns and propose the corresponding pattern solutions.

The rest of this paper is structured as follows. Section 2 provides preliminary knowledge about transformation engineering and an overview of the transformation engineering frameworks. Section 3 introduces the concept of transformation patterns and anti-patterns and offers four examples of anti-patterns and their corresponding pattern solutions. In Sect. 4, the MUPPIT approach is explained using a motivation scenario. Section 5 presents the framework implementation. MUPPIT evaluation is presented in Sect. 6, followed by threats to validity in Sect. 7. Section 8 reviews the related work. Finally, Sect. 9 concludes the paper by summarizing the main contributions and indicating areas for future work.

2 Background on transformation engineering

2.1 Transformation engineering

In MDE, models are the main artifacts that drive software development [26]. A key aspect of MDE is the ability to convert models from one type (or domain) to another. Model-

ing approaches employ different specification languages for defining new modeling languages (meta-models), specifying models, and defining the transformations between models at different levels of abstraction. Model transformation is seen as code written using a transformation language to transform a source model into a target model. Developing transformations tends to be a challenging and error-prone process [3]. This is due to the complexity of the syntax of the transformation languages, the need to understand the target and source model syntax and semantics, the lack of best practices, and the limited expertise in these languages. In MDE, once the transformations are written and deployed, they are treated as a black box that does the transformation magic. Any error in the transformation code can break the whole MDE solution, not to mention that any inefficiencies would result in important performance issues.

For these reasons, developing high-quality transformations is crucial for the successful adoption of the MDE paradigm in software engineering. To this end, many researchers have examined the application of best software development practices to the development of model transformations [3, 4,6,27,28], which led to the emergence of a relatively new field, often referred to as model transformation engineering or "transformation engineering" for short [3].

In the following section, we review two state-of-the-art transformation engineering frameworks, and elaborate on how our study in related to these frameworks.

2.2 Transformation engineering frameworks

Transformation engineering frameworks aim to enforce the adoption of best practices of software engineering when developing transformation models, meaning that transformations should be analyzed, designed, implemented, tested, and maintained based on sound software engineering techniques. This paper uses two transformation engineering frameworks, namely, transML [3,21] and MeTAGeM [22], which, to the best of our knowledge, are the most comprehensive transformation engineering frameworks to date.

transML is a family of modeling languages, which covers the whole life cycle of transformation development, i.e., requirements analysis, architecture, design, implementation, and testing. These phases result in engineering the transformation generation. transML provides a complete transformation development environment, including notation, methods, and tools. For each phase of this framework, there is a meta-model which provides notations for the transformation engineer to create models that conform to the meta-model of that phase. transML constructs the transformation following the MDE approach in a semi-automatic manner. To develop a transformation using transML, first the transformation requirements are specified as a requirement model. This model is then transformed into other models,

and finally, the implementation models of the transformation are built. In the design phase of transML, the design models can be expressed in two levels of abstractions: high level and low level [3,21].

MeTAGeM is another transformation framework that implements transformations based on the MDE principles [29]. MeTAGeM works on the levels defined in model-driven architecture (MDA) [30]. That means, implementing a transformation starts from platform-independent transformation (PIT), which describes relations between the source and target meta-models. After that, the platform specification transformation (PST) model is created from the PIT automatically. The generated intermediate model contains the definition of the transformation rules based on the high-level specifications presented in the PIT model. The next step is creating a platform-dependent transformation (PDT) that facilitates migration between different abstract levels. This model refactors the PST model based on the selected transformation language. Finally, the transformation code is generated from the PDT [6,22].

In this research, we used transML and MeTAGeM to generate transformation design models (TDMs) in our case studies. A TDM, which specifies a transformation, is the main input of the MUPPIT process. A TDM can be a high-level model, such as a mapping model of transML or a PIT model in MeTAGeM, or it can be a low-level design model of transML or PST model in MeTAGeM. Generating the transformation code from TDMs in MUPPIT is performed using transML or MeTAGeM. Therefore, scheduling the transformation rules, managing the execution schema, and maintaining the transformation behavior are dependent on these frameworks and are out of the scope of MUPPIT. transML uses a behavioral design model, in addition to a TDM, to define traces between the models and action language rules for generating a transformation code. MeTAGeM employs PDT models for specifying the design model elements in the action language. More information on the steps for converting these models into transformation code and managing the execution of the transformations in these frameworks can be found in [3,6,21,22].

3 Transformation patterns and anti-patterns

Similar to software development, the design of model transformations can benefit from the concepts of patterns and anti-patterns. Iacob et al. define a transformation pattern as a reusable solution to a general model transformation problem [11].

This is similar to the concept of design patterns in software development, which is defined as a reusable solution to a commonly occurring design problem [14].

We define a transformation anti-pattern as a common form of transformation that may lead to negative consequences. This definition is inline with the definition of Brown et al. [31] when referring to an anti-pattern in software development, as a pattern in an inappropriate context, which can result in symptoms and consequences.

It should be noted that while the software engineering community seems to agree on the definition of what a good pattern is, the community seems to use different terminologies to describe a bad pattern or a repeatable code or design that may result in bad consequences, such as an anti-pattern, code clone, and code smell. For example, Tahir et al. [32] use the terms anti-pattern and code smell interchangeably.

These concepts are, however, different. A code smell is defined as "a surface indication that usually corresponds to a deeper problem in the system" [33]. It is an indicator, a gauge, a meter, or a measure. An anti-pattern, on the other hand, is the reason behind the problem and also the possible reason behind a bad smell. For example, a poor performance that is measured through the transformation execution time is an indicator of a deeper problem. If we can correlate this with a set of repeatable code instructions or design constructs, then we identified an anti-pattern (e.g., the *Return-First Command* anti-pattern). A simple and known example in software engineering is the *Large Class* bad smell [34] which refers to a class trying to perform too much. A *Large Class* indicates some maintainability difficulties which can be caused by an adverse design or programming solutions, such as *Swiss Army Knife* anti-pattern [33]. *Swiss Army Knife* happens when the developer specifies or implements an interface class for every need of the software. This bad solution can be indicated by A *Large Class* smell.

Identifying anti-patterns and using the appropriate patterns in response can help in (i) restructuring complex transformations into modular sub-transformations, (ii) simplify individual mapping rules of a transformation, (iii) improve the efficiency of a transformation by removing redundant and duplicated evaluations, (iv) optimizing execution strategies, and (v) simplifying complex model navigation [16].

In the next section, we identified four model transformation anti-patterns (those are linked to matching rules), namely the *Spaghetti Transformation*, *Frequent Invocation*, *Return-First Command*, and *Boat Anchor*. For each anti-pattern, we also suggest a pattern solution, which will increase the quality of the transformation. Each anti-pattern might be resolved by several solutions. In this paper, we propose one pattern solution to each anti-pattern, except for one of the case studies where we propose two pattern solutions. We intend to extend MUPPIT to the detection of more anti-patterns and the recommendations of pattern solutions in the future. We show the effectiveness of MUPPIT in identifying these anti-patterns in the case study section.

We selected these four anti-patterns because they are commonly found in model transformations as shown by Cuadrado et al. [12] and Lano et al. [13]. The MUPPIT process can be applied to other transformation scenarios in a similar way. In addition, we selected these anti-patterns because they are based on mapping-based transformation languages, and hence they can be used with transML and MeTAGeM, the transformation engineering frameworks currently supported by MUPPIT. In the following, we present the four anti-patterns and their corresponding pattern solutions that are covered in this paper.

3.1 Spaghetti Transformation and Phased Construction

– **Anti-pattern** The *Spaghetti Transformation* anti-pattern occurs when the developer performs several transformation steps all in one phase. This usually happens in complex transformation rules. There are several signs in the transformation code that tells you if a developer is falling into this anti-pattern. Example of these signs: (i) if a transformation rule contains an alternation of quantifiers ($\forall \exists \forall$), or uses a long alternation sequence, (ii) if the transformation rule is creating more than one target instance at once (in particular, if the rule is referring to target elements at more than one hierarchical level). This anti-pattern reduces the comprehension of the transformation rule, which makes it difficult to maintain, verify, or reuse the rule.

– **Pattern solution** *Phased Construction* [13] is a pattern which decomposes one complicated transformation to separate rules. Each rule relates one source model element (or a group of source model elements) to one target model element. In fact, each rule works on one level of the target meta-model and does not navigate more than one step in the entity composition hierarchy [13]. There are two variations in the *Phased Construction* solution in constructing the target elements using transformation rules: bottom-up and top-down approaches [13]. These two approaches define the order of generating target elements or executing transformation rules. In this paper, we used the top-down approach for the *Phased Construction* pattern solution, meaning that we first generate the top elements in the target meta-model hierarchy and then construct their dependent elements (i.e., lower elements).

3.2 Frequent Invocation and Object Indexing

– **Anti-pattern** The *Frequent Invocation* anti-pattern occurs when a transformation expression frequently accesses objects or set of objects using a unique identifier. Example of such expression: $C.allInstances() \rightarrow select(id =$
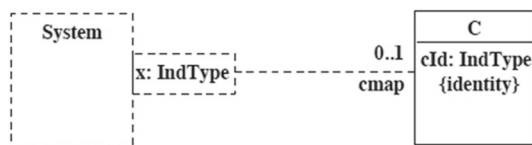
Fig. 1 *Object Indexing* pattern [13]

$v) \rightarrow any()$. This anti-pattern can negatively affect the transformation execution performance with a worst-case time complexity proportional to the number of the invoked instances.

– **Pattern solution** The *Object Indexing* pattern [13] provides a solution for the *Frequent Invocation* anti-pattern. It presents an index map data structure to be used instead of the selection command for accessing objects. This makes it possible to look up the objects using the map structure and the entity primary key. The structure of the *Object Indexing* pattern is shown in Fig. 1, in which each entity of C is stored in the index map data structure of *cmap* in a form like *IndType* $\rightarrow$ *C* where *IndType* is the type of the entitys primary key. Then, access to a C object with a key value of v is obtained by applying *cmap*to v like in *cmap.get(v)*. Hence, a map lookup is substituted for the select expression. This pattern decreases the complexity of the transformation syntax and execution time of the lookup [13].

3.3 Return-First Command and Usage of Iterators

– **Anti-pattern** In functional style-based transformation languages, such as the object constraint language (OCL) [35], the access to objects can be implemented using different iterators (e.g., any, exists, forAll). Using the wrong iterator or the wrong order of operations can significantly impact the performance of the transformation. The *Return-First Command* anti-pattern is a common inefficient transformation anti-pattern that occurs when the developer tries to access one element of a collection that satisfies a condition by using the wrong command. Particularly, by using the *select* command followed by the command *first* in OCL. In this anti-pattern, *select* is not the appropriate command to be used since *select* does not terminate as soon as the condition is satisfied; instead, it returns all the elements that satisfy the condition. For instance, using the *select* command in $collection \rightarrow select(e|e.condition) \rightarrow first()$, when all the elements after the middle of the collection satisfy the condition, returns $n/2$ elements, while the caller of the expression needs just one element.

– **Pattern solution** Cuadrado et al. [12] propose *Usage of Iterators* pattern as part of the recommendations for

performance patterns that can be used to optimize OCL-based model transformations. This pattern suggests that appropriate iterators, which terminate the calculation, by finding the first element, should be employed when there is no need to visit all the elements of a set. Cuadrado et al. [12] suggested using any() in the ATL language as a solution in case of requesting an object with the unique identifier attribute out of all instances. We also used any() in both ATL and ETL languages for the *Return-First Command* anti-pattern.[1]

3.4 Boat Anchor and Filtering

– **anti-pattern** The size of the input transformation model has an impact on the performance and cost of a transformation. *Boat Anchor*[2] anti-pattern occurs when a large input model is transformed into a target model while many of the elements in the input model are not used in the transformation. In fact transformation t transforms input model of *M* to target *Z* while *M* consists $\{m1, m2, , mn\}$ and *Z* consists $\{z_1, z_2, , z_n\}$. *Boat Anchor* happens when some of elements in the set $\{m1, m2, , mn\}$ are not transformed (directly or indirectly) into the elements in the set $\{z_1, z_2, , z_n\}$. This case happens when the target (*Z*) is generated based on a subset of elements in *M*.

– **Pattern solution** *Filtering* pattern, retrieved from [13] is an architectural transformation pattern solution,[3] which removes unused elements from the input model of a transformation. This solution checks the transformation rules and exclude idle concepts in the input model, which are not transformed to the target model. This pattern has also the ripple effect of reducing the size of the input model.

## 4 MUPPIT: a method for using proper patterns in transformations

In the previous section, we elaborated on the role of transformation patterns in improving the quality of a transformation.

---

[1] Cuadrado et al. [12] believe that *any* can be used instead of *select.first()* command in ATL whenever we are looking an object up with a unique attribute. For the ATL language, they implement a fixed *any* version as well to improve the performance more. However, the current paper employs the original version of *any* in ATL. We checked the Epsilon language and identified that *any* is shortcut as soon as an element validating condition is found. Therefore, it is a well-defined iterator in contrast to the select command in *Return-First Command.* More detail about *any* syntax in Epsilon can be found in [36].

[2] *Boat Anchor* is a known anti-pattern in traditional software development, which refers to a piece of software that serves no useful purpose in the current project [31].

[3] Architectural model transformation patterns address solutions to the organizing of transformations systems in order to enhance the modularity, verifiability and efficiency of these systems [13].

Fig. 2 MUPPIT approach
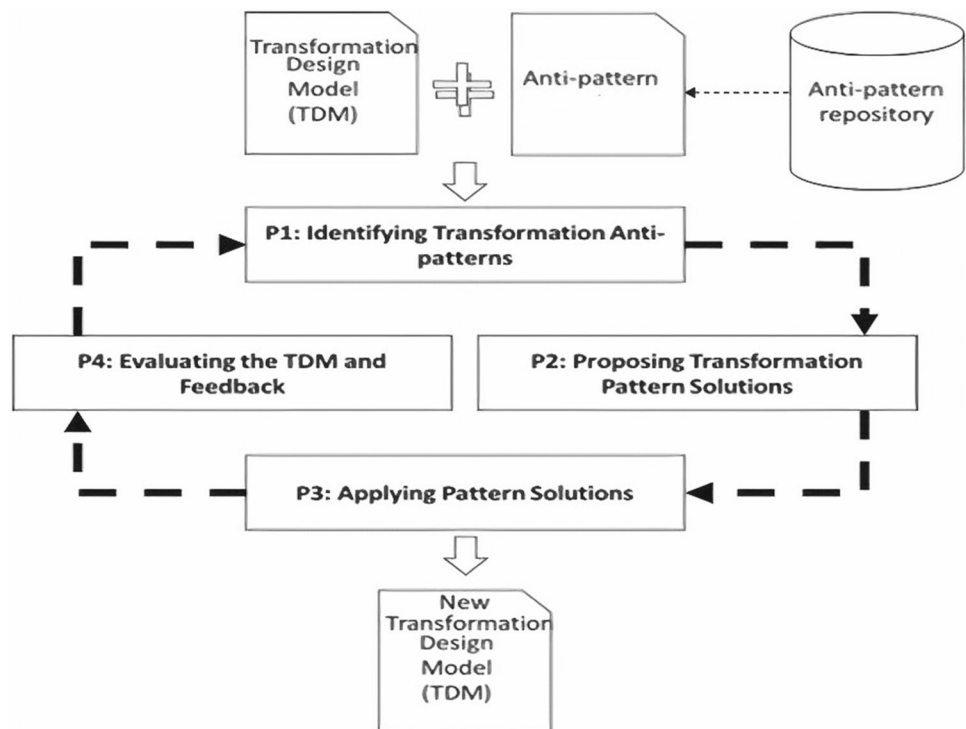
Unfortunately, while a large number of patterns have been developed, in the literature, to address several transformation scenarios, many of these patterns are still not used in practice. The transformation development lacks awareness of the current patterns, or scenarios where these patterns need to be applied. In this section, we present an approach that aims to integrate transformation patterns into model transformation frameworks to enable transformation engineers to assess the developed transformations and use the correct transformation pattern when applicable. The proposed process is called MUPPIT, which stands for "method for using proper patterns in transformations." Figure 2 illustrates the MUPPIT approach, which consists of four phases, namely (P1) identifying the transformation anti-patterns, (P2) proposing transformation pattern-solutions, (P3) applying the pattern-solutions to the original TDM, and (P4) evaluating the new TDM and providing feedback.

In a nutshell, MUPPIT takes a TDM as the main input and uses a repository of anti-patterns to create a new TDM by applying the four mentioned phases. The anti-pattern repository comprises a set of anti-patterns and their corresponding pattern solutions. In the following, these four phases are described.

P1: In the first phase, MUPPIT checks the presence of the selected anti-pattern in the transformation design model to verify if applying a transformation pattern is necessary to improve the quality of the input TDM. If an anti-pattern is detected in the TDM (i.e., a common form of transforma-

tion flaw is detected), this warrants the need for applying the corresponding transformation pattern solution.

P2: In the second phase, MUPPIT inspects the input TDM to see whether or not the pattern solution is used. If the pattern solution is not used in the input model, MUPPIT will propose the pattern solution as an option to improve the input TDM.

P3: In the third phase, the proposed pattern is automatically applied to the TDM after taking permission from the transformation engineer. Accordingly, a new TDM is created.

P4: In the fourth phase, MUPPIT evaluates new generated TDM by measuring several quantitative performance metrics to assess the effectiveness of the applied pattern. These performance metrics are introduced in Sect. 6.4.1.

MUPPIT is a general process that is currently implemented using transML and MeTAGeM frameworks. We intend to explore the use of other frameworks as part of future work. In this paper, we show how MUPPIT was integrated with transML and MeTAGeM frameworks. Accordingly, all models, anti-patterns, and pattern solutions are specified according to the specifications of transML and MeTAGeM. The detailed steps of the MUPPIT approach are shown in Fig. 3. To better understand these steps, a motivation example of a model transformation scenario will be presented; then, the example will be used to explain each of the MUPPIT phases in detail.
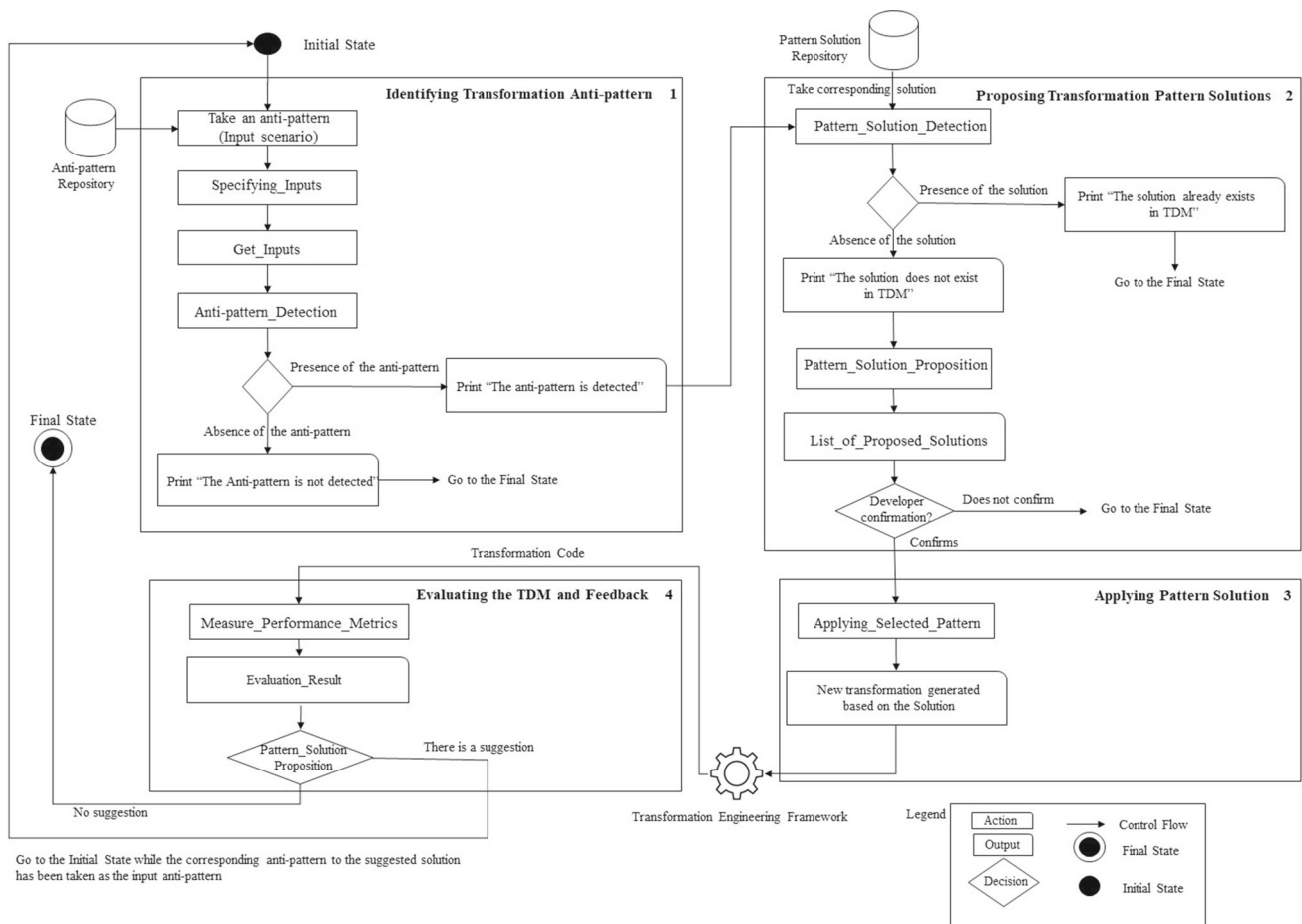
Fig. 3 MUPPIT process

4.1 Motivation scenario: transforming UML class diagrams to relational database tables

A TDM is a model that specifies the rules to transform a source model to a target model. Transforming a UML class diagram into a relational database table is a simple yet complete example, which has been commonly used as a case study by many researchers [3,6,16,37] to clarify the novelty of their new approaches. The same TDM, which is called UML2DB, is employed in this paper as a motivation scenario to explain how MUPPIT works. In the transformation UML2DB, the classes of a given UML class diagram are converted into their corresponding tables in a relational database schema. Each class attribute is transformed into a column in the related table. Moreover, every table needs a specific column as a primary key. The UML class diagram meta-model that represents the source meta-model and the relational database meta-model that represents the target meta-model are shown side by side in Fig. 4.

A schematic view of UML2DB TDM is presented in Fig. 5. This model represents the conceptual structure of UML2DB TDM. It describes the design model in an easy way. This TDM specifies the relations between the elements of the source and target meta-models in the UML2DB transformation. UML2DB includes two mappings. These are Class-Map and Attribute-Map, which transform classes to tables and attributes to columns, respectively. For every class, the Class-Map generates a primary key in the related table.

As explained earlier, MUPPIT uses TDMs that are defined according to transML or MeTAGeM specifications. In other words, MUPPIT uses TDMs that conform to the transML or MeTAGeM metamodels. For this, it uses the transML or MeTAGeM frameworks to specify the input TDMs. Each of these frameworks has two design abstraction levels: a high abstraction level that is used to specify the mapping relationships between the source and target model elements, and a low abstraction level that is used to specify the detail implementation of the transformation. Figure 6 shows a TDM in EMF [38] format that corresponds to the UML2DB transformation and is generated through the mapping phase using the transML framework. Figures 21, 22 and 23 in "Appendix" show the meta-models of transML and MeTAGeM that are used in MUPPIT. The TDM models (i.e., instance models) that are specified according to the high-level meta-model
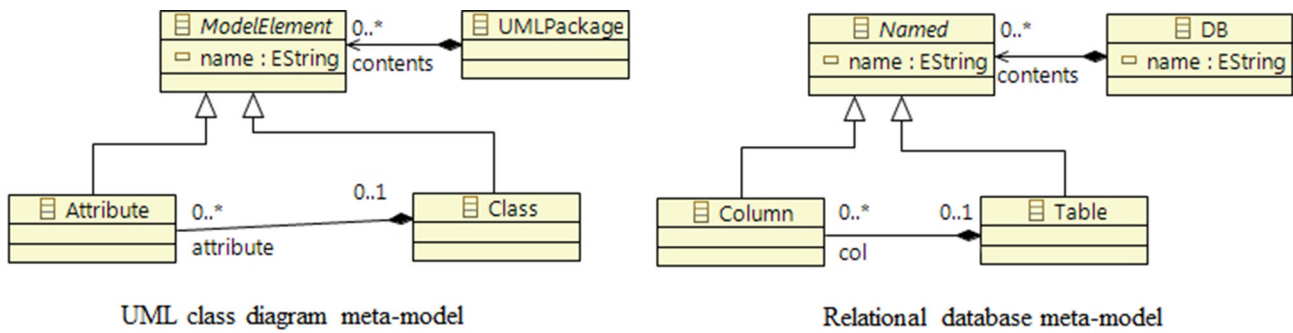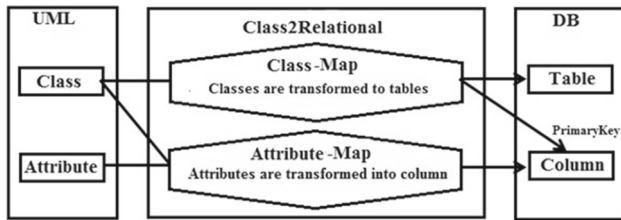
Fig. 4 UML2DB transformation meta-models

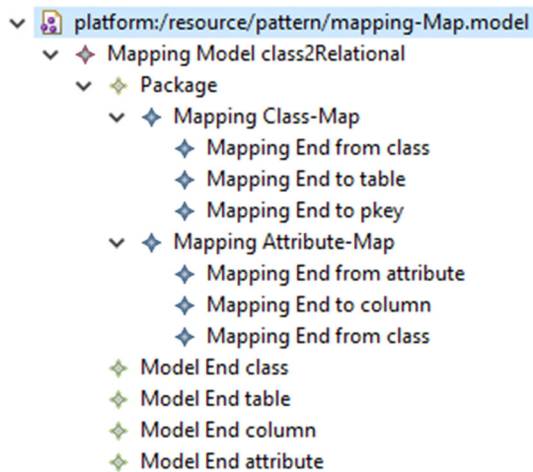**Fig. 5** Structure of UML2DB design model

Fig. 6 Structure of UML2DB design model (EMF format)

is referred to as "platform-independent model" (PIM) in MeTAGeM, and mapping diagram in transML, while a low-level TDM is referred to as "platform specific model" (PSM) in both frameworks.

In the next subsections, the phases of MUPPIT are explained, and then every phase is elucidated using the UML2DB example. We used the first two cases explained in Sect. 3 as a sample set for transformation anti-patterns and pattern solutions. Accordingly, we used MUPPIT to refine the UML2DB transformation structure and generate a new TDM.

4.2 Identifying transformation anti-patterns

The input requirements for MUPPIT to be able to identify transformation anti-patterns are the TDM, the transformation design meta-model, transformation source and target meta-models, and the anti-patterns catalogue. The first phase of MUPPIT starts by the transformation engineer selecting an anti-pattern from the anti-pattern catalogue to check its presence in the input TDM. After selecting an anti-pattern, MUPPIT requires the TDM, the TDM meta-model, the source model meta-model, and the target model meta-model. However, some anti-patterns (e.g., the *Boat Anchor* anti-pattern) require access to the source model as well. We expect that a transformation engineering framework that supports the MUPPIT process would allow enough flexibility for transformation engineers to specify these models, which need to be specified only once. Transformation engineers can apply MUPPIT multiple times on the TDMs that work on the same models.

After providing the required inputs, MUPPIT triggers the "Anti-pattern Detection" task shown in Fig. 2. This task checks if the anti-pattern appears in the input TDM. If there is no matching anti-pattern in the TDM, MUPPIT prints the message "The anti-pattern" is not detected and proceeds to the final state (i.e., select anti-pattern from the repository). If a match is identified, MUPPIT prints the message "The anti-pattern" is detected and proceeds to the next phase (i.e., "proposing transformation pattern solutions"). The "Anti-pattern Detection" task uses structural constraint-based pattern matching, in which a matching anti-pattern is defined as a set of constraints on the TDM meta-model. Here we can distinguish between two types of matching constraints: *relational mapping* at the high-level abstraction design model and *operational* at the low-level design model. The rules for identifying the *Spaghetti Transformation* and *Boat Anchor* anti-patterns are examples of *relational mapping* rules, while the rules for *Frequent Invocation* and *Return-First Command* anti-patterns are examples of *operational* rules.

In MUPPIT, the syntax to specify the pattern domain and perform the matching in the "Anti-pattern-Detection" task is based on the Epsilon pattern language (EPL) [25]. The syntax of the EPL language contains three main parts including match, onmatch, and nomatch blocks. Listing 1 shows the EPL syntax for defining a pattern.

```
1    pattern patternName
2      Definition of roles {
3        match : PatternSpecification
4      }
5    onmatch{}
6    nomatch{}
```

Listing 1 The EPL syntax for defining a pattern

In the above listing, the patternName is the name that is assigned to the pattern. Roles are those metamodel domains, i.e., instance elements in execution time, involved in pattern specification. The match block includes a formal definition of the pattern in Epsilon language. This definition represents a conditional constraint on the subject meta-model (e.g., TDM meta-model), which will be satisfied if an instance model (e.g., TDM) conforms to the pattern definition. The onmatch and nomatch blocks represent the actions that will be executed when the condition is satisfied or violated, respectively.

As MUPPIT uses TDMs generated using transML and MeTAGeM frameworks, the anti-patterns need to be specified using the design meta-models of transML or MeTAGeM. The initial anti-pattern repository provides EPL codes defining the anti-patterns explained in this paper according to transML and MeTAGeM. These EPL codes perform the anti-pattern matching on the target TDM. Table 1 shows the

definitions of the anti-patterns used in this paper as defined in the anti-pattern catalogue.

Figure 7 shows the definition of *Spaghetti Transformation* anti-pattern to detect the anti-pattern on TDMs that conforms to transML meta-model. This figure presents a part of the transML meta-model (the complete meta-model is shown in "Appendix," Fig. 21), as well as the pattern matching rule (on the arrow). The *Spaghetti Transformation* anti-pattern transformation is specified as a *relational mapping* constraint rule at the high-level abstraction design model; hence, the related metamodel (e.g., transML or MeTAGeM) is required for the anti-pattern definition. The rule on the arrow checks if the *Spaghetti Transformation* anti-pattern is presented in the instance TDM, by checking if the TDM has a mapping rule with more than one target MappingEnd. The target MappingEnd elements are recognized by the Boolean attribute "navigable," which has a true value for target MappingEndTDMs. In other words, the mapping rule access more than one level of the target meta-model or create more than one target element at once in one mapping rule.

Frequent Invocation and *Return-First Command* anti-patterns both address issues regarding the usage of appropriate operations (e.g., *select, any*) in a TDM. These operations are part of the syntax of the transformation-code. Consequently, the constraint rules for these anti-patterns are defined at the operational low-level design metamodel. Figure 8 is part of the low-level design metamodel of MeTAGeM (the complete meta-model is shown in "Appendix", Fig. 23). The figure shows the part related to defining the operations in a TDM developed in MeTAGeM. As shown in the figure, each

Table 1 Anti-pattern catalogue

Anti-pattern	Anti-pattern detection	Level of application/framework
Spaghetti Transformation	$mapping : Mapping.ends.select(mapend : MappingEnd|mapend.navigable = true).size() > 1$	Relational/mapping meta-model of transML
Frequent Invocation	$"select".isSubstringOf(op : Operation.body)$	Operational/low-level design model of MeTAGeM
Return-First Command	$"select().first()".isSubstringOf(op : Operation.body)$	Operational/low-level design model of MeTAGeM
Boat Anchor	$if(not(InputMetamodel.allInstances().equal(ModelRoot.Relations.source))$	Relational/high-level design model of MeTAGeM

Fig. 7 A partial definition of the *Spaghetti Transformation* anti-pattern in EPL
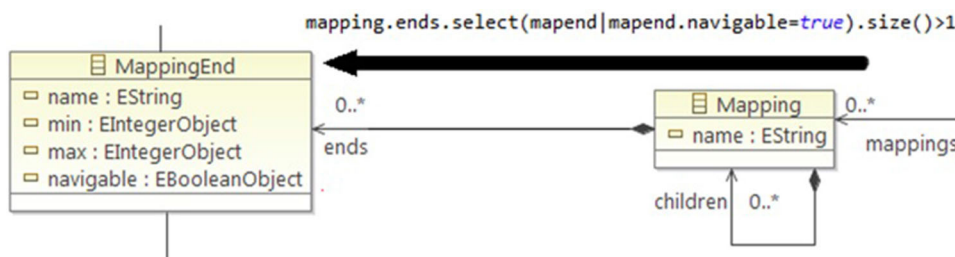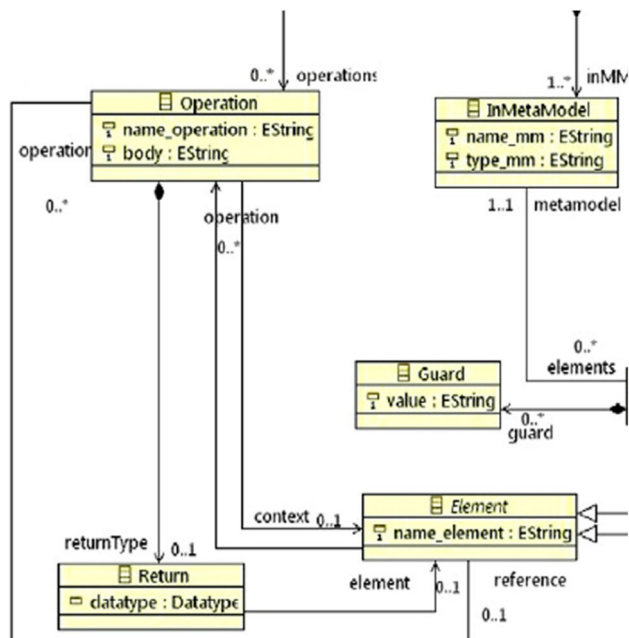
Fig. 8 A part of MeTAGeM metamodel representing operation parts

element of a TDM has an operation concept. An example of an operation concept is the "select" and "select().first()" operation. The anti-pattern catalogue provides the EPL codes for *Frequent Invocation* and *Return-First Command* to explore source code of a TDM and, respectively, identify any inappropriate usage of the select() and select().first() commands.

Boat Anchor is an anti-pattern which refines the high-level TDM. Table 1 shows how this anti-pattern can be defined for the high-level design metamodel of MeTAGeM. This anti-pattern searches the weaving model, high-level TDM designed by MeTAGeM, to find elements in the transformation input model which are not used in TDM. Figure 22 in "Appendix" presents the high-level design metamodel of MeTAGeM related to specifying weaving relations or mappings. As the figure shows, each high-level TDM in MeTAGeM has a Model Root element, which consists of some relations. Each relations defines a mapping which weaves the source elements to target elements. Thus, each relation can have various kinds of source and target elements. The EPL code checks the transformation input model against the source elements in the TDM. *Boat Anchor* is detected when the transformation input model comprises of elements not employed in specifying the weaving relations. Input-Metamodel.allInstances() returns a set including all instances of the input metamodel contained in the transformation input model. *Boat Anchor* is found when this returned set is not equivalent to the set of source elements in weaving relations of the TDM.

In our motivation scenario (i.e., UML2DB), to assess the input TDM against the *Spaghetti Transformation* anti-

pattern, the transformation engineer starts by selecting the *Spaghetti Transformation* anti-pattern. Accordingly, MUP-PIT prompts the transformation engineer for inputs. In this scenario, the TDM of UML2DB specified using transML, shown in Fig. 5, and the transML design meta-model, which is shown in "Appendix" Fig. 21, along with transformation source and target meta-models are taken by MUPPIT.

The "Anti-pattern-Detection" task will validate the existence of the anti-pattern by executing the EPL code in Fig. 7. In the case of UML2DB, the *Spaghetti Transformation* anti-pattern is detected in the TDM. This is because the Class-Map in the TDM accesses the table and column levels in the hierarchy of the relational database meta-model. This match the anti-pattern definition. Accordingly, MUPPIT will show the message "The anti-pattern" is detected, then proceed to the second phase (i.e., "proposing transformation pattern solutions").

4.3 Proposing transformation pattern solutions

In the second phase, "proposing transformation pattern solutions", MUPPIT starts by searching the TDM for possible transformation pattern solutions that correspond to the identified anti-pattern in the first phase. If a pattern solution is detected, the process terminates, else MUPPIT suggests a list of possible pattern solutions that can be applied to improve the TDM, then waits for a confirmation. If the transformation engineer selects one of the proposed patterns, MUPPIT proceeds to the third phase (i.e., "applying pattern solutions"). The transformation engineer has the option at any point to provide a custom solution or terminate the process.

As explained earlier, while each anti-pattern may have one or a list of corresponding pattern solutions, the presence of an anti-pattern in a TDM does not mean that the TDM designer did not consider the pattern solution. Separating the process of detecting an anti-pattern from detecting the presence of the solution has several advantages. It improves the approach modularity and usability by enabling the TDM engineers to provide different levels of matching rules when available and as needed. More importantly, it enables incremental and continuous improvement in the TDM through feedback loops with guaranteed termination.

Similar to the anti-pattern detection process, in our work the pattern solution detection uses structural constraint-based pattern matching that is specified according to the syntax of the Epsilon pattern language (EPL). Also, the solutions pattern matching rules can be at the relational and operational levels. Table 2 shows the catalogue of the pattern solutions used in this paper. In some cases such as *Phased Construction*, the pattern solution is specified by constraints reversing the anti-pattern specification. The *Phased Construction* pattern is specified in a similar way as the *Spaghetti Transformation* anti-pattern with negation constraint in EPL.

The EPL match block of the solution includes a condition to ensure that each mapping contributes to generating 'one' level of the target model (i.e., each Mapping instance in the TDM has one MappingEnd instance), while the detector condition of the *Spaghetti Transformation* anti-pattern looks for mappings with 'more than one' mappingEnds. In more complex scenarios, such as the *Frequent Invocation* scenario and *Object Indexing*, the matching constraints used in detecting the pattern solution are more than reversing or complementing the constraint matching specifications of the related anti-pattern detector. Table 2 shows the pattern solution catalogue of *Object Indexing* as a case, which needs searching TDM against the mapping structure for accessing the elements.

Back to the UML2DB scenario, MUPPIT detected the *Spaghetti Transformation* anti-pattern in the previous phase. Next, in "proposing transformation pattern solutions," MUPPIT looks for the corresponding solution, (i.e., *Phased Construction* solution pattern) to see if it already exists in the TDM. In this case, the *Phased Construction* pattern has not been identified in the TDM. Thus, the *Phased Construction* pattern is proposed to the transformation engineer, and MUPPIT proceeds to the third phase (i.e., "applying pattern solutions").

4.4 Applying pattern solutions

Applying the pattern solution to the input TDM is a complex process. It includes model comparison, merging, validation, and model-to-model transformation. The input to this phase is the selected pattern solution from the second phase. MUPPIT uses the solution pattern to modify the original TDM and generates a new TDM that conforms to the pattern solution. For relational anti-patterns/solutions introduced in Table 1, applying a pattern is a model to model transformation while the source and target models are a relational TDM, such as mapping design model in transML. This transformation, i.e., applying the pattern solution, is performed on hybrid TDMs for operational scenarios. Hybrid models are rule-based specifications for model-to-model transformations, using OCL to encode the transformation logic, such as the low-level design models of MeTAGeM.

For each solution pattern, a transformation is developed that takes the initial TDM as input and transforms it into a target TDM based on the pattern solution specification. This transformation has been implemented in Epsilon and mainly Epsilon transformation language (ETL) [25] in MUPPIT. The transformation that converts the TDM into the desired one is the main operator of this step. However, this transformation might be integrated into some pre/post-configurations. Usually, we need to provide some pre-configurations such as comparing the TDM with the pattern solution and desired target. A configuration might also

be required as post-condition to refine the generated TDM after applying the pattern solution.

For example, if the *Phased Construction* pattern was selected to be applied as a solution, a transformation code is needed to split those rules accessing more than one level of the target metamodel and generate one to one mapping instead. Listing 2 shows an ETL code snippet of applying the *Phased Construction* pattern solution. Particularly, it shows the process of retrieving then rebuilding the mappings in a TDM based on the suggested solution. Line 3 retrieves all mappings (i.e., transformation rules) in the original TDM. Then refine those mappings that require restructuring based on the suggested pattern solution (i.e., *Phased Construction* pattern). The "patternSet" in line 5 includes only those mappings that require applying the pattern solution (those with several transformation steps all in one phase). Based on the *Phased Construction* pattern, each transformation rule should generate only one target element. To do that, in lines 7 and 8 the target elements generated by each mapping are collected then counted. Accordingly, one mapping is created for each collected target elements. Lines 10–22 show the process of splitting a mapping based on the number of target elements in that mapping, starting from creating a mapping rule for each target in lines 12 and 13, then append the source element in lines 14–19, and finally, append the target element in line 20.

```
1  post
2  {
3  for (elem in mapping!Mapping.allinstances())
4  {
5     if(not elem.instanceOf
6      (patternSet!phasedconstruction))
7     {
8        var mapEnds:=elem.ends.
9         select(mapend|mapend.navigable=true);
10       var count:= mapEnd.size();
11        //count of mappingEnd
12       var i:=1;
13       while(i<=count-1)
14       {
15          var map:= mapping.createInstance
16          (elem.type().name);
17          map.name:= elem.name;
18          var navigableFalse:= elem.ends.select
19           (mapend|mapend.navigable=false);
20          for(e in navigableFalse)
21          {
22           var t:= emfTool.getECoreUtil().
23           copy(e);
24           map.ends.add(t);
25          }
26          map.ends.add(mapEnds.at(i));
27          mapping!Package.all.first().
28           mappings.add(map);
29          i:= i+1;
```

Listing 2 ETL code snippet of applying the *Phased Construction* pattern solution

Table 2 Pattern solution catalogue

Solution pattern	Solution pattern detection
Phased Construction	**select all** mapping: Mapping
	for each mapping **do**
	if mapping.ends.**one**(mapend|mapend.navigable=true)
	this transformation rule satisfies the Phased Construction solution
	else
	return violation of Phased Construction solution
	end for
	return presence of Phased Construction solution
Object Indexing	**select all** operation: Operation
	for each operation **do**
	if operation.body.includes(Map-Structure)&&
	operation.body.includes(get function on Map-Structure)
	then boolean ObjectIndexingsolution=1
	else
	if operation.**instanceOf**(patternSet!FrequentCall)
	return violation of Object Indexing solution
	end for
	if ObjectIndexingsolution=1
	return presence of Object Indexing solution
Usage of Iterators	**select all** operation: Operation
	for each operation **do**
	if "any".**isSubstringOf**(operation.body)
	return presence of Usage of Iterators solution
Filtering	**select all** InputMetamodel.**allInstances**()
	for each element **do**
	if ModelRoot.Relations.source.**asSet**().**includes**(element)
	return "presence of Filtering solution"

Applying this transformation on the initial TDM will automatically identify the mappings that expose the anti-pattern and then restructure them based on the suggested pattern solution.

In the case of the UML2DB transformation, MUPPIT applies the selected pattern solution (i.e., the *Phased Construction* pattern) by executing the transformation in listing 2. The generated output TDM is shown in Fig. 9. The TDM is shown in EMF format and conforms to the transML design model. As the figure shows, the new TDM has one more extra mapping compared to the original TDM. This extra mapping is called Class-MapII. The Class-MapII is the result of dividing the Class-Map into two mappings: Class-Map and Class-MapII. This new mapping restructured the *Spaghetti Transformation* anti-pattern in the UML2DB TDM so that it conforms to the *Phased Construction* pattern solution.

Applying some of the pattern solutions is a more complex task. For example, applying *Object Indexing* needs pre-configuration. We need to define a map structure for the detected commands in the transformation rules, which include frequently accessing instances of an element. Then,

this map structure is substituted for the commands in the TDM according to the *Object Indexing* specification.

Listing 3 shows a code snippet of applying *Object Indexing* solution pattern to low-level TDMs generated by MeTAGeM. Applying *Object Indexing* is triggered by the transformation engineer confirmation if TDM contains the frequent calls. As Fig. 23 in "Appendix" shows, the low-level TDM of MeTAGeM expresses the commands in the body operations within rules. If a low-level TDM generated in MeTAGeM contains commands looking up objects by value in their operation body, MUPPIT is proceeded to the second phase. The second phase, consisting proposing transformation pattern solution, checks the TDM against the presence of a map structure. Checking the operations body in TDM is performed by an implemented Java plug-in. As mentioned before, we integrated Java and Epsilon in MUPPIT implementation. Adding *Object Indexing* to MUPPIT is one of these cases that needs to use Java with Epsilon. To this end, we extended EPL classes when implementing MUPPIT. The Java plug-in uses a regular expression library to find a map structure pattern in the operations body string.
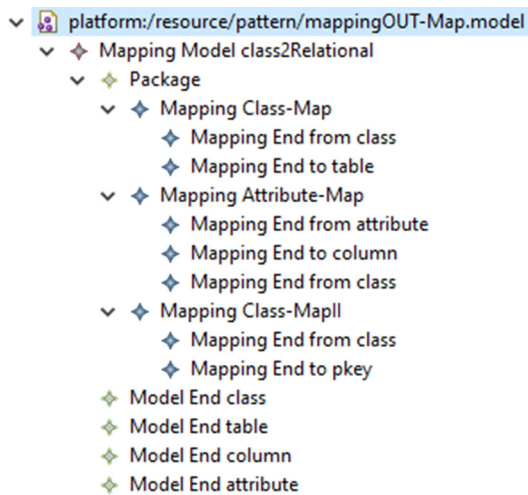
Fig. 9 Structure of UML2DB design model after applying the *Phased Construction* pattern (EMF format)

Applying *Object Indexing* after the confirmation made by the transformation engineer is performed by an ETL code, which calls a Java plug-in. The input of this phase is a patternSet, including operations in the TDM by a 'select' command for access to an object by values. The patternSet is the output of "identifying transformation anti-pattern" phase, as explained in Listing 2. Then according to Listing 3, for each operation in the pattern set, a map structure is built. First, in line 7, the context of the invoked element, the element of TDM that owns the operation, is extracted. Line 9 puts all instances of that context in the setOfContexts. Then, the generated Java plug-in, ObjectIndexingJavaTool, is invoked by

generating an instance named *sample* in line 11. This is how we connected Epsilon to Java in our implementation. This plug-in has a function that obtains an operation body as string and checks it to find the looked up property in the select command. Listing 4 shows in more detail how the plug-in works. Lines 15–18 build a map structure for setOfContexts and the looked up properties for this context. Finally, another Java plug-in is called to replace the map structure with the select command. The result is shown in Sect. 4.5 when discussing the evaluation of the TDM. This substitution is performed by considering the body of operation and select command as a string type and managing that using regular expressions in Java.

```
1  --foreach operation inside of the patternSet
2    pre
3  {
4    var map:=new Map;
5  }
6  --Return the context of the invoked element
7  var context:= operation.context.name_element;
8  --Extract all instances of invoked element
9  var setOfContexts:=
10   Transformation_source_meta-model.
11   getAllOfKind(context)
12 --Extract the invoked property using a java
13   plug-in
14 var sample = new Native('objectIndexingJavaTool.
15   ObjectIndexingJavaToolClass');
16 -- property is set of looked up properties
17   in the select command
18 var Property:=sample.func(op.body);
19 --constructing the map structure
20 for (p in Property)
21 {
```

Fig. 10 UML2DB before (**a**) and after (**b**) applying the *Object Indexing*
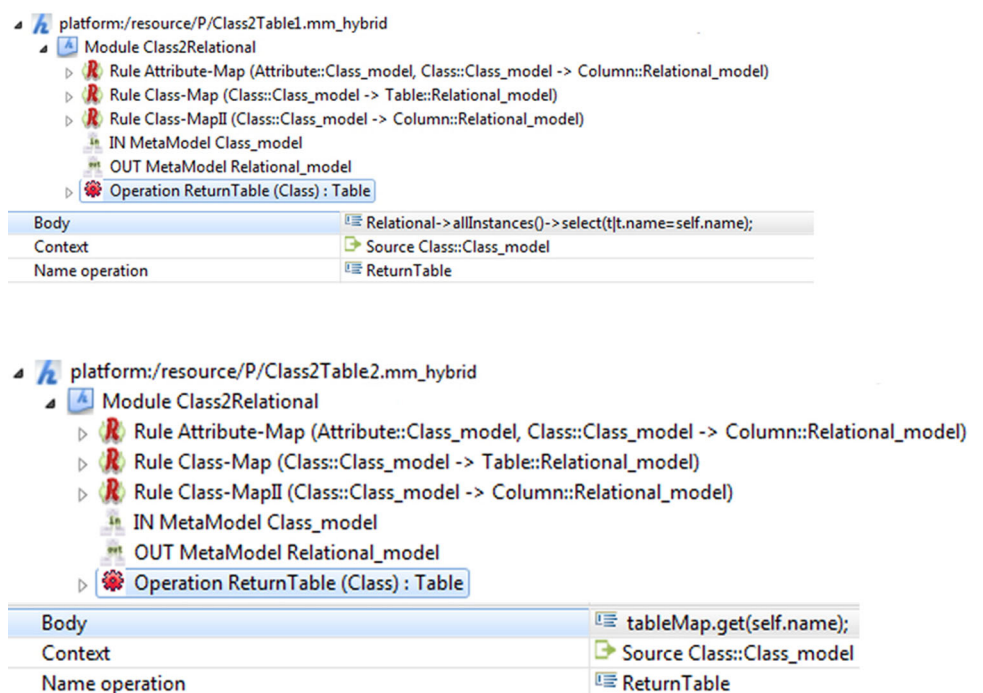
```
22  map:=setOfContexts.mapBy(m|
23  Transformation_source_meta-model.
24    propertyGetter.invoke(m,Property.at(count)));
25  count:=count+1;
26  }--end of for
27  //call a java plug-in for substituting
28    the map instead of select command
```

Listing 3 Epsilon code snippet of applying the *Object Indexing* pattern solution

```
1   public List<String> Func(String body) {
2       List<String> set = new ArrayList<String>();
3       Pattern p = Pattern.compile("select\\([a-zA-Z0-9]+\\|
4       [a-zA-Z0-9]+\\.(.*?)[<>|=|>|>=|<|<=]");
5       Matcher m = p.matcher(body);
6       while (m.find()) {
7        set.add(m.group(1)) ;
8       }
9       return set;
10      }
```

Listing 4 Java partial code for parsing operations' body in TDMs

We have considered the presence of several sequential selects in an operation body of a TDM or invoking for several features of a context. However, the MUPPIT implementation has some limitations in the current version. For instance, the current implementation cannot resolve nested 'select' commands as frequent calling of an object.

4.5 Evaluating the TDM and feedback

The TDM evaluation and feedback phase is an independent phase in MUPPIT that can be accessed directly by the transformation engineer to assess an existing TDM (i.e., before the first phase), or after generating a new TDM in the third phase. This phase uses a set of key performance indicators, such as the syntactical complexity, modularity, and exaction time as a base to evaluate the quality of a TDM and suggests some of the possible root causes (anti-patterns) that a TDM may exhibit. Then, based on the key performance indicators and the available possible pattern solutions, the TDM can go into a number of consecutive refinements until no further improvement is possible. The key performance indicators are explained in detail in Sects. 6.4.1 and 6.4.2. The evaluation of these metrics is done at the source code level of the TDM. transML and MeTAGeM are used to generate the transformation source code.

In the case of the UML2DB transformation, the new TDM generated in the third phase was evaluated against the aforementioned key performance indicators; the results (see Table 5 in the evaluation section) show that the syntactical complexity of the UML2DB TDM is high based on a predefined threshold. Comparing the metrics with predefined threshold by the transformation engineer, helps he/she to select next anti-pattern to be checked. The *Frequent Invocation* could be as a possible root–cause, and hence, the *Object Indexing* could be used as a solution to reduce the syntactical complexity [13]. By selecting the *Frequent Invocation* anti-pattern in MUPPIT, MUPPIT directs the process to the first
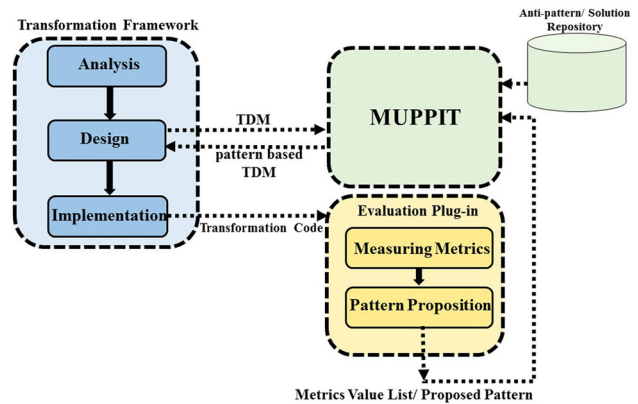

Fig. 11 Overview of MUPPIT connection to the transformation engineering frameworks

phase to detect the presence of the anti-pattern in the TDM and the engineer can check if this anti-pattern is the root cause behind the high syntactical complexity. In this case the anti-pattern was detected, and the pattern solution (i.e., *Object Indexing*) is not used. Hence, MUPPIT automatically applies the *Object Indexing* pattern on UML2DB TDM. Figure 10 shows the UML2DB TDM before and after applying the *Object Indexing* pattern, where the *select* command is substituted by the map structure. This improves the UML2DB TDM syntactical complexity [39], analyzed in Sect. 6.4.1.
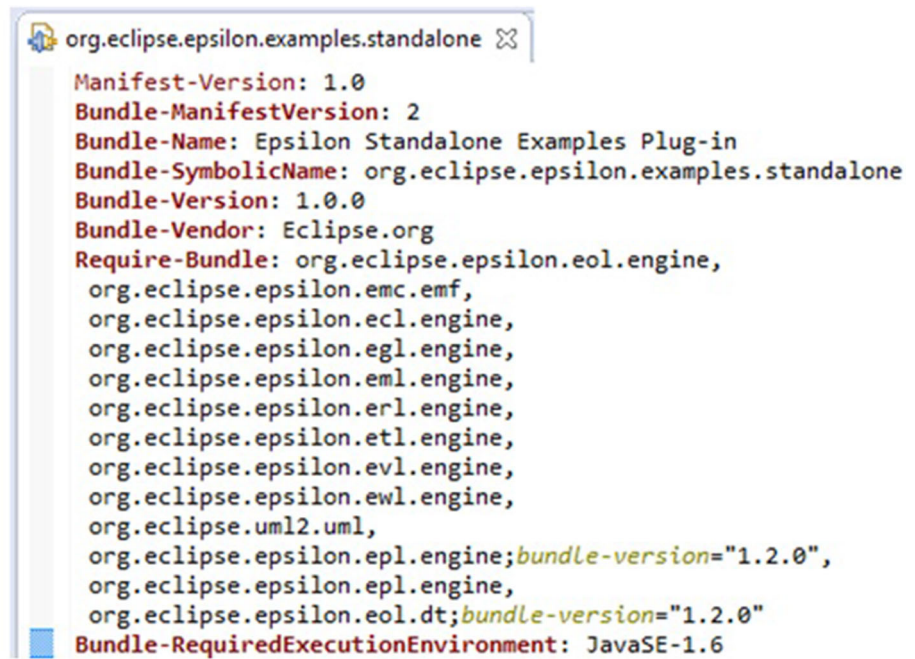
This section presented the MUPPIT process and showed through a simple scenario, how MUPPIT works. In the next section we explain the MUPPIT implementation and tool support.

5 MUPPIT implementation

MUPPIT defines the steps required for identifying anti-patterns, detecting and applying pattern solutions, and evaluating pattern application in a transformation. MUPPIT has been realized as a framework that facilitates TDM evaluation and restructuring based on best practices. The MUPPIT framework extends some of the existing transformation frameworks, integrates several plug-ins, and provides a repository of transformation anti-patterns and their corresponding pattern solutions. Figure 11 shows the high-level architecture of the MUPPIT framework integrated to the transformation engineering frameworks.

The backbone of MUPPIT consists of two main components (plug-ins) and a pattern repository. The first component is the MUPPIT core component, which was implemented as an extension of Epsilon [25]. Epsilon is a family of languages and tools, which offers comprehensive facilities in the realm of model-driven engineering. Among Epsilon languages, Epsilon transformation language (ETL), Epsilon

Fig. 12 List of required plug-ins for interaction between Epsilon and Java


```
org.eclipse.epsilon.examples.standalone  ⊠

    Manifest-Version: 1.0
    Bundle-ManifestVersion: 2
    Bundle-Name: Epsilon Standalone Examples Plug-in
    Bundle-SymbolicName: org.eclipse.epsilon.examples.standalone
    Bundle-Version: 1.0.0
    Bundle-Vendor: Eclipse.org
    Require-Bundle: org.eclipse.epsilon.eol.engine,
     org.eclipse.epsilon.emc.emf,
     org.eclipse.epsilon.ecl.engine,
     org.eclipse.epsilon.egl.engine,
     org.eclipse.epsilon.eml.engine,
     org.eclipse.epsilon.erl.engine,
     org.eclipse.epsilon.etl.engine,
     org.eclipse.epsilon.evl.engine,
     org.eclipse.epsilon.ewl.engine,
     org.eclipse.uml2.uml,
     org.eclipse.epsilon.epl.engine;bundle-version="1.2.0",
     org.eclipse.epsilon.epl.engine,
     org.eclipse.epsilon.eol.dt;bundle-version="1.2.0"
    Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

object language (EOL), and Epsilon pattern language (EPL) are used in instantiation of MUPPIT.

The second component is the evaluation plug-in that has been implemented as Java plug-in to make it reusable for other research purposes. This plug-in measures performance metrics and analysis them to propose proper patterns for transformation refinement. The current pattern repository is preloaded with the definitions of three anti-patterns and their corresponding pattern solutions, those referred to in this paper. The anti-pattern definitions were specified in the EPL language. EPL facilitates pattern matching in models that conform to specific a meta-model. As MUPPIT uses TDM generated in transML and MeTAGeM frameworks, the anti-pattern scenarios are specified based on the transML and MeTAGeM design meta-models. Applying Pattern Solutions is performed by executing transformation codes implemented in ETL. These transformations take the input TDM and change it to a new one, which embedded pattern solution in its structure. Depending on the complexity of the pattern, a combination of ETL code along with EOL and EPL may be needed.

In addition to the main components, MUPPIT integrates several plug-ins, provides high-level abstraction of some of the Epsilon formal code in order to increase usability, implements a parser and analyzer for the EPL commands in Java, and alleviates some of tedious plumbing work that the transformation engineers need to do, such as using some Epsilon operations, calling other Epsilon code as well as importing and using some of the prerequisite Java plug-ins. Figure 12 shows a list of the plug-ins in MUPPIT manifest.

In the next section, we explain how we evaluated MUPPIT.

6 Evaluation

The goal of this section is to evaluate the *effectiveness* of MUPPIT through its ability to generate the desired TDMs and the quality (e.g., maintainability, comprehension, reusability, and performance) of the generated TDM output. To illustrate MUPPITs ability to generate the desired TDMs and verify the flow and logic of the MUPPIT steps, a walkthrough example is used to illustrate how MUPPIT can be applied to automatically detect common transformation problems that affect the quality of a TDM and then utilize best practices to restructure the TDM to produce a new TDM.

To assess the quality of the generated results, we compare the quality of the original TDM and the one generated after applying MUPPIT on the TDM. The quality of the TDMs is compared based on the set of key performance indicators (metrics) in Sect. 6.4. These metrics provide quantitative measures regarding the transformation rules, such as the syntactical complexity, the number of "create" actions per rule, the MSC, modularity, and the execution time of running the transformation. These quantitative measures will be used then to reason about some of the qualitative measures of the TDM generated through MUPPIT, such as the transformation code maintainability, comprehension, reusability, and performance.

In order to test the behavior of a generated TDM by MUPPIT, i.e., TDM after applying the pattern solution, we have compared the transformation target model generated from the TDM before and after using MUPPIT. Comparing the result after the changes with the original source is recognized as one approach for evaluating behavior preservation in refac-
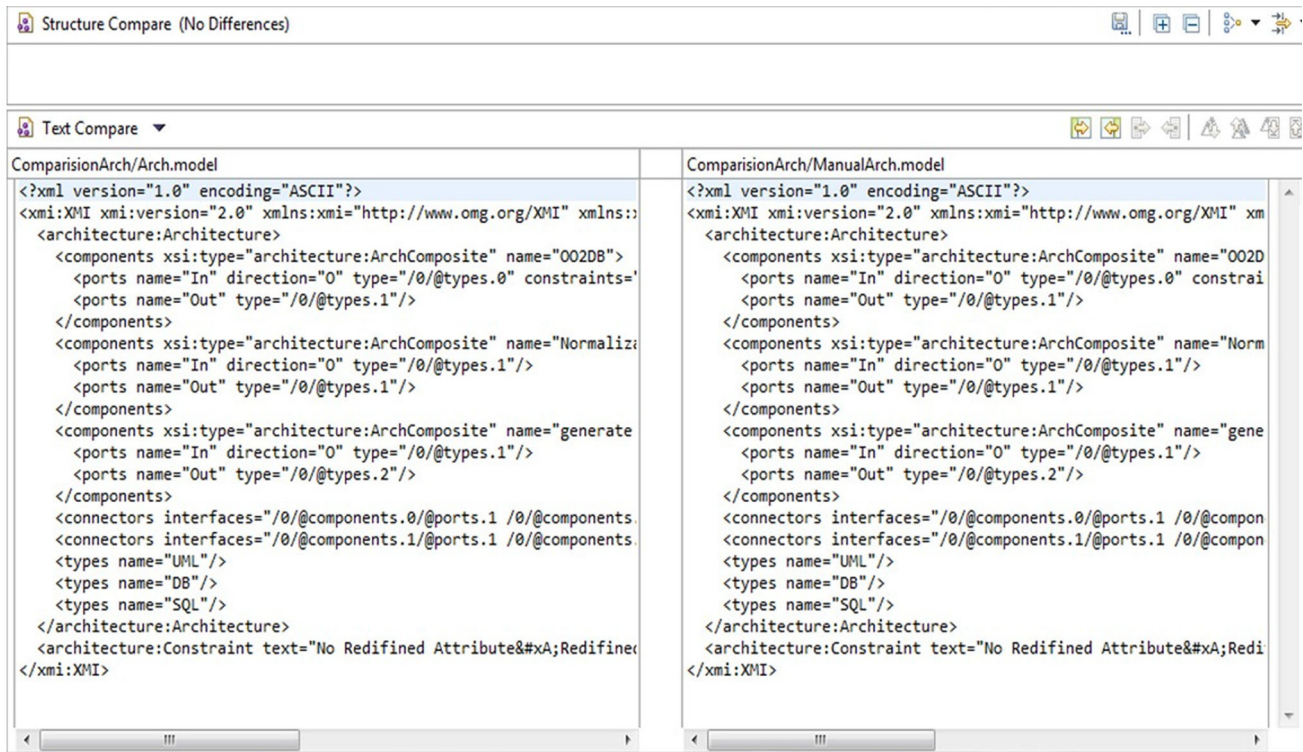
Fig. 13 Comparing a transformation target model before and after using MUPPIT in EMF Compare

toring [40] and was used in model transformation refactoring by Wimmer et al. [41]. We use EMF Compare[4] to compare the two TDMs. EMF Compare is a component for comparing models in EMF format. The differences between two given models are highlighted in this tool. Figure 13 shows a snapshot of EMF Compare when comparing two transformation target EMF models, generated from a TDM before and after using MUPPIT. As we can see in the figure, the two models are the same which means that the transformation behavior is preserved by MUPPIT.

In the next subsections, we present two different case studies including transforming FIXML models to Object models (FIXML2Obj) and Ecore to relational schema (Ecore2Schema). Then, we show the results of using MUPPIT instance on one of the case studies, FIXML2Obj, by walking through the MUPPIT process. The second phase of our evaluation consists of an assessment on the quality of the TDMs generated by MUPPIT compared to the original TDMs. This assessment is performed for the introduced motivation example, as well as two case studies by checking different anti-patterns.

6.1 Case study: FIXML to object model

The goal of the FIXML2Obj transformation is to transform FIXML *models* to the *object models*. FIX is the Financial Information eXchange protocol for transmitting pre-trade and trade communication messages between brokers and asset managers in the global equity market [42]. FIXML models are XML models for specifying the financial transaction messages and financial information exchange data. On the other hand, an object model is a logical interface that is modeled through the use of object-oriented techniques. An object model is normally specified as class definition in an object-oriented programming language, such as Java, C#, or C++.

The FIXML2Obj transformation is an industrial project that was proposed in the transformation tool contest (TTC) in 2014.[5] It is an industrial use case of the application model-driven development (MDD) in the financial field. The aim of the project is to enable the rapid upgrade of the user software when FIXML definitions are upgraded or modified [43].

Several transformations have been proposed to address the 2014's TTC. Among them, the winner transformation [44] was the one developed by SIGMA [45]. SIGMA FIXML2Obj transformation consists of three distinct stages; namely, text-

---

Fig. 14 Partial low-level TDM of FIXML2Obj

```
platform:/resource/evaluation-fixml2obj/transformation/Fixml2obj.mm_hybrid
  Module Fixml2xml
    Rule Node2Model (XMLNode::Fixml_model -> Model::obj_model)
      Source XMLNode::Fixml_model
      Target Model::obj_model
    Rule node2obj (XMLNode::Fixml_model -> objs:Set::obj_model)
      Source XMLNode::Fixml_model
      Target objs:Set::obj_model
    Rule At2Obj (XMLAttribute::Fixml_model -> objs :Set::obj_model)
      Source XMLAttribute::Fixml_model
      Target objs :Set::obj_model
    IN MetaModel Fixml_model
    OUT MetaModel obj_model
    Operation  NodeToObject (XMLNode) : objs:Set
      Return objs:Set
    Operation  AttributeToObject (XMLAttribute) : objs :Set
      Return objs :Set
```

```
Selection  Parent  List  Tree  Table  Tree with Columns
```

```
Problems    Properties      Error Log    Console
```

Property	Value
Body	var c = obj!Clazz.all().select(c | c.name == self.tag).first(); if (c == null) { c = new object
Context	Source XMLNode::Fixml_model
Name operation	NodeToObject
Right Pattern	Right Pattern R5_rightPattern ()

to-model (T2M), model-to-model (M2M), and model-to-text (M2T). First, the FIXML messages are parsed and transformed into an XML model that conforms to the XML meta-model. Second, the generated XML model is transformed into an object model. Third, the object model is transformed into the corresponding object-oriented language source code. (i.e., Java, C#, or C++) [42].

Our case study uses the TDM used in the second stage of SIGMAs solution. However, since the current implementation of MUPPIT extends the MeTAGeM and transML frameworks, the TDM was first migrated to these frameworks.

6.2 Case study: Ecore models to relational schema (Ecore2Schema)

Transforming Ecore models to relational schema is a well-known transformation case. We have employed a simple case of this mapping as a motivation example in the current study. In this section, we validate MUPPIT using a more intricate version of this mapping, Ecore2Schema, which covers a large part of Ecore metamodel.[6] Ecore2Schema is defined using both Ecore metamodel and relational schema meta models [46].

The solution for transforming Ecore2Schema is derived from the QVT solutions presented by Westfechtel [46] for Ecore models with a single package and inheritance. In Ecore2Schema, a package in Ecore model is mapped to a schema which contains tables. Classes and attributes are transformed into tables and columns similar to UML2DB,

while in Ecore2Schema, references, dependencies, containments, and relations between classes are managed by transforming them to columns and foreign keys in the corresponding tables. Moreover, cross-reference classes generate tables. Ecore2Schema covers transformation of inherited classes and sub-classes by mapping them to the foreign keys in root and sub-tables. The relational schema has Property and Event concepts for specifying the dynamic behavior of tables and columns. Properties define features on columns and events define conditions and actions for the foreign keys. The Ecore metamodel contains behavioral properties of classes and specifies them by EOperational classes. However, all concepts of the Ecore metamodel cannot be mapped to the target schema, for example, interface classes or EOperational classes.

In this paper, we generated a TDM for Ecore2Schema, which conforms to the solution presented by Westfechtel [46] in MeTAGeM. We also applied the MUPPIT process to this TDM to validate the process.
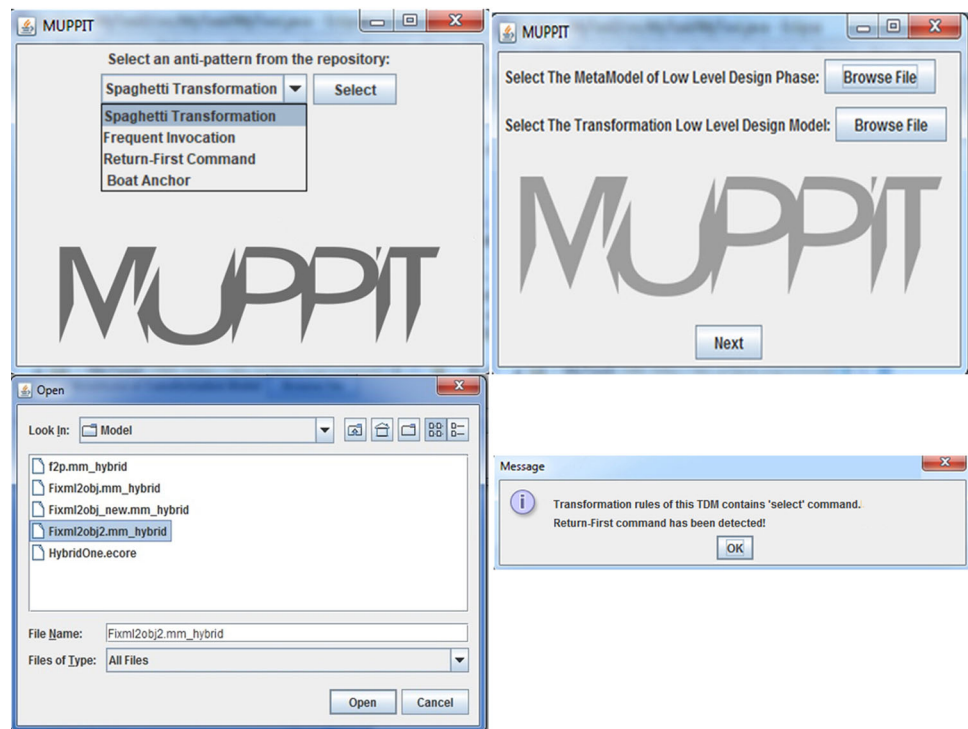
6.3 Walkthrough example: applying MUPPIT to the FIXML2Obj TDM

In this walkthrough example, we use the FIXML2Obj TDM to demonstrate the ability of MUPPIT in detecting transformations anti-patterns in a TDM, then restructuring the TDM to generate a new one that applies the pattern solution.

As explained earlier, to apply MUPPIT, the transformation engineer starts by selecting an anti-pattern from the anti-pattern repository. This can be done arbitrarily in an iterative way or based on evaluating the quality of the TDM. Let us assume that the selected anti-pattern is the *Return-*

---

[6] https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore/model/Ecore.ecore.

Fig. 15 MUPPIT windows
during the "identifying
transformation anti-pattern"
phase for FIXML2Obj

First Command anti-pattern. MUPPIT will prompt the user for the required inputs to detect the anti-pattern in the TDM. Since this anti-pattern requires checking if the TDM uses the appropriate operations, i.e., it is an operational anti-pattern, the FIXML2Obj low-level TDM is required. FIXL2Obj TDM, low-level design meta-model along with transformation source and target meta-models are inputs taken by MUPPIT. Figure 23 shows the meta-model of the low-level design phase, while Fig. 14 shows part of the FIXML2Obj low-level TDM generated using the MeTAGeM framework. Figure 14 consists of two parts: the top part shows the low-level TDM (i.e., the design of the FIXML2Obj). The bottom part displays the properties of the highlighted element (i.e., the NodeToObject operation).

Once the required inputs are uploaded, MUPPIT proceeds to the anti-pattern detection step. Here MUPPIT explores the TDM for *Return-First Command* anti-pattern. FIXML2Obj contains a select iterator in the NodeToObject operation, followed by first (as shown in the first row of the properties table in Fig. 14); hence, MUPPIT detects the *Return-First Command* anti-pattern and prints the result in a message box. Figure 15 shows the screenshots of different steps of the first phase of MUPPIT from selecting an anti-pattern to selecting the required inputs and finally printing the message that shows the identification of the anti-pattern in the TDM.

Once the message is confirmed, MUPPIT proceeds to the next second phase. Based on the pattern catalogue, the corresponding solution to the *Return-First Command* anti-pattern is the *Usage of Iterators* pattern. Accordingly, MUPPIT searches the FIXML TDM for *Usage of Iterators* pattern solution. In this case, the FIXML2Obj TDM exhibits the anti-pattern; however, the solution pattern was not detected. Accordingly, the pattern solution "*Usage of Iterators*" is proposed to the transformation engineer and proceeds to the third phase. Figure 16 shows the list of proposed patterns for FIXM2Obj TDM. As it is shown, MUPPIT also proposes alternative patterns with the same concern. In fact MUPPIT has been designed to propose additional solutions if available, but the current implementation support it only for *Return-First Command*, in which the 'select' command can be restructured based on two different solutions of *Usage of Iterators* and *Object Indexing*.

In this case, the user selects the pattern solution and apply it to the TDM. Figure 17 shows the result of applying the *Usage of Iterators* pattern solution. The first message shows the path of the new generated TDM, while the second message shows the result of evaluating the quality attributes of the new TDM.

Figure 18 shows newly generated TDM. In new TDM, the *select().first* command was replaced with the *any* expression. Using *any* instead of the *select* reduces the execution time for running the transformation; hence improves the transformation performance.

As illustrated through this walkthrough example, the new FIXML2Obj was automatically generated and evaluated. After evaluating the new TDM, MUPPIT will proceed to the first phase if it has any new suggested solution. The pro-
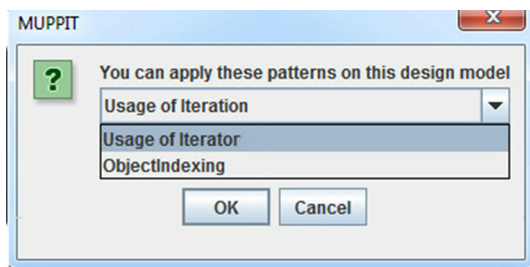
Fig. 16 A MUPPIT screenshot during the "proposing transformation pattern solution" pattern-solution

Fig. 17 **a** Result of applying *Usage of Iterators* on the FIXML2Obj- **b** Measured metrics for newly generated FIXML transformation

cess will continue iteratively until no more refinements are possible.

6.4 Quantitative evaluation of the MUPPIT

In this section, the effectiveness of MUPPIT in terms of the quality of generated TDMs is quantitatively assessed. First, several quality metrics are introduced. Then, the metrics are used to compare the quality of the original TDM in comparison to the one generated after applying MUPPIT.

6.4.1 Metrics definition

The following are the metrics used in the evaluation plug-in to assess the quality of a TDM. These metrics are widely used in the literature to assess the quality of a transformation model.

Syntactic complexity (SC) [13,16,39] This metrics is calculated by summing the number of entity type references, feature references, and operator occurrences in a transformation rule. This metric is adopted from Kolahdouz-Rahimis work [39]. The less the SC of a transformation rule, the simpler it is, which leads to better comprehension and hence maintainability.

Maximum subexpression complexity (MSC) This metric measures the SC of the most complex read subexpression in the transformation rules. It is known in the literature as the *maximally complex read subexpression* measure [13,16]. Similar to SC, this metric can be used as an indication of the complexity of a transformation. A transformation with lower MSC is less complex and more comprehensible.

Number of create actions per rule (CAPR) This metric measures the number of distinct element creation actions within a transformation rule. It is known in the literature as the *multiple creation* measure [13,16]. A transformation rule describes how a fragment of the source model can be transformed into a fragment of the target model. The rule of thumb is that each transformation rule creates one specific target element. However, in complex transformations, a transformation engineer may write a rule with several creation actions in the same rule. CAPR keeps track of the
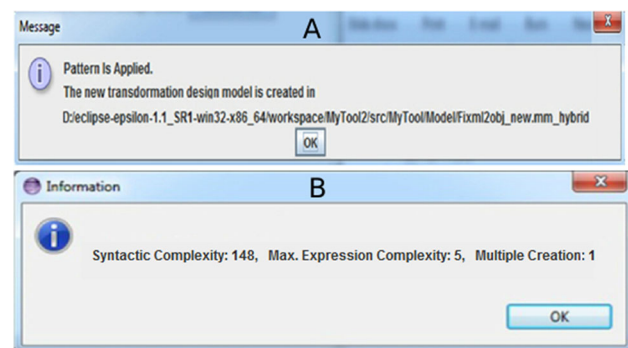
number of creation actions. A good TDM is the one where most rules have a CAPR value of one.

Modularity Modularity refers to the degree to which a system's components (i.e., transformation rules in a TDM) may be decomposed or recombined. Modularity is one of the widely used metrics in measuring the quality of a TDM [13, 16,39,42] as is directly related to flexibility, ease of change and reusability. Modularity is measured using the following equation as defined by Lano et al. [43] or Hoyos et al. [42]: $m = 1 - (d/r)$, where "m" is the Modularity of a TDM; "d" is the number of dependencies between the transformation rules (a.k.a., mappings). A dependency can be any of the following: implicit or explicit call, ordering dependency, inheritance or any kind of access control, and data dependency; and "r" is the number of rules in a TDM.

Execution time The average execution time for running a transformation code a number of times. In this paper, each transformation has been executed ten times.

Resource usage The amount of resource which a transformation needs to be executed. In this paper, we have considered the size of TDM as an indicator of memory usage.

In the next subsection, we show how these metrics have been used to evaluate the quality of the output TDM.

6.4.2 Evaluation plug-in

To collect the metrics introduced in Sect. 6.4.1, we implemented an evaluation eclipse plug-in, an enhanced version of our previous work [17]. The plug-in can be used as a standalone component to evaluate any TDM source code (in ATL and ETL) against the proposed metrics. In this work, we use the plug-in two different ways: (i) to evaluate the TDM code before and after applying MUPPIT and (ii) to guide the process of prioritizing the search for anti-patterns in a TDM (i.e., which anti-pattern to evaluate first).

The evaluation Java plug-in measures SC, CAPR, and MSC metrics by analyzing the transformation code, generated from a TDM. It takes the transformation code as a string

Fig. 18 New low-level TDM of FIXML2Obj
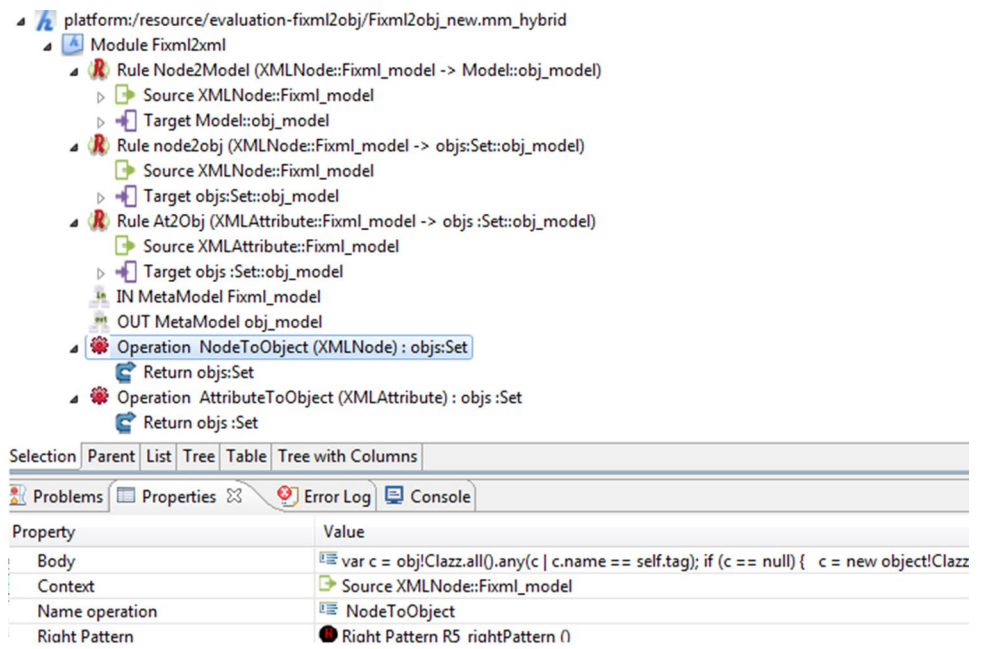
and checks it against Java regular expressions to find matches. The regular expressions are defined as search patterns, which find SC, CAPR, and MSC indicators based on their definition, and count them. For instance, the regular expression for measuring SC is a search pattern which finds and counts the number of references to the element types or features, and invoking operations in the transformation code. Referring to the types or features can be found by a pattern such as the one presented in line 1 in Listing 5. Operator occurrence is defined as matching rules for each operator based on the transformation language. In this paper, ETL and ATL syntax have been checked and the list of their operations has been extracted to be used in search pattern strings for finding the called operation. Listing 5 shows a partial code on how the occurring of each syntactic complexity indicator increases the SC measure.

```
1   syncComplexity+=string.
2       split(".",-1).length-1
3   syncComplexity+=printer.
4       split("<>",-1).length-1;
5   syncComplexity+=printer.
6       split("=",-1).length-1;
7   syncComplexity+=printer.
8       split(".allInstancesFrom\\(",-1).length-1;
9   syncComplexity+=printer.
10      split(".toString\\(",-1).length-1;
11  syncComplexity+=printer.
12      split(".oclType\\(",-1).length-1;
13  .....
```

Listing 5 Search pattern expressions in counting SC

To evaluate the effectiveness of using MUPPIT, the first step is to generate the transformation code from the input (original) and output (after applying MUPPIT) TDMs, which is generated using MeTAGeM. Then, the transformation codes of the input and output TDMs are evaluated using the proposed metrics. Particularly, for any ATL or ETL transformation code, the plug-in automatically measures the values

for the SC, CAPR, MSC. Moreover, the ATL profiler is used for measuring the execution time of the transformation.

In addition, we compare the metrics obtained from to the new TDM (i.e., after applying the pattern solution) with predefined thresholds that are provided by transformation engineers. This extra step (though optional) helps transformation engineers assess the benefits of applying the pattern solution proposed by MUPPIT. It may happen that the solution incurs overhead that a transformation engineer did not anticipate in which case he or she can select another pattern solution, if available. In other words, MUPPIT supports some sort of a feedback loop, which can be useful if multiple pattern solutions are considered.

Table 3 summarizes the possible impacts of applying each pattern solution on the metric values when the corresponding anti-pattern is identified in a TDM.

In this section, the effectiveness of MUPPIT in terms of the quality of the generated TDM is assessed using three presented scenarios in this paper, UML2DB, FIXML2Obj, and Ecore2Schema. Table 4 shows characteristics of the transformations and input models used for evaluation. The second column presents the number of transformation rules in the original case study. Average, minimum and maximum number of involved elements in each rule by considering elements in the called methods are shown in the third column. "Not Applicable" in the table means that our evaluation is independent of the corresponding characteristic. Measuring the number of executions is not applicable for Ecore2Schema case study since indicators of the the tested scenario in this case are not execution-based metrics.

Table 3 Relation of pattern solutions and introduced performance metrics

Pattern solution	Main concerns (improves)	Positive effects on	Negative effects on
Phased Construction	Modularity, CAPR	Execution time	SC
Object indexing	SC, Execution time	Modularity	–
Usage of Iterators	Execution time	SC	–
Filtering	Resources usage (Size of the source model)	Memory usage (it might reduce other resources as well)	–

Table 4 Characteristics of the evaluation scenarios

Case study	Transformation size (LOC)	Number of transformation rules	AVG/min/max of involved elements in transformation rules	Transformation source model size (#element)	Number of executions
UML2DB	18	2 rules and 1 function	3.5/3/4	100 elements	10 times
FIXML2Obj	49	3 rules and 4 functions	2.6/2/3	10,000 elements, 15,000 elements	10 times
Ecore2Schema	89	12 rules and 3 functions	8.75/2/12	55 elements	Not applicable

UML2DB

As explained earlier, the evaluation plug-in works on the source code of the TDM. MeTAGeM was used to generate the ATL code from the input UML2DB TDM. Listing 6 show pseudo-code of the UML2DB transformations, before and after using MUPPIT. Particularly, two pattern solutions were applied to the source TDM; namely, the *Phased Construction* and *Object Indexing* patterns. Listing 7, shows that applying the *Phased Construction* pattern solution divided the *Class-Map* into two rules. Moreover, the *select* command was replaced by the map structure using the *Object Indexing* pattern.

```
1    Class-Map:
2    for each c : Class Create t :
3        Table satisfying t.name = C.name
4    and Column!exists (k | k.name =
5        c.name + " Key" and k : t .column)
6    Attribute-Map:
7    for each c : Class; a : c.attribute;
8    t : Table.allInstances()!select
9        (table|table.name = c.name)
10   Create k : Column satisfying k.name =
11       a.name and k : t.column
```

Listing 6 Pseudo-code of the UML2DB TDM before using MUPPIT

```
1    Class-Map:
2    for each c : Class Create t :
3    Table satisfying t.name = c.name
4    Class-MapII :
5    for each c : Class Create k :
6    Column satisfying k.name = c.name + "Key" and
7    k : Table[c.name].column
8    Attribute-Map:
9    for each c : Class; a : c.attribute and
10   Create k : Column satisfying k.name =
11       a.name and k: Table[c.name].column
```

Listing 7 Pseudo-code of the UML2DB TDM after using MUPPIT

As shown in Listing 6, the *Class-Map* rule creates two elements. Accordingly, the number of create actions for that rule before applying MUPPIT (CAPR metric) is two. The CAPR metric was reduced to one after applying the *Phased Construction* pattern, as shown in Listing 7. However, while the *Phased Construction* pattern solution reduces the CAPR value, it does not improve the "SC" in total. After applying the *Phased Construction* pattern MUPPIT suggested applying the *Object Indexing* pattern to reduce the overall SC of the Transformation. Table 5 shows a summary of the measured performance metrics for the UML2DB case study before and after the MUPPIT.

As it is shown in Table 5, the overall SC and read subexpression complexity (MSC) have been decreased after applying both of the proposed pattern solutions. Consequently, the modularity and reusability are increased. Moreover, the execution time of the transformation was improved. Generally speaking, we can conclude that applying MUPPIT improved the quality of the UML2DB transformation. The execution time was measured in an average of 10 times execution for a UML source model containing 100 elements.

FIXML2Obj

In this section, we evaluate the quality of the FIXML2Obj TDM, before and after applying MUPPIT. Recall that in this case MUPPIT suggested and applied the *Usage of Iterators* pattern solution to improve the quality of the TDM. Table 6 shows the measured metrics for the FIXML2Obj before and after applying MUPPIT. The metrics show that applying the *Usage of Iterators* solution on the TDM significantly reduces the execution time of the transformation. In this case, the improvement was more than three folds. On the other hand, the *Usage of Iterators* has slight positive impact on SC and no impact on other metrics. The execution time was measured in average of 10 times execution for FIXML2Obj source models containing 10,000 and 15,000 elements.

Ecore2Schema

This section addresses the evaluation of MUPPIT for Ecore2Schema case. The explored anti-pattern in Ecore2Schema is *Boat Anchor*. As mentioned in Sect. 6.2, the Ecore metamodel, as the transformation source meta-model of Ecore2Schema, has some classifiers such as EOperation or some kinds of classes such as interface and abstract. These elements are popular in a UML model generated from Ecore meta-model, but they are not mapped to a corresponding element in the target schema model generated by Ecore2Schema. This scenario reveals the presence of *Boat Anchor* anti-pattern. In this case study, we use MUPPIT to search the Ecore2Schema TDM against the *Boat Anchor* anti-pattern. Figure 19 shows the high-level TDM of Ecore2Schema generated in MeTAGeM. This model maps source elements of the source meta-model to target ones, called weaving model as well. The transformation of Ecore2Schema is established based on this TDM. Then, a transformation source model generated from the Ecore meta-model is taken by Ecore2Schema and a target relational schema model is generated. We have used a UML class diagram in the domain of campus management as source model of Ecore2Schema transformation. Figure 20, left side, presents this source model. MUPPIT compares this TDM of Ecore2Schema with the source model of this transformation, presented in Fig. 19, and finds if there is any unused elements in the source model. In this case, operations in the source are not employed in transformation mappings. Therefore, MUPPIT identifies *Boat Anchor* and proposes *Filtering* as a solution to the transformation engineer. By applying the solution, the source model of Ecore2Schema is changed to the filtered version. Figure 20, right side, shows the source model of campus management after *Filtering*. In exploring this anti-pattern, *Boat Anchor*, MUPPIT seeks source model of a transformation beside of other MUPPIT inputs

Table 5 Evaluation of UML2DB before and after using MUPPIT

Transformation	SC	MSC	CAPR	Modularity	Execution time (ms)
Before applying patterns	41	10	2	0.5	237
After applying patterns in MUPPIT	31	7	1	0.7	157

Table 6 Evaluation of FIXML2Obj before and after using the MUPPIT

Transformation	SC	MSC	CAPR	Modularity	Execution time (ms)
Before applying patterns	150	5	1	1	529
After applying patterns in MUPPIT	148	5	1	1	147

and applying the MUPPIT changes this model, as shown in Fig. 20. In the practice, MUPPIT does this change by generating a new transformation source model which do not contain unused element in the transformation. Therefore, MUPPIT performance in proposing and applying the *Filtering* solution can be measured by measuring the metrics on source model of the transformation before and after applying MUPPIT, rather than evaluating TDMs. As Table 3 shows, *Filtering* pattern is a solution for reducing the size of source models. In this case, source model size was decreased almost 30% over 7374 bytes as the size of the original source model of Ecore2Schema.

Overall, this evaluation proves again the effectiveness of MUPPIT in terms of its ability to generate new TDMs or transformation environment with higher quality based on applying best practices.

7 Threats to validity

In this section, we discuss the threats to the validity of the MUPPIT approach.

7.1 Threats to external validity

MUPPIT relies on transML and MeTAGeM, which limits the application of MUPPIT to exogenous transformations using rule-based transformation and OCL-based languages such as ATL and ETL. We need to support other transformation engineering frameworks to enable the generalizability of MUPPIT. This said, Bollati et al. [6] compared various transformation frameworks and concluded that transML and MeTAGeM are the most mature in terms of their support to the transformation development cycle. Another threat to external validity lies in the fact that MUPPIT has been tested on only four anti-patterns. We need to apply MUPPIT to detect more anti-patterns in complex transformation scenarios. Finally, for each anti-pattern, we should experiment with multiple source models for some of the metrics that we presented such as the execution time metric.

7.2 Threats to internal validity

To decide when to apply a pattern solution, a transformation engineer needs to provide thresholds to which we compare the collected metrics. These thresholds may vary depending on the experience of the transformation engineers, which may lead to different results. We mitigate this threat by adding an extra step in which the transformation engineer checks the resulting transformation before deciding to apply it. Another threat to internal validity lies in our extension of the Epsilon compiler to implement the *Object Indexing* pattern. Implementation errors may have occurred. To mitigate this threat, we tested our code many times to reduce the chances of programming errors.

Possible effects achieved by the suggested pattern solutions might be affected by different factors, such as size of the transformation source model. *Usage of iterator* is a pattern solution for big models. In fact, this solution performs well in decreasing the execution time, when it is applied on big source models. Although we have not considered small models in our evaluation for the Usage of Iterator scenario, two model sizes with different ranges of impacts were considered. Both model sizes with 10,000 and 15,000 elements were selected according to Cuadrado et al. [12] study. Cuadrado et al. concluded that models in these size ranges show, respectively, low and appreciable change in reduced execution time by applying *Usage of Iterator*. Therefore, we tried to make our experiment reliable by measuring the execution time changes in two different bands of model size with different impact and reporting the result on average.

Moreover, the order of applying the pattern solutions can threaten the result of the pattern process. In fact, change in the order of applying the pattern solutions can result in different final transformations. In the current work, the order of execution is determined by the developer, who selects one anti-pattern at a time. However, we are planning to tackle this issue as part of our future research.
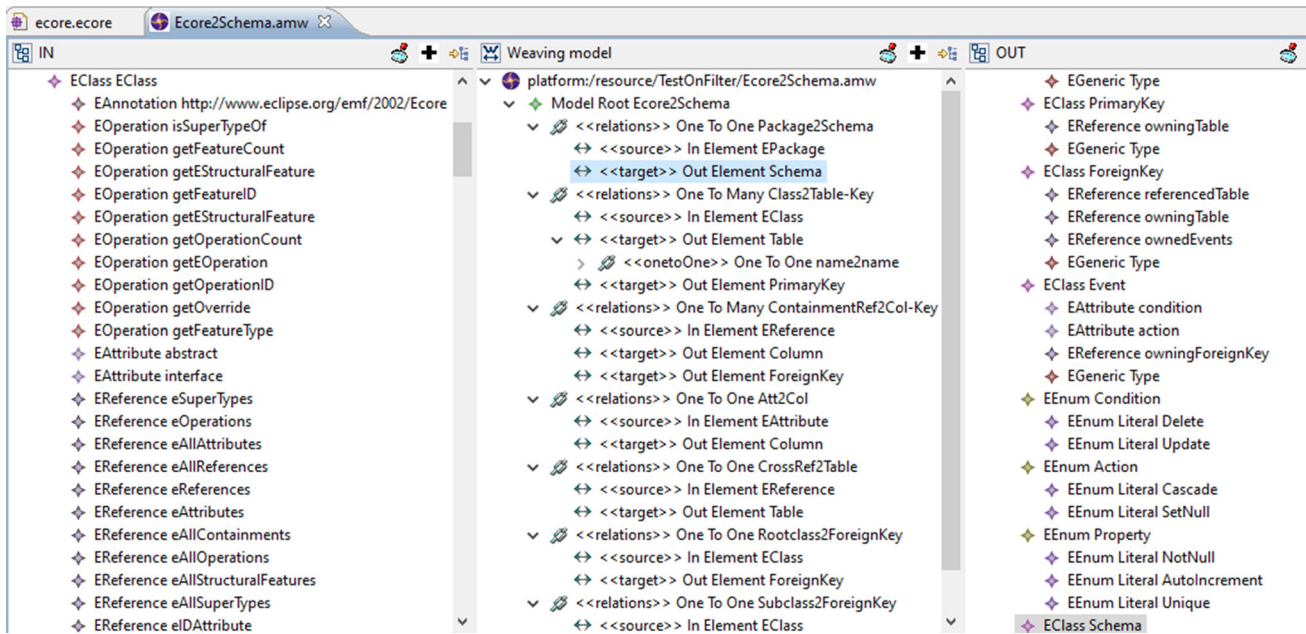
Fig. 19 Platform-independent TDM, low-level TDM, of Ecore2Schema generated in MeTAGeM
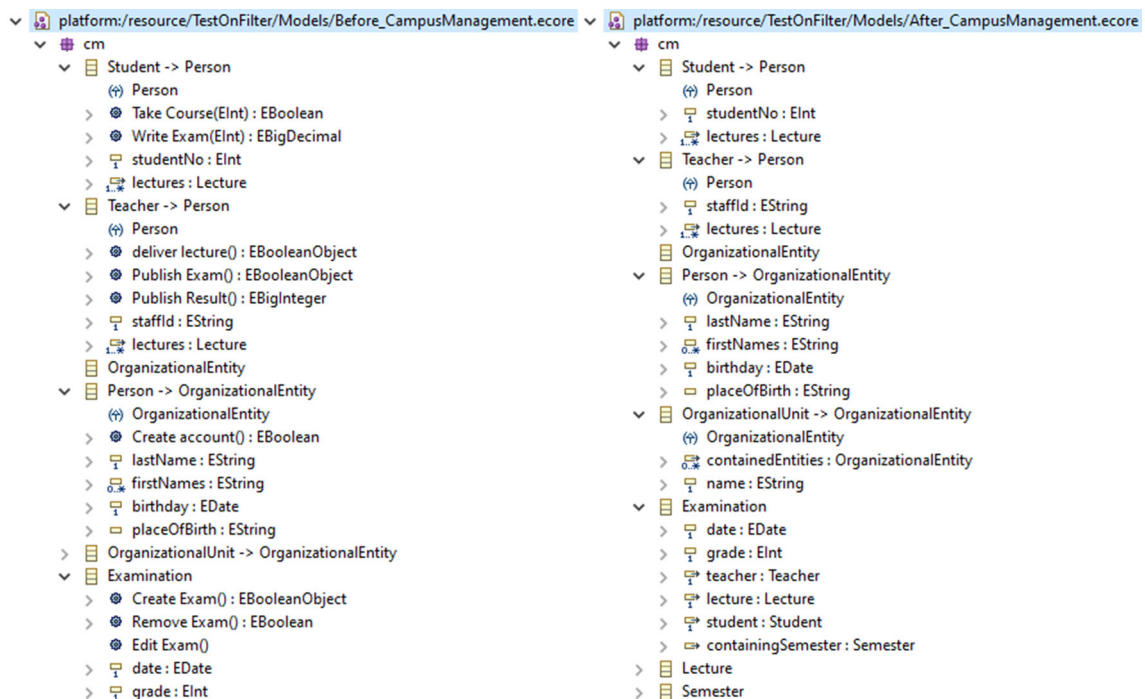

Fig. 20 Transformation source model of Ecore2Schema before and after using MUPPIT

8 Related work

There have been several efforts in the model-driven engineering community to identify and formalize transformation patterns. In this section, we address the works that are close to our work and try to compare them with ours.

Ergin et al. [47] presented a template for transformation pattern description similar to what is known for software patterns. Lano et al. [48,49] reviewed the most common transformation patterns, their benefits, trends, applications, and languages. They created a catalogue of pattern solutions and provided guidelines on how to detect them manually.

Inspired by the Gang of Four (GoF) design patterns [14] for object-oriented programming, Bezivin et al. [9] put together a collection of MDE patterns related to the design of meta-models, transformations, compositions, and other model-based operations. In their initial work, they recommend two patterns to implement high-quality transformations using the ATL[7] transformation language. The first pattern explains the case where some auxiliary information is needed (i.e., additional input meta-model that contains auxiliary variables of a transformation), while the second pattern is used in the implementation of transformations and represents the case of multiple matching problem, a problem that has been already solved in the later versions of ATL. In order to present more functional patterns, Cuadrado et al. [12] present some patterns for using the object constraint language (OCL) properly. They document five patterns to optimize the performance of model transformations. While the benefits of these patterns are evaluated with several benchmarks, these are introduced in the context of the ATL language. Similarly, Iacob et al. [11] proposed six model transformation design patterns. These patterns are related to some recurring problems in the QVT[8] transformation language. In the context of graph transformation languages, Agrawal et al. [10] presented a simple graph transformation language, called GREAT, to solve graph transformation problems the same way that software problems are solved by using design patterns. Two patterns were created to access the graph objects. These patterns are specific to the problems that arise in the context of graph transformation models.

All the aforementioned studies are limited to identifying and defining transformation patterns for a specific scenario in a specific transformation language. While these studies represent a great body of knowledge in transformation patterns that is necessary when creating transformation pattern catalogues, these studies do not address the concern of how to integrate these patterns as part of the development process, which includes how to select or use these transformation patterns properly and how to automatically generate high quality transformations based on these patterns. While there have been several efforts in the software engineering community to utilize transformation patterns as part of the transformation development process [50,51], the sheer volume of the research focused on a specific activity of the pattern-process and tried to optimize it. MUPPIT, on the other hand, is trying to fill this gap by providing a complete pattern-process that covers anti-pattern identification, pattern proposition, pattern application (generating a TDM based on the pattern), and TDM evaluation.

Perhaps the most related work to MUPPIT is the work by Ergin et al. [7,18,20], Mokaddem et al. [8], Lano et

al. [13,16,19,52], Gabriele et al. [53], and Tichy et al. [54]. Table 7 summarizes the main contributions of MUPPIT in comparison to these related research projects. These projects address a way of more than introducing anti-patterns and solutions, i.e., they provide using these concepts. The comparison criteria are defined based on the activities defined in the MUPPIT process. In Table 7, the works appeared in order of publication date. The letter M in the table means manual, and letter A declares an automatic or semi-automatic method. The label "Inc" means that the idea is not supported completely in our study, and "Dep" identifies cases, which are dependent on other tools to proceed. In the following, we will introduce each work in detail.

One of the most rigorous research in this field is conducted by Lano and Rahimi [13]. They presented several patterns, collected and classified into classes, for model transformations [55], then documented them in a pattern catalogue containing 29 patterns. In pattern application, they improved their UML-RSDS framework [56] by integrating that with the transformation patterns [23,55,57], which finally resulted in a tool support to use patterns in a proper way [52]. The UML-RSDS framework is considered as one of the most comprehensive works in this domain. In UML-RSDS, developers define a transformation based on patterns that are specified using a formal-mathematical language. As shown in Table 7, UML-RSDS project has been developed through several iterations. While in the early versions of UML-RSDS, choosing the proper pattern that corresponds to a specific problem was the responsibility of the developer and automatic model generation was not supported, later, Lano and Rahimi [13] defined metrics to measure the model transformation complexity to guide the process of selecting and applying design patterns [16]. Then, they defined a meta-model-based language [13] for pattern specification based on the UML meta-models and formal methods which provides a heuristic pattern selection approach based on their previous research [16]. Moreover, they added pattern verification and a synthesis process to UML-RSDS as well, to generate design and implementation model of a transformation from its specification, automatically [19]. Using formal specification for specifying patterns makes it difficult for the average developer to use the tool.

Ergin et al. [7,18,20] presented Delta, a domain-specific and agnostic specification language for transformation patterns. The first version of Delta [18] supports five transformation languages and four transformation patterns, in which patterns are limited to graph transformations and need to be selected by the developer and be applied manually during the design phase. In the second version of Delta, Ergin et al. [7,20] have improved their semi-formal language and made it extensible to support adding other patterns. They added support for 15 patterns, 14 extracted from literature and one introduced by themselves, and provided a tool to

---

Table 7 Comparison of related works on transformation patterns (M: Manual, A: (Semi-)Automatic, Inc: Incomplete, Dep: Dependent)

Criteria	UML-RSDS Lano et al. [52]	Gabriele et al. [53]	Tichy et al. [54]	Lano et al. [19]	Lano et al. [13]	UML-RSDS Lano et al. [19]	Ergin et al. [18]	Ergin [7,20]	Mokaddem et al. [8]	MUPPIT
Identifying transformation anti-pattern	–	M	–	–	–	–	–	–	–	+(A)
Pattern solution detection	–	–	–	–	–	–	–	–	+(A)	+(A)
Accuracy of detected pattern solution	–	–	–	–	–	–	–	–	+(A)	-
Pattern solution proposition	–	–	–	+(M)	+(M)	+(A)	–	–	–	+(A)
Recognizing alternative patterns	–	–	–	–	–	–	–	–	–	+(A, Inc)
Applying pattern solution	+(M)	+(A)	+(A)	+(M)	+(M)	+(A)	+(M)	+(A)	–	+(A)
Tracking effective metrics related to each pattern solution (providing feedback)	–	+(M)	–	+(M)	+(M)	+(M)	–	–	–	+(A)
Presenting a specific pattern usage process	–	–	–	–	–	–	–	+	+	+
Automatic generation of transformation code from the output of the method	–	+	+	–	–	+	–	+	+	+(Dep)
Independent of transformation Language	–	–	–	–	–	–	+	+	+	–

help developers to generate model transformation excerpts automatically. Using Delta, a developer selects a pattern and then customizes it according to the problem. Next, according to the selected transformation language, a model transformation excerpt, which describes the pattern, is generated. The work done by Ergin et al. [7,20] is comprehensive; however, it does not cover aspects such as anti-pattern detection and pattern proposition.

Mokaddem et al. [8] proposed a pattern detection approach based on the patterns introduced by Ergin [7]. Their approach detects the partial and complete Delta patterns [7] in declarative model transformations. In addition to detecting patterns, the accuracy level of the detection process is measured and presented to the transformation developer. The detection process consists of four phases. First, the Delta patterns are encoded as rules to be applied to the transformations. Second, a transformation is specified at the abstraction level as a set of components. These two phases are implemented using the Delta language [7]. In the third phase, the approach explores which rule of a transformation or component can participate in the pattern rules. Finally, the execution flow of the participant rules will be checked if it satisfies the scheduling scheme in the pattern structure or not. Accordingly, the scheduling scheme will be generated. While Mokaddems research can be extended to cover more languages and patterns, the current implementation is limited to detect the control structure of patterns. Moreover, the approach does not cover activities such as anti-pattern detection, pattern application, and transformation evaluation.

In the domain of bad smells and anti-patterns, we must introduce the work of Tichy et al. [54]. They have studied several scenarios in the Henshin engine [58], each containing a problem in the Henshin rules and a corresponding detector, which represents the solution in the Henshin meta-model. Using their approach, a transformation engineer can generate Henshin rules from the Henshin meta-model such that the transformation rules do not contain the introduced problems. This work is dependent on the graph-based Henshin transformation rules. In addition to introducing the possible problems in Henshin transformations, the authors provide automatic conformance to pattern solutions. The authors used the terms bad smell and anti-pattern interchangeably. They defined bad smells as scenarios which "can negatively affect the performance of the application of model transformations." Hence, the introduced bad smells are close to the definition of anti-patterns in our paper. Their work differs from ours in the sense they do not identify anti-patterns by analyzing TDMs. Instead, they directly embed the pattern solutions into the Henshin engine.

Gabriele et al. [53] defined smells as quality improvement indicators that may take the form of metrics or patterns. They introduce the concept of quality smells as the basis for improving the quality of rule-based model-to-model endoge-

nous graph transformations. They defined many quality metrics such as conciseness, compatibility, and changeability, that they refer to as metric-based smells. Then, they introduced six scenario that affect these metrics as pattern-based smells. A pattern-based smell scenario mainly specifies a problem, affected metrics, and the solution. These scenarios are similar to the concept of anti-pattern/pattern solution in our work. According to the quality metrics that are considered in this work, the pattern-based smells were defined to explain problems related to the size and redundancy of transformations. In contrast to Tichy et al [54], Gabriele et al. [53] have described scenarios in a formal way. Their formal description contains detection and a refactoring scenario which are similar to identifying the transformation anti-patterns and applying the pattern-solutions in MUPPIT. The refactoring step in Gabriele et al. study considers semantic preservation of transformation rules as well. To integrate these refactoring scenarios to the Henshin engine, Gabriele et al. have defined some rules on top of the Henshin rules in the transformation model using EMF Refactor.[9] Therefore, applying the solution, or refactoring, is performed in an automatic way. This study provides the detection of pattern-based smells, or anti-patterns, manually by a detection description in scenarios.

Another research close to our work is the study by Wimmer et al. [41]. The authors provided a catalogue in a format suggested by Fowler [33] including 27 refactoring scenarios for model-to-model transformations. Each scenario describes a problem and a corresponding refactoring solution. The catalogue was extracted based on the available ATL transformations, but many of the problems can occur in other transformation languages as well. Wimmer et al.'s research is similar to MUPPIT in the sense that they also consider the detection of anti-patterns, as well as the use of pattern solutions to improve the transformation quality. Moreover, the evaluation has been performed similar to a part of MUPPIT transformation and by measuring metrics, including bad smell metrics. However, implementing this refectory catalogue is suggested as in-place transformation and shown in ATL language. This study cannot do automatic anti-pattern identification, pattern proposition, and solution application. To preserve the transformation behavior, authors have used the same approach that we used and compare transformations target models before and after refactoring by tools such as EMF Compare.

In a nutshell, the work presented in this paper (MUPPIT) is in continuation of our previous study [17] and is influenced by the works by Lano et al. [19] and Ergin et al. [7,20]. MUPPIT enables the transformation engineers to use high-level abstraction models based on transML or MeTAGeM specifications as opposed to formal methods. It provides a complete round trip for model transformation development from iden-

---

[9] https://www.eclipse.org/emf-refactor/.

tifying transformation anti-patterns to pattern application, TDM generation, and evaluation. MUPPIT has improved our previous work [17], in terms of the process, automation, and implementation to provide a process and a semi-automated pattern-based integration development tool for model transformations. In our previous work, the tool was able to propose the *Phased Construction and Auxiliary Model* patterns by checking the TDM structure against the pattern specifications. The current paper has improved our previous work by adapting the concept of anti-patterns, as possible weaknesses in the designed TDMs, and proposing a different iterative process which automates all the steps from identifying the anti-pattern to applying the solution pattern. The evaluation in the previous work was performed by fewer performance metrics and applied only to a simple case study. In the current paper, we have instantiated the introduced process with four scenarios and evaluated it using three different case studies.

9 Conclusions and future work

In this paper, we presented MUPPIT, a systematic process for identifying anti-patterns in model transformations and applying pattern solutions with the overall goal of being the quality of the transformations. MUPPIT is an Eclipse plug-in that extends the Epsilon language. It relies on two transformation engineering frameworks, transML and MeTAGeM, which support rule-based transformations. Moreover, the quality of the TDMs that are generated using MUPPIT was evaluated by comparing the TDMs before and after applying MUPPIT using various metrics including syntactical complexity, the number of create actions per rule, the maximum sub-expression complexity, modularity, and execution time. When applied to three cases studies involving four anti-patterns and their corresponding solutions, the results show that the TDMs generated by applying MUPPIT are more efficient, modular, and less complex than the original one.

One future direction is to extend MUPPIT to other transformation engineering framework to support other types of transformation. Regarding MUPPIT usability, MUPPIT requires to be fully integrated to the transformation engineering frameworks. The idea helps in automatically transmitting inputs and outputs between MUPPIT and frameworks. To this end, MUPPIT needs to be released as an executable plug-in inside of frameworks. In result, TDM and related meta-models, i.e., MUPPIT inputs, can be taken by MUPPIT instead of asking the developer. On the other hand, the generated TDM will be also automatically replaced with the original one in frameworks. We also need to experiment with more scenarios (anti-patterns and their corresponding patterns) to extend included anti-pattern and pattern scenarios as an large open-source catalogues. In addition, we need to define more indicators (i.e., bad smells) that we can use to

reveal the presence of anti-patterns in a more efficient way. Moreover, we the current version of MUPPIT detects the exact match of an anti-pattern. To support multiple variants of the same anti-pattern, we need to implement a matching mechanism that relies on similarity measures. Furthermore, we intend to work with transformation engineers to evaluate the usefulness of MUPPIT in practice. Finally, we need to investigate the use of other techniques for automatic detection of anti-patterns such as the use of tracing [59,60] and software debugging based on log analysis [61,62]. These dynamic analysis approaches are based on the analysis of the flow of execution (or simulation) of transformations and therefore have the potential to detect anti-patterns that are hard to profile through mere use of performance metrics.

Appendix

A Supplementary images

See Figs. 21, 22 and 23.

B Scheduling anti-pattern and pattern solutions

The anti-pattern identification and pattern application in this paper were implemented using Epsilon language. Therefore, the execution schema for each one follows Epsilon execution semantics, which can be found for both EPL[10] and ETL[11] on the Epsilon website. In the following we elaborate on the scheduling rules of anti-patterns and pattern solutions that are used in this paper. Each scheduling rule is a task performed by Epsilon statements in form of pre, post, transformation rule, do-blocks, and/or a function call. Our implementation is based on the iterative mode of Epsilon execution semantics, in which the anti-pattern identification is repeated until no more matches have been found in the TDM elements.

The *Spaghetti Transformation* anti-pattern starts with defining a global variable to keep the status of mapping elements in the TDM. Then, a rule iteratively checks all the mapping elements of a TDM against the high-level design meta-models. Each mapping that does not satisfy the constraints shown in Table 1 is added to a set, called patternSet which keeps mappings with *Spaghetti Transformation* scenario. Also, the global value is changed to show the presence of *Spaghetti Transformation* scenario in the TDM, i.e, it is changed from false to true. Finally, after matching all the mappings, the result is printed for the developer. In Epsilon, the result of an EPL code, i.e., anti-pattern identification, can

---

[10] https://www.eclipse.org/epsilon/doc/epl/.

[11] https://www.eclipse.org/epsilon/doc/etl/.

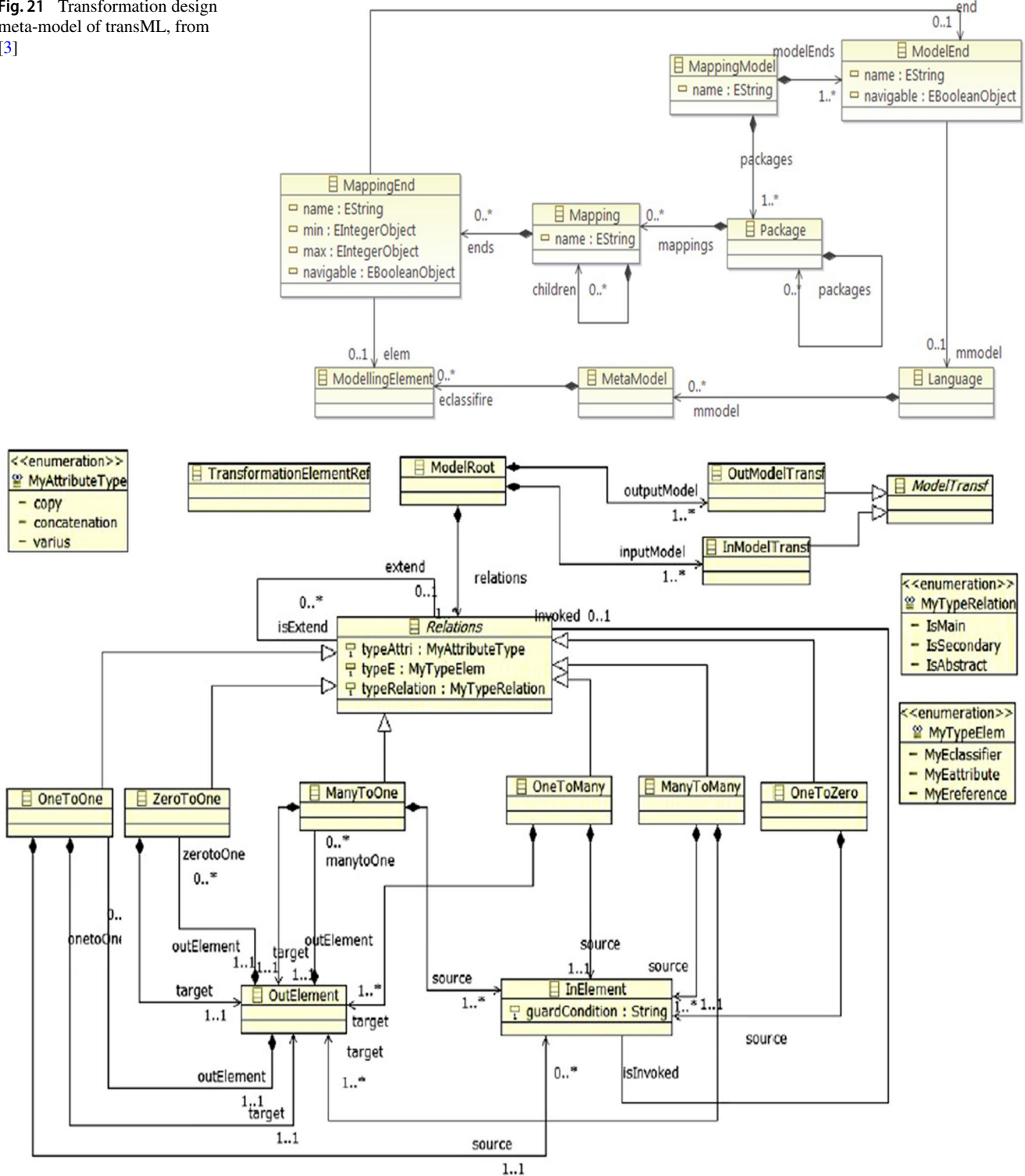**Fig. 21** Transformation design meta-model of transML, from [3]

Fig. 22 Transformation high-level design meta-model of MeTAGeM [6]
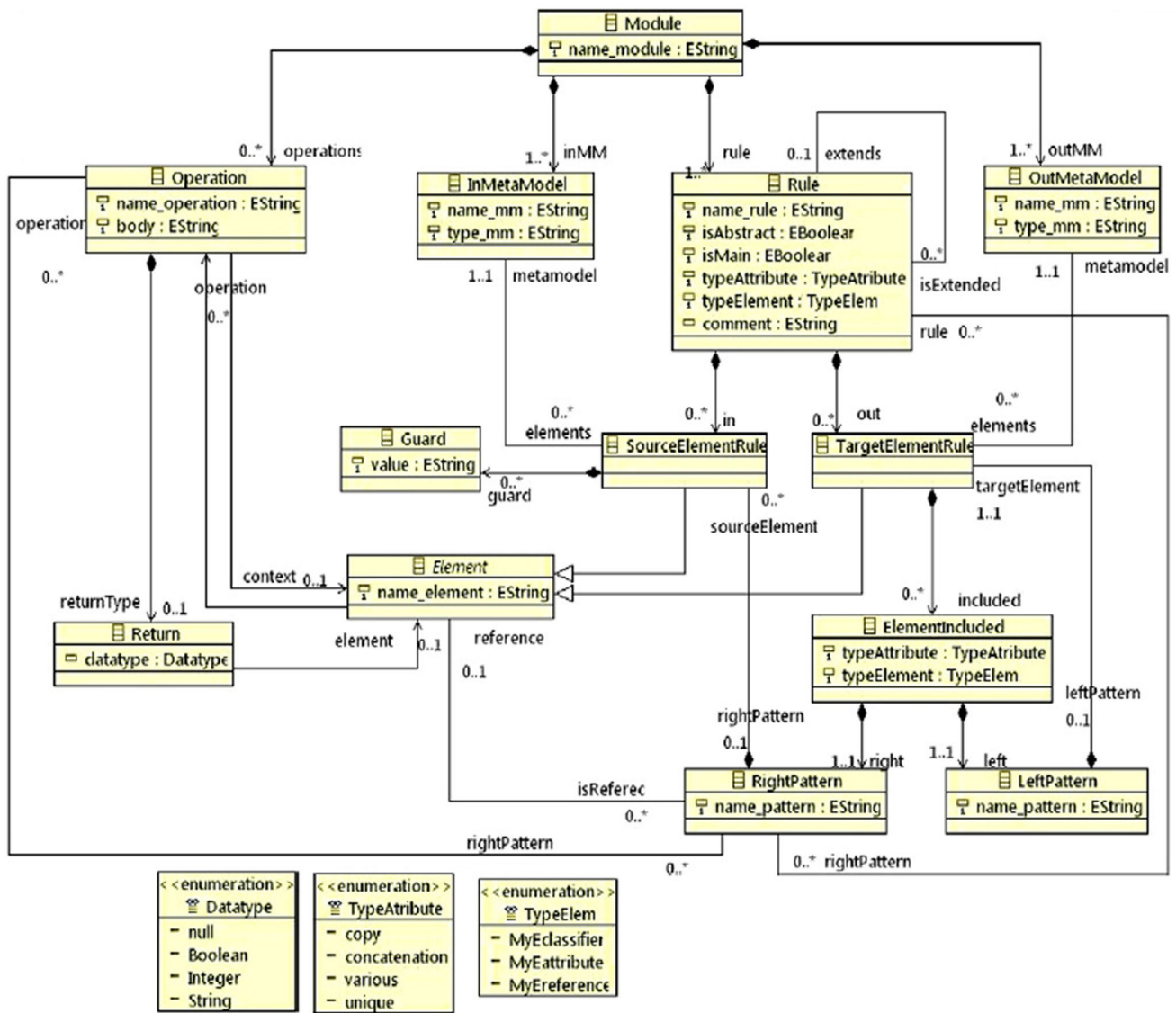
Fig. 23 Transformation low-level design meta-model of MeTAGeM [6]

be stored and transformed between the other languages of the Epsilon family using a set, patternSet.

Phased Construction An ETL code is executed to apply *Phased Construction* solution to the original input TDM. First, the patternSet, provided by identifying *spaghetti*, including the mappings with *Spaghetti* scenario is taken as an input. To keep the original TDM, a new TDM is generated and *Phased Construction* solution is applied to a new one. An initial rule generates independent elements on mapping concept in the TDM. Then, A rule recreates those mappings which are not a member of the patternSet as a copy of these elements in the original TDM. Moreover, features and dependencies of these mappings are set same as the original TDM. Then, each mapping in the patternSet, i.e., a mapping which needs to be restructured, is processed. A rule creates a new mapping

element for each mapping in the patternSet while the created element has the same features as the original mapping, but it only accesses one level of the target meta-model. In fact, for each mapping in the patternSet, the number of involved target elements are counted using the conditional statement in Table 2, and for each one, a new mapping is generated. The rest of the features and dependencies of the created mapping element are populated using the original mapping. It is noticeable that this paper employs the top-down approach for restructuring *spaghetti* mappings. However, mappings in patternSet are processed in order that they have been defined by the developer in the TDM. In both transML and MeTAGeM, considering the mappings order in TDM is a developer's task.

Frequent Invocation Regarding the TDM meta-model, each low-level TDM defines the transformation rule body

inside of the Operation elements. Identifying the *Frequent Invocation* is performed by a rule which for each transformation rule, in the order that they have been defined in the TDM, checks whether its Operation body contains the *select* command. If so, it puts the Operation element in a set, patternSet. If the patternSet contains any Operation, then the developer is informed of the presence of *Frequent Invocation*.

Object Indexing Scheduling schema of applying *Object Indexing* on TDMs starts with a rule which regenerates elements of the original TDM, except the Operation elements. Operation elements are generated if those are not a member of the patternSet. Otherwise, identified Operations with *Frequent Invocation* scenario, are differently scheduled. For each Operation in the patternSet, a rule of the schema extracts the context of the *select* command, instances of the invoked context, and invoked property by the *select* command. Then, a map structure is generated by the invoked property value as the index associated with the invoked instances of the context type. This map is stored and to be used instead of the *select* command. Then, the schema rule restructures the TDM by changing the Operation body and substitutes the *select* command by the generated map structure. The changed Operation is added to the related element of the TDM and dependencies are set same as the original TDM. Finally, schema is repeated to check the next Operation in the patternSet.

Return-First Command The scheduling schema for identifying the *Return-First Command* is same as the scheduling for *Frequent Invocation*. However, in this scenario, the scheduling rule checks the body of all Operations against the *select().first()* command. Then, if Operations contain the checked command, the scheduling rule puts them in the patternSet.

Usage of Iterators Applying the *Usage of Iterators* is similar to the *Object Indexing*. A scheduling rule regenerates all the elements of the original TDM. However, Operation elements or transformation rule bodies are regenerated if they are not included in the patternSet. Otherwise, a rule is scheduled to find the conditional statement of the *select* command in the Operation body. Then, inside the Operation element, the present *select(condition).first()* command is substituted with the *any(condition)* statement. Finally, the changed Operation is added to the related transformation rule in the new TDM and dependencies are set same as the original TDM. The schema is repeated for all Operation elements in the patternSet.

Boat Anchor A rule collects all element types of the transformation source model and puts them in a set called Source. Then, a secondary rule iteratively checks each member of the Source set against the TDM elements. If the Source member is not a member of the TDM elements, it will be added to the ExcludedSet. Finally, another rule checks the ExcludedSet

and informs the developer of the existing *Boat Anchor* if the set is not empty.

Filtering solution The ExcludedSet in identifying *Boat Anchor* is transmitted to the applying *Filtering* solution in patternSet. In this scenario, the transformation source model is restructured. A premier rule regenerates transformation source model elements except element types included in the patternSet, i.e., elements that need to be excluded from the transformation source model. Each element in patternSet, i.e., excluding element, is ignored by the premier rule if it does not have any dependent element. Otherwise, a secondary rule is called to remove the excluding element dependency in the model hierarchy. In fact, the secondary rule regenerates the excluding elements and its dependent element in a flat view and adds them to the generated model. Next, the execution schema is returned to the premier rule and it removes the excluding element. Then, the premier rule continues by checking and regenerating the rest of the transformation source model elements.

References

1. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5), 19–25 (2003)
2. Brown, A.W., Iyengar, S., Johnston, S.: A rational approach to model-driven development. IBM Syst. J. **45**(3), 463–480 (2006)
3. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: Engineering model transformations with transML. Softw. Syst. Model. (SoSyM) **12**(3), 555–577 (2013)
4. Bzivin, J., Farcet, N., Jezequel, J.-M., Langlois, B., Pollet, D.: Reflective model driven engineering. In: UML2003—The Unified Modeling Language. Modeling Languages and Applications, vol. 2863, pp. 175–189. Springer, Berlin Heidelberg (2003)
5. Kusel, A.: TROPIC-a framework for building reusable transformation components. In: Proceedings of the Doctoral Symposium at MODELS, pp. 22–27
6. Bollati, V.A., Vara, J.M., Jimenez, A., Marcos, E.: Applying MDE to the (semi-)automatic development of model transformations. Inform. Softw. Technol. **55**(4), 699–718 (2013)
7. Ergin, H., Syriani, E., Gray, J.: Design pattern oriented development of model transformations. Comput. Lang. Syst. Struct. **46**, 106–139 (2016)
8. Eddine Mokaddem, C., Sahraoui, H., Syriani, E.: Towards rule-based detection of design patterns in model transformations. In: System Analysis and Modeling. Technology-Specific Aspects of Models, pp. 211–225. Springer International Publishing
9. Bzivin, J., Jouault, F., Palies, J.: Towards model transformation design patterns. In: Proceedings of the 1th European Workshop on Model Transformations (EWMT 2005). ATLAS group (2005)
10. Agrawal, A., Vizhanyo, A., Kalmar, Z., Shi, F., Narayanan, A., Karsai, G.: Reusable idioms and patterns in graph transformation languages. Electron. Notes Theor. Comput. Sci. **127**(1), 181–192 (2005)
11. Iacob, M.-E., Steen, M.W.A., Heerink, L.: Reusable model transformation patterns. In: Enterprise Distributed Object Computing Conference Workshops, 2008 12th, pp. 1–10. IEEE (2008)
12. Cuadrado, J.S., Jouault, F., Garcia Molina, J., Bzivin, J.: Optimization Patterns for OCL-Based Model Transformations, vol. 5421, pp. 273–284. Springer, Berlin Heidelberg (2009)

13. Lano, K., Kolahdouz-Rahimi, S.: Model-transformation design patterns. IEEE Trans. Softw. Eng. **40**(12), 1224–1259 (2014)

14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education, London (1994)

15. Fowler, Martin: Analysis Patterns: Reusable Object Models. Addison-Wesley, Boston (1997)

16. Lano, K., Kolahdouz-Rahimi, S.: Optimising model-transformations using design patterns. In: International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 77–82. SciTePress (2013)

17. Panahandeh, M.S., Zamani, B.: Automatic pattern proposition in transformation life cycle. Int. J. Inf. Technol. Syst. Approach (IJITSA) **10**(2), 1–16 (2017)

18. Ergin, H., Syriani, E.: Towards a Language for Graph-Based Model Transformation Design Patterns, vol. 8568, pp. 91–105. Springer, Berlin (2014)

19. Lano, K., Kolahdouz-Rahimi, S., Poernomo, I., Terrell, J., Zschaler, S.: Correct-by-construction synthesis of model transformations using transformation patterns. Softw. Syst. Model. (SoSyM) **13**(2), 873–907 (2014)

20. Ergin, H.: Design Pattern Driven Development of Model Transformations. PhD Thesis, University of Alabama, Alabama, USA (2017)

21. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: transML: A Family of Languages to Model Model Transformations, vol. 6394, pp. 106–120. Springer, Berlin Heidelberg (2010)

22. Jiménez, Á., Vara, J.M., Bollati, V.A., Marcos, E.: MeTAGeM-Trace: improving trace generation in model transformation by leveraging the role of transformation models. Sci. Comput. Program. **98**, 3–27 (2015)

23. Lano, K., Kolahdouz-Rahimi, S.: Specification and verification of model transformations using UML-RSDS. In: International Conference on Integrated Formal Methods, pp. 199–214. Springer (2010)

24. Del Fabro, M.D., Valduriez, P.: Towards the efficient development of model transformations using model weaving and matching transformations. Softw. Syst. Model. (SoSyM) **8**(3), 305–324 (2009)

25. Kolovos, D.S., Rose, R., Garca-Domnguez, A., Paige, R.: The Epsilon Book. Eclipse, Amman (2018)

26. Bzivin, J.: Model Driven Engineering: An Emerging Technical Space, vol. 4143, pp. 36–64. Springer, Berlin Heidelberg (2006)

27. Siikarla, M., Laitkorpi, M., Selonen, P., Systa, T.: Transformations Have to be Developed ReST Assured, vol. 5063, pp. 1–15. Springer, Berlin Heidelberg (2008)

28. Kolahdouz-Rahimi, S., Lano, K.: A Model-Based Development Approach for Model Transformations, vol. 7141, pp. 48–63. Springer, Berlin Heidelberg (2011)

29. Schmidt, D.C.: Model-driven engineering. Comput. IEEE Comput. Soc. **39**(2), 25 (2006)

30. Brown, A.W.: Model driven architecture: principles and practice. Softw. Syst. Model. (SoSyM) **3**(4), 314–327 (2004)

31. Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. ITPro collection. Wiley, Hoboken (1998)

32. Tahir, A., Yamashita, A., Licorish, S., Dietrich, J., Counsell, S.: Can you tell me if it smells? A study on how developers discuss code smells and anti-patterns in stack overflow. In: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, pp. 68–78 (2018)

33. Martin Fowler, D.R.J.B.W.O.K.B., Becker, P., Fowler, M., Beck, K., Shanklin, J.C.: Addison-Wesley, E. Gamma, Safari Tech Books Online (Online service). Brant, J., Opdyke, W.., et al.: Refactoring: Improving the Design of Existing Code. Addison-Wesley object technology series. Addison-Wesley (1999)

34. Brown, W.H., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, Hoboken (1998)

35. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, Boston (2003)

36. Begaudeau, T.: Acceleo/OCL operations reference. Available at https://wiki.eclipse.org/Acceleo/OCL_Operations_Reference (2019/04/1)

37. Cabot, J., Clariso, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. J. Syst. Softw. **83**(2), 283–302 (2010)

38. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Pearson Education, London (2008)

39. Kolahdouz-Rahimi, S.: A comparative study of model transformation approaches through a systematic procedural framework and goal question metrics paradigm. PhD Thesis, King's College London (University of London) (2013)

40. Opdyke, W.F.: Refactoring object-oriented frameworks. PhD Thesis, University of Illinois at Urbana-Champaign Champaign, IL, USA (1992)

41. Wimmer, M., Perez, S.M., Jouault, F., Cabot, J.: A catalogue of refactorings for model-to-model transformations. J. Object Technol. **11**(22), 1–2 (2012)

42. Hoyos, H., Chavarriaga, J., Gomez, P.: Solving the fixml case study using epsilon and java. In: Proceedings of the 7th Transformation Tool Contest, pp. 87–92. Part of the Software Technologies: Applications and Foundations (STAF 2014) Federation of Conferences (2014)

43. Lano, K., Maroukian, K., Tehrani, S.Y.: Case study: FIXML to Java, C# and C++. In *TTC@ STAF*, pp. 2–6 (2014)

44. Krikava, F., Collet, P.: Solving the TTC'14 FIXML case study with SIGMA. In: TTC@ STAF (2014)

45. Krikava, F., Collet, P., France, R.: Manipulating models using internal domain-specific languages. In: 29th Annual ACM Symposium on Applied Computing, pp. 2–6 (2014)

46. Westfechtel, B.: Case-based exploration of bidirectional transformations in QVT relations. Softw. Syst. Model. (SoSyM) **17**(3), 989–1029 (2018)

47. Ergin, H., Syriani, E.: A unified template for model transformation design patterns. In: PAME@ STAF pp. 27–30 (2015)

48. Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S., Sharbaf, M.: A survey of model transformation design pattern usage. In: Theory and Practice of Model Transformation, pp. 108–118. Springer International Publishing (2017)

49. Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S., Sharbaf, M.: A survey of model transformation design patterns in practice. J. Syst. Softw. **140**, 48–73 (2018)

50. Fontana, F.A., Zanoni, M.: A tool for design pattern detection and software architecture reconstruction. Inf. Sci. **181**(7), 1306–1324 (2011)

51. Albin-Amiot, H., Cointe, P., Guéhéneuc, Y.-G., Jussien, N.: Instantiating and detecting design patterns: putting bits and pieces together. In: Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), pp. 166–173. IEEE (2001)

52. Lano, K., Kolahdouz-Rahimi, S.: Design patterns for model transformations. In: The 6th International Conference on Software Engineering Advances, pp. 263–268. IARIA (2011)

53. Taentzer, G., Arendt, T., Ermel, C., Heckel, R.: Towards refactoring of rule-based, in-place model transformation systems. In: Proceedings of the First Workshop on the Analysis of Model Transformations, pp. 41–46 (2012)

54. Tichy, M., Krause, C., Liebel, G.: Detecting performance bad smells for Henshin model transformations. Amt@ models, p. 1077 (2013)

55. Kolahdouz-Rahimi, S.: Model transformation specification in UML-RSDS. In: ICST PhD Symposium (2010)
56. Lano, K.: Agile model-based development using UML-RSDS. CRC Press, Boca Raton (2017)
57. Lano, K.: A compositional semantics of UML-RSDS. Softw. Syst. Model. (SoSyM) **8**(1), 85–116 (2009)
58. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place emf model transformations. In: International Conference on Model Driven Engineering Languages and Systems, pp. 121–135. Springer (2010)
59. Hojaji, F., Zamani, B., Hamou-Lhadj, A., Mayerhofer, T., Bousse, E.: Lossless compaction of model execution traces. Softw. Syst. Model. (SoSyM) **19**(1), 199–230 (2019)
60. Hojaji, F., Mayerhofer, T., Zamani, B., Hamou-Lhadj, A., Bousse, E.: Model execution tracing: a systematic mapping study. Softw. Syst. Model. (SoSyM) **18**(6), 3461–3485 (2019)
61. El-Masri, D., Petrillo, F., Guhneuc, Y.-G., Hamou-Lhadj, A., Bouziane, A.: A systematic literature review on automated log abstraction techniques. Inf. Softw. Technol. (IST) **122**, 106–276 (2020)
62. Dhoolia, P., Mani, S., Sinha, V.S., Sinhae, S.: Debugging model-transformation failures using dynamic tainting. In: Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP), pp. 26–51 (2010)

Mahsa Panahandeh received her B.Sc. and M.Sc. in Software Engineering from University of, Isfahan, Iran. Currently, she is a Ph.D. candidate in software engineering at the university of Alberta, Edmonton, Canada. She is also a member of MDSE research group at the University of Isfahan, Iran. Her research interest includes model-driven software engineering (MDSE), model transformations, and studying software architectures and patterns.

Mohammad Hamdaqa is an Assistant Professor at the Department of Computer Engineering and Software Engineering at Polytechnique Montréal, Canada, and an Adjunct Professor at the Department of Computer Science at Reykjavik University, Iceland. Hamdaqa is interested in the interplay between emerging technologies and software applications and has made contributions in the areas of model-driven software engineering, cloud computing, and blockchain.

Bahman Zamani holds a Ph.D. in Computer Science from Concordia University, Montreal, QC, Canada, for his work on the pattern language verification. Currently, he is an associate professor in the Department of Software Engineering, University of Isfahan, Isfahan, Iran. His main research interest is model-driven software engineering (MDSE). He is the founder and director of the MDSE Research Group at the University of Isfahan.

Abdelwahab Hamou-Lhadj is a Professor in the Department of Electrical and Computer Engineering, Gina Cody School of Engineering and Computer Science. His research interests are in software engineering, software tracing and logging, applications of artificial intelligence to software run-time data, and model-driven engineering. He holds a Ph.D. degree in computer science from the University of Ottawa. He is a long-lasting member of IEEE and ACM, and a frequent technical contributor to OMG programs.