Towards Rule-Based Detection of Design Patterns in Model Transformations

Chihab eddine Mokaddem, Houari Sahraoui, and Eugene Syriani(✉)

University of Montreal, Montreal, Canada
{mokaddec,sahraoui,syriani}@iro.umontreal.ca

Abstract. Model transformations are at the very heart of the Model-Driven Engineering paradigm. As modern programs, they are complex, difficult to write and test, and overall, difficult to understand, maintain, and reuse. In other paradigms, such as object-oriented programming, design patterns play an important role for understanding and reusing code. Many works have been proposed to detect complete design pattern instances for understanding and documentation purposes, but also partial design pattern instances for quality assessment and refactoring purposes. Recently, a catalog of design patterns has been proposed for model transformations. In this paper, we propose to detect these design patterns in declarative model transformation programs. Our approach first detects the rules that may play a role in a design pattern. Then, it ensures that the control flow over these rules corresponds to the scheduling scheme in the design pattern. Our preliminary evaluation shows that our detection mechanism is effective for both complete and partial instances of design patterns.

1 Introduction

Model-driven engineering (MDE) is a recent software development approach that is rapidly growing in popularity [14]. At its core, it makes intensive use of models as a means for automation and reuse. MDE developers use model transformations to perform operations on models, such as: evolving, refactoring, and simulating them [16]. Model transformations, which uses generally a rule-based declarative paradigm [9], are still manually developed. Therefore, like any hand-written software programs, model transformations must be well-designed and implemented in order to be understandable by other developers, be re-used in other projects, and reduce maintenance efforts.

In other paradigms, such as object-oriented programming (OOP), design patterns play an important role in software design [13]. They are proven solutions to recurring design problems that complement practices of developers. Design patterns are described at a higher level of abstraction than the implementation language to ease communication and comprehension. They are considered as micro-architecture building blocks from which more complex designs can be built, thus promoting modularity and reuse. Recently, Lano et al. proposed a thorough catalog of over 20 design patterns for model transformations [17]. They

© Springer International Publishing AG 2016
J. Grabowski and S. Herbold (Eds.): SAM 2016, LNCS 9959, pp. 211–225, 2016.
DOI: 10.1007/978-3-319-46613-2_14

showed that these design patterns reduce complexity and execution time, as well as improve the flexibility and modularity of model transformations. Although the intent and application conditions of each pattern are described rigorously, they chose to define the solution part of the design pattern using a formal notation. To facilitate their understanding for model transformation engineers and to enable the automatic instantiation of design patterns in model transformation implementations, Ergin et al. [10] proposed a dedicated modeling language DelTa with both a graphical and a textual [12] notation.

With the increasing scale and complexity of utilizing models in MDE, the model transformations developed are also increasing in scale and complexity. Furthermore, as with any software product, model transformations are evolving constantly in development projects. This tends to deteriorate their architecture and design, which is a burden of maintenance tasks. Nevertheless, design patterns expressed in DelTa impose structure thanks to the abstraction they use. Therefore, the identification of design patterns implemented in an existing model transformation can tremendously help the developer in understanding the design, as well as document the transformation [22]. Even if a design pattern was not implemented in its integrity in the model transformation, identifying some of its participants provides valuable feedback to the developer: (1) a missed opportunity to implement it in order to improve the quality, (2) a suggestion to correctly implement it through refactoring, or (3) the presence of a modified version of the design pattern, since any design pattern may be implemented with endless variations [20]. Various design pattern detection mechanisms have proven to be very efficient [2,4,7,22]. However, these techniques have been applied to imperative OOP code. Detecting design patterns on model transformations comes with many challenges because they are described (1) declaratively, (2) at the level of meta-models dealing with types and relations rather than instances, and (3) with non-deterministic execution of rules.

In this paper, we present an approach to detect complete or partial instances of design patterns in concrete model transformation implementations. It is a model finding approach based on a rule engine, where we map model transformations to an abstract representation and design patterns to rules that these representations must satisfy. After identifying individual participants of a design pattern, we verify that the scheduling scheme described in the pattern is satisfied in the transformation. We compute an accuracy score at each detection step that is finally aggregated and reported. We implemented a prototype where we encode design patterns defined with the DelTa language as rules and that automatically maps a complete model transformation implemented in a specific model transformation language to the abstract representation. We report preliminary results that show our detection mechanism is effective for both complete and partial instances of design patterns.

In Sect. 2 we provide the necessary background on model transformation and their design patterns. In Sect. 3 we describe our approach on an example. We report the results on the effectiveness of our approach in Sect. 4. Finally, we conclude in Sect. 5.

2 Background

We first review background on model transformations and their design patterns, and then discuss different techniques for detecting design patterns in programs.

2.1 Model Transformation

In MDE, a model transformation is the automatic manipulation of a model following a specification defined at the level of metamodels [16]. A model transformation can be outplace, when it produces a target model from a source model, such as in a translation, or it can be inplace when it modifies a model and the result is an updated version of the source model, such as in a simulation. Typically, a model transformation is defined by a set of declarative rules to be executed. A rule consists of a pre-condition and a post-condition pattern. The pre-condition pattern determines the applicability of a rule: it is the pattern that must be found in the input model to apply the rule. Optionally negative patterns may be specified in the pre-condition to inhibit the application of the rule if present. The post-condition imposes the pattern to be found after the rule is applied. Patterns are made up of structural elements (i.e., model fragments) and of constraints on their attributes. Rules follow a scheduling scheme that defines the order in which they are applied when a transformation is executed. The scheduling can be made explicit by the language with a control flow structure partially ordering rules, such as in Henshin. In some languages, such as ATL, rules are scheduled implicitly, depending on the causal dependence between the post-condition of a rule and the pre-condition of another. Features that model transformation languages support are listed in [9]. A comparison of existing model transformation tools can be found in [18]. Possible scheduling schema of model transformations are described in [21].

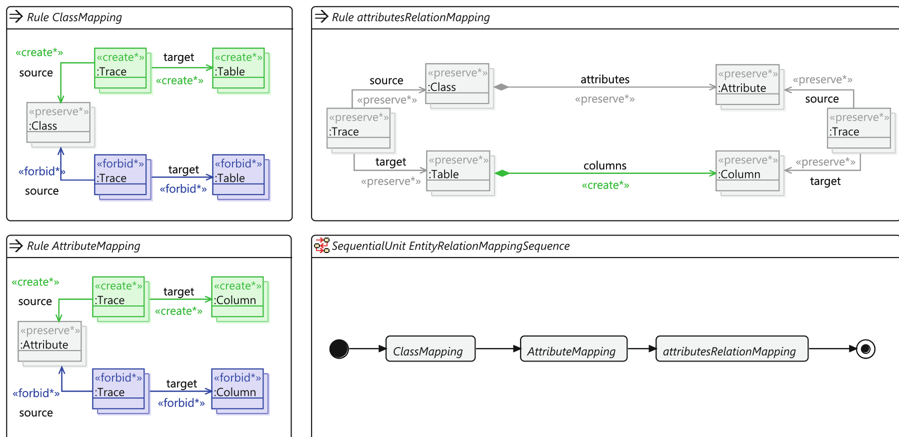


Fig. 1. Model transformation of entity relation in Henshin

For example, consider the model transformation defined in the Henshin language in Fig. 1. It contains three rules that are scheduled to execute in sequence, as depicted on the bottom right. This is an excerpt of transformation that creates database tables and columns from a class diagram. The first rule in the top left states that if a class is present then create a table and link it with a trace element unless such a trace already exists for the class.

2.2 Model Transformation Design Patterns

A design pattern expresses a means of solving a common model transformation design problem: it describes the transformation structure (rules, condition patterns, and scheduling) that constitute the solution idea. A design pattern includes also a description of the problem which motivated the pattern, how such problems can be detected, and the benefits and negative consequences to consider when using the pattern.

In the mid-2000s, several works proposed design patterns for model transformation. Agrawal et al. [8] defined design patterns for graph transformation described in a specific model transformation language. Iacob et al. [15] defined other design patterns for outplace transformations. Levendovszky et al. [19] proposed domain-specific design patterns for model transformation and different domain-specific languages.

More recently, Lano et al. [17] presented the most comprehensive model transformation design pattern study and defined a catalog of 29 patterns classified into five categories. For example, these include a design pattern to map objects before links, to decompose a transformation into phases based on the target model, the criteria to separate rules so they can be executed in parallel, to ensure that elements created by a rule are unique, or to individually process all nodes of a model recursively.

At the same time, Ergin and Syriani [11] presented similar design patterns, as well as new ones, such as modifying a model iteratively until a fixed point is reached, or the execution of a modeling language by translating it into another modeling language that can be simulated.

2.3 DelTa to Describe the Structure of Design Patterns

Lano et al. [17] presented the structure design patterns using a formal language TSPEC in the form of contracts with pre- and post-conditions that a concrete model transformation implementing the pattern should satisfy. However, Ergin and Syriani [12] engineered a domain-specific language, *DelTa*, dedicated to represent the structure of model transformation design patterns. Because an implementation is already available in EMF, we opted to use the DelTa implementations of Lano et al.'s design patterns.

DelTa is a language to define model transformation design patterns with its own syntax and semantics. It is independent from existing model transformation languages. In terms of abstraction, DelTa borrows concepts from various MTLs

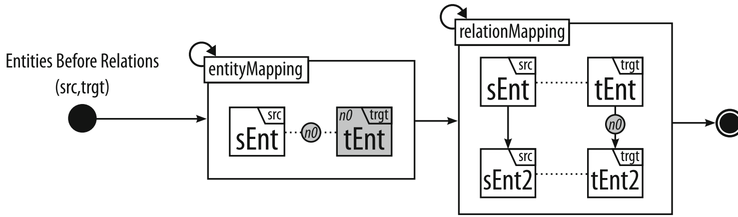


Fig. 2. Entities before relations design pattern

to create a more understandable and common language. Figure 2 represents a model transformation design pattern in its graphical syntax as described in [10].

A DelTa model specifies the minimal rules (the large rectangles) and necessary rule scheduling (the connections between them) that a concrete model transformation implementing it should have. Rules consist of the minimal constraints and actions on elements of the metamodel that concrete transformation rules implementing them should specify. Constraints and actions refer to variables that are typed as entities (rectangles like `sEnt`) or relations (arrows between entities) of a metamodel, or traces (dotted lines). For example, in rule *entityMapping*, there is a constraint stating that there must be an entity (`sEnt`). Furthermore, the *n0* symbol on rule elements indicates that trace and the entity `tEnt` are part of a negative constraint. These two entities come from different metamodels (`src` and `trgt`). In DelTa, we only reason about entities and relations, independently from specific metamodel types and relations. Entities are represented using a UML class notation and their metamodel appears on the top right. An "x" symbol on an element inside a rule means that this element should not appear in the concrete transformation rule implementing the DelTa rule.

Color coding of entities and relations inside the rules indicates whether they are part of the constraint or a type of action of the rule. White elements form the minimal application pre-condition that a concrete transformation rule implementing it should have. Gray elements are the minimal elements to be created in the concrete transformation rule. For example, the `tEnt` and the trace between it and `sEnt` must be created. Therefore, the rule *entityMapping* dictates that the concrete transformation rule implementing it should look for an entity from one metamodel and create a new entity from another metamodel, as well as a trace between them. Elements in black are the minimal elements to be deleted in the concrete transformation rule.

When a self loop symbol appears on the top left (as it is the case with both rules in Fig. 2), the DelTa rule is exhaustive: the concrete transformation rule implementing it should be applied on all of its matches. This may require to have more than one rule implementing this DelTa rule, for example to match different metamodel types.

In DelTa, the scheduling is depicted using a control flow notation. The start node (filled ball) indicates the initial rule of the design pattern. Arrows between rule blocks indicate a predence order: the concrete transformation rule

implementing the *entityMapping* rule should be performed before the one implementing the *relationMapping* rule. A dashed box containing rules specifies that the order of execution of the rules it contains is irrelevant to the design pattern. Entities, rules, and scheduling represent the *participants* of a model transfromation design pattern. In this paper, we use model transformation design patterns expressed in DelTa from [10].

2.4 Design Patterns Detection in Software Engineering

To the best of our knowledge, there is no previous work that tackles the detection of design patterns in model transformation. Most of the detection approaches target the patterns of Gamma et al. in object-oriented programs [13]. These approaches target primarily the structural patterns as these can be detected by matching the structure of code to one of the pattern [3,22]. To improve the detection, some projects combine multiple strategies as in [7]. The detection of behavioral patterns also attracted the interest of the research community. In De Lucia et al. [2], the authors use model checking to improve the detection of behavioral patterns. A work similar to our is one in [5]. In this paper, the authors first identify pattern key participants using a machine learning technique. Then, they check for the other participants of the pattern and the relations between them.

3 Design Pattern Detection for Model Transformation

We propose an approach to detect complete and partial instances of design patterns in concrete model transformations. We consider design pattern detection as a constraint satisfaction problem where a design pattern imposes a specific structure that a concrete model transformation should contain, and we solve it using a declarative strategy based on an inference rule engine.

3.1 Overview

As shown in Fig. 3, the detection of a design pattern is encoded as a set of *rules*. These rules apply to a set of *facts* representing the model transformation. The facts conform to *fact templates*: a generic abstract representation of transformation components relevant to design pattern detection. This abstract representation makes our approach independent from a specific model transformation language. The mapping to of a concrete model transformation is performed by a model-to-text transformation.

The detection process is performed in three automated steps. First, the transformation is mapped to an abstract representation (i.e., facts) using a higher-order transformation. Second, we identify which rules of the model transformation can play the role of the participants of the design pattern. Third, once the participant candidates are identified, we verify that their execution satisfies the scheduling scheme specified in the design pattern.

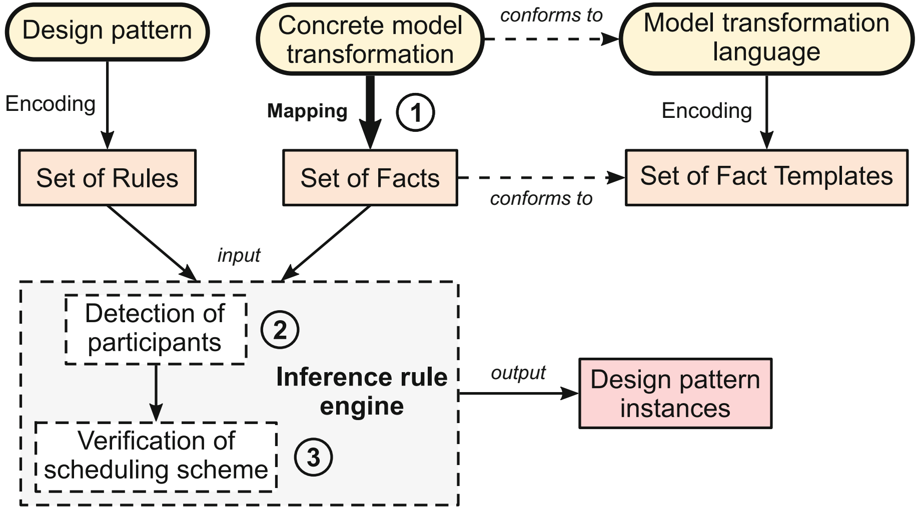


Fig. 3. Architecture overview of design pattern detection

In the remainder of this section, we describe how concrete model transformations are mapped to generic facts and then explain the two steps of the detection process.

3.2 Mapping Model Transformations to Generic Facts

Fact Template. To describe model transformations, we defined a fact-based language inspired by the Henshin transformation language [1]. The motivation behind this decision is that design patterns, as defined in [17], deal mainly with the manipulation (creation/modification/deletion) of model elements by rules as well as with the rule execution scheduling. All these constructs can be described by the Henshin concepts.

The main fact template to describe a transformation is `Rule`. A `Rule` is composed of nodes, each corresponding to an action on a model element present in the pre- or post-condition of a model transformation rule. Nodes are described by the fact template `Node`. Nodes have several attributes to define the element name and type they represent, a reference to the rule in which they appear, and also an action. If the action slot is assigned "create", "update" or "delete", then the node is part of the post-condition of the transformation rule. If it is assigned "preserve" or "forbid", then the node is part of the pre-condition (positive or negative constraint, respectively) of the transformation rule. Nodes may also be related with the `Edge` fact template when the action in one node depends on another node, e.g., an element is created and its attributes are set according to those of another element. For rule execution scheduling, we define the fact template `Sequence` that specifies the precedence between two rules.

The precedence relationship may also involve control events such as the beginning and the end of a loop.

Listing 1.1 shows fact templates for `Rule`, `Node`, and `Sequence` expressed in the Jess language [6]. In Jess, each template has a name and a set of slot definitions. When asserting a fact, the slots must be set with values. Some slots are used to describe the fact properties such as Name in Rule and Action in Node. Others are used to connect facts. For example, the slot `RuleId` in Node is set with the `Id` of the `Rule` which the node belongs to. Similarly, `SourceId` and `TargetId` in Sequence refer respectively to the `Ids` of the preceding and following rules.

Listing 1.1. Fact Templates representing a model transformation language

```
1 (deftemplate Rule (slot Id)(slot Name))
2
3 (deftemplate Node (slot Id)(slot RuleId)(slot Action)
4 (slot Occurrences)(slot Name)(slot Type))
5
6 (deftemplate Sequence(slot SourceId)(slot TargetId))
```

Fact. Listing 1.2 shows the Jess facts of a rule having two nodes.

Listing 1.2. Fact representing a concrete model transformation

```
1 (Rule (Id"R1")(Name"Class2TableMapping"))
2
3 (Node (Id"N1")(RuleId"R1")
4 (Action"preserve")(Occurrences"n")(Name"")(Type"Class"))
5
6 (Node (Id"N2")(RuleId"R1")
7 (Action"create")(Occurrences"n")(Name"")(Type"Table"))
```

To be effective for large transformations, we automate the mapping of a given concrete model transformation to a set of facts. Therefore, we need to write a fact generator for each model transformation language considered. To this end, we use Acceleo[1], a template-based model-to-text transformation tool in EMF. These code generation templates encode the semantic equivalence between the transformation language constructs and our fact templates. Listing 1.3 illustrates an example for generating of a fact `Rule` from a Henshin rule. Although our implementation currently supports Henshin, adapting to another model transformation language simply requires to create a new Acceleo template for it.

Listing 1.3. Fact representing a concrete model transformation

```
1 [template public generateRule(rule:Rule, position:Integer)]
2     (Rule (Id \"["R" + position]\") (Name \"[rule.name/]\"))
3 [/template]
```

[1] https://eclipse.org/acceleo/.

3.3 Encoding Design Patterns as Detection Rules

As mentioned in Sect. 2.2, the participants of a model transformation design pattern are DelTa rules, the elements they contain in their constraint and actions, and their scheduling scheme. The pattern *Entities Before Relations* in Fig. 2, for instance, consists of two DelTa rules: *entityMapping* and *relationMapping*. It also mandates that the former must be executed before the latter. Consequently, our detection strategy starts by finding concrete model transformation rules that match the ones in the DelTa model, and then verify if the scheduling specified in the patterns holds for the concrete matched rules.

Listing 1.4. Rule encoding the complete entityMapping rule of Entities before Relations design pattern

```
1  (defrule CreateEntityMapping_Rule
2    (Rule (Id ?r_1)(Name ?r_2))
3    (Node (Id ?sEnt_1)(RuleId ?r_1)(Action"preserve")
4      (Occurrence ?sEnt_4)(Name ?sEnt_5)(Type ?sEnt_6))
5    (Node (Id ?tEnt_1)(RuleId ?r_1)(Action"forbid")
6      (Occurrence ?tEnt_4)(Name ?tEnt_5)(Type ?tEnt_6))
7    (Node (Id ?tEnt_2)(RuleId ?r_1)(Action"create")
8      (Occurrence ?tEnt_4)(Name ?tEnt_5)(Type ?tEnt_6))
9    (Edge (Id ?ed_1)(RuleId ?r_1)(SourceId ?sEnt_1)
10     (TargetId ?tEnt_1))
11   (Edge (Id ?ed_2)(RuleId ?r_1)(SourceId ?sEnt_1)
12     (TargetId ?tEnt_2))
13 =>
14   (assert
15    (EbR_entityMapping
16     (Id (str-cat ?r_1 ?sEnt_1 ?tEnt_1 ?tEnt_2 ?Ed_1 ?Ed_2))
17     (RuleId ?r_1)
18     (sEnt_1Id ?sEnt_1)
19     (tEnt_1Id ?tEnt_1)
20     (tEd_1Id ?ed_1)
21     (tEnt_2Id ?tEnt_2)
22     (tEd_2Id ?ed_2)
23     (accuracy 1))
24   )
25 )
```

The detection of instances of a DelTa rule is encoded as a *rule* in Jess. For example, Listing 1.4 rule detects complete instances of *entityMapping*. The Jess rule first filters all transformation rule facts that have a “preserve” node connected to a “forbid” node and to a “create” node. For each rule satisfying these conditions, it asserts a fact *EbR_entityMapping*. Another Jess rule will filter the concrete rules that can play the role of *relationMapping* and asserts for each match a fact *EbR_relationMapping*. The encoding of DelTa rules into Jess rules can be implemented with Acceleo templates.

Once the potential participants are detected, the next step is to ensure if the execution schedule of the concrete rules corresponds to the one of the pattern. In the case of the pattern *Entities Before Relations*, a Jess rule filters facts

EbR_entityMapping and *EbR_relationMapping*, and a `Sequence` fact relating the rules respectively involved in the participant facts.

3.4 Accuracy for Complete and Partial Instances

In the case of complete instance detection, all the conditions (participants and scheduling) should be fully satisfied, i.e., accuracy equals 1.

When detecting partial instances, rules variants are defined for participants and scheduling detection. These rules may omit one of the conditions and adjust the value of fact accuracy accordingly. For example, in the detection of *entityMapping* participants, a variant rule can consider rules with "preserve" and "create" nodes, but without a "forbid" node. This is depicted in Listing 1.5. The accuracy is then adjusted to 0.66 for example. The scheduling verification rule, calculate the global accuracy of the pattern instance from the accuracy values of the participants facts and one of the scheduling itself.

Listing 1.5. Rule encoding a partial entityMapping rule of Entities before Relations design pattern

```
1  (defrule CreateEntityMapping_Rule
2    (Rule (Id ?r_1)(Name ?r_2))
3    (Node (Id ?sEnt_1)(RuleId ?r_1)(Action"preserve")
4      (Occurrence ?sEnt_4)(Name ?sEnt_5)(Type ?sEnt_6))
5    (not (Node (Id ?tEnt_1)(RuleId ?r_1)(Action"forbid")
6      (Occurrence ?tEnt_4)(Name ?tEnt_5)(Type ?tEnt_6))
7    ...
8  =>
9    (assert
10    (EbR_entityMapping
11     (Id (str-cat ?r_1 ?sEnt_1 ?tEnt_1 ?tEnt_2 ?Ed_1 ?Ed_2))
12     (RuleId ?r_1)
13     (sEnt_1Id ?sEnt_1)
14     (tEnt_1Id"")
15     (tEd_1Id"")
16     (tEnt_2Id ?tEnt_2)
17     (tEd_2Id ?ed_2)
18     (accuracy 0.66))
19   )
20 )
```

4 Preliminary Evaluation

4.1 Setup

A preliminary evaluation of this work consists in selecting a subset of design patterns and detect their instances on a sample of model transformations. The goal here is to analyze qualitatively how our detection approach applies to concrete transformations.

We selected 13 Henshin transformations[2] with different characteristics (see Table 1). As we had to analyze manually the results, we opted for small-medium transformations having 1 to 13 rules. We also paid attention to the control complexity as most of the transformation design patterns deal with the rule execution control. Indeed, some of the selected transformations use default implicit control (no control specified), and others have up to 13 rule scheduling units with loops and calls between the units. Additionally, we varied the complexity of the rules with respect to the number of involved model elements, with an average number of nodes per rule between 3 and 11.

Table 1. Selected transformations.

Model transformations	# rules	# sch-unit	# nodes	# relations	# calls	# loop
bank	3	0	12	12	0	0
bankmap	1	0	5	4	0	0
comb	2	1	22	38	1	1
diningphils	4	0	22	34	0	0
ecore2genmodel	8	6	55	59	12	2
gossipingGirls	2	0	7	9	0	0
grid-full	4	5	18	27	8	3
grid-sparse	3	4	11	16	6	2
java2statemachine	13	13	77	59	27	5
petriM	2	0	15	27	0	0
sierpinski	1	0	6	12	0	0
sort	1	1	3	2	1	1
entityRelationMapping	3	1	16	14	3	0

In this preliminary evaluation we experimented with the detection of three patterns, selected from the catalog of [17]. Two of them deal with the rule modularization (*Entities Before Relations* and *Construction and cleanup*), and one with optimization (*Unique Instantiation*).

Entities Before Relations. The goal of this pattern (Fig. 2), also called *Map Objects Before Links*, is to create the entities and then their relations. As mentioned in Sect. 3.3, three rules are defined for the detection of this pattern: (1) detection of entities creation, (2) detection of relations creation, and (3) precedence checking between the two creations. In addition to the detection of complete instances, we implemented the detection of one kind of partial instance, i.e., the situation in which the transformation program have rules for creating the entities before the creation of their relations, but does not check if an entity exists before it creates a new one (see Sect. 3.4).

[2] https://www.eclipse.org/henshin/examples.php.

Construction and Cleanup. As shown in Fig. 4, this pattern consists in separating rules which create model elements from those which delete elements [17]. Like for the previous patterns, the detection is done in three phases: (1) finding element creation rules, (2) finding element deletion rules, and (3) precedence checking between the two.

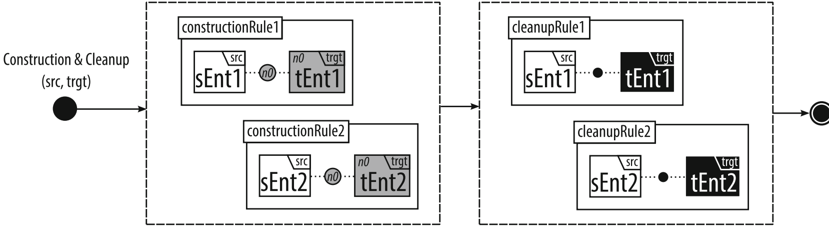


Fig. 4. Construction & cleanup - Structure in DelTa

Unique Instantiation. This pattern, sketched in Fig. 5, aims at avoiding multiple creations of the same model element. This may happen in two situations: (1) two rules creating the same model element or (2) a rule creating a model element, and that appears in a loop inside a rule execution schedule. We defined detection rules for each situation, i.e., identifying element-creation rules, and checking duplications and loops.

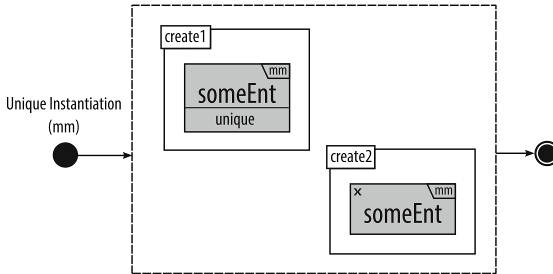


Fig. 5. Unique instantiation - structure in DelTa

4.2 Qualitative Analysis

Entities Before Relations. Surprisingly, our prototype did not find complete instances of the pattern *Entities Before Relations*. To understand this, we manually inspected the automatically detected partial instances. We noticed that, in many cases, the *EntityMapping* participants were identified with an accuracy of 1. However, the *relationMapping* participants did not satisfy the condition of the non-existence of a relation before its creation. All the detected partial instances satisfied the execution schedule conditions with perfect accuracy. Figure 1 illustrates two examples of partial instances found in the *entityRelationMapping* e rules transformation. The rules *ClassMapping* and *AttributeMapping*

are both complete instances of *entityMapping*. Conversely, in rule *attributeRelationMapping*, the relation between "Class" and "Attribute" is mapped to a relation between "Table" and "Column" without ensuring that such a relation does not already exist (not a "forbid" action). Although the scheduling is perfectly accurate, i.e., both *ClassMapping* and *AttributeMapping* rules precede *attributeRelationMapping*, the aggregated accuracy is lower than 1.

Construction and Cleanup. The prototype found many instances of the design pattern *Construction and cleanup*. An interesting instance is one found in the *Java2StateMachine*. In this transformation, only one rule has a "delete" action (*updateAction* on the right of Fig. 6). All the other rules create elements. This rule appears at the last step of the execution schedule (on the left of Fig. 6). This is a non trivial instance to detect because of the modularization of the execution schedule. In our detection program, we implemented a function that reconstructs a flat schedule by resolving the schedule step references.

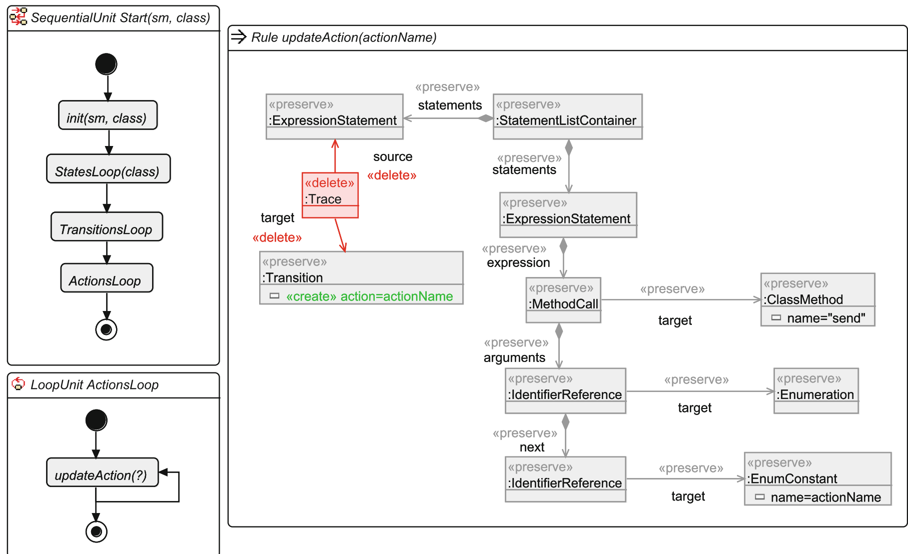
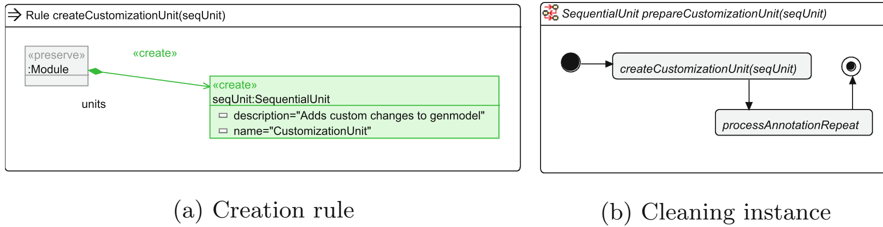


Fig. 6. An example of instance of the pattern *Const*ruction and Cleanup

Unique Instantiation. This is by far the most frequent pattern and many of its instances were found in almost all the considered transformations. Some of them have a high accuracy. An example of a complete instance was found in the *Ecore2GenModel* high-order transformation. The *createCustomizationUnit* rule creates an element, which is not created by other rules (Fig. 7a). Moreover, this rule does not appear in a loop in the execution schedule (Fig. 7b).



(a) Creation rule

(b) Cleaning instance

Fig. 7. Unique instantiation instance detected in Ecore2Genmode transformation

5 Conclusion

In this paper, we propose an approach and a preliminary implementation for the detection of complete and partial instances of design patterns in model transformations. Our approach follows a declarative strategy which consists in identifying transformation rules that play the roles of design pattern participants and then check if their execution sequence conforms to the schedule specified in the pattern.

We conducted a preliminary evaluation which consisted in applying our detection rules on a set of transformations and in qualitatively analyzing the detection results. Although the obtained results are encouraging, our evaluation revealed some limitations. First, we define explicitly rules for detecting pattern variants [20]. The advantage of this strategy is that we identify acceptable variants of a design pattern. The drawback is that our detection code is very verbose with very similar rules. We plan in the future to have a generic detection of variants by allowing weights to the pattern participants.

Another limitation of our approach resides in the limited number of control structures we handle. In our current implementation, we do not consider alternatives structures. Thus for the pattern *Unique Instantiation*, if two rules respectively in the two branches of the alternative create the same element, we do not detect a valid instance. Handling more control structures is a part of our future work.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place emf model transformations. In: Model Driven Engineering Languages and Systems, pp. 121–135 (2010)
2. De Lucia, A., Deufemia, V., Gravino, C., Risi, M.: Improving behavioral design pattern detection through model checking. In: European Conference on Software Maintenance and Reengineering, pp. 176–185 (2010)
3. Dong, J., Zhao, Y., Peng, T.: A review of design pattern mining techniques. Int. J. Softw. Eng. Knowl. Eng. **19**(06), 823–855 (2009)
4. Guéhéneuc, Y.G., Guyomarc'h, J.Y., Sahraoui, H.: Improving design-pattern identification: a new approach and an exploratory study. Softw. Qual. J. **18**(1), 145–174 (2010)

5. Gueheneuc, Y.G., Sahraoui, H., Zaidi, F.: Fingerprinting design patterns. In: Working Conference on Reverse Engineering, pp. 172–181. IEEE (2004)
6. Hill, E.F.: Jess in Action: Java Rule-Based Systems. Manning Greenwich, Greenwich (2003)
7. Rasool, G., Mäder, P.: Flexible design pattern detection based on feature types. In: International Conference on Automated Software Engineering, pp. 243–252 (2011)
8. Agrawal, A.: Reusable idioms and patterns in graph transformation languages. In: International Workshop on Graph-Based Tools, ENTCS, vol. 127, pp. 181–192. Elsevier (2005)
9. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. Spec. Issue Model-Driven Softw. Dev. **45**(3), 621–645 (2006)
10. Ergin, H., Syriani, E., Gray, J.: Design pattern oriented development of model transformations. Comput. Lang. Syst. Struct. **46**, 106–139 (2016). doi:10.1016/j.cl.2016.07.004
11. Ergin, H., Syriani, E.: Identification and application of a model transformation design pattern. In: ACM Southeast Conference, ACMSE 2013. ACM (2013)
12. Ergin, H., Syriani, E.: Towards a Language for Graph-Based Model Transformation Design Patterns. In: Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 91–105. Springer, Heidelberg (2014). doi:10.1007/978-3-319-08789-4_7
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional, Boston (1994)
14. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: International Conference on Software engineering, pp. 471–480. ACM (2011)
15. Iacob, M.E., Steen, M.W.A., Heerink, L.: Reusable model transformation patterns. In: Enterprise Distributed Object Computing Conference Workshops, pp. 1–10. IEEE Computer Society (2008)
16. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G.M., Syriani, E., Wimmer, M.: Model transformation intents and their properties. Softw. Syst. Model. **15**(3), 647–684 (2014)
17. Lano, K., Rahimi, S.K.: Model-transformation design patterns. IEEE Trans. Softw. Eng. **40**(12), 1224–1259 (2014)
18. Lano, K., Rahimi, S.K., Poernomo, I.: Comparative evaluation of model transformation specification approaches. Int. J. Softw. Inf. **6**(2), 233–269 (2012)
19. Levendovszky, T., Lengyel, L., Mészáros, T.: Supporting domain-specific model patterns with metamodeling. Softw. Syst. Model. **8**(4), 501–520 (2009)
20. Prechelt, L., Krämer, C.: Functionality versus practicality: employing existing tools for recovering structural design patterns. J. Univ. Comput. Sci. **4**(11), 866–882 (1998)
21. Syriani, E., Vangheluwe, H.: A modular timed graph transformation language for simulation-based design. Softw. Syst. Model. **12**(2), 387–414 (2013)
22. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.: Design pattern detection using similarity scoring. Trans. Softw. Eng. **32**(11), 896–909 (2006)