Software and Systems Modeling (2022) 21:1241–1269
https://doi.org/10.1007/s10270-021-00936-4

REGULAR PAPER

A generic approach to detect design patterns in model transformations using a string-matching algorithm

Chihab eddine Mokaddem[1] · Houari Sahraoui[1] · Eugene Syriani[1]

Received: 30 June 2020 / Revised: 21 September 2021 / Accepted: 5 October 2021 / Published online: 15 November 2021

Abstract
Maintaining software artifacts is a complex and time-consuming task. Like any other program, model transformations are subject to maintenance. In a maintenance process, much effort is dedicated to the comprehension of programs. To this end, several techniques are used, such as feature location and design pattern detection. In the particular case of model transformations, detecting design patterns contributes to a better comprehension as they carry valuable information on the transformation structure. In this paper, we propose a generic approach to detect, semi-automatically, design patterns and their variations in model transformations. Our approach encodes both design patterns and transformations as strings and use a string-matching algorithm for the detection. The approach is able to detect complete and partial implementations of design patterns in transformations, which is useful to refactoring and improving model transformations.

Keywords Design pattern · Model transformation · Pattern detection · String matching · Bit-vector · Model-driven engineering

1 Introduction

Model transformation is now the mainstream paradigm to manipulate models in model-driven software engineering (MDE) [6]. Designing model transformations is a tedious task. Moreover, like any other code artifact, model transformations evolve and should be maintained. To assist developers in writing and maintaining model transformations, several design patterns have been proposed [11,22]. In general, design patterns facilitate the comprehension and manipulation of software programs [1]. In the special case of model transformations, they help improving the quality of model transformation specifications and designs, as stated by Lano et al. [22].

Detecting instances of a pattern in a transformation provides valuable information to the developer, such as understanding high-level concepts used and identifying refactoring and reuse opportunities. However, as for general programs, developers do not always implement perfectly a pattern in model transformations. Hence, design pattern detection should identify both complete and incomplete occurrences. Detecting various forms of a design pattern, including incomplete forms, offers refactoring opportunities to improve transformations by completing a form or by replacing one form by a more appropriate one.

Detecting design patterns in model transformations did not get much attention so far from the modeling community. To the best of our knowledge, only our previous work in [28] has attempted to automatically detect design patterns in model transformations using manually written detection rules. Preliminary results showed that this is an effective technique to find complete and approximate design pattern occurrences. However, this technique has performance limitations as it relies on a rule inference engine that is time and memory consuming. Another limitation of this technique is the need to specify a set of detection rules for each pattern.

To find inspiration on how to detect patterns in transformations, we looked at the active community of design pattern detection in object-oriented programs. As reported in [1], there are dozens of detection approaches for this family of programs. However, as mentioned in [13], these approaches also suffer from performance problems, because detecting

Communicated by Gabor Karsai.

✉ Chihab eddine Mokaddem
cemo.mokaddem@umontreal.ca

Houari Sahraoui
houari.sahraoui@umontreal.ca

Eugene Syriani
eugene.syriani@umontreal.ca

1 Université de Montréal, Montréal, Canada

complete and incomplete occurrences is generally costly in time, due to the large search space that includes all possible combinations of classes. These approaches are also prone to return many false positives, impeding program comprehension, and cluttering the maintainers' cognitive capabilities. To address the performance issues, the work by Kaczor et al. [20] uses a string matching technique inspired by pattern matching algorithms in bioinformatics to identify pattern occurrences in object-oriented programs. These algorithms allow to efficiently process a large amount of data if the problem to solve can be encoded as a string matching one.

In this paper, we propose a generic technique to detect design pattern occurrences in declarative model transformation implementations, without writing detection code for each design pattern, its variants, and approximations. Like in Kaczor et al., we rely on a bit-vector algorithm that has proved to be efficient for string matching problems [30]. The challenge we faced is how to encode model transformations, which are sets of rules linked by control schemes, as strings. The same challenge arises also in the encoding of the patterns as strings. We succeed to encode the participants of a patterns as strings, but had to complete our approach by a manual step to combine the identified participant instances to form pattern occurrences. Thus, the detection consists in an automated step that matches the participant strings of a pattern with rule strings of transformation, and a manual step to complete the occurrences. In addition to the performance, an advantage of using this approach is the fact both complete and incomplete occurrences can be detected.

We evaluated our approach for the detection of various forms of 10 design patterns in 18 transformations, collected from open source repositories. We first manually checked to which extent design patterns are used in these transformations. Then, we evaluated the ability of our approach to detect various instances of the patterns. We also studied the co-occurrences of different patterns. Our results show that patterns are effectively used in transformations and that our approach is able to detect them, in less time than the rule-based approach. Moreover, we found that patterns are not always used independently, but in combination with other patterns.

The rest of the paper is structured as follows. In Sect. 2, we first introduce the basic notions used in our work, and then, discuss the related work. Section 3 details different steps of our approach, whereas Sect. 4 describes different forms of patterns that can be detected by our approach. We provide an evaluation of the detection approach in Sect. 5. Finally, we discuss the limits of our approach in Sect. 6 and conclude in Sect. 7.

2 Background and related work

In this section, we briefly give some background on model transformations and corresponding design patterns and illustrate these concepts with a running example. This example also allows us to highlight the challenges of design pattern detection in model transformations. The rest of the section is dedicated to the discussion of the related work on design pattern detection.

2.1 Model transformation: an illustrative example

To help understanding the main concepts involved in our research, we use as example the exogenous transformation `Java2StateMachine`, implemented in Henshin [5]. This transformation was written for the Transformation Tool Contest of 2011 [19]. An excerpt of this transformation is depicted in Fig. 1.

We are specifically interested in rule-based model transformations. This kind of transformation is typically defined with 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 and contains two types of conditions, positive and negative. The positive application condition (PAC) represents the pattern that must be found in the input model to apply the rule. Optionally, negative application conditions (NAC) may be specified to inhibit the application of the rule if these conditions are verified. The rule `createTransition` in Fig. 1-left illustrates the elements involved in a transformation in Henshin. A rule is represented as a graph representing both the pre- and post-conditions. Gray elements, annotated with the *"preserve"* stereotype, indicate the PAC pattern, whereas the blue elements, annotated with the *"forbid"* stereotype, form the NAC conditions. Henshin supports multiple NAC groups. In this rule, the `Trace` and `Transition` objects together with their adjacent associations are part of the same NAC group identified as *#default*. The post-condition imposes the pattern to be found after the rule is applied. In our rule, the green elements, annotated with the *"create"* stereotype, must be created by the rule. It is also possible to delete elements (represented by the color red and the stereotype *"delete"* (not present in our example). In summary, patterns are made up of structural elements (i.e., model fragments) and of constraints or actions on them. Therefore, roughly speaking, the rule `createTransition` states that if a method call exists between a source class and a target class, create a transition between the states corresponding to these classes if a transition was not created already for this method call. To check for the pre-existence of the transition (NAC), the transformation uses a traceability auxiliary meta-model to keep a trace of all created transitions (as highlighted by the `Trace` node). Rules in Henshin can be parameter-

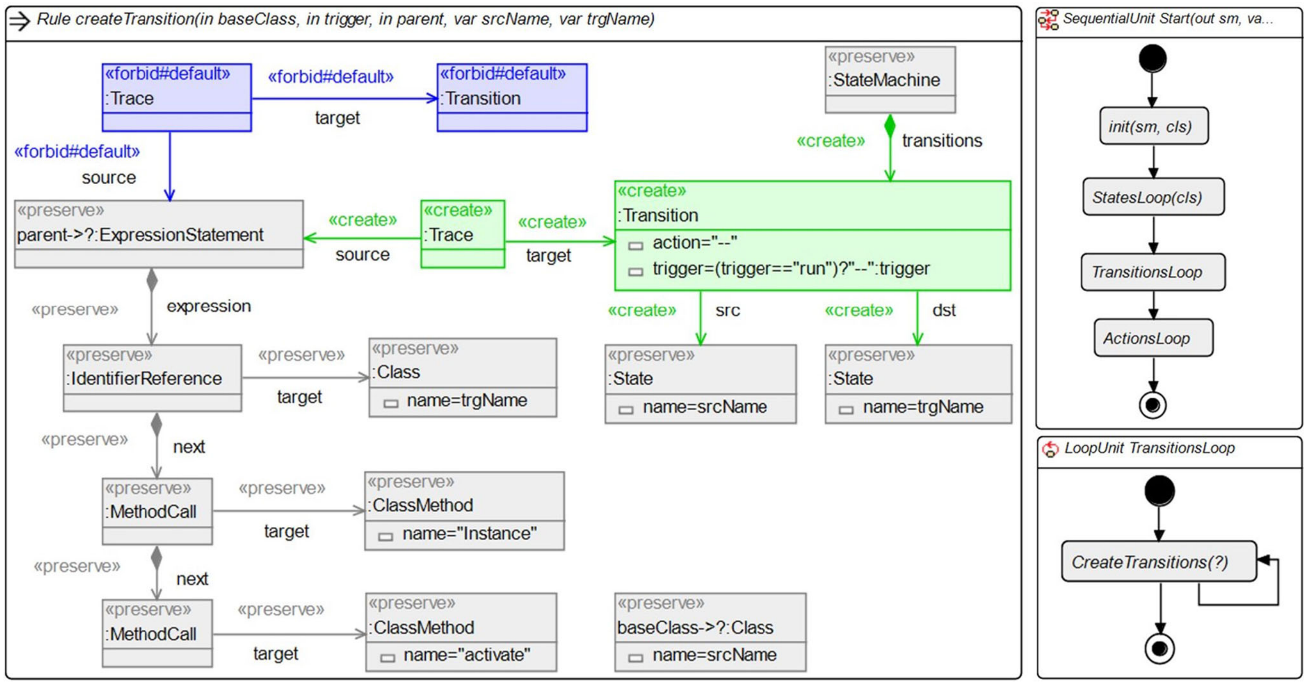


Fig. 1 Excerpt of the `Java2StateMachine` transformation implemented in Henshin, showing the `createTransition` rule

ized to bind elements across rules or pass values. This rule has three input (`in`) parameters. `baseClass` is bound to a class type that must be preserved, `parent` is bound to an expression statement type that must be preserved, and `trigger` is a string value. The latter is used in attribute constraints or assignments. For example, if the value of the `trigger` parameter is the string `"run"`, then the trigger value of the transition is –, otherwise it will be assigned the parameter value. A rule may also declare local variables, like `srcName` and `trgName` to refer to objects within the rule. For example, the source of the created transition must be the state whose name is the same as the base class name.

The rule `createTransition` is fired according to a scheduling scheme that defines the order in which the rules are applied when a transformation is executed. In Henshin, the scheduling can be specified explicitly through one or more control flow structures to partially order the rule executions. In our example, a first control structure specifies that state rules must be executed before transition rules (Fig. 1 top right). The second control structure, in the bottom right of the figure, states that the transition rule must be fired iteratively until all the transitions are created. As it can be seen in our approach, the detection of design patterns is done semi-automatically: (1) automated detection of patterns in the rules and (2) manual verification of the compliance with the control scheme and attribute constraints.

2.2 Design patterns for model transformations

A model transformation design pattern expresses a means of solving a common model transformation design problem [22]. 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.

The idea of proposing design patterns for model transformation gained popularity by the late 2000s. Small sets of patterns were proposed by different research teams such as ones in [16,17]. Later in [22], Lano et al. presented a larger catalog of 24 design and specification patterns. To allow the representation of patterns in a way that ease their usage in development processes, Ergin et al. [11] defined a domain-specific language, DelTa. DelTa represents design pattern solutions in a platform-independent model: independent from the model transformation language. Since we are interested in detecting design patterns in concrete model transformations, we must transform the design pattern into a format that will correspond to its implementation in a specific model transformation language. Figure 2 shows an example pattern from the catalog in [22] that was described in DelTa, and adapted for Henshin. For example, in this representation specific to Henshin, we explicitly represent a trace object with source and target links. In DelTa, this is originally represented by a trace link, which is not possible to define in

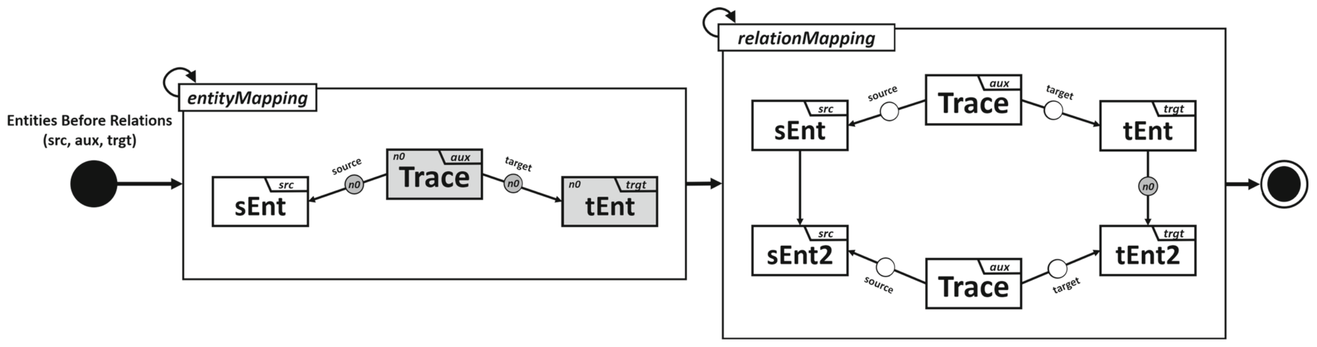


Fig. 2 *Entity-before-Relation* design pattern with explicit trace elements

Henshin. This pattern states that, when transforming a model into another model, rules mapping entities should be executed before ones mapping relations.

In DelTa, color coding is used to indicate the elements to be preserved (in white), elements to be created (in gray), and elements to be deleted (in black) in a transformation. A model transformation design pattern consists of participants and their scheduling. **Participants** represent rule templates that shall be implemented in a concrete model transformation. In the pattern *Entities before Relations* (E-R), there are two participants: *entityMapping* and *relationMapping*. In a participant, entities and relations play a **role**. *entityMapping* indicates that an entity `sEnt` has to be mapped if an entity `tEnt` was not previously created from it. Negative application conditions (NAC) can exist in any participant. It indicates the pattern that should not be found by the concrete transformation rule implementing it. Elements considered by the NAC are labeled by *n0* in DelTa. The second participant *relationMapping* states that if two source entities `sEnt` and `sEnt2` were mapped into two target entities `tEnt` and `tEnt2`, a relation between `sEnt` and `sEnt2` can be mapped into a relation between `tEnt` and `tEnt2`, if this was not done before. Note that both rule templates use the auxiliary type to keep track of the previous mapping by means of the `Trace` element. As shown in Fig. 2, DelTa also allows to express the rule **scheduling** scheme in the form of edges between the rule templates. For the E-R pattern, the *entityMapping* participant must precede the *relationMapping* participant. We refer the reader to [11] for the complete description of design patterns in DelTa.

2.3 Design pattern detection

Up to date, design patterns were primarily used for model transformation writing [11,24]. To the best of our knowledge, few research contributions targeted their detection. In a previous work, we proposed an approach to detect patterns in transformations [28]. Design patterns are encoded into rules and transformations into facts on which the rules are applied. Although the detection results were very encouraging, this approach is not generic enough to be applicable to any pattern. Indeed, while the mapping of transformations into fact sets is automated, the detection rules for each pattern have to be written manually. The detection of design pattern was also mentioned in [24]. In this study, the authors defined detection criteria for each pattern to be applicable manually by the authors to check which patterns are used in transformations. The criteria are given in natural language at a high-level of abstraction.

Outside the model transformation community, there is a large body of work on design pattern detection, especially in object-oriented programs [1]. The detection strategies differ in many ways. One difference is the intermediate representation of the code used to perform the detection. The most-used representation is the Abstract Syntax Tree (AST) [25,40,47] or the Abstract Syntax Graph (ASG) [12,32]. Some strategies map the code into high-level graph representation such as UML [36,46]. Other representations include, among others, matrices [10,44], Prolog assertions [21,34], and strings [20].

Another difference between the strategies is the detection technique. Query-based detection with mainly SQL is used, for example, in [4,38]. The authors use SQL queries to represent and detect the design patterns. This methodology is limited to the structural patterns. Some techniques use a combination of metrics and structural relations to find patterns [9,45]. Another alternative is to use ontologies as in [2,27].

One of the most important limitations of the above-mentioned techniques is the lack performance when the input programs are very large. Attempts have been made in [3] and [14] to improve the detection efficiency. Their idea is to reduce the search space by removing entities that obviously do not participate in an occurrence of design pattern according to expected metrics values. The detection performance issues are experienced in many domains. For example, in bioinformatics, pattern detection is also used to locate different genes in long DNA sequences. To this end, powerful solutions were adopted, such as vectorial algorithms [7,30], automata simulation [15], and dynamic programming alignment [31,41]. Yet, these solutions cannot be applied directly to our problem since we are dealing with complex structures and not strings. Inspired by these algorithms, Kaczor et al.[20] chose to represent object-oriented programs, as well as design patterns, as strings. This allowed the use of string-matching algorithms to detect pattern occurrences with a good accuracy.

Finally, an important property of a detection strategy is the genericity with respect to the patterns to detect. Most of the existing work requires to write a specific code for each pattern, being queries [38], structural rules [47], or pattern-matching rules [26]. There are few strategies that provide the description of the patterns as an input of the detection algorithm such as in [20,44].

Overall, the aim of the approach proposed in this paper is to detect design patterns in model transformations efficiently and without the need of writing specific detection code for each pattern. To address time and memory consumption issues, we rely on a string-matching algorithm proven to be efficient for large genomic databases. Additionally, the fact that the patterns to detect are specified as strings to match in these algorithms eliminates the need to write new code for each pattern.

3 Design pattern detection by string matching

String matching techniques and algorithms are popular in bioinformatics, given they are highly scalable in terms of input size and performance [30,35]. Kaczor et al. [20] showed the benefits of this approach by applying them to detect design patterns in source code. In this paper, we adapt this technique to detect design patterns in model transformation.

Figure 3 shows an overview of our approach separated into three phases. A key aspect of the string matching technique is to represent input artifacts as string sequences. Given a model transformation implementation and a set of design patterns, our goal is to find instances of the design patterns in the transformation. To automate the process, we consider a rule of the transformation (e.g., `createTransition` in Fig. 1) and a participant of a given design pattern (e.g., `entityMapping` in Fig. 2) as inputs. The first phase encodes both inputs as strings following an identical encoding strategy. The choice of string representation will have a direct impact on the performance of the detection algorithm and is, therefore, a crucial step.

The second phase detects the occurrence of the participant string in the rule string using a string matching algorithm. A naive detection algorithm would traverse all elements in both structures. Each element in one would be compared with all elements in the other to find the matching element structurally and the matching properties. To resolve this combinatorial problem, previous approaches using a bit-vector algorithm (like in [20]) have shown to be an efficient solution by bounding the number of vector operations independently from the length of the structures. We, therefore, employ a bit-vector algorithm to match the encoded strings. This outputs all the instances of the participant string in the rule string.

After finding all instances of the participants of the design pattern in all the rules of the model transformation separately, the last phase recovers the complete design pattern instances inside the model transformation. While the first two phases are automated in our approach, the recovery phase is performed manually to consider the scheduling aspect of the transformation and the design pattern.

3.1 Encoding phase

The goal of the encoding phase is to represent a rule and a participant each in a unique circular string to employ a bit-vector algorithm to match them. We first transform the input artifacts into Eulerian directed graphs. Then, we derive the optimal Eulerian circuit of each graph. Finally, we encode the circuit as a unique circular string.

3.1.1 Rules and participants as Eulerian directed graphs

It is natural to represent model transformation rules as graphs. For example, the graph transformation paradigm defines rules explicitly as graphs, such as in Henshin (see Sect. 2.1). Other model transformation paradigms, like ATL [18], can also be represented as graphs [39]. Therefore, without loss of generality, we consider model transformation rules as directed graphs for this work. Following the Henshin representation in Fig. 4, the *rule graph* consists of nodes and edges. Nodes and edges are typed by metamodel types, e.g., `ExpressionStatement` or `expression`, and attributed with an action, e.g., preserve, create, delete, forbid. Nodes and edges may be typed by special types, such as `Trace`, that are not found in the input/output metamodel of the model transformation, but used as auxiliary structures to facilitate the transformation.

Similarly, we represent design pattern participants as directed graphs. We follow the notation used by DelTa, with the difference of explicitly representing NACs like in Fig. 5. The nodes and edges in the *participant graph* are typed by the role name, e.g., `sEnt`, and attributed with an action. DelTa also employs built-in roles, such as `Trace`. We consider these nodes in our graph representation as *fixed nodes* for detection purposes.

It is important to convert the directed graph into a unique string representation of rules and participants. Furthermore,

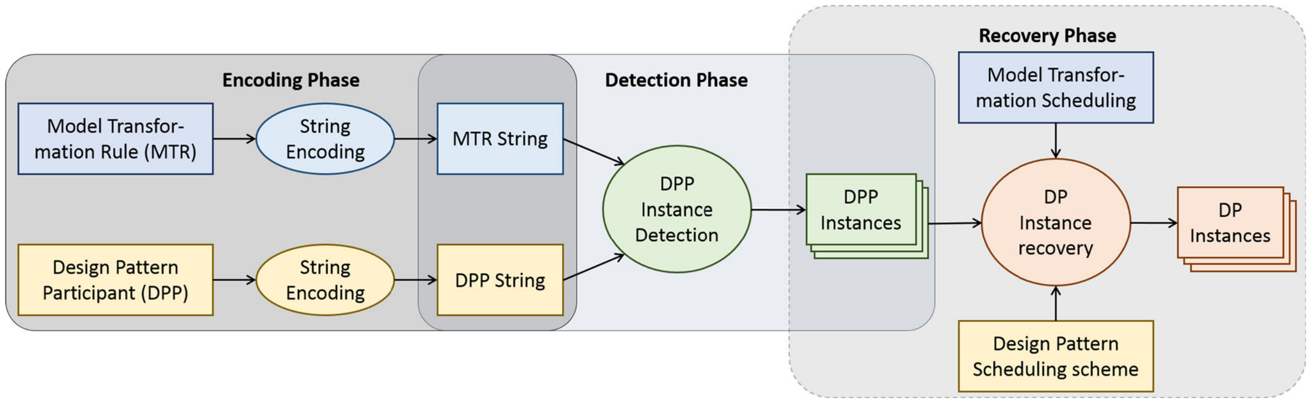


Fig. 3 Overall process of the approach

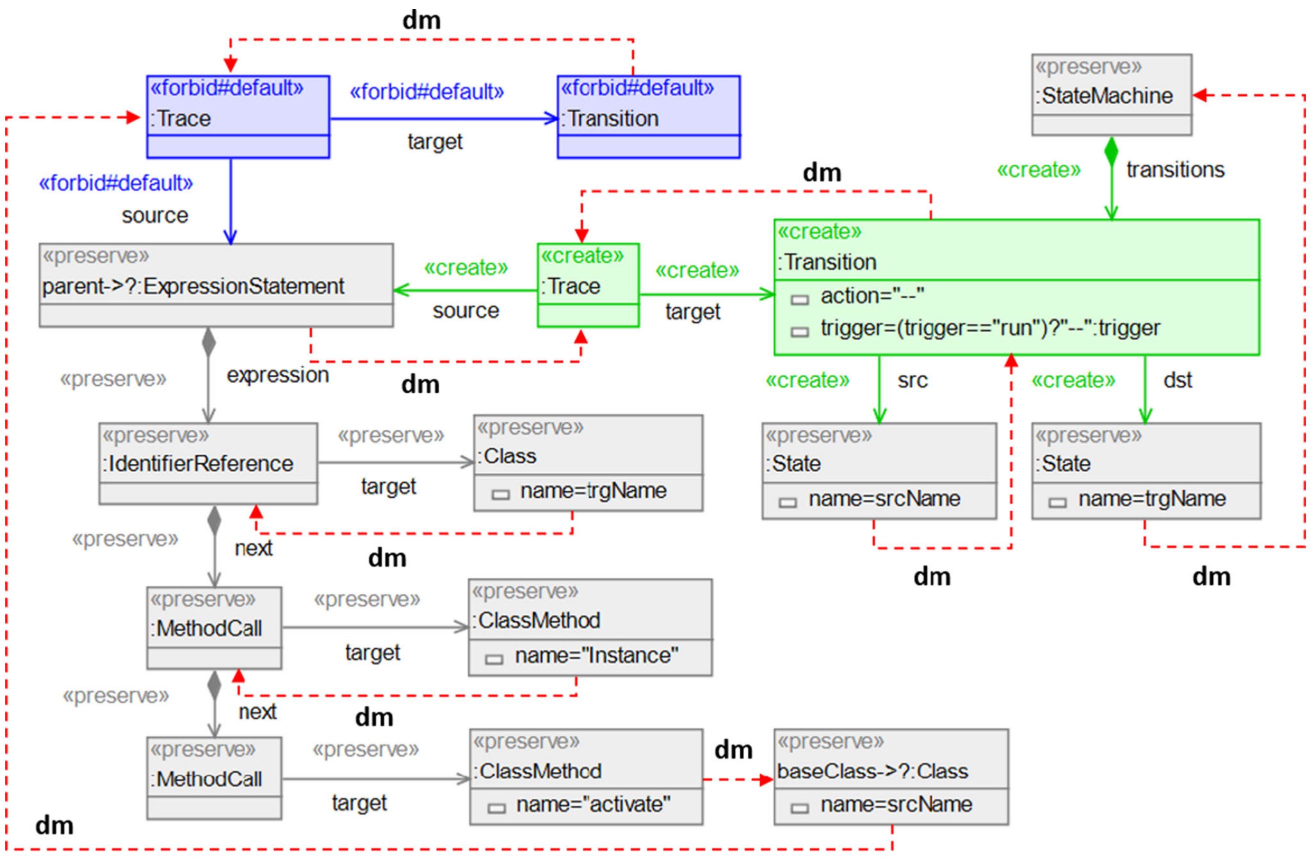
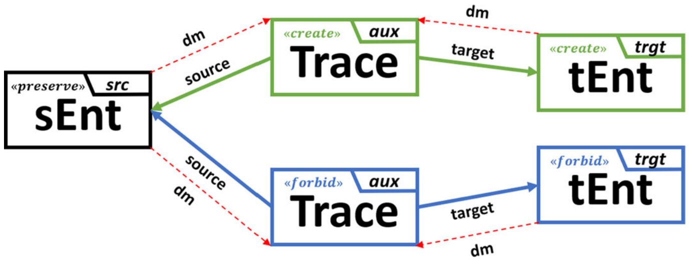


Fig. 4 `createTransition` rule as directed graph

Fig. 5 Participant graph for the `entityMapping` participant of the E-R design pattern



the strings must be circular so that a participant string may match any part of the rule string (and vice versa). We can achieve this by computing the minimum Eulerian circuit of each graph [20]. However, the directed graphs we obtain are not necessarily Eulerian, i.e., they do not contain a Eulerian circuit: a cycle that uses every edge of the graph exactly once. Following the method employed in [20] for class diagrams, we transform[1] the participant and rule graphs into unweighted directed Eulerian graphs. A graph is Eulerian if and only if every node is balanced: has equal in- and out-degrees. Therefore, we consistently add a *dummy edge* between unbalanced node. They are represented by dashed arrows in Figs. 4 and 5. Furthermore, in the case of an isolated node (like `baseClass` in Fig. 5), we add dummy edges to ensure its degree is at least two. There are many ways of making a graph Eulerian. The only requirement is to use the same strategy to make the rule and participant graphs Eulerian.

3.1.2 Computing the rule and participant strings

Given that the rule and participant graphs are Eulerian, the optimal solution to the Chinese postman problem gives a Eulerian circuit [42]. This circuit starts and ends at the same node and traverses each edge exactly once: listing the nodes and edges, we obtain a Eulerian trail. The circuit can start from any node, though our experiences have shown that starting with a node connected to a dummy edge may reduce the performance. Then, we transcribe the trail into a string representation. Note that, unlike edges, a node may appear more than once in the trail. This string is circular, and a graph may have many possible Eulerian trails. However, it is critical that the trail for the rules and participants are computed using the same strategy. Thus, we chose one strategy to obtain a unique trail for both. When traversing the graph, multiple trails are possible when a node has at least two outgoing edges. This is where we opt for a strategy that prioritizes the order in which to traverse edges. In our case, we give priority to non-dummy edges over dummy edges. Then, we base the order depending on the action of the edge/node: preserve, create, delete, forbid in this order. Finally, if a choice is available, we follow the alphabetical order of the values (e.g., `"source"` is chosen before `"target"`). The exact strategy is not relevant as long as it is the same strategy used for both rules and participants graphs to compute a unique Eurlerian circuit.

To represent the nodes and edges of the trail as string, we inspired ourselves from the string representation used for class diagrams in [20]. However, their approach is too simple to tokenize model transformations: we need to not only represent the connectivity of the graph, but also encode semantics specific to model transformations. Figure 6 depicts the string encoding a Eulerian subgraph of a rule and a participant. We encode each node into a string token following the format $[id, type, action, fixed]$ for participants and $[id, type, action]$ for rules. We encode edges into string tokens similarly but between parentheses. We refer to these strings as *role token* and *element token*, respectively. id uniquely identifies the element. $type$ is the type of the element. For the participant string, we use the name of the element as type, like `sEnt`. For the rule string, we use the name of the metamodel element as type, like `Transition`. In case of fixed nodes and edges for the participant string, like `Trace`, we set $fixed = 1$ otherwise 0. Fixing nodes enforces the detection mechanism to match exactly the same element type. For example, in Fig. 4, the *Transition* object should not match the *Trace* role because the transition is from the same metamodel as its *State* objects neighbors. The $action \in \{c, d, p, f, \_\}$ represents the create, delete, preserve, or forbid actions, respectively. The underscore is used for dummy edges, for which no action is required, as they are not part of the original rule or participant. The graph is encoded by concatenating the tokens of the source node of an edge, followed by the edge, and then by the target node of the edge in the trail.

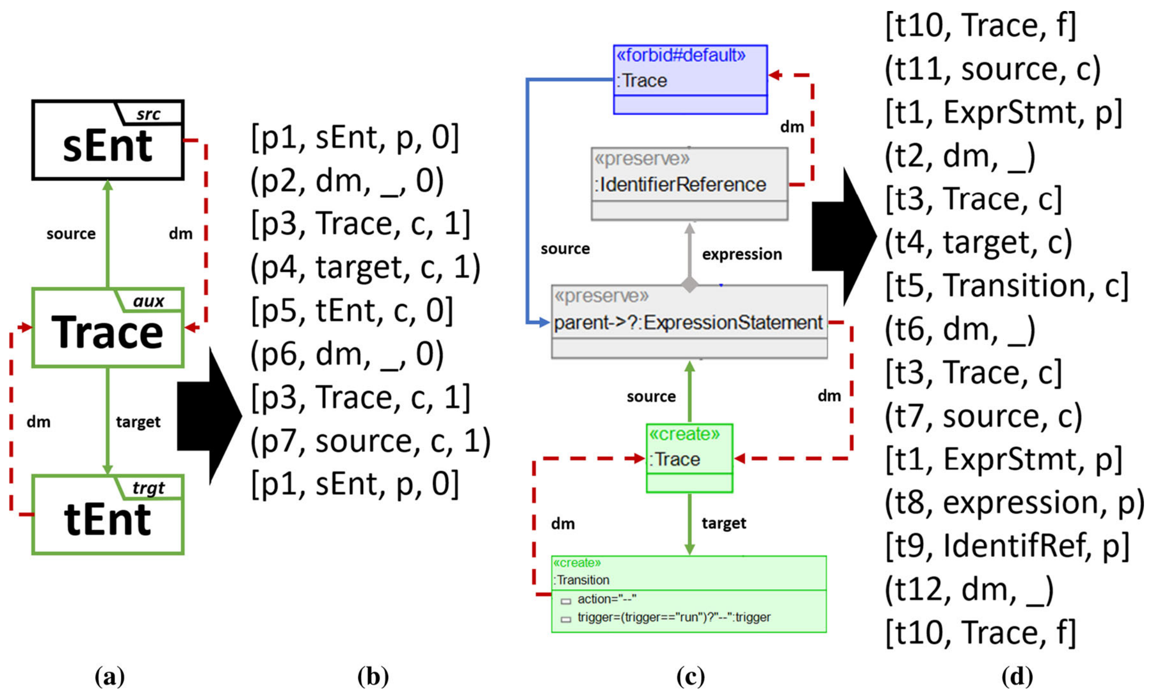
3.2 Detection phase

The goal of the detection is to find all instances of a design pattern participant, encoded in the participant string, that are present in the rule string. Bit-vector algorithms are known to be very efficient to approximate word matching in strings [30]. This requires to encode the strings into bit vectors. Kaczor et al. [20] have derived an iterative bit-vector processing algorithm to find exact and approximate occurrences of an object-oriented design pattern in a class diagram. Like for object-oriented patterns, this algorithm is also suitable for model transformations where pattern participants are encoded into short strings that can be found in longer strings representing transformation rules. However, the algorithm presented in [20] only deals with classes and associations in structural patterns. Since rule-based model transformation is a declarative paradigm, the original algorithm cannot be applied directly. Therefore, we significantly modify it to tackle the complexity that that model transformations and their design patterns provide.

3.2.1 Characteristic bit-vector

To employ a bit-vector algorithm, we represent the participant and rule strings into vectors of bits. Let t be any a token appearing in an encoded string $s = s_1 \ldots s_n$, where n is the length of the trail. We define the *characteristic vector* of t,

[1] The algorithm is similar to the one presented in [20] based on the transportation simplex [8].



[p1, sEnt, p, 0]
(p2, dm, _, 0)
[p3, Trace, c, 1]
(p4, target, c, 1)
[p5, tEnt, c, 0]
(p6, dm, _, 0)
[p3, Trace, c, 1]
(p7, source, c, 1)
[p1, sEnt, p, 0]

[t10, Trace, f]
(t11, source, c)
[t1, ExprStmt, p]
(t2, dm, _)
[t3, Trace, c]
(t4, target, c)
[t5, Transition, c]
(t6, dm, _)
[t3, Trace, c]
(t7, source, c)
[t1, ExprStmt, p]
(t8, expression, p)
[t9, IdentifRef, p]
(t12, dm, _)
[t10, Trace, f]

Fig. 6 **a** Eulerian trail of a part of the `EntitiesMapping` participant. **b** The corresponding participant string encoding. **c** Eulerian trail of a part of the `createTransition` rule. **d** The corresponding rule string encoding

denoted by $\mathbf{t} = t_1 \ldots t_n$, as

$$t_i = \begin{cases} 1 \text{ if } s_i = t \\ 0 \text{ otherwise.} \end{cases}$$

Conceptually, it represents the positions of the token in the participant and rule strings. For example, in Fig. 6(d), the characteristic vector of `t1` and `t4` is:

$$\begin{aligned} \texttt{[t1,ExprStmt,p]} &= 001\,\underbrace{0000000}_{7}\,100 \text{(i.e., two occurrences)} \\ \texttt{(t4,target,c)} &= \underbrace{00000}_{5}\,1\,\underbrace{0000000}_{7} \text{(i.e., one occurrence)} \end{aligned}$$

Characteristic vectors are sequences of bits on which we can apply standard bit operations: bit-wise logical AND/OR operators and left/right shifts. A notable property of the way we construct the participant and rule strings is that tokens always appear in the same order modulo a shift. Thus, we consider that characteristic vectors are circular. For a characteristic vector $\mathbf{t} = t_1 t_2 \ldots t_{n-1} t_n$, the left/right shifts are defined by shifting the position of the bits circularly by one to the left or the right as:

$$\underset{\text{Left shift by 1}}{\mathbf{t} \ll 1 := t_2 \ldots t_n t_1} \quad \underset{\text{Right shift by 1}}{\mathbf{t} \gg 1 := t_n t_1 \ldots t_{n-1}}$$

3.2.2 Matching participants in rules

We rely on characteristic vectors to find the tokens of a transformation rule that play the role of a design pattern participant. Our algorithm, iteratively reads triplets of tokens ⟨*node*, *edge*, *node*⟩ in the participant string. It then identifies all possible matching triplets in the rule string by performing conjunctions and shifts. To ensure unification, we reuse tokens that are already matched in a triplet. Algorithm 1 separates the algorithm into two steps described in the subsequent algorithms.

Algorithm 1: DetectPatternInstances(participant, rule)

```
1 candidates ← FindCandidates(participant, rule)
2 maxSize = max |candidates[c]| ∀c ∈ candidates.keys()
3 matches ← ∅                                    // empty dictionary
4 foreach partTripl ∈ candidates.keys() do
5     pEdge, pBefore, pAfter ← partTripl.unpack()
6     matches[pEdge], matches[pBefore], matches[pAfter] ←
      Array(maxSize)
7 return Detect(candidates, 0, matches)
```

The first step is to find potential candidate rule tokens for each participant token. Algorithm 2 takes as input a participant and a rule strings to output a dictionary pairing each participant triplet to corresponding rule triplets. For exam-

Algorithm 2: FINDCANDIDATES(participant, rule)

```
candidates ← ∅                                        // empty dictionary
foreach partToken ∈ participant.getEdges() do
    tripl ← (partToken, partToken.before(), partToken.after())
    candidates[tripl] ← ∅
    foreach edgeTok ∈ rule.getEdges() do
        if not MATCH(partToken, edgeTok) then continue
        before ← ∅, after ← ∅                         // ordered sets
        partToken ≫ 1
        foreach afterTok ∈ rule do
            conj1 ← afterTok ∧ partToken
            if conj1 ≠ 0 and MATCH(partToken.after(), afterTok) then
                after ← after ⊎ {afterTok}            // append
                conj1 ≪ 2
                foreach beforeTok ∈ rule do
                    conj2 ← beforeTok ∧ conj1
                    if conj2 ≠ 0 and MATCH(partToken.before(), beforeTok) then
                        before ← before ⊎ {beforeTok}  // append
        for i = 0...MIN(|before|, |after|) do       // discarded if before or after are empty
            candidates[tripl] ← candidates[tripl] ∪ {(edgeTok, before[i], after[i])}
return candidates
```

ple, suppose we wish to find the candidates of the triplet ⟨p3, p7, p1⟩ in Fig. 6. We start by searching matches for the edge, i.e., partToken = p7 on line 6. The MATCH function matches a token p of the participant string to a token t of the rule string following these rules:

1. If p is a node then t must be a node. If p is an edge then t must be an edge.
2. $t.action = p.action$
3. If $p.fixed = 1$ then $t.type = p.type$

In this case, p7 matches t7 and t11. The shifts and conjunctions on lines 8–17 ensure that the nodes before and after also match; otherwise, the edge is discarded. The following shows how Algorithm 2 finds the rule triplet ⟨t3, t7, t1⟩ to match our participant triplet from the characteristic vectors:

$$\begin{aligned}
\textbf{edgeTok} = \textbf{t7}: &\quad \textbf{p7} \gg 1 = 001\overbrace{0000000}^{7}100 \\
\textbf{beforeTok} = \textbf{t3}: &\quad \textbf{p3} \gg 2 = 0010001000100 \\
\textbf{conj1}: &\quad (\textbf{p7} \gg 1) \wedge (\textbf{p3} \gg 2) = 0010000000100 \\
\textbf{afterTok} = \textbf{t1}: &\quad \textbf{p1} = 0010000000101 \\
\textbf{conj2}: &\quad (\textbf{p7} \gg 1) \wedge (\textbf{p3} \gg 2) \wedge \textbf{p1} = 0010000000100
\end{aligned}$$

On the last equation, we see how **conj2** rules out the candidate matching triplet ⟨t10, t11, t1⟩. On lines 18–19 of Algorithm 2, we only consider candidates that match the complete triplet of tokens.

Algorithm 3: DETECT(candidates, index, matches)

```
if index > |candidates.keys()| then return matches
partTripl ← candidates.keys()[index]
pEdge, pBefore, pAfter ← partTripl.unpack()
if pEdge.isDummy() and index > 0 then
    return DETECT(candidates, index+1, matches)       // skip this triplet
for i = 0...|candidates[partTripl]| do
    rulTripl ← candidates[partTripl].pop()
    rEdge, rBefore, rAfter ← rulTripl.unpack()
    if matches[pEdge][i] = ∅ or index = 0 then
        matches[pEdge][i] ← rEdge
    if matches[pBefore][i] = ∅ then
        matches[pBefore][i] ← rBefore
    if matches[pAfter][i] = ∅ and (not pEdge.isDummy() or index > 0) then
        matches[pAfter][i] ← rAfter
    if VALID(partTripl, (matches[pEdge][i], matches[pBefore][i], matches[pAfter][i])) or index = 0 then
        DETECT(candidates, index+1, matches)
    else CLEARMATCH(matches, i)                       // discard this match
return matches
```

The candidate matches that Algorithm 2 outputs only consider triplets of tokens. Therefore, we must ensure that the whole participant matches cohesively among triplets. Algorithm 3 detects instances of each role of the participant. It creates a set of matches where a match pairs every role token from the participant string to an element token from the rule string. As shown on lines 4–6, Algorithm 1 initializes the data structure matches as dictionary where keys are individual roles of all participant triplets and values are empty arrays. Algorithm 3 recursively traverses each participant triplet and assigns the corresponding rule triplets. The loop on line 6 can easily be run in parallel since it processes each match of the participant independently. On lines 9–14, when assigning an element to a role, we first check if we have already assigned an element to that role before looking into the candidates output by Algorithm 2. This ensures the unification between the triplets of the same participant. On line 4, if a participant triplet contains a dummy edge, this triplet is skipped since it does not contribute to any role of the participant. The only exception is if the dummy edge is part of the initial triplet we are processing. In this case, we only assign the node before because the node after will be processed by the next triplet.

To illustrate Algorithm 3, Fig. 7 shows how we construct the matches for the participant string in (b) and the rule string in (d) of Fig. 6. Each row shows the current values of the matches dictionary at each iteration of the recursive calls (lines 5 and 16). The values of the iteration column corre-

Triplets	(p2,dm,_,0)	(p4,target,c,1)		(p6,dm,_,0)	(p7,source,c,1)	
	[p1,sEnt,p,0]	[p3,Trace,c,1]	[p5,tEnt,c,0]		[p3,Trace,c,1]	[p1,sEnt,p,0]
Iterations						
1	[t1,ExprStmt,p]					[t1,ExprStmt,p]
2	[t1,ExprStmt,p]	[t3,Trace,c]	[t5,Transition,c]		[t3, Trace,c]	[t1,ExprStmt,p]
3	[t1,ExprStmt,p]	[t3,Trace,c]	[t5,Transition,c]	«skip»	[t3, Trace,c]	[t1,ExprStmt,p]
4	[t1,ExprStmt,p]	[t3,Trace,c]	[t5,Transition,c]		[t3, Trace,c]	[t1,ExprStmt,p]
1	[t9,IdentifRef,p]					[t9,IdentifRef,p]
2	[t9,IdentifRef,p]	[t3,Trace,c]	[t5,Transition,c]		[t3,Trace,c]	[t9,IdentifRef,p]
3	[t9,IdentifRef,p]	[t3,Trace,c]	[t5,Transition,c]	«skip»	[t3,Trace,c]	[t9,IdentifRef,p]
4	[t9,IdentifRef,p]	[t3,Trace,c]	[t5,Transition,c]		~~[t3, Trace,c]~~	~~[t9,IdentifRef,p]~~

Fig. 7 Iterations of the Algorithm 3 showing matching nodes

spond to the values of the `index` variable. At iteration 1, the initial triplet ⟨`p1`, `p2`, `p3`⟩ contains a dummy edge. Therefore, we only match `p1`. Algorithm 2 returns two possible candidates, `t1` and `t9`. They will be part of different matches in the match sets (line 2). Let us first consider the case of `t1`. At iteration 2, we process the next participant triplet to which we assign `t3` and `t5` to `p3` and `p5`, respectively. At iteration 3, the next triplet contains a dummy edge, so it is ignored. At iteration 4, we notice that `p3` and `p1` are already assigned an element for the last triplet. On line 15, we verify the integrity of this match making sure they are well-formed according to Algorithm 2. Therefore, the function VALID verifies that the rule triplet matched exists in the `candidates` dictionary. Since it is the case, this match is an occurrence of the participant in the rule.

Let us now consider the second partial match where `p1` is assigned to element `t9`. The process is similar for iterations 2 and 3. At the last iteration, the integrity check fails since there is no edge from `t3` to `t9` that matches `p7`. This match is therefore discarded and line 17 only outputs the first match we described. The function CLEARMATCH removes the values at index `i` in all the arrays in `matches`.

3.3 Recovery phase

The detection phase outputs the rule elements matched to each role of a design pattern participant. Therefore, we obtain the set of rules that correspond to each participant. To reduce false positives, the recovery phase identifies the sets of rules corresponding to all the participants of a design pattern. Furthermore, the detection phase is based on the structure of individual participants.

To recover the complete design pattern instances, we must consider the global description of the design pattern and not only one of the participants. For instance, the scheduling scheme indicates whether the detected individual participants form a valid instance of the design pattern. DelTa offers several scheduling schema to order participants, such as sequencing, conditional bifurcation, choosing any participant, or not enforcing an order. Some model transformation languages, like Henshin, express rule ordering explicitly using similar schema. Others, like ATL, keep the order of the rules implicit. Therefore, a deep knowledge of the semantics of the transformation language is needed to decide whether the detected rules are executed in the order prescribed by the design pattern. There are other components, such as constraints and actions on attributes of rule elements or profiles/tags in DelTa participants, that should also be considered to recover complete design pattern instances. The mapping between the description of these additional elements (scheduling and other components) and the concrete transformations is not straightforward, because many options can be used for their implementations. This is why in the current state of our research, we perform the recovery phase manually.

To illustrate the recovery phase, let us consider the case of the E-R pattern. Suppose that, during the detection phase, we found two instances E1 and E2 of the *entityMapping* participant, and two instances R1 and R2 of the *relationMapping* participant. The pattern suggests that the former should precede the latter (see Fig. 2). After analyzing the scheduling scheme of the model transformation, we identify that E1 is executed before R1, but E2 is executed after R2. Therefore, we conclude that the E-R pattern has only one valid instance E1-R1.

4 Detecting other design pattern specifications

Typically, the solution described by a design pattern is a generic template used as a guideline to understand how to implement the design pattern. In practice, it happens often that we implement a design pattern in a different way than the template solution. Therefore, a design pattern detection approach should consider not only the exact template solution provided in the definition of the design pattern, but should also detect other implementation variants that may be encountered in practice. Furthermore, existing model transformations may only partially implement a design pattern. Our detection process in Fig. 3 takes as input the specification of a design pattern participant. However, the participant does not necessarily need to be the complete or exact participant defined in the solution template of the design pattern. It can be an approximation of that participant, that is missing some roles or variant with an alternative structure that still preserves the the global description of the design pattern. Another important aspect to consider is that patterns may use other patterns in their implementation. Patterns can be specified independently, but their detected occurrences can be checked with respect to the conformity with the expected usage relationships. In the remainder of this section, we characterize variants and approximations of model transformation design patterns and show that we can reuse the exact same detection process presented in Sect. 3 to detect them. Moreover, we illustrate some of the common usage relations between the considered patterns.

4.1 Variants

Variants of a design pattern modify the template solution, but they must preserve the pattern description. For example, the observer pattern in object-oriented design has one variant for the pull and another for the push approach. There is no general way of deriving all possible variants of a design pattern.

Many model transformation design pattern propose possible variants (called “variations” in [11,22]). They can take various forms, such as using more roles in a participant, changing the type of a relation, or changing the action of a role. For example, for the E-R design pattern, the relation between the source and target entities can be many-to-one instead of the relation mapping one source to one target entity, as depicted in Fig. 2.

Consider the Henshin rule `createLinks` in Fig. 8. Given a trace mapping objects, if on one side of the trace two objects are linked, the rule links the two objects on the other side of the trace and maps both links by a reference. It implements a variant of the participant *relationMapping* in the E-R design pattern. First, we note that the single relation between `sEnt` and `sEnt2` is implemented by means of a `WLink` object with `wMembers` and `wTarget` associations. Secondly, we note that a single relation between `tEnt` and `tEnt2` is also implemented by means of a `WLink` object. Thirdly, this object has an additional `eStructuralFeature` association that is not required by the design pattern participant. This rule implements a typical variant of the design pattern where a participant or role of the design pattern is implemented by more than one element.

4.2 Approximations

Sometimes, a design pattern is not implemented in its entirety in a transformation [28]. Although the underlying implementation may match part of a design pattern description, some roles of the participants may be missing. This typically has a negative effect on the quality of the transformation, such as reuse, cohesion, or coupling [22]. We consider an implementation as an approximation of a design pattern if it is missing or misusing roles.

Consider the Top-Down design pattern that decomposes a transformation into phases based on the target model composition structure [11]. It is composed of two participants, *formerPhase* and *latterPhase*, sequenced in this order. The *formerPhase* states that a rule shall create a `tContainer` entity in the target metamodel. This entity shall be traced to the corresponding `sContainer` entity in the source metamodel that contains an `sComponent` entity. The *latterPhase* (represented in Fig. 9) states that a rule shall create a `tComponent` entity contained in `tContainer`. This entity shall be traced to the corresponding `sComponent` entity in the source metamodel.

Figure 10 shows a Henshin rule stating that, if a `GenClass` is associated with an `EClass` containing an `EAttribute`, then the `GenClass` should contain a `GenFeature` and associate it to the `EAttribute`. Intuitively, we understand that it corresponds to the *latterPhase* participant of the Top-Down design pattern. However, we note differences between the two structures. To be a perfect match, the rule is missing a relation from `GenClass` to the forbidden `GenFeature` to complete the NAC of *latterPhase*. Furthermore, the participant requires using a `Trace` element to link the created `GenFeature` to the `EAttribute`. According to our representation, `Trace` would be a fixed role. However, this role is played by the `ecoreFeature` link and the `Rel` element. Since the rule reuses an element from the metamodel to implement a fixed role, they would not match. We consider this situation as a misuse of a role. Therefore, for these two reasons, this rule is an approximation of the `latterPhase` participant in the Top-Down design pattern.

Design pattern participant variants or approximations can be derived manually to ensure their conformance with the corresponding design pattern. Some types of approxima-

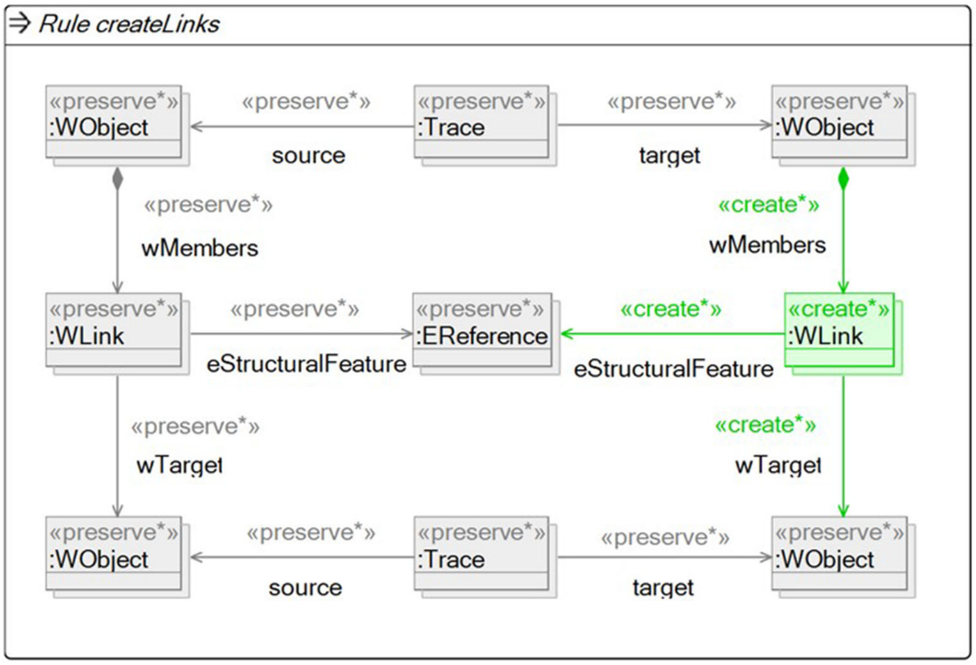


Fig. 8 Instance of a variant of the `relationMapping` participant

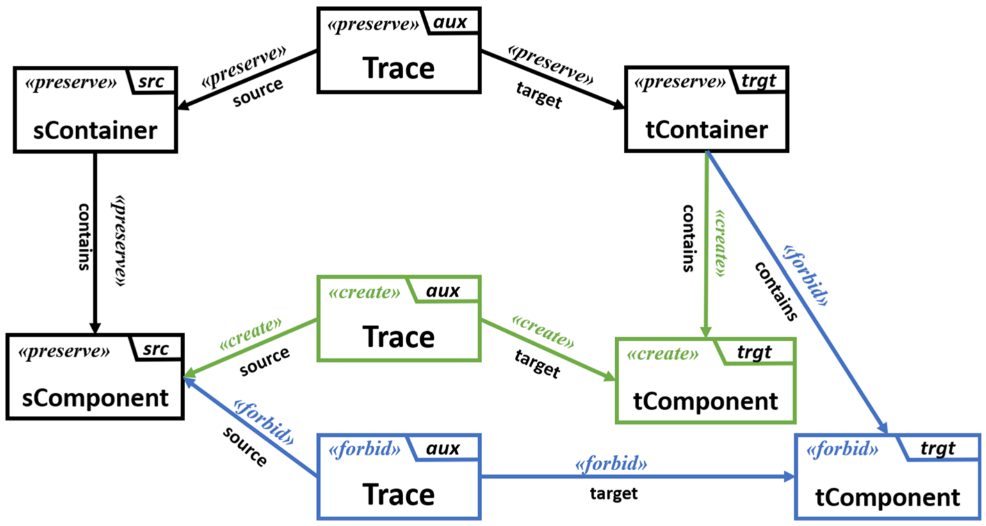


Fig. 9 *latterPhase* participant of the Top-Down design pattern

tion can be derived automatically by, e.g., removing some roles. Some variants can be derived automatically once the transformation language is fixed. Kaczor et al. [20] define four different types of approximations for object-oriented design patterns, though they acknowledge there are more. These can be summarized in two major categories: when the role of a participant is missing or when a role is misused. Our approach can automatically detect model transformation design pattern approximations that fall under the first category. Given the Eulerian trail of a participant, we can automatically derive approximations by removing elements of the trail. The bit-vector algorithm remains the same. Therefore, for some approximations, we do not require to manually specify them (as opposed to [28]). The second category of approximations covers cases when the use of a role deviates from the original specification of the design pattern. Kaczor et al. propose ways to modify their algorithm to handle these cases. In some cases, they acknowledge that the approximation must be specified manually. Our detection approach can recover variants and approximations of a design pattern as long as they are explicitly defined. We apply the process in Fig. 3 where the input is a variant.

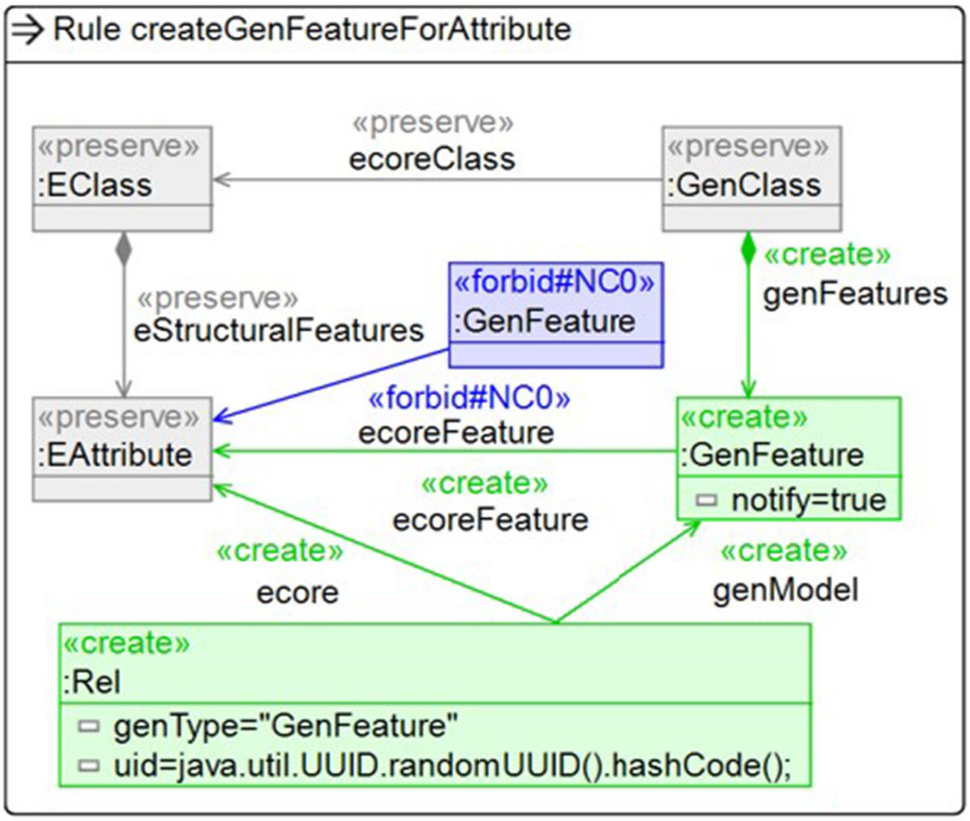


Fig. 10 Approximation of the `latterPhase` participant in Fig. 9

4.3 Co-occurrence of design patterns

Looking at the design patterns catalog, we notice that the structure of the participants of some design patterns are present in participants of other design patterns. This suggests that some design patterns are related with others. As outlined in Fig. 11, we distinguish at least three types of relations. The relations we present are the result of a critical analysis of all catalogued design patterns in [11,22] complemented with the notion of design pattern relations introduced in [24].

When analyzing existing model transformation implementations, we identified variants of design patterns used in practice. As defined in Sect. 4.1, all variants of a design pattern share the same goal, but differ in the structure of their participants. A predominantly used design pattern in practice is the *Auxiliary Metamodel*. *"This pattern proposes to create an auxiliary metamodel for temporary elements used in the transformation that do not belong to either the source or target metamodels"* [22]. We have noted two main variants. The *Trace type* is a variant of this pattern that relies on an entity to keep a trace correspondence between entities of different metamodels. This variant is mostly used in exogenous transformations. However, for in-place transformations, we noted that developers often rely on a specific entity (from the same or another metamodel) to temporarily mark entities to be transformed, such as the rule in Fig. 12 taken from the `Ecore2GenModel` transformation.[2] The generic solution of the *Auxiliary Metamodel* pattern presents three possible variants of how it can be used when creating a new correspondence to an existing entity, deleting the corresponding entity, or modifying an attribute of the corresponding entity. We, therefore, obtain the six variants mentioned in Fig. 11.

The *Object Indexing* pattern uniquely indexes an entity in a first participant to enable its efficient lookup in a second participant. We identified that it comes in two variants: one using flag to mark an entity to index and another using a temporary marker pointing to the entity. The latter variant essentially uses the *Marker type* variant of the *Auxiliary Metamodel* pattern to index an entity. For this relation, if a pattern A *uses* a pattern B, we will often find the structure of a participant of B subsumed in a participant of A. For example, the *Visitor* pattern requires to mark an entity as already visited. This essentially uses an instance of the *Object Indexing* pattern. Another example of this relation is that the *E-R* pattern (see Fig. 2) presents an instance of the *Trace type* variant of the *Auxiliary Metamodel* pattern. More precisely,

[2] https://www.eclipse.org/henshin/examples.php?example=ecore2genmodel.

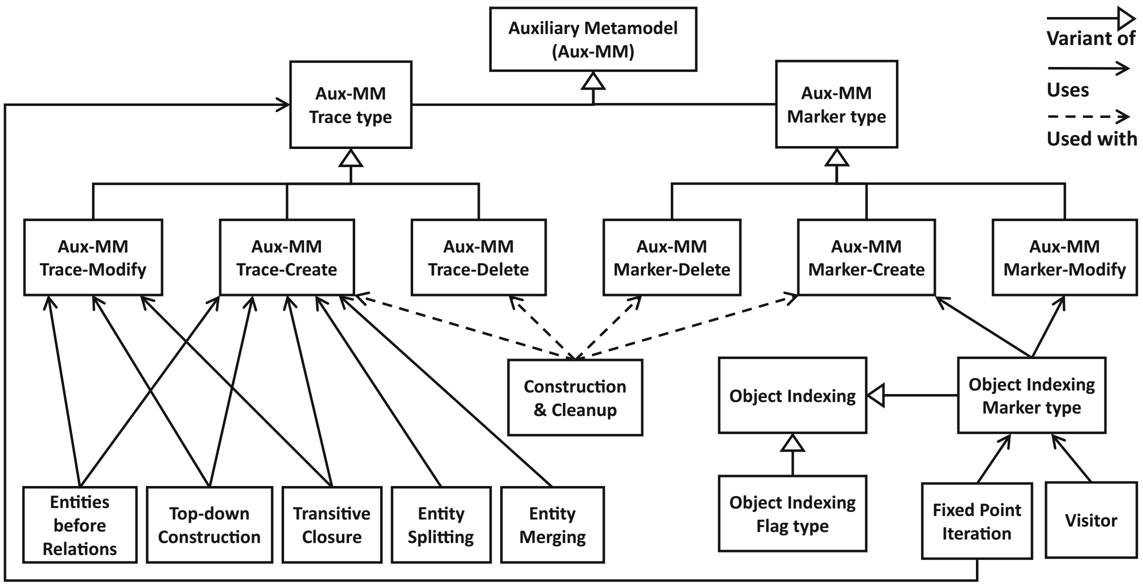


Fig. 11 Relations between some design patterns

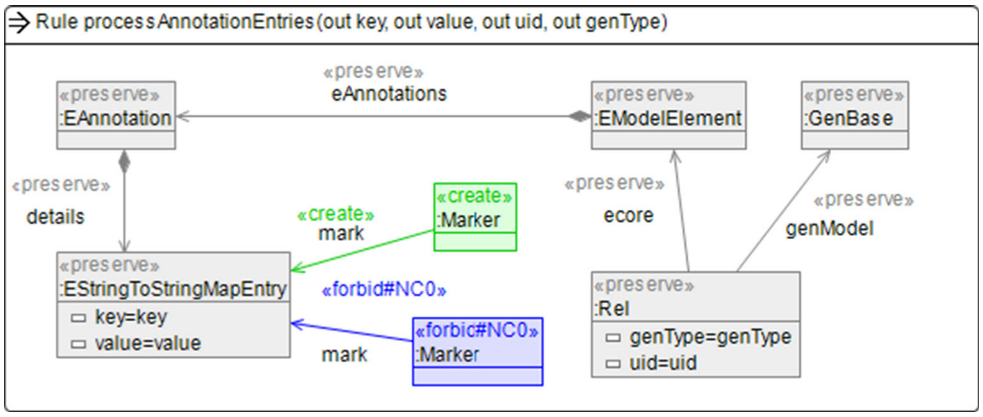


Fig. 12 A rule using markers

the *entityMapping* participant uses the create variant and the *relationMapping* uses the modify variant.

The third type of relation we identified is when a design pattern is often *used with* another one. For example, the *Construction & Cleanup* pattern often creates temporary entities at the beginning of the transformation and deletes them at the end. Therefore, it is often used with the *Auxiliary Metamodel* pattern to achieve its goal.

Although the detection process does not consider explicitly usage relations between patterns, the detected occurrences can be checked to ensure that patterns are used correctly by other patterns. If not, these situations can be considered as potential refactoring opportunities.

5 Evaluation

To evaluate the usefulness and the relevance of our approach, we defined the following research questions:

RQ1: How are design patterns used in model transformations? The goal of this question is to understand if design patterns are actually employed in concrete transformations and in what form they are used (complete, variants, approximations). This also serves as a motivation for our approach.

RQ2: How effective is our approach to detect design pattern variants and approximations? With this question, we evaluate the ability of our approach to detect complete and partial pattern occurrences, whether they are standard patterns or variants. When reporting the results, the number of variants also includes the standard version.

RQ3: What design patterns are typically used in combination with each other? This question allows us to identify relations between design patterns and validate the co-occurrence premise that we describe in Sect. 4.3.

5.1 Setup

In our evaluation, the goal is to analyze quantitatively and qualitatively how our detection approach applies to concrete transformations. We relied on model transformations gathered from three sources as shown in Table 1. The selected transformations are described by the number of rules they contain, as well as by the total number of Henshin nodes and relations, respectively, involved in these rules. We also provide the minimum and maximum number of nodes per rules for each transformation.

The first source we used is the Henshin website.[3] We selected six exogenous transformations. To add more diversity to our transformation sample, we also considered ATL transformations from the ATL Zoo.[4] Although the literature proposes higher-order transformations from ATL to graph transformation [33,39], we were unable to reuse their tool support. Therefore, we limited ourselves to three ATL transformations because we had to reimplement them manually in Henshin following the guidelines of [33,39]. Finally, the third source of transformations is the ReMoDD project reported in [37]. This project contains 10 model transformations that map a secure business process model, expressed in SecBPMN2, to a preliminary architectural UML model enriched with security policies, expressed in UMLsec. The size characteristics of these transformations are much higher than ones of the transformations from the two previous sources. It is worth mentioning that the selected transformations from the above-mentioned sources are exogenous transformations as most of the design patterns are present in exogenous transformation by nature. We did not consider endogenous transformations in these sources.

We considered all 15 design patterns from [11] to detect their instances on our sample of model transformations. However, we found no occurrences of five of them in the transformations we consider. Thus, Table 2 presents the sample of 10 design patterns we selected with a brief description and benefits. This information is extracted from the design pattern catalogs presented in [11,22,24]. The transformation patterns we use come from different categories [22]. The first category deals with rule modularization patterns. It includes the design patterns: *Auxiliary Metamodel*, *Construction & Cleanup*, *Entities before relations* (also called *Map objects before links* in [24]), *Entity Merging*, and *Entity splitting*. These patterns aim at organizing the dependencies and relationships between rules in a transformation. *Unique Instantiation* and *Object Indexing* are two patterns from the optimization category. They are used to improve the execution efficiency at the rule or transformation level. According to Lano et al. [22], *Unique Instantiation* and *Auxiliary Metamodel* can be seen as fundamental patterns which forms a separate category. From the classical/external patterns category, we selected *Visitor*. The latter is heavily used in model-to-text transformations to navigate through a model. Finally, we selected two patterns from the architecture category: *Top-down Phased Construction* and *Fixed-point Iteration*. Both patterns are used to organize systems of transformations at the inter-transformation level to improve modularity and processing capabilities of systems. For more information about the selected patterns, we refer the reader to [23], which presents a survey on how these pattern are actually used in real transformations.

5.2 Design pattern usage in model transformation (RQ1)

To answer the first research question, we considered all the transformations described in Table 1. We performed a manual inspection of these transformations to detect all existing instances of the design patterns in Table 2. For each pattern, one co-author identified its usage in all the transformations and the other two validated the results. When manually detecting design patterns we first identify the rules matching the participants of the design pattern, and then, we verify that the logic of the rule scheduling and attribute constraints conform to the constraints of the pattern. We had to consider complete and approximate patterns, as well as variants. The occurrences we found are presented in Table 3.

A quick glance at the results reveals that the *SecBPMN2-to-UMLsec* transformations contain higher numbers of pattern occurrences than those of the other transformations. This is expected considering the size of these transformations. Among the design the patterns, *Auxiliary Metamodel* is the most used. This was expected as this is fundamental pattern that allow different rules to manipulate the same objects in exogenous transformations. We distinguish between the three variants of the pattern encompassing the *Trace* and *Marker* versions (see Fig. 11). The *Create variant* is consistently used in all transformations because they are all creating elements in the target model. The *Modify variant* is found when a rule updates elements it matches. Since we found many instances of the *Create variant*, we expected to find many instances of the *Modify variant* as well. That is because the *Modify variant* preserves the existence of the elements created. Usually, the *Auxiliary Metamodel* variants are used together. We

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

[4] https://www.eclipse.org/atl/atlTransformations/.

Table 1 Selected model transformations

Sources	Model transformations	# rules	# nodes	# relations	# min nodes/rule	# max nodes/rule
Henshin Website	ecore2genmodel	8	55	59	4	6
	ecore2rdb	2	38	49	13	25
	ecore2uml	2	29	40	9	20
	java2statemachine	13	77	59	2	15
	wrap copy	3	20	19	4	9
	classDiagram2relationSchema	3	16	14	5	6
ATL Zoo	book2publication	5	20	16	2	6
	ATOM2RSS	17	73	59	2	6
	RSS2ATOM	6	28	27	3	6
SecBPMN2-to-UMLsec	GR2_classOperation	31	262	267	6	10
	GR3_associations	35	640	687	15	20
	GR4_dependency	28	393	440	12	15
	GR5_abac	15	131	131	7	9
	GR5_secureDependency	33	295	292	5	10
	GR6_secureLinks	29	323	374	2	15
	GR7_secureLinks2	30	426	510	11	16
	GR7_secureLinks3	24	396	444	15	18
	GR7_secureLinks4	17	240	261	6	18

Table 2 Selected design patterns

Patterns	Summary	Benefits
Auxiliary metamodel	This pattern proposes to create an auxiliary metamodel for temporary elements used in the transformation that do not belong to either source or target metamodels	Improves clarity, flexibility, and modularization
Construction & cleanup	This pattern structures a transformation by separating rules which construct model elements from those which delete elements	Improves modularity
Entities before relations	This pattern is used in exogenous transformations to encode a mapping between two languages. It creates the elements in a language corresponding to elements from another language and establishes traceability links between the elements of source and target languages	Improves debugging, error localization, and avoids circularity in processing
Entity merging	Two rules each create/update elements of the same target entity type, using different source entity types or elements	Organizes instance data integration
Entity splitting	Two rules (with disjoint application conditions) map (instances of) the same source entity type to instances of different target entities	Distinguishes cases in source data
Fixed-point iteration	Pattern for representing a "do-until" loop structure. It solves the problem by modifying the input model iteratively until a condition is satisfied	Organizes fixed-point processing
Object indexing	All objects of an entity are indexed by a primary key value, to permit efficient lookup of objects by their key	Reduces syntactic complexity
Top-down phased construction	This pattern decomposes a transformation into phases or stages, based on the target model composition structure. These phases can be carried out as separate subtransformations, composed sequentially	Organizes processing
Unique instantiation	This pattern makes sure the created elements in a rule are unique and eliminates redundant creation of the same element by reuse	Avoids duplicating instances
Visitor	This pattern traverses all the nodes in a tree and processes each entity individually	Improves extensibility

found only few instances of the *Delete variant* because not all transformations intend to remove elements. This is expected as we are dealing with exogenous transformations.

The *Construction & Cleanup* pattern is based, in general, on the combination of the *create* and *delete* variants of the *Auxiliary Metamodel* pattern. Most of developers use the *delete* variant in a *Construction & Cleanup* process. Thus, the instances we found for this pattern are almost the same as ones we found for the *Auxiliary Metamodel*.

Entity Merging and *Entity Splitting* are mostly used in *SecBPMN2-to-UMLsec* transformations. Similarly, we found a high number of instances of *Entities before Relations* within the third source and very few in transformations of the other sources. The same observations also hold for the *Top-down Phased Construction* pattern.

Among the remaining patterns, *Object Indexing* does not seem to be used in the majority of transformations. This is not surprising as this pattern is more useful for endogenous

Table 3 Existing design pattern instances fetched by hand

Model transformations	*Auxiliary Meta-model (Create variant)*	*Auxiliary Meta-model (Modify variant)*	*Auxiliary Meta-model (Delete variant)*	*Construction & Cleanup*	*Entities before Relations*	*Entity Merging*	*Entity Splitting*	*Fixed-Point Iteration*	*Object Indexing*	*Top-down Phased Construc-tion*	*Unique Instantia-tion*	*Visitor*
ecore2genmodel	4									3	3	
ecore2rdb	3	3								2	1	
ecore2uml	5	4				1				4	4	
java2statemachine	10		1	1				X		1		
wrap copy	1	3			1					3	3	
classDiagram2relation Schema	2	2			1						2	
book2publication	3	3	1	1	1				1		2	
ATOM2RSS	8	17	3	3	2		1		4		4	X
RSS2ATOM	6	4			2		2				5	
GR2_classOperation	19	41			12	3	6			18		
GR3_associations	35	58			23	1	3			58		
GR4_dependency	28	56			37	1	1			56		
GR5_abac	15	15				1	2			15		
GR5_secure Dependency	30	33				1	2			30		
GR6_secureLinks	20	39		6	14	4	6			10		
GR7_secureLinks2	30	48		9	18	2	2			48		
GR7_secureLinks3	24	48			24	1	1			48		
GR7_secureLinks4	17	20			9	2	1			20		

transformations. We found instances of the *Unique Instantiation* pattern in almost all transformations from Henshin and ATL. However, we found none in *SecBPMN2-to-UMLsec* because, given the nature of its large transformations, the same element is created several times. Finally, we found a very limited number of instances of *Fixed-point iteration* and *Visitor* patterns because they do not fit the implementation of most of the transformations in our sample. Since these two patterns are generally implemented by the whole transformation, and not an isolated subset, we indicate their occurrence with an "X" in Table 3.

From the perspective of the pattern nature, most of the found occurrences are different variants that implement the generic pattern. The other occurrences are approximations. Figure 13 summarizes the distribution of the nature of occurrences found for each transformation. For each transformation rule involved in a design pattern, we report if it implements a variant of the pattern (white) or an approximation (black). The Mix category (gray) means that we found both variants and approximations occurrences of the same design pattern in the transformation. Except for *ecore2genmodel*, *java2statemachine*, and *RSS2ATOM*, rules involved in variants account for more 60%, reaching 100% for three transformations. We found rules involved in both variants and approximations in only four transformations.

For RQ1, we can conclude that design patterns are used in transformations, which concurs with the survey in [24]. Some of the patterns occur more frequently than others. Variants are more frequent than approximations. Finally, transformation rules can be involved in more than one occurrence and sometimes in both variants and approximations. This result clearly motivates the need for design pattern detection approaches able to detect variants and approximations.

5.3 Detecting design pattern variants and approximations (RQ2)

To answer the second research question, we applied our detection approach on all the selected model transformations in two steps: first for variants, then for approximations. Table 4 presents the results of both steps. Note that we consider two kinds of accuracy in the detection of patterns: one for the detection of participants and one for the recovery of the complete design pattern. In our experiments, there are no false positives for the former. However, when taking into account the constraints of the pattern (e.g., scheduling, attribute constraints), we manually filtered participants and their scheduling leading to false positives. In addition, since our approach detected all pattern occurrences that we labeled in each transformation. Therefore, there are no false negatives.

5.3.1 Detecting design pattern variants

In a first step, we specified all design patterns and their different variants (described in Sect. 4) as input. Note that the *Auxiliary Metamodel* pattern does not involve rule scheduling. Hence, our approach, in its current state, is able to detect it fully automatically without the need for a manual recovery phase. For the other patterns, we combined automated detection of pattern participants with the recovery phase, as explained in Sect. 3.3.

From RQ1, most of the design pattern instances we expect are variants. The numbers on the left side of the slash in Table 4 show the results of detecting the variants of each design pattern. Our approach has detected all the expected instances of the design pattern variants and we found no false positive. The manual recovery phase, in addition to the verification of the scheduling, allows us to eliminate pattern participants for which the other participants to complete the pattern occurrences are not present.

5.3.2 Detecting design pattern approximations

The remaining instances reported in Table 3 are approximations. Therefore, in the second step, we performed the detection on the 18 transformations with the pattern approximations of Sect. 4 as input. The results of detecting approximations are presented in Table 4 .

Like for the variants, we detected all the expected instances of design pattern approximations. However, we found false positives for the *Auxiliary Metamodel* pattern approximations. When examining these occurrences, we discovered that they are resulting from the combination of two factors: (1) the specification of the approximation of a design pattern is very similar to the specification of one of its variants and (2) the inability of our approach to express negative conditions in the detection.

More specifically, the variants using *Marker* allow, among others, to tag elements as already processed when executing a rule on a list of elements. For example, in the rule on the top left of Fig. 14, a `Chapter` is marked after adding the number of its pages in the `Publication`. In contrast, in some transformations, the *Trace* variant is used for the same purpose, like in the rule on the bottom right of Fig. 14. However, the *Trace* is not connected to a target element as defined by the design pattern. Thus, we consider this use of a `Trace` as an approximation. This approximation results in some false positives as shown in Fig. 15, because with our detection approach, it is not possible to express the negative condition that `Trace` should not be connected to a target element.

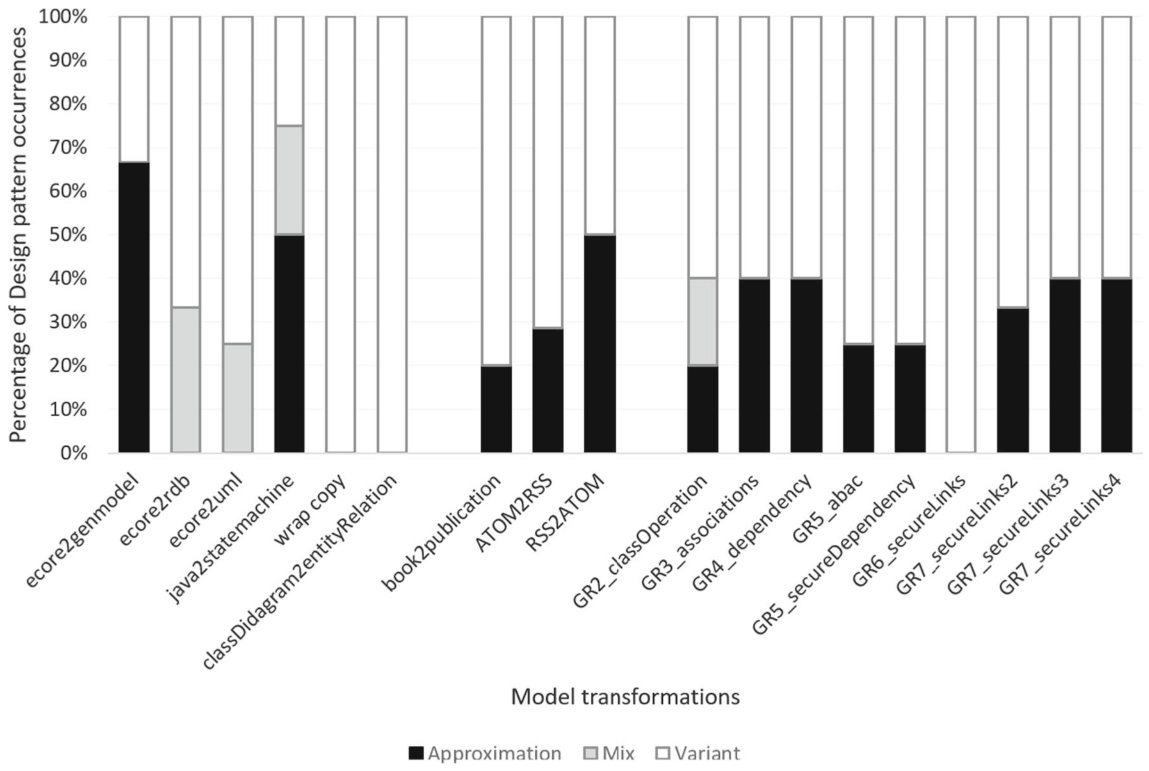


Fig. 13 Nature of design pattern occurrences in model transformations

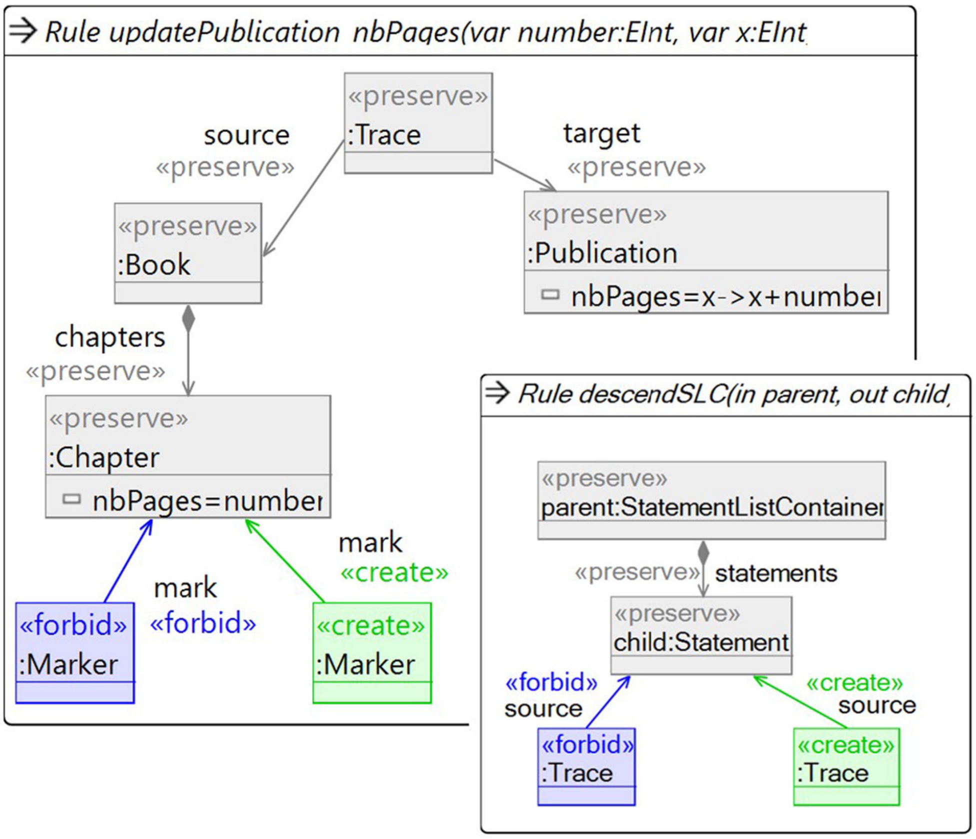


Fig. 14 *Trace* as an approximation of *Marker* for *Auxiliary metamodel* pattern

Table 4 Variants and approximations of design pattern instances detected by our approach. x/y denotes x variants and y approximations

Model transformations	*Auxiliary Meta-model (Create variant)*	*Auxiliary Meta-model (Modify variant)*	*Auxiliary Meta-model (Delete variant)*	*Construction & Cleanup*	*Entities before Relations*	*Entity Merging*	*Entity Splitting*	*Fixed-Point Iteration*	*Object Indexing*	*Top-down Phased Construction*	*Unique Instantiation*	*Visitor*
ecore2genmodel	/4									/3	3/	
ecore2rdb	3/	2/1								/2	1/	
ecore2uml	5/	4/				1/				4/	4/	
java2statemachine	1/9		/1	/1				/X		/1		
wrap copy	1/	3/			1/					/3	3/	
classDiagram2relation Schema	2/	2/			1/						2/	
book2publication	3/	3/	1/	1/	1/				/1		2/	
ATOM2RSS	8/	17/	3/	3/	2/		/1		4/		4/	/X
RSS2ATOM	6/	4/			2/		/2				5/	
GR2_classOperation	19/	41/			/12	3/	6/			11/7		
GR3_associations	35/	58/			/23	1/	3/			/58		
GR4_dependency	28/	56/			/37	1/	1/			/56		
GR5_abac	15/	15/				1/	2/			15/		
GR5_secure Dependency	30/	33/				1/	2/			/30		
GR6_secureLinks	20/	39/		6/	14/	4/	6/			10/		
GR7_secureLinks2	30/	48/		9/	/18	2/	2/			/48		
GR7_secureLinks3	24/	48/			/24	1/	1/			/48		
GR7_secureLinks4	17/	20/			/9	2/	1/			/20		

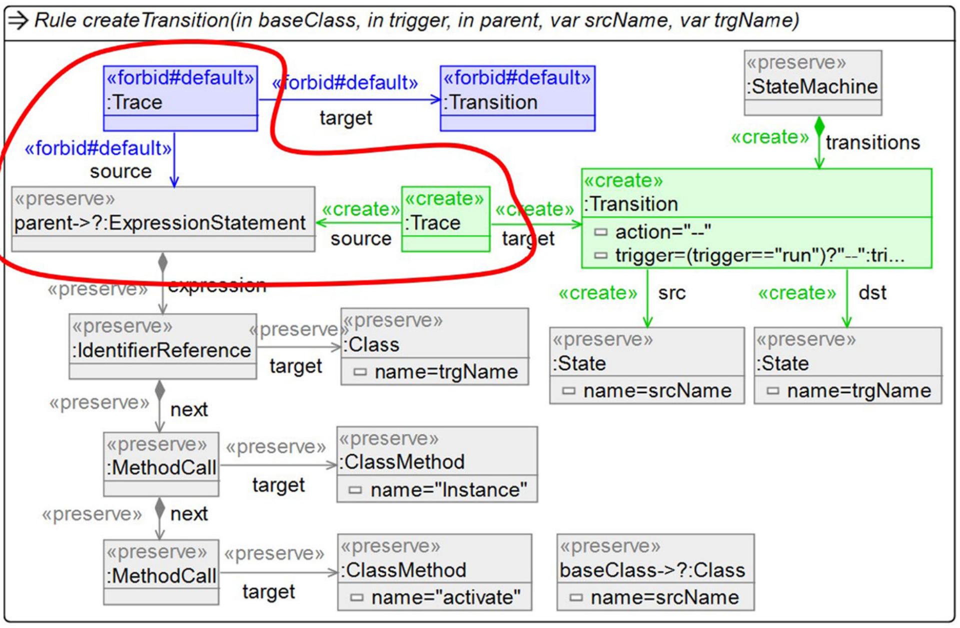


Fig. 15 Example of a false positive with the *Trace* vs *Marker* approximation

5.4 Pattern combinations (RQ3)

To answer the third research question, we analyzed the pattern instances detected by our approach to assess whether *uses* dependencies between patterns depicted in Fig. 11 actually exist in transformations. More specifically, we examined which pattern participants are detected in each model transformation rule and checked the scheduling to determine the co-occurrences.

To measure the consistency with which co-occurrences of patterns are found, for each pair of patterns, we calculate the percentage of times the used pattern instances appear with the using pattern instances. For a pair of patterns $\langle userP, usedP\rangle$ from Fig. 11, let us consider the set P of pairs of pattern occurrences of, respectively, $userP$ and $usedP$ in a transformation T_k:

$$P = \left\{(p_{ik}, p_{jk}) \mid p_{ik} \in userP \wedge p_{jk} \in usedP\right\} \qquad (1)$$

For each pair in P, we consider that p_{ik} uses p_{jk} if all the participants of p_{jk} appear simultaneously with the participants of p_{ik} in the same rules of T_k. Furthermore, we distinguish between different usage levels depending on whether p_{jk} is an instance of a variant or an approximation of $usedP$. Thus, we assign a usage level $uselev(p_{ik}, p_{jk})$ between p_{jk} and p_{jk} as follows:

$$uselev(p_{ik}, p_{jk}) = \begin{cases} 1 & p_{ik} \text{ uses a variant instance } p_{jk}, \\ 0.5 & p_{ik} \text{ uses an approximation instance } p_{jk}, \\ 0 & p_{ik} \text{ does not use } p_{jk}. \end{cases} \qquad (2)$$

Following these definitions, we calculate, for each transformation T_k in our transformation set, the usage score $usage(userP, usedP, T_k)$ as the average usage level of pairs of instances of $userP$ using instances of $usedP$. Note that, by the pattern specification, an instance of $userP$ may use more than one instance of $usedP$. Then, the average is calculated with respect to the number of instances of $userP$ in T_k multiplied by the number $mf(userP, usedP)$ of instances of $usedP$ an instance of $userP$ is expected to have. Formally:

$$usage(userP, usedP, T_k) = \frac{\sum_{(p_{ik}, p_{jk})\in P} uselev(p_{ik}, p_{jk})}{|\{p_{ik} \mid p_{ik} \in userP\}| \times mf(userP, usedP)} \qquad (3)$$

Table 5 shows the usage scores between the user patterns (in columns) and the used patterns (in rows) for the considered transformations. We only report a score when the user patterns exist in the transformations. For the sake of conciseness, we aggregated the scores of all the transformations of *SecBPMN2-to-UMLsec*.

A first observation is that in all but a few cases, we confirmed empirically the use relationships of Fig. 11. For example, the instances of *Entities before Relations* consistently use the instances of the two variants *Trace Create* and *Trace Modify* of the *Auxiliary Metamodel* pattern, as indicated by a usage score of 100%. Similarly, although less frequent, *Entity Splitting* and *Entity Merging* also use in 100% of the cases *Trace Create* variant. We also obtained perfect usage scores (100%) for three other user patterns. *Construction & Cleanup* always uses *Create* and *Delete* for both *Trace* and *Marker* variants of the *Auxiliary Metamodel*

Table 5 Usage scores for pairs of patterns

	Entities before relations	*Entity splitting*	*Entity merging*	*Top-down construction*	*Construction & Cleanup*	*Object indexing*	*Visitor*	*Fixed-point iteration*	Transformations
Aux-MM Trace-Delete					100%				Java2StateMachine
Aux-MM Trace-Modify				50%					Ecore2GenModel
	100%			100%					Ecore2Rdb
				100%					Ecore2Uml
				100%					Java2StateMachine
	100%			67%					wrap-copy
	100%								classDiag2relationSche
	100%								Book2Publication
	100%							100%	ATOM2RSS
	100%								RSS2ATOM
	100%			100%					SecBPMN2-to-UMLsec
Aux-MM Trace-Create				50%					Ecore2GenModel
	100%			100%					Ecore2Rdb
			100%	100%					Ecore2Uml
				100%	100%				Java2StateMachine
	100%			67%					wrap-copy
	100%								classDiag2relationSche
	100%								Book2Publication
	100%	100%							ATOM2RSS
	100%								RSS2ATOM
	100%	100%	100%	100%					SecBPMN2-to-UMLsec
Aux-MM Marker-Delete					100%				Book2Publication
					100%				ATOM2RSS
Aux-MM Marker-Modify						100%			Book2Publication
						100%	100%	100%	ATOM2RSS
Aux-MM Marker-Create						50%			Java2StateMachine
					100%	100%			Book2Publication
					100%	100%	100%	100%	ATOM2RSS
Object indexing							100%	100%	ATOM2RSS

pattern. *Visitor* and *Fixed-point Iteration* use the *Marker Create* and *Marker Modify* variants of *Marker* type, as well as *Object Indexing* in all the detected instances.

The only two patterns for which we did not observe perfect scores for all the cases are *Top-down Phased Construction* and *Object Indexing*. The former uses the *Create* and *Modify* variants with scores of 100% in all the transformations but two. In `Ecore2GenModel`, the developer implementing *Top-down construction* used approximations of variants of the *Auxilary Metamodel* pattern, which explains the scores of 50%. In the second transformation `wrap-copy`, the score is slightly better (67%). We found that the developer used approximations of variants of this pattern only in some cases.

For *Object Indexing*, the only transformation in which we had unexpected results is `Java2StateMachine`. This design pattern is expected to use the *Marker Create* to index elements and *Marker Modify* to manipulate them in another rule. In the case of create, the developer always used *Trace* as an approximation which led to the score of 50%. The *Modify* variant was not used at all as the developer exploited another mechanism to access the indexed objects, i.e., rule invocation in the scheduling with the indexed elements passed as parameters.

It is worth mentioning here that the *user* patterns may have different variants, and these variants may use different patterns. In our evaluation, we rely on the detected variant of the *user* pattern to determine what are the expected *used* patterns. For example, we noticed that in *SecBPMN2-to-UMLsec* transformations, the variant implemented for *Construction & Cleanup* does not use *Auxiliary Metamodel*. Thus, we did not report any score of that pattern for this transformation.

In summary, we can conclude that, for our transformation sample, the usage relations between patterns, depicted in Fig. 11, are generally respected by developers. Moreover, we believe that detecting missing or incomplete usage relations may offer good refactoring opportunities to improve the transformations. In the example of the `Ecore2GenModel` transformation mentioned earlier, `Trace` was used as an approximation of `Marker` as part of *Top-down Phased Construction*, which led to a score of 50%. The transformation can be refactored by changing `Trace` to `Marker`.

5.5 Threats to validity

The objective of our approach is to detect model transformation design patterns. We implemented and evaluated our approach for Henshin where model transformations are specified as nested graphs. It is legitimate to question whether our approach is applicable on transformations written in other rule-based languages. To address at least partially this concern, we showed in our evaluation that some ATL transformations can be mapped to graph-like representations, on which we can apply our detection approach. Moreover, the transformations in our sample are of medium size, so it does not allow us to have a thorough time performance evaluation of our detection approach. Experiments with larger transformations are needed to draw conclusions on performance.

For the representativeness of the explored design patterns, our evaluation used most of the design patterns in the catalog of Ergin et al. [11]. We acknowledge, however, that other design patterns may exist or can be defined in the future. Moreover, considering the variety of model transformation languages, instances of patterns may take various forms, which makes it challenging to specify all these forms as input to our approach. More specifically, some of the design patterns considered in this work depend heavily on the transformation rule engine and the way the rule execution control is performed. For instance, this is the case of *Auxiliary Metamodel* which helps using objects across rules in the context of Henshin.

Another threat to validity of our results is the way we define the approximations. Although our approach is able to detect approximations of fully specified patterns in the case of missing elements, we had to specify explicitly the approximations in our evaluation.

Finally, the fact that the recovery phase is performed manually biases somehow the evaluation of our approach accuracy during the evaluation. Potential false positives can be discarded when aligning the participants and evaluating the schedule. However, we believe that the accuracy with which our approach detects the patterns participants alleviates the burden of assembling the pattern instances. Yet, there is a burden on the manual recovery phase where we should check all combinations of the detected participants for each pattern. In the future, we plan to visualize the detected instances in the transformation directly to significantly reduce the manual workload.

6 Discussion

Our experience with the model transformation design pattern detection approach has raised benefits and limitations that we discuss in what follows.

6.1 Comparison with other techniques

The only other automated approach we are aware of that is able to detect design patterns in model transformation is our previous rule-based approach [28]. Although the rule-based approach does not need a manual recovery phase, every design patterns must be manually encoded as rule facts. This means that every variant and approximation of a design pattern must be manually encoded. Instead, our proposed string

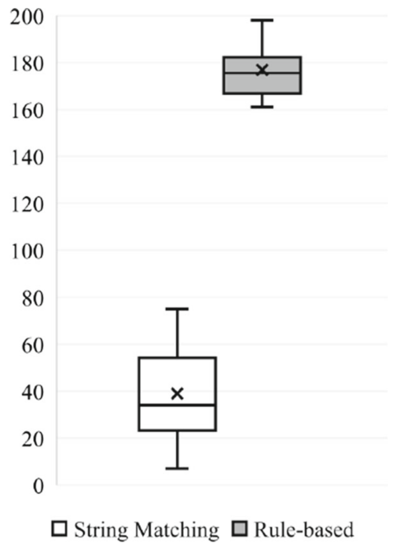


Fig. 16 Execution time for string matching vs. rule-based detection

matching approach is able to automatically detect the participants of a design pattern, its variants, and its approximations.

To compare the time performance of both approaches, we chose the three variants of the *Auxiliary Metamodel* design pattern because they do not specify a scheduling or attribute constraints; thus, they do not require a manual recovery phase. We executed both approaches on the same set of model transformations presented in Table 1. Both approaches were able to detect the same variants and approximations. As shown in Fig. 16, the average detection time for each pattern in a transformation is 39 *ms* for the string matching detection as compared to 177 *ms* for the rule-based detection. This represents a gain of around 80% in time performance for the string matching approach, concurring the findings in [20]. We ran all executions on a Windows 10 machined equipped with an Intel i7-8750H CPU clocked at 2.20 GHz with 16 GB of RAM. To fairly compare both approaches, we only measured the detection phase (see Fig. 3).

6.2 Improving the performance

During our experiments, the detection procedure executed instantaneously, in the order of milliseconds. Nevertheless, there is still room for improvement. We identified useful heuristics to increase the time performance of the detection mechanism. The order in which the triplets are read in Algorithm 2 influences the overall detection time. In general, treating less frequent roles first reduces the number of potential occurrences early in the detection process. For instance, a heuristic is to favor fixed roles and roles that occur less often in the Eulerian path. Also, it is preferable to avoid starting with a triplet containing a dummy edge because there are numerous occurrences of this edge in the participant and rule strings.

Figure 17 shows two alternative participant strings of the same participant *entityMapping*. Although they will produce the same result, the representation in (b) will produce results in fewer iterations. That is because it starts with a fixed role (`Trace`) and the first triplet does not contain a dummy edge. Also, there are typically fewer `Trace` elements in concrete model transformation rules than elements from the input/output metamodel of the transformation.

There are different possibilities to automatically apply these heuristics in our approach. For example, once we obtain a participant or rule string, we can post-process it to apply the heuristics. Another possibility is to assign different weights to specific nodes and edges in the Eulerian graph and solve the Chinese Postman Problem using these weights.

6.3 Recovering the scheduling scheme

The catalog of model transformation design patterns can be partitioned into three categories: (i) design patterns not relying on a scheduling scheme, (ii) design patterns relying on a simple scheduling scheme, and (iii) design patterns heavily relying on a complex scheduling scheme.

In category (i), the design patterns contain either only one or a set of unordered participants to detect. This category comprises patterns like: *Auxiliary Metamodel*, *Entity Splitting*, and *Unique Instantiation*. For these design patterns, our approach detects their instances completely automatically. The recovery phase is not needed.

In category (ii), the complexity of the design patterns is in the roles of each participants, rather than in their scheduling. This category comprises patterns like: *Entity before Relations*, *Transitive Closure*, *Object Indexing*, *Phased construction*, *Entity Merging*, and *Construction & Cleanup*. In most of these design patterns, the participants must be ordered sequentially. Our approach detects individual participants regardless of the complexity of the structure of its roles. Nevertheless, the recovery phase simply has to check whether the order of the rules matching the participants is satisfied.

In category (iii), the complexity of the design patterns is mainly in the scheduling of the participants. This category comprises patterns like: *Fixed-point Iteration*, *Visitor*, *Simulating Universal Quantification*, and *Execution by Translation*. For these design patterns, the detection phase is very simple as we can see by the structure of the roles in the participants of Fig. 18. However, most of the workload is concentrated in the manual recovery phase. Nevertheless, this category comprises very few design patterns. Future research should therefore investigate how to automate this phase. Such an automated process should address the few scheduling scheme that design patterns rely on (conditional

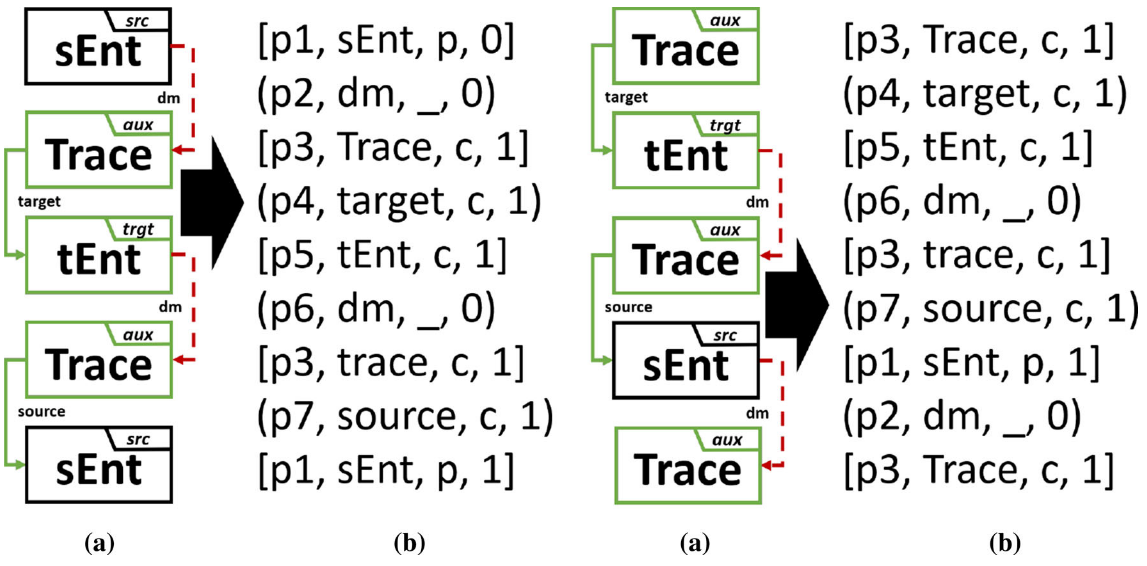


Fig. 17 Alternative string representations of the *entityMapping* participant

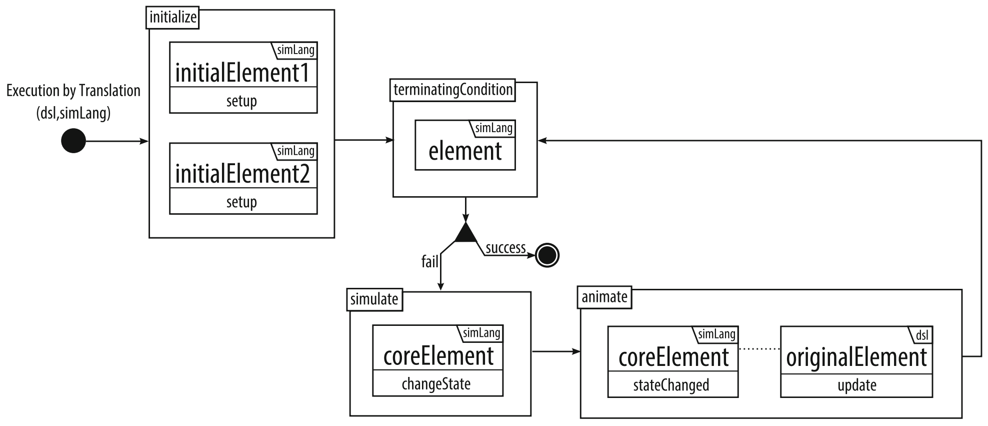


Fig. 18 *Execution by Translation* design pattern with implicit trace elements (from [11])

branching, parallelism, etc.). Furthermore, it should cope with the many different scheduling schemes offered by different model transformation languages. For example, on the one hand, languages like MoTif [43] offer explicit and advanced constructs such as recursion. On the other hand, languages like ATL [18] often rely on implicit rule scheduling.

6.4 Detecting other components of a design pattern

Currently, we detect automatically structural features of participants, that is the graph structure of the roles. The detection of other features is left for the manual recovery phase. Consider the *Execution by Translation* pattern. According to [11], the description of this design pattern assumes the entities of a metamodel (`dsl`) are already mapped[5] to entities of another intermediate metamodel (`simLang`). Then, `simLang` is simulated and the corresponding entities in `dsl` are modified accordingly to animate the simulation. Figure 18 presents the generic solution of the pattern. The *initialize* participant sets up the initial state of a model to start its simulation. The *terminatingCondition* participant checks if a certain condition on the model is satisfied to stop the simulation. If it is not satisfied, the *simulate* participant is activated to modify entities according to a specific criterion. The *animate* participant finds the entities in `dsl` corresponding to the entities modified in `simLang` and applies the necessary changes on them. The *terminatingCondition* participant is checked again and the simulation goes on.

The generic solution of this design pattern shows participants with one or two roles when, in practice, they could be played by a variable number of elements and relations in a model transformation implementation, depending on the application. Therefore, to reduce the false negative instances

[5] For example, through *Entities before Relations* or *Auxiliary Metamodel*.

of design patterns to detect, we should not only consider the generic solution, but also look at different reification possibilities of the design pattern. In our detection approach, we can specify variants of each participant by varying the number of roles to detect most instances. Nevertheless, this also emphasizes the importance of the recovery phase that must correctly interpret the detected participants. There is important information about the design pattern that is not expressed in the generic solution. In particular, the intention of the design pattern is typically defined in natural language. Future work should investigate a procedure for capturing the intention to enhance the detection.

A participant may be played by more than one rule. For example, `initialize` in Fig. 18 can be implemented in multiple rules to initialize the state of the elements needed for the simulation. On the contrary, some participants are optional, like `initialize`. For example, for specific application, the model transformation may not need to explicitly initialize the simulation because it relies on default values from the metamodel.

Another observation concerns the use of *tags* in design patterns expressed in DelTa. A tag expresses a condition on a role, e.g., `stateChanged`, or an action on a role, e.g., `changeState`. They are abstract components that may be implemented in three different ways in a model transformation implementation: (i) ignored, (ii) with intermediate elements, or (iii) with non-structural elements. For (i), a typical example where a tag can be ignored is the `setup` action. The implementation of this tag can be implicit in the logic of the transformation, without explicitly having an attribute of an element of a rule implementing the tag. For (ii), tags may be implemented by intermediary entities and relations. For example, `changeState` and `stateChanged` can be implemented by a marker or a link like in Fig. 14. In this example, we also see the necessity of adding a NAC in the rule. Therefore, tags in this category can be implemented by various elements depending on the semantic of the model transformation. For (iii), tags can be implemented using other types of elements, such as constraints and actions on attributes of elements in a rule. This is typically the case when a transformation heavily uses helpers expressing OCL constraints in ATL. When a design pattern falls in one of these three categories, we rely on the recovery phase to detect occurrences of the pattern.

Finally, very few design patterns, like *Unique Instantiation* and *Parallel Composition*, rely on forbidden constraints. For example, in the former pattern, if a role appears in a participant, then it cannot appear in another one. Currently, our approach is not able to detect such constraints because string matching cannot detect the absence of occurrences.

7 Conclusion

Detecting design patterns in model transformations can facilitate the comprehension and the maintenance of these transformation programs. In this paper, we propose a generic two-steps approach to detect design patterns, their variations, and their approximations in model transformation implementations. In the first step, our approach takes as input a set of model transformation rules and the participants of a design pattern, both converted into strings, and uses a string-matching algorithm to automatically find occurrences of pattern participants in the transformations. This automated step is completed by a manual step that consists in assembling participant occurrences and checking for the control schemes to form the pattern occurrences. The ability of our approach to detect approximations, i.e., incomplete pattern occurrences, may help to propose potential opportunities to improve the model transformation implementation. Other possibilities of refactoring opportunities can be found when examining detected occurrences of different design patterns used jointly in a transformation.

To evaluate our approach, we applied it on 18 transformations written in Henshin to detect 10 design patterns. Our evaluation results showed that design patterns are actually used in transformations, and that our approach is able to detect their different forms, being variants or approximations. They also showed that, except for a few cases, developers comply with the usage relations between patterns.

To improve our work, we plan to investigate some research directions. First, we will study other algorithms to increase the automation of the recovery phase. To broaden the applicability of our approach to other transformation languages, we expect to explore mapping strategies for each language to encode transformations into strings that can be handled by the matching algorithm. We also work on a recommending strategy to propose refactoring solutions to incomplete pattern occurrences, similar to the work in [29]. Finally, to better assess the performance of our approach, we plan to test it on large transformations.

References

1. Al-Obeidallah, M., Petridis, M., Kapetanakis, S.: A survey on design pattern detection approaches. Int. J. Softw. Eng. **7**, 41–59 (2016)
2. Alnusair, A., Zhao, T., Yan, G.: Rule-based detection of design patterns in program code. STTT **16**(3), 315–334 (2014). https://doi.org/10.1007/s10009-013-0292-z

3. Antoniol, G., Fiutem, R., Cristoforetti, L.: Design pattern recovery in object-oriented software. In: 6th International Workshop on Program Comprehension (IWPC '98), June 24–26, 1998, Ischia, Italy, pp. 153–160 https://doi.org/10.1109/WPC.1998.693342 (1998)
4. Arcelli, F., Perin, F., Raibulet, C., Ravani, S.: Design pattern detection in java systems: a dynamic analysis based approach. In: Evaluation of Novel Approaches to Software Engineering: 3rd and 4th International Conferences, ENASE 2008/2009, Funchal, Madeira, Portugal, May 4-7, 2008 / Milan, Italy, May 9–10, 2009. Revised Selected Papers, pp. 163–179 . https://doi.org/10.1007/978-3-642-14819-4_12 (2009)
5. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: D.C. Petriu, N. Rouquette, Ø. Haugen (eds.) Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3–8, 2010, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 6394, pp. 121–135. Springer . https://doi.org/10.1007/978-3-642-16145-2_9 (2010)
6. Batot, E., Sahraoui, H.A., Syriani, E., Molins, P., Sboui, W.: Systematic mapping study of model transformations for concrete problems. In: S. Hammoudi, L.F. Pires, B. Selic, P. Desfray (eds.) MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19–21 February, 2016, pp. 176–183. SciTePress. https://doi.org/10.5220/0005657301760183 (2016)
7. Bergeron, A., Hamel, S.: Vector algorithms for approximate string matching. Int. J. Found. Comput. Sci. **13**(1), 53–66 (2002). https://doi.org/10.1142/S0129054102000947
8. Dantizig, G.B.: Linear Programming and Extensions. Princeton University Press, Princeton (1991)
9. von Detten, M., Becker, S.: Combining clustering and pattern detection for the reengineering of component-based software systems. In: 7th International Conference on the Quality of Software Architectures, QoSA 2011 and 2nd International Symposium on Architecting Critical Systems, ISARCS 2011. Boulder, CO, USA, June 20–24, 2011, Proceedings, pp. 23–32 (2011). https://doi.org/10.1145/2000259.2000265
10. Dong, J., Lad, D.S., Zhao, Y.: Dp-miner: design pattern discovery using matrix. In: 14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2007), 26–29 March 2007, Tucson, Arizona, USA, pp. 371–380 (2007). https://doi.org/10.1109/ECBS.2007.33
11. Ergin, H., Syriani, E., Gray, J.: Design pattern oriented development of model transformations. Comput. Lang. Syst. Struct. **46**, 106–139 (2016). https://doi.org/10.1016/j.cl.2016.07.004
12. Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: Columbus: reverse engineering tool and schema for C++. In: 18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3–6 October 2002, Montreal, Quebec, Canada, pp. 172–181 (2002). https://doi.org/10.1109/ICSM.2002.1167764
13. Guéhéneuc, Y., Guyomarc'h, J., Sahraoui, H.A.: Improving design-pattern identification: a new approach and an exploratory study. Softw. Qual. J. **18**(1), 145–174 (2010). https://doi.org/10.1007/s11219-009-9082-y
14. Guéhéneuc, Y., Sahraoui, H.A., Zaidi, F.: Fingerprinting design patterns. In: 11th Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8–12, 2004, pp. 172–181 (2004). https://doi.org/10.1109/WCRE.2004.21
15. Holub, J.: Simulation of NFA in approximate string and sequence matching. In: Proceedings of the Prague Stringology Club Workshop 1997, Prague, Czech Republic, July 7, 1997, pp. 39–46. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University (1997). http://www.stringology.org/event/1997/p5.html
16. Iacob, M., Steen, M.W.A., Heerink, L.: Reusable model transformation patterns. In: Workshops Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference, ECOCW 2008, 16 September 2008, Munich, Germany, pp. 1–10 (2008). https://doi.org/10.1109/EDOCW.2008.51
17. Johannes, J., Zschaler, S., Fernández, M.A., Castillo, A., Kolovos, D.S., Paige, R.F.: Abstracting complex languages through transformation and composition. In: A. Schürr, B. Selic (eds.) Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4–9, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5795, pp. 546–550. Springer (2009). https://doi.org/10.1007/978-3-642-04425-0_41
18. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008). https://doi.org/10.1016/j.scico.2007.08.002
19. Jurack, S., Tietje, J.: Solving the TTC 2011 reengineering case with henshin. In: Proceedings Fifth Transformation Tool Contest, TTC 2011, Zürich, Switzerland, June 29–30 2011, pp. 181–203 (2011). https://doi.org/10.4204/EPTCS.74.17
20. Kaczor, O., Guéhéneuc, Y., Hamel, S.: Identification of design motifs with pattern matching algorithms. Inf. Softw. Technol. **52**(2), 152–168 (2010). https://doi.org/10.1016/j.infsof.2009.08.006
21. Kramer, C., Prechelt, L.: Design recovery by automated search for structural design patterns in object-oriented software. In: Proceedings of WCRE '96: 4rd Working Conference on Reverse Engineering, pp. 208–215 (1996). https://doi.org/10.1109/WCRE.1996.558905
22. Lano, K., Rahimi, S.K.: Model-transformation design patterns. IEEE Trans. Softw. Eng. **40**(12), 1224–1259 (2014). https://doi.org/10.1109/TSE.2014.2354344
23. Lano, K., Rahimi, S.K., Tehrani, S.Y., Sharbaf, M.: A survey of model transformation design pattern usage. In: Theory and Practice of Model Transformation: 10th International Conference, ICMT 2017, Proceedings, LNCS, vol. 10374, pp. 108–118. Springer (2017). https://doi.org/10.1007/978-3-319-61473-1_8
24. Lano, K., Rahimi, S.K., Tehrani, S.Y., Sharbaf, M.: A survey of model transformation design patterns in practice. J. Syst. Softw. **140**, 48–73 (2018). https://doi.org/10.1016/j.jss.2018.03.001
25. Lucia, A.D., Deufemia, V., Gravino, C., Risi, M.: Design pattern recovery through visual language parsing and source code analysis. J. Syst. Softw. **82**(7), 1177–1193 (2009). https://doi.org/10.1016/j.jss.2009.02.012
26. Mayvan, B.B., Rasoolzadegan, A.: Design pattern detection based on the graph theory. Knowl. Based Syst. **120**, 211–225 (2017). https://doi.org/10.1016/j.knosys.2017.01.007
27. Mhawish, M.Y., Gupta, M.: Design pattern detection using ontology (2018)
28. 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: 9th International Conference, SAM 2016, Saint-Melo, France, October 3–4, 2016, Proceedings, *LNCS*, vol. 9959, pp. 211–225. Springer, Saint-Malo (2016). https://doi.org/10.1007/978-3-319-46613-2_14
29. Mokaddem, C., Sahraoui, H., Syriani, E.: Recommending model refactoring rules from refactoring examples. In: A. Wasowski, R.F. Paige, Ø. Haugen (eds.) Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018, pp. 257–266. ACM (2018). https://doi.org/10.1145/3239372.3239406
30. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. J. ACM **46**(3), 395–415 (1999). https://doi.org/10.1145/316542.316550

31. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. **48**(3), 443–453 (1970). https://doi.org/10.1016/0022-2836(70)90057-4
32. Niere, J., Schäfer, W., Wadsack, J.P., Wendehals, L., Welsh, J.: Towards pattern-based design recovery. In: Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19–25 May 2002, Orlando, Florida, USA, pp. 338–348 (2002). https://doi.org/10.1145/581339.581382
33. Oakes, B.J., Troya, J., Lúcio, L., Wimmer, M.: Full contract verification for ATL using symbolic execution. Softw. Syst. Model. **17**(3), 815–849 (2018). https://doi.org/10.1007/s10270-016-0548-7
34. Paakki, J., Karhinen, A., Gustafsson, J., Nenonen, L., Verkamo, A.: Software metrics by architectural pattern mining. In: Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress) (2000)
35. Pandiselvam, P., Marimuthu, T., Lawrance, R.: A comparative study on string matching algorithm of biological sequences. Computing Research Repository (CoRR) (2014). arXiv:1401.7416
36. Philippow, I., Streitferdt, D., Riebisch, M., Naumann, S.: An approach for reverse engineering of design patterns. Softw. Syst. Model. **4**(1), 55–70 (2005). https://doi.org/10.1007/s10270-004-0059-9
37. Ramadan, Q., Salnitri, M., Strüber, D., Jürjens, J., Giorgini, P.: From secure business process modeling to design-level security verification. In: 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17–22, 2017, pp. 123–133. IEEE Computer Society (2017). https://doi.org/10.1109/MODELS.2017.10
38. Rasool, G., Philippow, I., Mäder, P.: Design pattern recovery based on annotations. Adv. Eng. Softw. **41**(4), 519–526 (2010). https://doi.org/10.1016/j.advengsoft.2009.10.014
39. Richa, E., Borde, E., Pautet, L.: Translation of ATL to AGT and application to a code generator for simulink. Softw. Syst. Model. **18**(1), 321–344 (2019). https://doi.org/10.1007/s10270-017-0607-8
40. Shi, N., Olsson, R.A.: Reverse engineering of design patterns from java source code. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18–22 September 2006, Tokyo, Japan, pp. 123–134 (2006). https://doi.org/10.1109/ASE.2006.57
41. Smith, T., Waterman, M.: Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981). https://doi.org/10.1016/0022-2836(81)90087-5
42. Srinivasan, G.: Operations Research: Principles and Applications. Prentice-Hall of India, Hoboken (2010)
43. Syriani, E., Vangheluwe, H.: A modular timed graph transformation language for simulation-based design. Softw. Syst. Model. **12**(2), 387–414 (2013). https://doi.org/10.1007/s10270-011-0205-0
44. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T.: Design pattern detection using similarity scoring. IEEE Trans. Softw. Eng. **32**(11), 896–909 (2006). https://doi.org/10.1109/TSE.2006.112
45. Uchiyama, S., Kubo, A., Washizaki, H., Fukazawa, Y.: Detecting design patterns in object-oriented program source code by using metrics and machine learning. J. Softw. Eng. Appl. (2014). https://doi.org/10.4236/jsea.2014.712086
46. Yu, D., Zhang, Y., Chen, Z.: A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. J. Syst. Softw. **103**, 1–16 (2015). https://doi.org/10.1016/j.jss.2015.01.019
47. Zanoni, M., Fontana, F.A., Stella, F.: On applying machine learning techniques for design pattern detection. J. Syst. Softw. **103**, 102–117 (2015). https://doi.org/10.1016/j.jss.2015.01.037

Chihab Mokaddem received the M.Sc. degree in Software and Information Systems Engineering from the University of Mostaganem, Algeria. He completed his Ph.D. in 2021 at the University of Montreal, Canada. His research interests include refactoring, design pattern detection, model transformation, and search-based software engineering.

Houari Sahraoui is a Professor at Université de Montréal. His research interests include AI Techniques Applied to Software Engineering, Search-based Software Engineering, Model-Driven Engineering, and Software Visualization.

Eugene Syriani is an Associate Professor at the department of computer science and operations research at University of Montreal. His main research interests fall in software design based on the model-driven engineering approach, the engineering of domain-specific languages, model transformation and code generation, simulation-based design, collaborative modeling, and user experience