Clone Detection for Graph-Based Model Transformation Languages

Daniel Strüber[1], Jennifer Plöger[1], Vlad Acreţoaie[2]

[1] Philipps-University Marburg, Germany
{strueber,ploeger1}@informatik.uni-marburg.de
[2] Technical University of Denmark, Kgs. Lyngby, Denmark
rvac@dtu.dk

Abstract. Cloning is a convenient mechanism to enable reuse across and within software artifacts. On the downside, it is also a practice related to significant long-term maintainability impediments, thus generating a need to identify clones in affected artifacts. A large variety of clone detection techniques has been proposed for programming and modeling languages; yet no specific ones have emerged for model transformation languages. In this paper, we explore clone detection for graph-based model transformation languages. We introduce potential use cases for such techniques in the context of constructive and analytical quality assurance. From these use cases, we derive a set of key requirements. We describe our customization of existing model clone detection techniques allowing us to address these requirements. Finally, we provide an experimental evaluation, indicating that our customization of ConQAT, one of the existing techniques, is well-suited to satisfy all identified requirements.

1 Introduction

Model transformation is of paramount importance to Model-Driven Engineering. Like all software artifacts, model transformation systems undergo a life-cycle including at least two main phases: an initial creation phase, followed by a long-term maintenance phase. Cloning, the development of transformations in the *copy-paste-modify* paradigm, provides key benefits for the creation phase; it is a fast, easy, and universally applicable practice. Still, cloning is related to substantial maintainability drawbacks. For instance, once a bug is found, many affected transformation rules may have to be updated correspondingly, a tedious and error-prone process. As maintenance tasks are estimated to account for 60% of all software costs [1], it seems advisable to address this trade-off explicitly.

The drawbacks of cloning are well-known from research on the more general issue of *software clones*. Yet, despite a substantial body of research [2], there is no universally accepted directive for how to proceed with clones. In the seminal work by Fowler [3], clones are deemed one particular kind of "bad smell". In this view, a refactoring towards a better suited abstraction is generally recommended. Empirical studies lead to a more nuanced view: Kim et al. [4] identify different types of clones, some of them warranting a refactoring towards suitable

abstractions, others rendering such efforts clearly unjustified. Still, despite controversy on the question of how to *proceed* with clones, there appears to be a consensus that software clones "should at least be *detected*" [5].

While numerous automated clone detection techniques for programming and modeling languages have been proposed [6], no specific ones have emerged for model transformation languages. The lack of such techniques is particularly surprising since existing model transformations may be affected heavily by cloning: Unlike in the case of most programming languages, reuse mechanisms for model transformations are just starting to become available [7]. Clone detection can be an enabling technology for the evolution of existing transformations towards these reuse mechanisms. But the variety of potential use cases for clone detection is even broader. It includes the quality assessment of existing transformations, performance optimizations, and even the identification of new design patterns.

The combination of different model transformation paradigms and clone detection use cases leads to a considerable design space for clone detection techniques. The goal of this paper is to approach this design space from a specific angle: We focus on graph-based transformation languages, one of the main model transformation paradigms [8]. Graph-based languages are popular since they allow to specify behavior in a high-level and intuitive manner.

Example. Consider three in-place model transformation rules expressed in a graph-based language. The rules, shown in Fig. 1, specify variants of the *move method refactoring*: Rule A describes the basic relocation of a method between two classes related through a field. Rule B additionally creates a "wrapper" method as a delegate for this method. Rule C adds an annotation to mark the wrapper as deprecated.

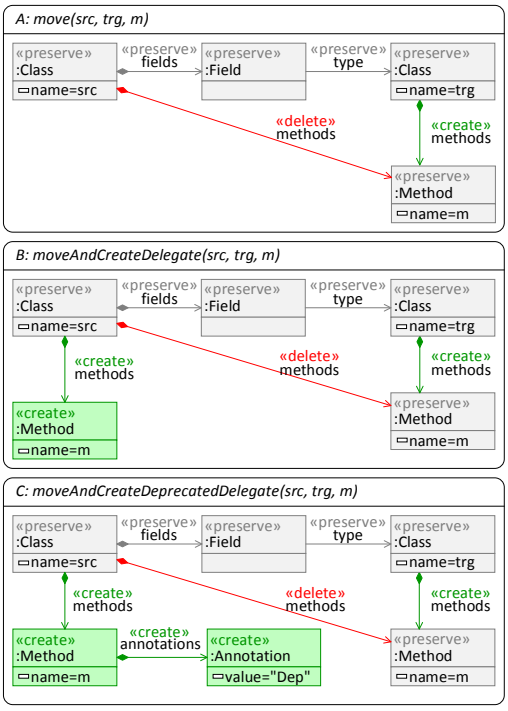

Fig. 1: Rules affected by cloning (from [9]).

Such rule sets are often created by copying a seed rule and modifying the copies. If a rule set contains many copied rules, maintaining it may be daunting and error-prone. It is advisable to provide dedicated support for the editing of such rules. For instance, the rules could be unified using a reuse concept provided by the transformation language. Alternatively, the consistent editing of the rules could be facilitated by tool support. In each case, clones need to be detected first.

Contributions. In this paper, we make the following contributions:

- We discuss use cases of clone detection for model transformation languages. The discussion is informed by recent developments in research on model transformations and software clones.
- Based on these use cases, we identify five key requirements for a clone detection technique for graph-based model transformations.
- We propose a customization of existing model clone detection techniques to address these requirements. To explore the feasibility of this idea, we provide experimental data and experiences.

This work is the first to investigate clone detection for model transformations systematically. While we have applied clone detection in an ad-hoc manner in a recent work [9,10], the outlined contributions, in particular the experimental data from adapting and applying different clone detection techniques, are new.

The rest of this paper is structured as follows. In Sect. 2, we outline the identified use-cases. In Sect. 3, we fix preliminaries. In Sect. 4, we propose requirements derived from the use-cases. We discuss our customization of existing techniques in Sect. 5 and our evaluation of this approach in Sect. 6. After discussing related work in Sect. 7, we conclude and suggest future research directions in Sect. 8.

2 Use cases

In this section, we introduce potential use cases. In each case, we pair a description of the use case with an account of the research state of the art.

Clone refactoring. The replacement of clones with a suitable reuse mechanism is a typical *refactoring* process [3]. Its outcome is a semantically equivalent, yet syntactically refined representation of the input artifacts. In the case of model transformations, reuse approaches such as *rule inheritance* [11], *refinement* [12] or *variability-based rules* [13] have emerged recently and are now available to developers. For instance, the rules in Fig. 1 can be expressed as one base rule with two sub-rules, or as one variability-based rule. Usually, such refactorings are performed manually. In legacy transformations with hundreds of rules, such a task is daunting and error-prone. An automated clone detection technique is an important prerequisite for automating this process.

Clone management. A suitable clone refactoring may not always be available. Even if the language provides a reuse mechanism, this mechanism may not match the scale or granularity of affected clones. For instance, an external reuse mechanism [7] does not help avoiding duplications in the same rule set, such as that shown in Fig. 1. We explore this issue further in Sect. 4. Furthermore, a refactoring may not always be *desirable*: It has been observed that expert developers create software clones intentionally with specific maintainability-related benefits in mind [5]. In these situations, the remaining maintainability drawbacks can be mitigated by tool support: A recent idea is to *manage* clones, using a system to monitor all clones constantly and to update affected artifacts automatically when one of them is edited [14,15].

Assessing specifications and languages. Clone detection can be used during the assessment of transformation specifications, for instance, in a quality assurance process or during the evaluation of solutions in a student assignment. Furthermore, the number of detected clones might be an indicator that the reuse mechanisms of the employed model transformation language are not adequate or not used enough. The detection of frequent patterns in transformation specifications can even lead to the identification of new design patterns and antipatterns. In contrast to object-oriented programming languages, where a catalog of fundamentally accepted patterns is available, the identification of transformation patterns is a recent idea [16]. Clone detection may contribute to this emerging branch of research by supporting the discovery of new design patterns.

Usability improvements. The level of support offered by most transformation editors to developers is below that offered by programming language IDEs. For instance, none of these editors benefits from advanced auto-complete functionality. Detecting clones introduced during an editing step could help providing such functionality by asking the developer if the reuse of an existing element is preferred. The clone detection algorithm would run in the background, much like the Java compiler runs in the background of Eclipse.

Performance improvements. While the impact of software clones on maintainability has been studied intensively, maintainability is by no means the only quality concern affected by cloning. Creating a large set of mutually similar rules may also entail a substantial computational effort during the application or analysis of these rules. As a result, cloning may give rise to longer execution times or even render entire transformations infeasible. Blouin et al. report on a case where a rule set of 250 similar rules was too large for execution [17]. While most existing performance optimizations for model transformations focus on accelerating the application of individual rules, clone detection might be highly useful in improving the performance of a whole model transformation system.

3 Preliminaries

In this section, we present formal preliminaries for clones in graph-based model transformation systems. To address the requirements identified later in this work, we extend our formalization from [9] by the distinction of *full* and *incomplete clones*, as well as *scopes*. We leave the notion of "graph" unspecified, which allows us to insert a graph kind with certain desired features. For instance, meta-model conformance and attributes can be expressed using *typed attributed graphs* [18].

Definition 1 (Rule) *A* rule $r = L \xleftarrow{le} I \xrightarrow{ri} R$ *consists of graphs* L*,* I *and* R*, called* left-hand side*,* interface graph *and* right-hand side*, respectively, and two embedding morphisms,* le *and* ri*. A* transformation system *is a set of rules.*

The rules in Fig. 1 conform to this definition, representing it in an integrated form: Elements of I are annotated with the action *preserve*, elements of $L \setminus I$ and $R \setminus I$ with the actions *delete* and *create*.

Our definition of clone reflects the idea that rules specify structural patterns: The left-hand side is a pattern to be matched in the source model. The right-hand side is a pattern specifying actions to derive the target model. Thus, we define "clone" as *common sub-pattern being present in a set of rules.* Such a sub-pattern is a fully formed rule itself, an idea captured by the concept of subrules.

Definition 2 (Subrule) *Given a pair of rules* $r_0 = (L_0 \xleftarrow{le_0} I_0 \xrightarrow{ri_0} R_0)$ *and* $r_1 = (L_1 \xleftarrow{le_1} I_1 \xrightarrow{ri_1} R_1)$ *with embeddings* le_i, ri_i *for* $i \in \{0,1\}$*, a* subrule morphism $s : r_0 \to r_1$*,* $s = (s_L, s_I, s_R)$ *comprises injective morphisms* $s_L : L_0 \to L_1$*,* $s_I : I_0 \to I_1$ *and* $s_R : R_0 \to R_1$ *s.t.* (1) *and* (2) *in Fig. 2 commute and*

(i) the intersection of $s_L(L_0)$ *and* $le_1(I_1)$ *is isomorphic to* I_0*,*
(ii) the intersection of $s_R(R_0)$ *and* $ri_1(I_1)$ *is isomorphic to* I_0*, and*
(iii) $L_1 - (s_L(L_0) - s_L(le_0(I_0)))$ *is a graph.*

Conditions (i)-(iii) ensure that a subrule always performs the same actions on related elements as the original rule.

For example, in Fig. 1, A is a subrule of B since A can be injectively mapped to B and the actions on the original and mapped elements are identical.

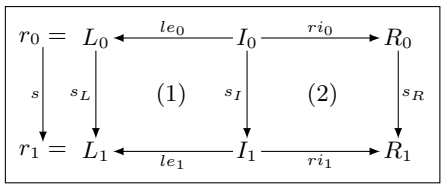

Fig. 2: Subrule morphism.

Given a set of rules, a clone is a subrule that can be embedded into a subset of this rule set.

Definition 3 (Clone) *Given a set* $\mathcal{R} = \{r_i | i \in I\}$ *of rules, a clone* $C_\mathcal{R} = (r_c, \mathcal{C})$ *over* $\mathcal{R}$ *consists of rule* r_c *and set* $\mathcal{C} = \{c_j | j \in J, J \subset I\}$ *of subrule morphisms* $c_i : r_c \to r_j$*. A clone* $C_\mathcal{R}$ *induces a set of* affected rules $\mathcal{R}_{aff}(C_\mathcal{R}) = \{r \in \mathcal{R} \mid \exists c \in \mathcal{C} : r_c \to r\}$.

In the example, any subrule of rule A is a clone over the entire rule set {A, B, C} since it can be embedded in each of these rules.

We discern full clones from partial clones. A full clone is a largest subrule, i.e., one not fully covered by another clone over the same subset.

Definition 4 (Full and partial clone) *A clone* $C_\mathcal{R} = (r_c, \mathcal{C})$ *over a set* $\mathcal{R}$ *of rules is a* full clone *iff there is no clone* $C'_\mathcal{R} = (r'_c, \mathcal{C}')$ *over* $\mathcal{R}$ *with a subrule mapping* $i : r_c \to r'_c$ *such that* $i \neq id$*. Non-full clones are called* partial *clones.*

The full clones present in the example rules are listed in Table 1. Clones are given by their *size*, calculated as the total number of involved nodes and edges. In particular, C2 represents all nodes and edges found in rule A. In addition, C1 incorporates the nodes and edges present in B, but not in A. All subrules of A except for the complete rule are partial clones. Please note that we omit attributes here for simplicity.

Name	Rules	Size
C1	{*B*, *C*}	10
C2	{*A*, *B*, *C*}	8

Table 1: Full clones in the running example.

In the established taxonomy of software clones [2], our definition includes *Type I* and *II* clones, *identical* and *almost identical* (except for naming) duplications. Furthermore, depending on the selected base graph kind, the definition may extend to *Type III* or *near-miss* clones, differing just in the presence or absence of certain attributes. In contrast, *Type IV* or *semantic* clones cannot be captured using syntactic properties, as we do. Identifying semantic clones in rule sets requires to analyze their behavior, an interesting avenue for future work.

We further distinguish clones based on their scope.

Definition 5 (Scope) *The scope of a clone is either* MICRO, INTERNAL *or* EXTERNAL.

$$scope(C_{\mathcal{R}}) = \begin{cases} \textsc{Micro} & |\mathcal{R}_{aff}(C_{\mathcal{R}})| = 1 \\ \textsc{Internal} & |\mathcal{R}_{aff}(C_{\mathcal{R}})| \geq 2 \text{ and } \exists \text{ transformation system } \mathcal{T} \\ & \text{s.t. } \mathcal{R}_{aff}(C_{\mathcal{R}}) \subset \mathcal{T} \\ \textsc{External} & \text{else} \end{cases}$$

This definition is illustrated in Fig. 3. Micro-clones are pattern duplications within the same rule. In the case of code clones, an effect has been observed that the last in a set of micro-clones is particularly prone to errors [19]. Internal clones, as exemplified in our running example, extend to multiple rules within the same model transformation system. Transformation systems are prone to internal clones if they capture multiple variants of a rule: Some included actions may be common to all variants, others optional. External clones shared between multiple transformation systems may occur if a system or parts of it are adapted for a new purpose, for instance in exogenous transformations: The target language of the transformation may be replaced while retaining the source language.

The reuse mechanisms found in transformation languages [7] correspond to these scopes. Micro-clones can be avoided by specifying multiplicity at the level of individual graph nodes and edges [20]. Internal clones can be replaced using reuse mechanisms such as rule inheritance [11], refinement [12], or variability-based rules [13]. A suitable alternative to the creation of external clones are external reuse approaches, such as generic model transformations [21].

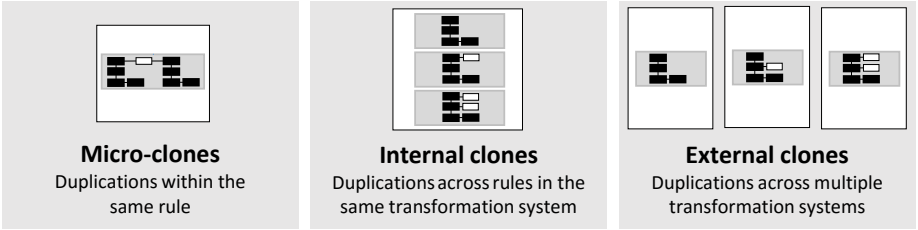

Fig. 3: Granularity of clones in model transformation systems.

4 Requirements

In this section, we present key requirements for a clone detection technique for graph-based model transformations. The requirements were identified from the use cases introduced in Sect. 2. We summarize them in Table 2.

Requirement	Summary	Target use case U1 U2 U3 U4 U5
R1: Pattern-based	*Must identify common structural patterns.*	■ ■ ■ ■ ■
R2: Performance	*Must be able to deliver results rapidly.*	■ ■ ■ ■ ■
R3: Exhaustiveness	*Must prefer full over partial clones.*	■ ■ ■ □ □
R4: Scope	*Must operate in a specific cloning scope.*	■ ■ ■ ■ ■
R5: Tool integration	*Must integrate with existing tool environments.*	■ ■ □ ■ ■

Table 2: Key requirements for clone detection techniques in the identified use cases: Clone refactoring (U1), clone management (U2), assessment (U3), usability improvement (U4), performance improvement (U5). ■ = Hard requirement, ■ = Soft requirement, □ = Not required.

(R1) Pattern-based. In accordance with our definition of clones, the identification of structural patterns is a hard requirement in all identified use cases. A detection technique capable of identifying cloned patterns is required, rather than one aimed at identifying pairs of similar elements. The latter typically assumes that individual elements contain a significant amount of information, such as names [22]. In rules, conversely, nodes and edges usually express only limited amounts of information, such as just a type and an action. Moreover, for the performance improvement use case, it is crucial to find patterns; individual elements in isolation are hard to handle efficiently during rule application [23].

(R2) Performance. Clone detection needs to support scenarios with many rules and large individual rules – arguably situations where maintainability is problematic [24]. In such scenarios, performance becomes a significant challenge. The task at hand is *pattern-mining*, the identification of structurally corresponding subgraphs, which boils down to the NP-complete sub-graph isomorphism problem [25]. Clearly, a high execution time in the range of hours or days would not be beneficial for use cases that are applied constantly, such as refactorings. Still, a high latency may be acceptable if clone detection is to be used in a nonrecurring manner: Performance optimizations can be carried out statically before running the transformation. Clone management may require a one-time setup of the transformation system. Yet even in such cases, execution time is not the only issue – a large search space may lead to memory-related program terminations.

(R3) Exhaustiveness. To deal with the computational cost, a clone detection tool might trade-off performance for exhaustiveness: It may apply a heuristic to trim its search space. As a result, certain duplications may not be considered, leading to the reporting of *partial clones* (Def. 4). In three use cases, this kind of outcome is problematic: In clone refactoring, using partial clones as a starting

point leads to unnatural results that retain certain duplications. A clone management tool that only propagates arbitrary updates to corresponding instances is undesirable. The quality of a specification may be assessed incorrectly if the full extent of cloning is not discovered. In contrast, exhaustiveness plays no evident role in auto-completion features and performance optimizations that normally operate on a best-effort basis.

(R4) Scope. Since all identified use cases operate on a specific scope, a clone detection technique needs to match this scope. For instance, during clone refactoring, it is essential that the upfront clone detection step operates in a scope where a suitable reuse mechanism is available for refactoring. The refactoring of internal clones requires an internal reuse mechanism, while that of external clones requires an external reuse mechanism (see the discussion after Def. 5).

(R5) Tool integration. It is best to enable the exploration of clones in the environment familiar to maintainers, that is, their transformation editor. Even in scenarios where clone detection is an upfront step to an automated refactoring, developers need to inspect the reported clones to influence the refactoring result. This requirement can be neglected in performance optimizations since they are usually transparent to the user, and to some extent in usability-oriented recommender systems that use clone detection as a background technique only.

5 Adapting Existing Clone Detection Techniques

In this section, we explore the idea that existing clone detection techniques can be adapted to the requirements of graph-based model transformations.

Since patterns are abstractions of model structures, the most suitable candidate techniques are those focusing on *model clone detection*. We consider two techniques, *eScan* [26] and *ConQAT* [27], as they allow us to address R1, the identification of identical patterns in their input models. Both techniques were originally devised for the domain of Simulink models. It is noteworthy that they may not seem a natural fit for our purpose: Simulink models are structured based on control flow, while rules do not prescribe a specific navigation order.

Both techniques apply the same basic process: First, a suitable encoding is provided as input. Second, the actual clone detection takes place. Third, the results are post-processed to retain only the most useful results.

Phase 1: Creating an encoding. Both eScan and ConQAT assume a directed, labeled graph as input data structure. We devised a suitable encoding of graph transformation rules: (i) To represent the graph spans constituting a rule as one graph, we use the integrated representation indicated in Fig. 1. The action assigned to an element is reflected in its label. This encoding allows us to capture the subrule relation: For instance, a clone never includes the left-hand side instance of a *preserve* node while neglecting the right-hand side counterpart, which would lead to invalid results during clone refactoring. (ii) To preserve the typing information of an element, we encode its type as part of the label. (iii) We represent attributes as additional elements in the graph. Each attribute

becomes a pair of a node and an edge, labeled with the attribute value, type and action. Encoding attributes as distinct elements allows us to account for reuse mechanisms that accommodate the attribute level.

Phase 2: Clone search. We use the search phases of the considered approaches in a black-box manner. For completeness, we still give a brief account of the internal workings of these approaches. Details are found elsewhere [26,27].

ConQAT proceeds by finding pairs of nodes with the same label and combining these node pairs to *clone pairs*. A clone pair represents two isomorphic sub-graphs of the input graph. To group only promising node pairs together, a heuristics is applied. To this end, a similarity function is used, comparing the neighborhoods of two input nodes. Starting with one of the node pairs with the highest similarity value, ConQAT executes a breadth first search to find a clone pair of the largest possible size, i.e., number of included node pairs. In each step, one of the node pairs of highest similarity is used to extend the clone pair.

In the example, there are 26 relevant node pairs.[1] The "src" nodes in Rules B and C are determined most similar as they share the largest number of common adjacent nodes and edges. Starting at this pair, phase 2 produces six clone pairs, four of size 4 (rule A with corresponding parts of rule B and C, and reversed) and two of size 5 (rule B with the corresponding part of rule C, and reversed).

eScan works by systematically deriving all *clone fragments*, i.e., sub-graphs with an isomorphic counterpart, contained in the input graph. Starting with sub-graphs comprising of just one edge and its source and target node, eScan produces larger sub-graphs incrementally. In each iteration, given the cloned sub-graphs with k edges, eScan finds the set of (k+1) edge sub-graphs by including additional edges from the graph. Sub-graphs without isomorphic counterparts are discarded. Isomorphy between sub-graphs is detected by comparing their canonical labels, an encoded representation of their elements. An optimization ensures that each sub-graph is used as a starting point just once.

In the example, the input graph contains 15 sub-graphs of size 1: four in rule A, five in rule B and six in rule C.[1] With the exception of the *annotations* edge in rule C, each of these sub-graphs is a clone fragment and is consequently used to derive sub-graphs of size 2. After termination, there are 14 clone fragments of size 1, 16 of size 2, 16 of size 3, 11 of size 4, and 2 of size 5.

Phase 3: Post-processing. In both approaches, the result of phase 2 is clustered, producing sets of isomorphic subgraphs. The result may contain sets that are completely covered by other sets. For instance, in the eScan result, the groups containing the sub-graphs of size 1, 2 and 3 are completely covered by the group of size 4. Covered groups are discarded since they are typically not useful to developers. Furthermore, ConQAT and eScan report only *connected* sub-graphs. Larger unconnected ones may be assembled from connected ones. To obtain clones (Def. 3), we map the results of phase 2 back to the rules.

In the example, both approaches produce the output shown in Table 1. In general, the employed strategy during Phase 2 may have implications for the

[1] In favor of simplicity, we neglect attributes and their encoding in these illustrations.

exhaustiveness of the result (R3). Since eScan eventually produces every possible sub-graph, it finds all full clones (Def. 4) – assuming unlimited memory and time. In practice, eScan has been shown not to scale up to larger models in the Simulink domain [27]. In contrast, ConQAT shows good scalability behavior, yet the employed heuristic might lead to some detected clones being incomplete.

6 Evaluation

In this section, we present an evaluation of our approach. We address the following research question: *Can the requirements for graph-based model transformation clone detection be satisfied by adapting existing clone detection techniques?*

Methods and Materials. Using our customization of ConQAT and eScan, described in Sect. 5. we addressed the requirements as follows:

- ConQAT and eScan are *pattern-based* (R1) by design. Since this requirement is important in all identified use-cases, we selected these particular techniques to investigate clone detection in model transformation rules.
- To study *performance* (R2), we applied each technique on rule sets from real model transformation systems and measured execution time.
- While eScan guarantees *exhaustiveness* (R3) by design, we devised a custom set-up to study the exhaustiveness of ConQAT: We fed the largest clones reported by ConQAT as input to *eScan-Inc* [26], an incremental variant of eScan that allows continuing the clone search from clones of a given size. This method, called *ScanQAT*, can find full clones missed by ConQAT. The number of full clones missed by ConQAT gives an indication of its exhaustiveness.
- To study *scope* (R4), we discuss how our customization of the existing techniques accounts for the different scopes of clones.
- To study *tool integration* (R5), we report on our experience with integrating the studied techniques in the existing tool environment of the Henshin model transformation language [28].

In the experiments for R2 and R3, we used rule sets from two transformation systems. The rule sets were chosen since they represent realistic, non-trivial rule sets available to the authors (convenience sampling). Ocl2Ngc is a set of rules from an OCL to nested graph constraint translator [29]. FmEdit is a set of editing rules for feature models, used in the context of model differencing [30]. We present statistical information on both rule sets in Table 3. The rules in Ocl2Ngc are organized in sets of 4 to 7 rules. The rules in FmEdit are organized in sets of 2 to 11 rules. In the case of Ocl2Ngc, we selected small, average, and large rules as samples for our experiments, presenting them in the table. In the case of FmEdit, we studied all rule sets. These sets provide a semantic grouping of the transformations without prescribing a particular control flow. In fact, the Ocl2Ngc transformation exhibits an elaborate control flow expressed using *units*, an activity-diagram-like control mechanism, which we neglected as it was orthogonal to the grouping into rule sets. To explore scalability, we also applied the considered techniques to the entire rule sets.

Rule set	#R	#N	#E	#A
trE04	4	8.0	10.0	2.3
trE0506	4	8.0	10.0	3.3
trE1112	4	14.0	18.0	7.3
trE09	4	11.0	16.0	4.3
trE10	4	10.0	13.0	3.3
trE13	6	19.5	29.5	10.0
trE16	4	20.0	29.0	12.3
trE17	7	26.7	41.7	17.9
all	54	19.7	30.7	10.0

(a) Ocl2Ngc

Rule set	#R	#N	#E	#A
a.arbitrary	7	3.9	5.1	2.7
a.generalize	9	3.2	4.3	2.2
a.refactor	2	2.0	1.0	2.0
a.specialize	9	3.1	3.6	3.0
c.arbitrary	4	5.3	9.3	4.5
c.generalize	8	6.9	35.8	8.5
c.refactor	11	6.6	17.0	4.7
c.specialize	7	8.1	39.9	7.4
all	57	5.2	15.8	4.6

(b) FmEdit

Table 3: Sample rule sets with number of rules (#R) and average number of nodes (#N), edges (#E), and attributes (#A) in each rule.

We created an implementation prototype for our experiments, implementing the customization outlined in Sect. 5. For Phase 2 and the clustering step of Phase 3, in the case of ConQAT we used the publicly available implementation[2]. We created our own implementation of eScan as no existing one was available to us. We ran all experiments on a Windows 7 system (3.4 GHz; 8 GB of RAM).

Results and Discussion. We applied the techniques on all rule sets, yielding the results shown in Table 4. For each combination of technique and rule set, we show the largest and the broadest clone. The largest clone is the one with the greatest number of common elements. The broadest clone is the one found in the greatest number of input rules; ties are broken by selecting the one with the greatest number of common elements.

Performance. ConQAT took between 1 and 544 msec for each individual rule set. For the full rule sets, it took 26.5 seconds and 783 msec. Our ScanQAT and eScan implementations took between 2 msec and 13.5 seconds for smaller rule sets. On the larger ones, they terminated with memory overflow errors or did not terminate within one hour. While our implementations could be flawed, this experience is in accordance with earlier experiments in the Simulink domain [27].

Exhaustiveness. Where available, the clones reported by ConQAT, ScanQAT and eScan were identical in size. Only in the case of two larger individual sets and the entire rule sets, both ScanQAT and eScan did not scale up. In these cases, we cannot evaluate the exhaustiveness of ConQAT. In all other cases, the largest and broadest clones reported by ConQAT were full clones. The largest clones found by ConQAT for all rules were larger than those in the individual rule sets – these clones spanned over several rule sets. In sum, it is indicated that ConQAT is generally suitable to address the exhaustiveness requirement.

Scope. The encoding described in Sect. 5 can be used to apply the considered techniques on all desired scopes: The input graph provided to the technique

[2] https://www.cqse.eu/en/products/conqat/install/

Rules	ConQAT				ScanQAT				eScan			
	R	N	E	A	R	N	E	A	R	N	E	A
trE04	2	7	8	1	2	7	8	1	2	7	8	1
	4	6	5	1	4	6	5	1	4	6	5	1
trE0506	2	7	8	2	2	7	8	2	2	7	8	2
	4	6	5	2	4	6	5	2	4	6	5	2
trE09	2	10	14	3	2	10	14	3		—		
	4	9	11	3	4	9	11	3		—		
trE10	2	9	11	2	2	9	11	2	2	9	11	2
	4	8	8	2	4	8	8	2	4	8	8	2
trE1112	2	13	16	6	2	13	16	6		—		
	4	12	13	6	4	12	13	6		—		
trE13	2	20	30	10		—				—		
	6	2	1	1		—				—		
trE16	2	19	27	11	2	19	27	11		—		
	4	18	24	11	4	18	24	11		—		
trE17	2	28	42	19		—				—		
	7	4	2	1		—				—		
all	2	33	55	16		—				—		
	31	2	1	1		—				—		

(a) Ocl2Ngc

Rules	ConQAT				ScanQAT				eScan			
	R	N	E	A	R	N	E	A	R	N	E	A
a.arbitary	2	3	2	0	2	3	2	0	2	3	2	0
	2	3	2	0	2	3	2	0	2	3	2	0
a.generalize	2	3	2	0	2	3	2	0	2	3	2	0
	2	3	2	0	2	3	2	0	2	3	2	0
a.refactor	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0
a.specialize	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0
c.arbitary	2	4	5	1	2	4	5	1	2	4	5	1
	3	2	1	0	3	2	1	0	3	2	1	0
c.generalize	2	5	7	2	2	5	7	2	2	5	7	2
	7	2	2	0	7	2	2	0	7	2	2	0
c.refactor	2	6	13	1	2	6	13	1	2	6	13	1
	10	2	1	0	10	2	1	0	10	2	1	0
c.specialize	2	5	7	2	2	5	7	2	2	5	7	2
	6	3	2	0	6	3	2	0	6	3	2	0
all	2	8	18	1		—				—		
	18	3	2	0		—				—		

(b) FmEdit

Table 4: Results. For each rule set, the *largest* (first row) and the *broadest* (second row) clones found are detailed with their number of rules (R) and number of nodes (N), edges (E), and attributes (A). "—" denotes memory-related program exits or execution times longer than one hour.

may represent one rule as well as multiple rules from the same or different transformation systems. An interesting edge case we observed in the larger rules of Ocl2Ngc includes clones that cover other clones of a separate scope: Internal clones may exhibit multiple embeddings to the same rule, i.e., cover a micro-clone. The preferable directive in this case depends on the use case. For instance, if adequate reuse concepts are available, clones can be refactored incrementally, first explicating the reuse inside the rule and then that across multiple rules.

Tool integration. To explore the integration with existing tools, we designed and implemented an Eclipse plug-in on top of the Henshin language [28]. We devised a custom *Clone Detection* view as an extension to the Henshin transformation editor, listing reported clones. When the user selects an entry in this view, the corresponding elements are highlighted in the editor. This view can be combined with most considered use-cases, for instance, by serving as an entry point for a clone refactoring. We describe the use of this plug-in in another work [31].

Threats to validity. A threat to external validity is our limited sample size of rule sets from two transformation systems. While the studied scenarios are heterogeneous, more examples are required to justify extensive generality claims. A threat to construct validity concerns our study of exhaustiveness. We have not compared the results against a list of known clones, which would be the most reliable strategy. Unfortunately, such lists are hard to produce manually for large rule sets. Furthermore, we focus on largest clones, neglecting smaller ones. While more comprehensive exhaustiveness studies are desirable, large clones are arguably the most relevant in refactorings and performance optimizations.

Conclusions. In conclusion, ConQAT, ScanQAT and eScan were on par with regards to all identified requirements except performance, where ConQAT outperformed the other approaches. Notably, the promising exhaustiveness results for ConQAT complement the findings from our recent work where we used this technique to construct rules in a performance optimization scenario [9]. This finding indicates that ConQAT is potentially useful in all considered use-cases, a hypothesis that still needs to be validated for larger industrial examples.

7 Related Work

Several more techniques for model clone detection have been proposed. While the approaches by Störrle [22,32] and Ekanayake et al. [33] enable the identification of groups of similar elements in UML and business process models, respectively, we focus on the detection of identical *patterns*. Liang et al. [34] propose a suitable technique based on identifying longest sub-sequences in paths through the input models. The technique shows a comparable accuracy to that of ConQAT while yielding a runtime improvement. We focus on ConQAT due to its publicly available implementation that fully satisfied the requirements in our experiments. The approach by Alalfi et al. [35] focuses on Type III clones in Simulink models.

A number of quality assurance approaches for model transformations are related. Van Amstel et al. [36] propose a variety of analytical methods, such as metrics and dependency graphs. Without mentioning specifics, they also foresee the use of clone detection. Kapová et al. [37] propose a set of quality metrics to evaluate QVT-R transformations; number of clones is mentioned as one metric. Wimmer et al. [38] introduce a refactoring catalog to improve the quality of M2M transformations; duplicate code is mentioned as a bad smell. Gerpheide et al. [39] present a quality model for QVT-O comprising 37 quality properties and four quality goals: functionality, understandability, performance, and maintainability.

8 Conclusion and Future Work

In this work, we present the first approach to address clone detection for model transformations, focusing on the graph-based transformation paradigm. We considered Type I and II clones, which are routinely produced when rules are created in a copy-and-paste manner. Our experiments indicate that our adaptation of ConQAT, a technique from the Simulink domain, is well-suited to satisfy the requirements of clone detection in graph-based model transformations.

There are several directions for future work. To validate the hypothesis that transformation developers can benefit from clone detection, a user experiment based on our prototypical tool is appropriate. Moreover, we aim to broaden the scope of our work towards additional transformation and clone detection features. First, to extend the expressiveness of the considered language, control flow and NACs can be addressed. Second, as our work focuses on graph-based model transformation, we aim to establish whether similar results can be obtained for other paradigms. Desirable clone detection features include support for Type III and IV clones and, addressing the performance optimization and usability improvement use cases, an incremental execution mode that reuses earlier results.

References

1. R. L. Glass, "Frequently forgotten fundamental facts about software engineering," *IEEE software*, no. 3, pp. 112–110, 2001.
2. R. Koschke, "Survey of research on software clones." Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software, 2007, p. 24.
3. M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2002.
4. M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 187–196.
5. C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.
6. D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.
7. A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reuse in model-to-model transformation languages: are we there yet?" *Software & Systems Modeling*, vol. 14, no. 2, pp. 537–572, 2013.
8. K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3. USA, 2003, pp. 1–17.
9. D. Strüber, J. Rubin, T. Arendt, M. Chechik, G. Taentzer, and J. Plöger, "RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules," in *Int. Conference on Fundamental Approaches to Software Engineering*. Springer, 2016, pp. 122–140.
10. D. Strüber, "Model-driven engineering in the large: Refactoring techniques for models and model transformation systems," Ph.D. dissertation, Philipps-Universität Marburg, 2016.
11. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger, D. S. Kolovos, R. F. Paige, M. Lauder, A. Schürr *et al.*, "Surveying rule inheritance in model-to-model transformation languages," *Journal of Object Technology*, vol. 11, no. 2, pp. 1–46, 2012.
12. A. Anjorin, K. Saller, M. Lochau, and A. Schürr, "Modularizing triple graph grammars using rule refinement," in *Int. Conference on Fundamental Approaches to Software Engineering*. Springer, 2014, pp. 340–354.
13. D. Strüber, J. Rubin, M. Chechik, and G. Taentzer, "A Variability-Based Approach to Reusable and Efficient Model Transformations," in *Int. Conference on Fundamental Approaches to Software Engineering*. Springer, 2015, pp. 283–298.
14. H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," *Software Engineering, IEEE Transactions on*, vol. 38, no. 5, pp. 1008–1026, 2012.
15. K. Narasimhan and C. Reichenbach, "Copy and paste redeemed," in *Int. Conference on Automated Software Engineering*. IEEE, 2015, pp. 630–640.
16. K. Lano and S. Kolahdouz-Rahimi, "Model-transformation design patterns," *Software Engineering, IEEE Transactions on*, vol. 40, no. 12, pp. 1224–1259, 2014.
17. D. Blouin, A. Plantec, P. Dissaux, F. Singhoff, and J.-P. Diguet, "Synchronization of models of rich languages with triple graph grammars: An experience report," in *Int. Conference on Model Transformation*. Springer, 2014, pp. 106–121.
18. R. Heckel, J. M. Küster, and G. Taentzer, "Confluence of typed attributed graph transformation systems," in *Int. Conference on Graph Transformation*. Springer, 2002, pp. 161–176.
19. M. Beller, A. Zaidman, and A. Karpov, "The last line effect," in *Int. Conference on Program Comprehension*. IEEE Press, 2015, pp. 240–243.

20. J. Bauer, I. Boneva, M. E. Kurbán, and A. Rensink, "A modal-logic based graph abstraction," in *Int. Conference on Graph Transformation*. Springer, 2008, pp. 321–335.
21. J. S. Cuadrado, E. Guerra, and J. De Lara, "Generic model transformations: write once, reuse everywhere," *Int. Conference on Model Transformation*, pp. 62–77, 2011.
22. H. Störrle, "Towards Clone Detection in UML Domain Models," *J. Software & Systems Modeling*, vol. 12, no. 2, pp. 307–329, 2013.
23. M. Tichy, C. Krause, and G. Liebel, "Detecting Performance Bad Smells for Henshin Model Transformations," *AMT workshop*, vol. 1077, 2013.
24. H. Störrle, "On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters," in *Int. Conference on Model Driven Engineering Languages and Systems*. Springer, 2014, pp. 518–534.
25. X. Yan and J. Han, "gspan: Graph-Based Substructure Pattern Mining," in *ICDM'03*. IEEE, 2002, pp. 721–724.
26. N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Complete and Accurate Clone Detection in Graph-Based Models," in *Int. Conference on Software Engineering*. IEEE, 2009, pp. 276–286.
27. F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz, "Model Clone Detection in Practice," in *Ws. on Software Clones*. ACM, 2010, pp. 57–64.
28. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: advanced concepts and tools for in-place EMF model transformations," in *Int. Conference on Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 121–135.
29. T. Arendt, A. Habel, H. Radke, and G. Taentzer, "From Core OCL Invariants to Nested Graph Constraints." in *Int. Conference on Graph Transformation*. Springer, 2014, pp. 97–112.
30. J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr, "Reasoning about product-line evolution using complex feature model differences," *Journal of Automated Software Engineering*, pp. 1–47, 2015.
31. D. Strüber and S. Schulz, "A tool environment for managing families of model transformation rules," in *Int. Conference on Graph Transformation*. Springer, 2016.
32. H. Störrle, "Effective and efficient model clone detection," in *Software, Services, and Systems*. Springer, 2015, pp. 440–457.
33. C. C. Ekanayake, M. Dumas, L. García-Bañuelos, M. La Rosa, and A. H. ter Hofstede, "Approximate clone detection in repositories of business process models," in *Business Process Management*. Springer, 2012, pp. 302–318.
34. Z. Liang, Y. Cheng, and J. Chen, "A novel optimized path-based algorithm for model clone detection," *Journal of Software*, vol. 9, no. 7, pp. 1810–1817, 2014.
35. M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, "Models are code too: Near-miss clone detection for simulink models," in *Int. Conference on Software Maintenance*. IEEE, 2012, pp. 295–304.
36. M. F. Van Amstel and M. G. Van Den Brand, "Model transformation analysis: staying ahead of the maintenance nightmare," in *Int. Conference on Model Transformation*. Springer, 2011, pp. 108–122.
37. L. Kapová, T. Goldschmidt, S. Becker, and J. Henss, "Evaluating maintainability with code metrics for model-to-model transformations," in *Research into Practice–Reality and Gaps*. Springer, 2010, pp. 151–166.
38. M. Wimmer, S. M. Perez, F. Jouault, and J. Cabot, "A catalogue of refactorings for model-to-model transformations." *Journal of Object Technology*, vol. 11, no. 2, pp. 1–40, 2012.
39. C. M. Gerpheide, R. R. Schiffelers, and A. Serebrenik, "Assessing and improving quality of QVTo model transformations," *Software Quality Journal*, pp. 1–38, 2014.