Softw Syst Model (2019) 18:995–1016
https://doi.org/10.1007/s10270-017-0625-6

SPECIAL SECTION PAPER

Model clone detection for rule-based model transformation languages

Daniel Strüber[1,2] · **Vlad Acreţoaie**[3,4] · **Jennifer Plöger**[2]

Received: 15 November 2016 / Revised: 14 April 2017 / Accepted: 3 September 2017 / Published online: 6 October 2017

Abstract Cloning is a convenient mechanism to enable reuse across and within software artifacts. On the downside, it is also a practice related to severe long-term maintainability impediments, thus generating a need to identify clones in affected artifacts. A large variety of clone detection techniques have 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 rule-based model transformation languages, including graph-based ones, such as Henshin, and hybrid ones, such as ATL. We introduce use cases for such techniques in the context of constructive and analytical quality assurance, and a set of key requirements we derived from these use cases. To address these requirements, we describe our customization of existing model clone detection techniques: We consider eScan, an a-priori-based technique, ConQAT, a heuristic technique, and a hybrid technique based on a combination of eScan and ConQAT. We compare these techniques in a comprehensive experimental evaluation, based on three realistic Henshin rule sets, and a comprehensive body of examples from the ATL transformation zoo. Our results indicate that our customization of ConQAT enables the efficient detection of the considered clones, without sacrificing accuracy. With our contributions, we present the first evidence on the usefulness of model clone detection for the quality assurance of model transformations and pave the way for future research efforts at the intersection of model clone detection and model transformation.

Keywords Quality assurance · Model clone detection · Model transformation · ATL · Henshin

1 Introduction

Model transformation is of paramount importance to Model-Driven Engineering [1]. 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 model transformations in the *copy–paste–modify* paradigm, provides key benefits for the creation phase; it is a fast, easy and universally applicable practice. On the downside, cloning presents severe maintainability challenges. 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 [2], it seems advisable to address this trade-off explicitly.

The drawbacks of cloning are well known from research on the general issue of *software clones*. Yet, despite a substantial body of research [3], there is no universally accepted directive for how to proceed with clones. In the seminal work by Fowler [4], clones are deemed one particular kind of "bad smell." In this view, a refactoring to a better suited abstraction is generally recommended. Empirical studies lead to a more nuanced view: Kim et al. [5] identify different types of clones, some of them warranting a

Communicated by Prof. Andrzej Wasowski and Pieter van Gorp.

✉ Daniel Strüber
strueber@uni-koblenz.de

1 University of Koblenz and Landau, Koblenz, Germany

2 Philipp University of Marburg, Marburg, Germany

3 Technical University of Denmark, Kgs. Lyngby, Denmark

4 Configit, Copenhagen, Denmark

refactoring towards suitable abstractions, others rendering such efforts clearly unjustified. Still, despite controversy on the question of how to *proceed* with clones, there seems to be a consensus that software clones "should at least be *detected*" [6].

While numerous automated clone detection techniques for programming and modeling languages have been proposed [7], 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 [8]. Clone detection can be an enabling technology for the evolution of existing transformation programs towards these reuse mechanisms. But the variety of potential use cases for clone detection are even broader. It includes the quality assessment of existing transformations, performance optimizations and even the identification of new design patterns. One contribution of this paper is an overview of such use cases.

The combination of different model transformation paradigms and use cases leads to a large design space for clone detection techniques. In this paper, we approach this design space from a specific angle: We focus on *rule-based model transformation languages*; in particular, we consider two language paradigms where rules play a key role [9]: *graph-based* languages, in which transformations are expressed in terms of graph rewriting rules, and *hybrid* languages, in which declarative rules are combined with imperative language constructs. This selection allows us to study cloning in two main paradigms of model transformations [9,10] which are related by a common denominator, the notion of rules.

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.

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. Therefore, it is advisable to provide dedicated support for the copying and editing of such rules. For instance, the rules could be unified using a reuse mechanism provided by the model transformation language. Alternatively, the consistent editing of the rules could be facilitated by tool support. In both cases, a prerequisite for an improved management of clones is their detection.

Contributions. This paper extends our recent work on clone detection for graph-based languages [12] in two main directions. First, in addition to graph-based languages, we consider hybrid ones. Hybrid languages are a particularly interesting complementary case where the notion of *rule* is of equivalent importance, but expresses a slightly different concept. Specifically, graph-based rules specify rewriting patterns, whereas the rules in hybrid languages can be seen as specifications of mappings between meta-models. The former concept lends itself to application in endogenous transformation scenarios; the latter is suitable for use in exogenous scenarios. Both paradigms are increasingly applied in industrial and academic contexts [13]. Second, we provide a considerably extended evaluation. To evaluate clone detection approaches for graph-based rules in a large-scale setting, we consider an additional scenario with a rule set of 1404 rules. For evaluation of the hybrid ones, we performed an in-depth analysis of rules from the ATL zoo,[1] comprising 2566 rules in total.

In this work, we make the following contributions:

- We discuss use cases of clone detection for model transformation languages. The discussion is informed by

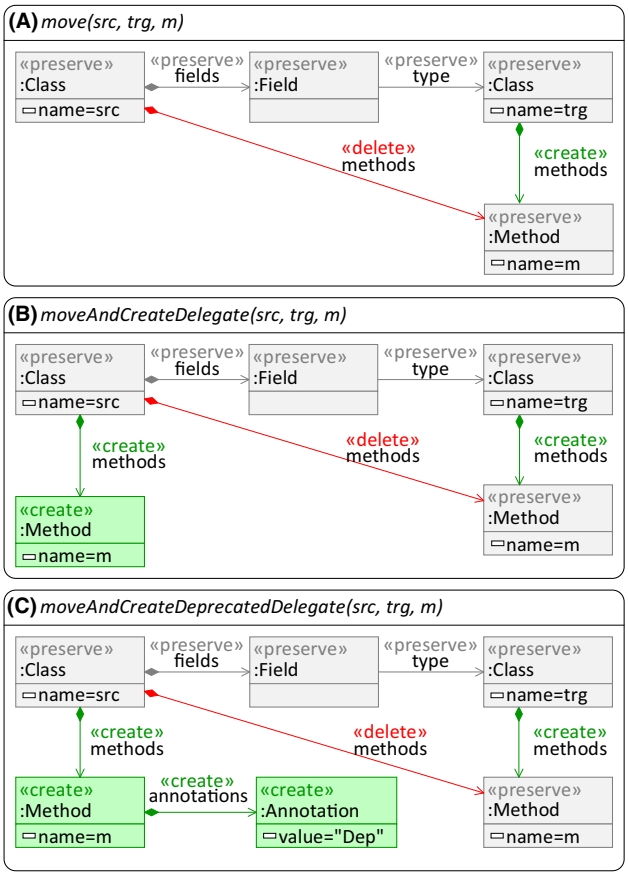

Fig. 1 Rules affected by cloning (from [11])

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

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 rule-based model transformations.
- To address these requirements, we introduce a set of adaptations of existing model clone detection techniques. We consider eScan, an a-priori-based technique, ConQAT, a heuristic one, and a hybrid composed of eSan and ConQAT.
  Our adaptations are tailored to the graph-based language Henshin [14,15] and the hybrid language ATL [16].
- We provide an extensive experimental evaluation based on three realistic graph-based transformations and a body of example transformations from the ATL transformation zoo.

An established taxonomy of software clones [3] distinguishes four types of clones based on their level of similarity: *Type I* clones are *identical* fragments; *Type II* are *almost identical* except for naming. *Type III* or *near-miss* clones are identical except for subtle differences, such as the presence or absence of certain elements, and *Type IV* or *semantic* clones represent duplicate increments of functionality where the duplication is not directly reflected on the syntactic level. Since Type I and II clones are routinely produced when transformation systems are developed in a copy-and-paste manner, this work focuses on these types of clones.

The rationale behind the selection of eScan and ConQAT is to compare an a-priori-based and a heuristic approach to the detection of Type I and II clones. A key finding of our evaluation is that the accuracy of ConQAT was nearly optimal when using the results of eScan and ScanQAT as a ground truth. At the same time, the heuristic approach was the only one scaling up to complete realistic rule sets, rather than just selected subsets. With this finding, we provide a first insight into the application of model clone detection to model transformations. In the future, we aim to study the case of Type III clones using the SIMONE clone detector [17], which has also been shown to produce excellent results for Type I and II clones. The detection of Type IV clones is another interesting research avenue, eliciting the question of how *semantically equivalent* model transformations can be identified.

The rest of this paper is structured as follows. In Sect. 2, we outline the identified use cases. In Sect. 3, we present the necessary 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 in Sect. 8 and suggest future research directions in Sect. 9.

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 [4]. Its outcome is a semantically equivalent, yet syntactically refined representation of the input artifacts. While the strategies used in specific refactorings may vary, they share the common requirement that a target reuse mechanism is assumed. In the case of model transformations, reuse approaches such as *rule inheritance* [18], *refinement* [19] or *variability-based rules* [20] have emerged recently and are now available to developers. For instance, the rules in Fig. 1 can be expressed using rule refinement: To this end, rule A becomes a basis rule, while the individual parts of rules B and C are captured via two subrules. Conversely, the same rules can be represented by one variability-based rule, augmenting rule C with variability annotations to denote individual parts of rules B and C. 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 *scope* or *granularity* of affected clones. For instance, an external reuse mechanism [8] 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 [6]. Despite these benefits, the existing drawbacks may remain. 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 [21,22].

Assessing specifications and languages. Clone detection can be used during the assessment of transformation specifications, for instance, in a quality assurance process [23] or to evaluate 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. Finally, clone detection might be useful to improve the detection of design-pattern and anti-pattern instances [24,25]: In contrast to model query engines, which generally find exact matches of a particular pattern, clone detection can identify common subpatterns as well. The detection of

frequent patterns in transformation specifications can even lead to the identification of new design patterns and anti-patterns. 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 [26]. 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 [27]. 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 rule-based model transformation languages, focusing on graph-based and hybrid languages with rules. While we consider clones in these two paradigms separately, a unifying idea is that clones are defined as *common parts of multiple rules*. In the graph case, this idea leads to the notion of subrule: Common parts of multiple rules are generally self-contained rules of their own. In the hybrid case, clones may be rule fragments which do not form a self-contained rule.

3.1 Graph-based model transformation languages

We first 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 [11,28] 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* [29].

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 subpattern being present in a set of rules*. Such a subpattern 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.

For simplicity, the formal treatment in this work does not address an advanced feature of graph-based model transformation called *negative application conditions* (NACs, see

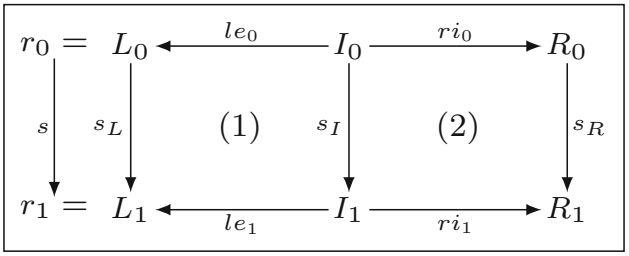

Fig. 2 Subrule morphism

[30]). In a recent work [31], we show how the concept of subrule can be extended to rules with NACs: A NAC is a graph with a graph morphism between the left-hand side of the rule and this graph. To define the subrule relation, the notion of *shifting* a NAC over a given graph morphism [32] is key: Each NAC of the subrule needs to have a counterpart in the superrule that results from shifting the NAC over the embedding morphism.

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

Definition 3 (*Rule 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, that is, 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 characterized 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.

In the established taxonomy of software clones [3], 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 based on their behavior is an interesting avenue for future work.

We further distinguish clones based on their scope.

Table 1 Full clones in the example graph-based rules

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

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

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

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 [33]. 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 [8] correspond to these scopes. Micro-clones can be avoided by specifying multiplicity at the level of individual graph nodes and edges [34]. Internal clones can be replaced using reuse mechanisms such as rule inheritance [18], refinement [19] or variability-based rules [20]. A suitable alternative to the creation of external clones are external reuse approaches, such as generic model transformations [35].

3.2 Hybrid model transformation languages

Our investigation of clones in hybrid languages is inspired by the ATL Transformation Language (ATL, [16]). ATL features a declarative rule concept called *matched rule*. A matched rule comprises a source pattern, specifying a set of *source elements*, and an optional OCL constraint called *guard*, preventing the rule from being applied if the constraint is not fulfilled. In addition, a rule has a target pattern, specifying a set of *target elements*. Each target element has a set of *bindings*, specifying values for references and attributes of the target element. Finally, rules may declare *variables* that can be used within the target pattern elements. In what follows, we focus on matched rules, leaving the imperative parts of ATL such as *helpers* and *statements* outside our scope.

Example. This example is motivated by a large real-life clone we identified during our evaluation. Consider a model-to-model transformation that refines *class models* to *program models*. Just as class models do, program mod-

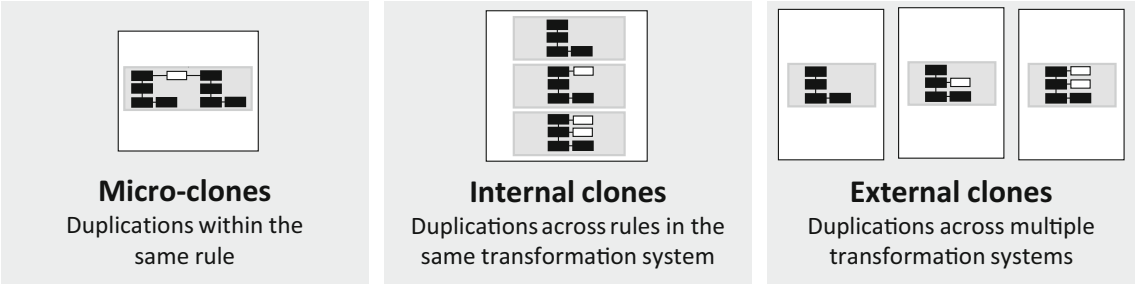

Fig. 3 Scope of clones in model transformation systems

```
1 module ClassModel2ProgramModel;
2 create OUT : ProgramModel from IN : ClassModel;
3
4 rule ClassNotDeprecated {
5     from
6         s : ClassModel!Class (not s.isDeprecated)
7     to
8         t1 : ProgramModel!Class (
9            name <- s.name,
10           superClass <- s.superClass,
11           isAbstract <- s.isAbstract,
12           isInterface <- s.isInterface
13        )
14 }
15
16 rule ClassDeprecated {
17    from
18        s : ClassModel!Class (s.isDeprecated)
19    to
20        t1 : ProgramModel!Class (
21           name <- s.name,
22           superClass <- s.superClass,
23           isAbstract <- s.isAbstract,
24           isInterface <- s.isInterface,
25           annotation <- t2
26        ),
27        t2: : ProgramModel!Annotation (
28           value <- "Deprecated"
29        )
30 }
```

Fig. 4 ATL module with two matched rules

els have classes, fields and methods. In addition, program models have some additional elements that can be used during code generation, such as annotations. In this context, Fig. 4 shows two matched rules dealing with the transformation of classes. The rules address the complementary cases where a boolean attribute called *deprecated* is set to *false* or *true*, respectively. The first rule deals with the *false* case: It ensures that names, super-classes and the values of the *abstract* and *interface* attributes are transferred correctly during the refinement of classes. The second rule addresses the *true* case. It does the same as the first rule. In addition, it establishes that an annotation `@deprecated` is created together with the class. Since these two rules share a significant number of commonalities, a refactoring seems recommended.

The following definition captures such rules. In particular, matched rules are required to have non-empty sets of source and target pattern elements. If this is not the case, we only speak of *fragments*. Each matched rule is also a fragment, but not vice versa.

Definition 6 (*Matched rule fragment*) A *matched rule fragment* $r = (S, g, V, T)$, in short: *fragment*, consists of a set S of atomic elements, called *source pattern elements*, a constraint g, called *guard*, a set V of key–value pairs, called *variables*, and a set T of *output pattern elements*. An output pattern element is a pair (t, B_t) of an atomic element t and a set B_t of key–value pairs, called *bindings*.
If both S and T are non-empty, r is called a *matched rule*.

Rule `ClassNotDeprecated` is a matched rule in which $S = \{\texttt{s}\}$, $g = \texttt{'not s.isDeprecated'}$, $V = \emptyset$ and $T = \{(\texttt{t1}, B_{\texttt{t1}})\}$ s.t. $B_{\texttt{t1}}$ contains the four shown entries. Note that the specification of guards is optional: If a rule does not specify a guard explicitly, the guard of that rule is `true`, that is, it is always fulfilled.

As a prerequisite for our notion of clones, we consider a subfragment relation:

Definition 7 (*Subfragment*) Given a pair of fragments $f = (S_f, g_f, V_f, T_f)$ and $s = (S_s, g_s, V_s, T_s)$, s is a subfragment of f iff

- $S_s \subseteq S_f$,
- $g_s \in \{g_f, true\}$,
- $V_s \subseteq V_f$
- and $\forall (t, B_t) \in T_s$ there $\exists (t, B'_t) \in T_f$ s.t. $B_t \subseteq B'_t$.

For instance, if the guard of rule `ClassNotDeprecated` was replaced with `true`, then this rule would be a subfragment of `ClassDeprecated`.

Definition 8 (*Clone of matched rules*) Given a set $\mathcal{R}$ of matched rules, a fragment c is called a clone over $\mathcal{R}$ if c is a subfragment of at least two rules in $\mathcal{R}$. A clone c induces a set of *affected rules* $\mathcal{R}_{aff}(c) = \{r \in \mathcal{R} \mid c \text{ is a subfragment of } r\}$.

Rules `ClassDeprecated` and `ClassNotDeprecated` are affected by clones: For instance, one clone is the fragment with $S = \{\texttt{s}\}$, $g = \text{true}$, $V = \emptyset$ and $T =$

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)

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.*	■ ■ □ ■ ▪

■ = Hard requirement, ▪ = soft requirement, □ = not required

$\{(\texttt{t1}, B_{\texttt{t1}})\}$ s.t. $B_{\texttt{t1}}$ contains the four bindings from rule `ClassNotDeprecated`.

Taking up the same distinction as in Definition 4, clones can be either full or partial.

Definition 9 (*Full and partial clones*) A clone c over a set of matched rules $\mathcal{R}$ is a *full clone* iff there is no clone c' over the same rule set so that c' is a subfragment of c. Non-full clones are called *partial* clones.

The aforementioned clone is a full one. If one of the four bindings was missing, the result would be a partial clone, since this clone would be a subfragment of the aforementioned one.

Similar to clones of graph-based rules, this definition may include *Type I* and *II* clones [6], *identical* and *almost identical* (except for naming) clones, depending on whether we consider the name of an element to be a part of its identity. As a *Type III* clone, we may consider a set of rules that only differ in a number of elements, where the number is below a certain user-defined similarity threshold. *Type IV* clones may be addressed by also considering the imperative part of an ATL transformation.

We can define the scope of a clone as either internal or external. An internal clone spans exactly one module, while an external one spans multiple ones.

Definition 10 (*Scope*) The scope of a clone is either INTERNAL or EXTERNAL.

$$scope(c) = \begin{cases} \text{INTERNAL} & | \exists\, module\ \mathcal{T}\ s.t.\ \mathcal{R}_{aff}(c) \subseteq \mathcal{T} \\ \text{EXTERNAL} & else \end{cases}$$

In contrast to Definition 5, micro-clones are not part of this definition, since we assume that elements can be uniquely identified. Therefore, there is generally exactly one way to map a subfragment to a matched rule.

4 Requirements

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

(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 [36]. 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 [37].

(R2) Performance. Clone detection needs to support scenarios with many rules and large individual rules—arguably situations where maintainability is problematic [38]. 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 subgraph isomorphism problem [39]. 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* (Definition 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 qual-

ity 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 Definition 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 our adaptations of existing clone detection techniques to the requirements of rule-based model transformations.

We considered the applicability of several clone detection techniques. Since rule patterns are abstractions of model structures, the most suitable candidate techniques are those focusing on *model clone detection*. We consider two techniques, *eScan* [40] and *ConQAT* [41,42], as they allow us to address R1, the identification of identical patterns in their input models, by supporting the identification of Type I and II clones. 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.

We selected eScan and ConQAT since they represent two complementary paradigms to the detection of Type I and II clones: eScan provides an *a priori* strategy that can generally achieve perfect precision and recall during clone detection, assuming unlimited memory and time. ConQAT provides an *heuristic* strategy that improves the efficiency during clone detection by trimming the search space to focus on the most viable alternatives, which comes at the cost of decreased recall. The study of alternative heuristic approaches such as the text-based one of SIMONE [17] is an interesting avenue for future work; it might become especially relevant when we extend the present work to address Type III clones, which are only supported by SIMONE so far.

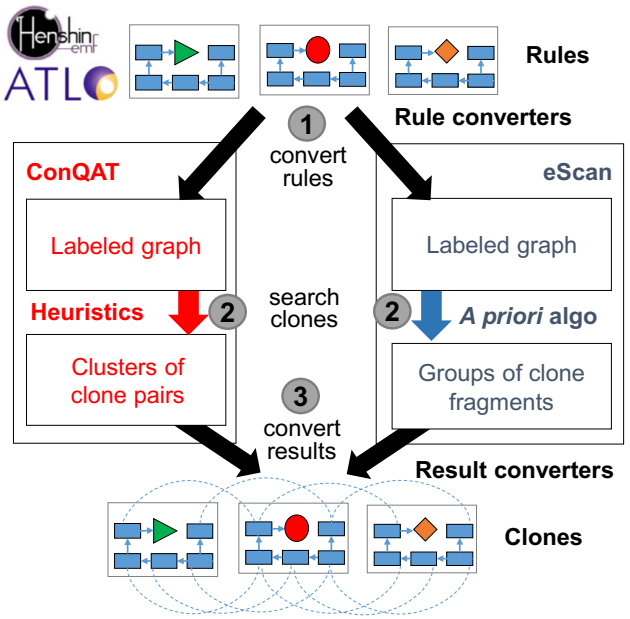

Fig. 5 Overview of adapted clone detection techniques

For both considered techniques, our adaptations follow the same basic process shown in Fig. 5. First, as both techniques assume a directed, labeled graph as input data structure, we provide rule converters that encode the input rules into such graphs. Second, the actual clone search takes place. Third, we provide converters to map the identified clones back to the original rules.

5.1 Adaptation for graph-based model transformation

5.1.1 Phase 1: Convert rules

Our rule converters produce the following encoding of Henshin rules, illustrated in Fig. 6.

- *Nodes*: Henshin rules arrive in the form of *multiple* graphs, following Definition 1. Our encoding of these graphs to just one graph is inspired by the representation shown in Fig. 1. The action and type assigned to a node are 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, thereby turning it into a *delete* node.
- *Edges*: Similar to nodes, each edge contained in the rule is represented by one edge in the graph, labeled with its action and type.

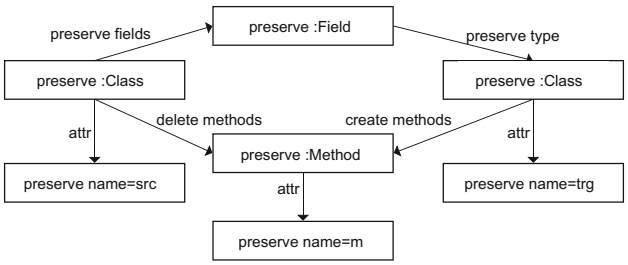

Fig. 6 Labeled graph for encoding the Henshin rule `move` (cf. Fig. 1)

– *Attributes*: Attributes are represented as additional graph elements. Each attribute becomes a pair of a node and an edge, labeled with the attribute action, type and value. Encoding attributes as distinct elements allows us to account for reuse mechanisms that accommodate the attribute level.

Note that in our current implementation, we focus on representing the elements of rules that can be encoded using the actions *delete*, *preserve* and *create*, which correspond to the rule parts stipulated in Definition 1. An additional rule feature not addressed yet are NACs. However, based on the extended formalization discussed in Sect. 3, an adaptation to NACs would be straightforward: Each NAC can be represented using one additional action. For example, in a rule with two NACs `n1` and `n2`, we would have two additional actions, *forbid#n1* and *forbid#n2*.

5.1.2 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 [40,42].

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 subgraphs 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, that is, 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.[2] 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 rules 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*, that is, subgraphs with an isomorphic counterpart, contained in the input graph. Starting with subgraphs comprising of just one edge and its source and target node, eScan produces larger subgraphs incrementally. In each iteration, given the cloned subgraphs with k edges, eScan finds the set of (k+1) edge subgraphs by including additional edges from the graph. Subgraphs without isomorphic counterparts are discarded. Isomorphy between subgraphs is detected by comparing their canonical labels, an encoded representation of their elements. An optimization ensures that each subgraph is used as a starting point just once.

In the example, the input graph contains 15 subgraphs of size 1: four in rule A, five in rule B and six in rule C.[2] With the exception of the *annotations* edge in rule C, each of these subgraphs is a clone fragment and is consequently used to derive subgraphs 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.

Both approaches include a post-processing of the clone search result. The two main actions are *clustering* and *filtering*. Clustering groups individual clones to sets of isomorphic subgraphs. From these sets, filtering removes entries that are completely covered by another one. For instance, in the eScan result, the groups containing the subgraphs 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.

Note that ConQAT and eScan report only *connected* subgraphs as clones. Where desirable, larger unconnected clones can be assembled from connected ones by enumerating the power set of the set of clones a particular rule is affected by.

5.1.3 Phase 3: Convert results

To obtain clones (Definition 3), we map the results of Phase 2 back to the rules, using a mapping that we created during Phase 1. In the example, both approaches produce the output shown in Table 1.

5.2 Adaptation for hybrid model transformations

5.2.1 Phase 1: Convert rules

Our rule converters produce the following encoding of ATL rules to the ConQAT and eScan input data structures, illustrated in Fig. 7:

[2] For simplicity, we neglect attributes in these illustrations.

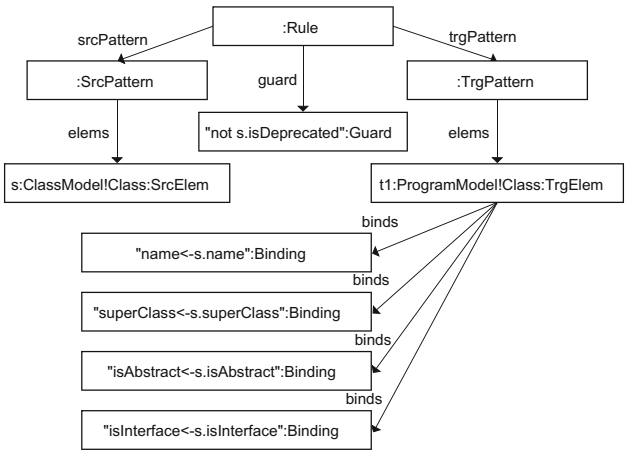

Fig. 7 Labeled graph for encoding the ATL rule `ClassNotDeprecated` (cf. Fig. 4)

– *Nodes*: Given a matched rule, we create one node for each of the following objects: the rule itself, its guard, the source and target patterns, each source pattern element, each variable, each target pattern element and each binding of each target pattern element. The labels specify all information required to determine elements identities as required for the subfragment relationship (see Definition 7). The node labels for rules as well as source and target patterns are simply `:Rule`, `:SrcPattern` and `:TrgPattern`, since the identity of these elements is irrelevant for clone detection. For the remaining elements, we use the following information: for source pattern elements, the name and type; for the *guard* constraint, the constraint value; for variables, the name, type and value; for target pattern elements, the name and type; for bindings, the type and value.
– *Edges*: Edges represent the containment references between the represented elements. Apart from containment ones, no other kind of reference is relevant in our definition of clones.

5.2.2 Phase 2: Clone search

For a description of the internal workings of ConQAT and eScan, see Sect. 5.1.2.

5.2.3 Phase 3: Convert results

Again we convert the clone detection results to actual rule clones, using stored mappings between both artifacts from Phase 1. In the case of the example, this step yields precisely one clone, indicated in Table 3. The clone comprises elements `s` as well as `t1` with its four bindings, leading to a size of 6 (`:Rule`, `:SrcPattern` and `:TrgPattern` are not part of the clone).

Table 3 Full clones in the example ATL rules

Name	Rules	Size
C1	{ClassNotDeprecated, ClassDeprecated}	6

In general, the employed strategy during Phase 2 may have implications for the exhaustiveness of the result (R3). Since eScan eventually produces every possible sub-graph, it finds all full clones (Definition 4)—assuming unlimited memory and time. In practice, eScan has been shown not to scale up to larger models in the Simulink domain [42]. 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 rule-based model transformation clone detection be satisfied by adapting existing model clone detection techniques?

Using our customizations of ConQAT and eScan, 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 conducted an experimental evaluation. To this end, we applied each technique on rule sets from realistic model transformation systems and measured execution time.
– In the course of our experimental evaluation, we also studied *exhaustiveness* (R3). While eScan guarantees exhaustiveness by design, we devised a custom setup to study the exhaustiveness of ConQAT: We fed the largest clones reported by ConQAT as input to *eScan-Inc* [40], an incremental variant of eScan that allows continuing the clone search from clones of a given size. This method that we call *ScanQAT* potentially allows any clones missed by ConQAT to be detected. 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 [14].

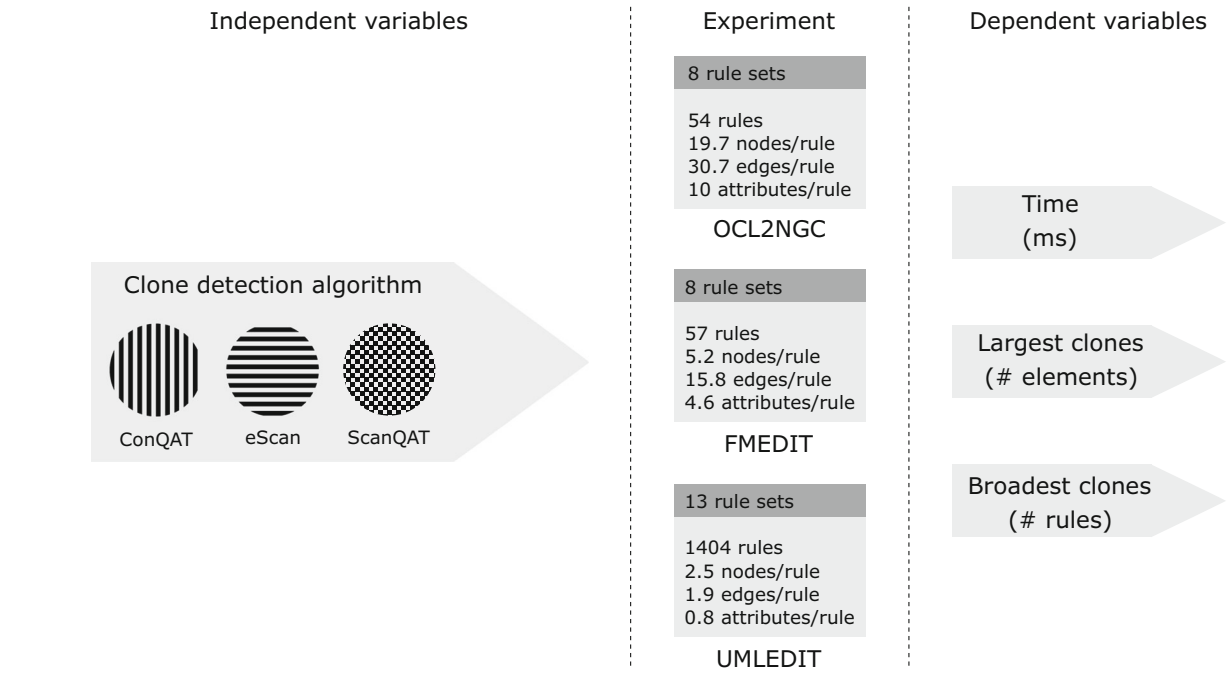

Fig. 8 Experimental design for graph-based model transformations

In the rest of this section, we first present the setup and results of our experimental evaluation, focusing on the requirements of *performance* (R2) and *exhaustiveness* (R3). We start with the case of graph-based model transformations in Sect. 6.1 and continue with hybrid ones in Sect. 6.2. In our wrap-up discussion in Sect. 6.3, we also address *scope* (R4) and *tool integration* (R5). Finally, in Sect. 6.4, we discuss threats to validity.

6.1 Graph-based model transformation

6.1.1 Methods and materials

An overview of our experimental design and detailed information on all rule sets is given in Fig. 8. As the independent variable, we varied the clone detection technique by applying ConQAT, eScan and the hybrid ScanQAT (described above). In our experiments, we used rule sets from three transformation scenarios. 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 [43]. FMEDIT and UMLEDIT are sets of editing rules for feature models and UML models, respectively, used in the context of model differencing [44]. All rule sets have recently been made publicly available as part of a benchmark set [45].

Table 4 provides detailed metrics information on the used rule sets. The rules in OCL2NGC are organized in sets of 4–7 rules. The rules in FMEDIT are organized in sets of 2–11 rules. The rules in UMLEDIT are organized in sets of 22–682 automatically generated rules, and 2–8 manually created ones. 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 and UMLEDIT, we studied all rule sets. These sets provide a semantic grouping of the transformations without prescribing a particular control flow. In addition, 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.

We created an implementation prototype for our experiments, implementing the customization outlined in Sect. 5. For Phase 2, in the case of ConQAT we used the publicly available implementation.[3] We created our own implementation of eScan as no existing one was available to us. We ran all experiments on a Windows 10 system (2.6 GHz; 8 GB of RAM).

6.1.2 Results

We applied the techniques on all rule sets, obtaining the results shown in Tables 5, 6 and 7. For each combination of technique and rule set, we show the largest and the broadest

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

Table 4 Sample rule sets with number of rules (#R) and average number of nodes (#N), edges (#E) and attributes (#A) in each rule

Rule set	#R	#N	#E	#A
(a) OCL2NGC				
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
(b) FMEDIT				
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
(c) UMLEDIT				
gen/ADD	26	2.1	1.5	0.0
gen/CHANGE	282	1.0	0.0	1.0
gen/CREATE	100	2.7	2.1	4.8
gen/DELETE	105	2.3	1.7	0.0
gen/MOVE	682	3.5	3.0	0.1
gen/REMOVE	26	2.1	1.5	0.0
gen/SET	136	1.2	0.3	0.8
gen/UNSET	22	2.1	1.3	0.0
man/CREATE	8	3.9	5.9	13.3
man/DELETE	8	3.8	5.6	0.0
man/MOVE	2	3.0	2.0	0.0
man/SET	4	1.8	0.8	0.3
man/UNSET	3.0	2.0	1.0	0.0
All	1404	2.5	1.9	0.8

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. The performance plots in Fig. 9a–c show the runtimes in relation to the sizes of the considered rule sets. Size was measured in terms of the accumulative number of elements in all included rules; note the logarithmic scale on the time axis. ConQAT took between 7 ms and 8.8 s for each individual rule set. When applied to the full rule sets, it took 64.4 s for OCL2NGC, 3.7 s for UMLEDITand 1.1 s for FMEDIT. Our ScanQAT and eScan implementations did not yield results for all rule sets: In some cases, they terminated with memory overflow errors or did not terminate within 1 h. In the result tables, these cases are indicated by dashes. In the cases where they produced results, they mostly did so in less than 10 s; yet the longest successful runs took 57 s for eScan and 16.7 min for ScanQAT, respectively. Note that our implementations of eScan and ScanQat may in principle be flawed. Yet our experience of memory issues is in line with earlier experiments in the Simulink domain [42].

Exhaustiveness. In the cases where eScan and ScanQAT did not yield results, we cannot evaluate the exhaustiveness of ConQAT. In the other cases, the clones reported by ConQAT, ScanQAT and eScan for the rule sets of OCL2NGC and FMEDIT were always identical in size. In the case of UMLEDIT, we observed a number of subtle differences between the reported results: The largest clone found by ConQAT for the `gen/CHANGE` and `gen/MOVE` rule sets spanned more or fewer rules, respectively, than the ones reported by ScanQAT and eScan did. In `man/DELETE`, the reported largest clone was a slightly smaller one. These observations indicate that in situations where perfect exhaustiveness is required, a scalable tool is still not available. In other cases, ConQAT seems generally suitable to address the exhaustiveness requirement. Note that 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.

6.1.3 Methods and materials

Figure 10 provides an overview of our experimental evaluation of clone detection techniques for hybrid languages. Again we considered our adaptions of ConQAT and eScan as well as the hybrid clone detector ScanQAT. We used samples from the ATL transformation zoo,[4] called ATLZOO in short. ATLZOO is an online collection of 103 transformation scenarios with 202 modules in total. Several modules occur in multiple versions, a particularly interesting scenario for clone detection.

We applied ConQAT, ScanQAT and eScan to ATLZOO in its entirety. In addition, to study performance and exhaustiveness in a more detailed manner, we applied them to suitable subsets of ATLZOO. Specifically, in a preparation step, we identified 21 "clusters" of modules that are mutually related via common clones. To this end, we applied our adaptation of ConQAT to get a first approximation of present clones. To group the modules based on the identified clones, we used a standard clustering algorithm: *hierarchi-*

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

Table 5 OCL2NGC results

Rules	ConQAT				eScan				ScanQAT			
	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	2	10	14	3
	4	9	11	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	–				–			

For each rule set, the *largest* (first row) and the *broadest* (second row) reported clones are denoted with their number of rules (R), nodes (N), edges (E) and attributes (A). "–" denotes memory-related program exits or execution times longer than 1 h

Table 6 FMEDIT results

Rules	ConQAT				eScan				ScanQAT			
	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	–				–			

See footnote of Table 5

Table 7 UMLEDIT results (excluding five rule sets for which no clone was found by either technique: gen/{SET,UNSET}, man/{MOVE, SET, UNSET})

Rules	ConQAT				eScan				ScanQAT			
	R	N	E	A	R	N	E	A	R	N	E	A
gen/ADD	2	2	2	0	2	2	2	0	2	2	2	0
	2	2	2	0	2	2	2	0	2	2	2	0
gen/CHANGE	9	0	0	1	5	0	0	1	5	0	0	1
	11	0	0	1	11	0	0	1	11	0	0	1
gen/CREATE	7	2	2	10	7	2	2	10	7	2	2	10
	9	0	0	7	9	0	0	7	9	0	0	7
gen/DELETE	2	3	4	0	2	3	4	0	2	3	4	0
	7	2	2	0	7	2	2	0	7	2	2	0
gen/MOVE	2	3	4	0	12	3	4	0	11	3	4	0
	34	2	2	0	34	2	2	0	34	2	2	0
Gen/REMOVE	2	2	2	0	2	2	2	0	2	2	2	0
	2	2	2	0	2	2	2	0	2	2	2	0
man/CREATE	3	6	7	28	–				–			
	3	6	7	28	–				–			
man/DELETE	3	6	7	0	2	6	9	0	2	6	9	0
	3	6	7	0	3	6	7	0	3	6	7	0
All	3	6	7	28	–				–			
	34	2	2	0	–				–			

See footnote of Table 5

cal agglomerative clustering [46], which has been used as a grouping mechanism in the MDE context before [47]. We configured the clustering algorithm to use the *average linkage* strategy and clone size as the similarity metric.

An overview of the clusters obtained from this preparation is shown in Table 8. We discarded clusters that only contained a single module; the number of modules in the remaining clusters ranged from 2 to 46. A first interesting observation is that 166 of all 202 modules are contained in one of the clusters—in other words, 82% of all modules are affected by at least one external clone (Definition 10). Subscripts indicate versions of the same module. Several groups of versions are part of the same cluster, e.g., two versions of `UML2MOF`. Conversely, several groups have been split over multiple clusters, e.g., the eight versions of `TypeA2TypeB` are split over two clusters.

6.2 Hybrid model transformation

6.2.1 Results

The results of this experiment are shown in Table 9. Again, the table indicates the *largest* and *broadest* reported clones, that is, the ones incorporating the greatest number of elements and spanning the largest number of rules, respectively.

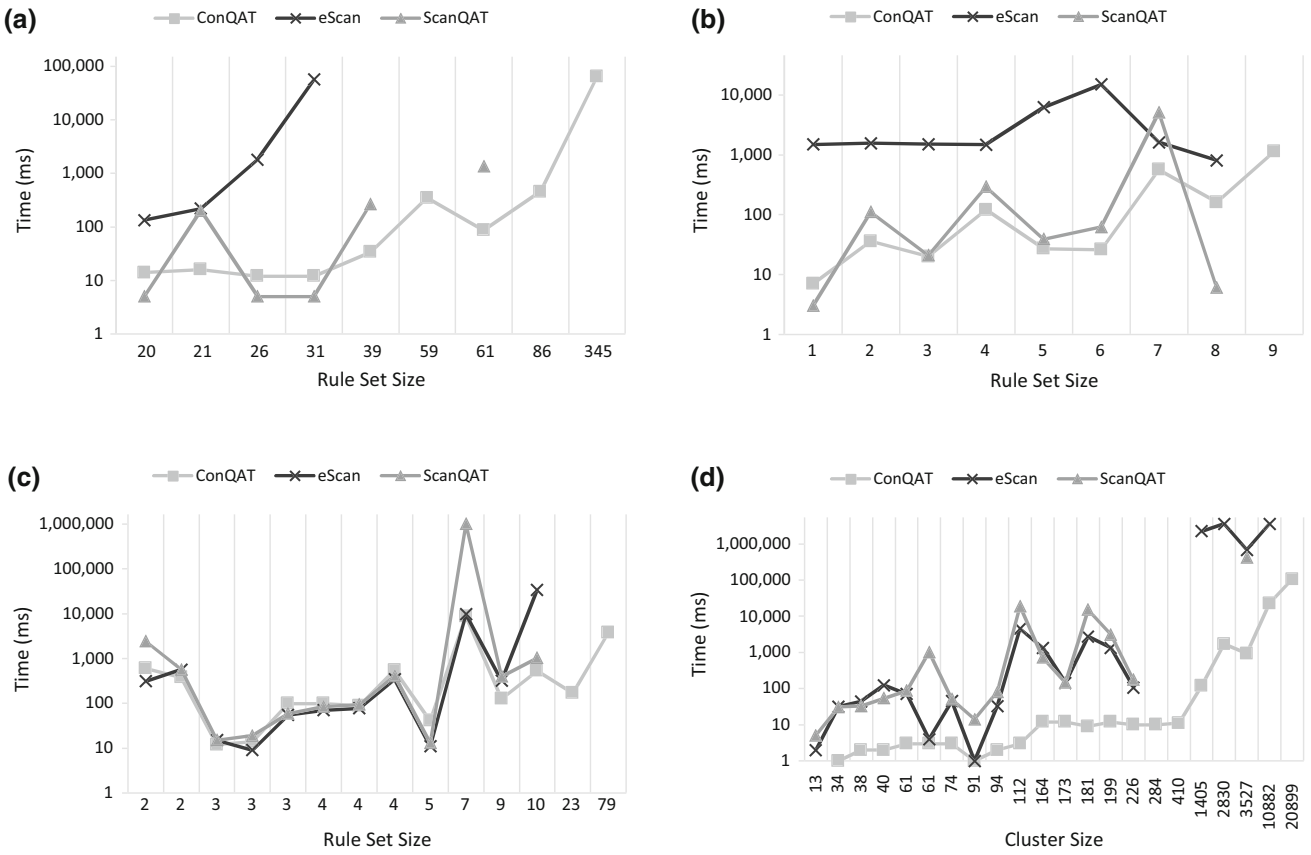

Fig. 9 Performance plots for the four example rule sets. **a** OCL2NGC, **b** FMEDIT, **c** UMLEDIT, **d** ATLZOO

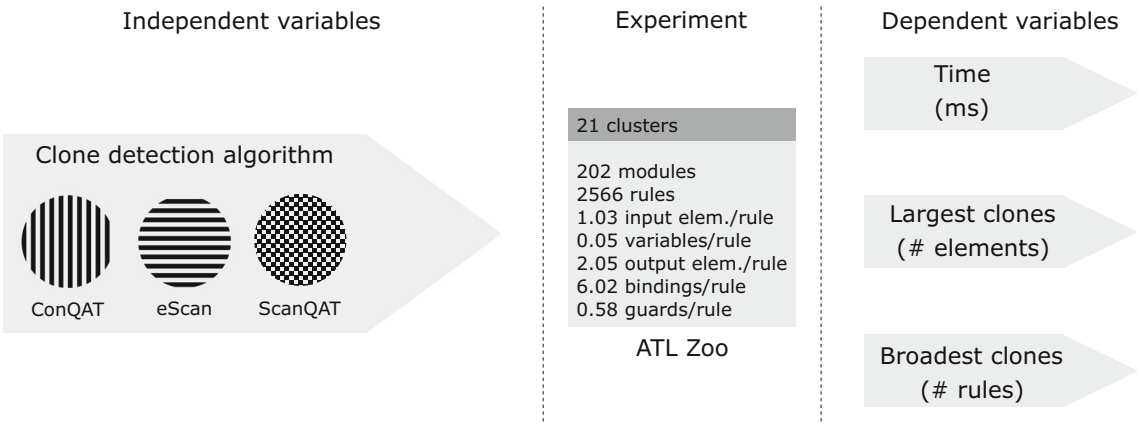

Fig. 10 Experimental design for hybrid model transformations

Performance. The performance plot in Fig. 9d relates execution time to the size of the considered modules. Size is given in terms of the accumulative number of elements in all rules of all included modules; note the logarithmic scale on the time axis. ConQAT took 111.3 s for the entire ATLZOO. For the individual clusters, it took between 0.2 and 1.3 s in three cases and less than 0.1 s in all other cases. eScan and ScanQAT did not produce results in six cases: the entire rule set, the three largest clusters and two clusters of medium size that included the modules `UML2MOF` and `MOF2UML`, respectively. A defining feature of the latter two cases is the relatively large size of their included rules, as characterized by the number of elements per rule. Large input graphs are challenging for eScan and ScanQAT as they suffer from the combinatorial explosion of opportunities for extending clone candidates during their search. In the cases where they reported a result, eScan took between 2 ms and 662 s; ScanQAT took between 7 and 340 ms.

Exhaustiveness. While eScan and ScanQAT are exhaustive by design, the exhaustiveness of ConQAT can only be assessed in the cases where a baseline result produced by either eScan or ScanQAT exists. Since eScan and ScanQAT did not terminate for the complete rule set and the five clusters mentioned above, we can only study the exhaustiveness in the remaining 16 of the 21 clusters.

In these cases, we find a complete agreement of all three techniques on the largest and broadest clones. In other words, the exhaustiveness provided by ConQAT was perfect in these cases. Among the determined clones were several ones of considerable size.

We consider some particular interesting data points more detailedly. The largest clone found for the entire ATLZOO concerns the rules `ConceptHasSuper` and `ConceptHasSuperAndisAbstract` in the `DSL2XML` module. This clone, a Type I clone including 39 common target pattern elements and 88 bindings is comparable to the one in the example shown in Fig. 4: It results from addressing two vastly similar cases of a translation that only differ in the handling of a meta-attribute of the considered source class. The broadest clone found for the complete ATLZOO, found in 340 rules, comprises a target pattern element called `o` of the type `XML!Element`. The broadest clone found in an individual set was found in the module `UML2Copy`, a guard of the value `(thisModule.inElements->includes(s))` found in each of the 167 rules in this module. Such small clones may appear to be not immediately useful to developers. However, it is worth noting that even clones of the smallest extent have been observed to be related to defects [33].

6.3 Discussion

Our results can be summarized as follows: ConQAT, ScanQAT and eScan were on par with regard to all identified requirements except performance, where ConQAT outperformed the other approaches. From the performance plots in Fig. 9, it is noticeable that the main issue of eScan and ScanQAT is scalability: while these techniques are capable of providing perfect exhaustiveness for inputs of a certain size, they do not cope with the combinatorial explosion during clone detection for larger inputs. Conversely, 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 [11]. The new findings indicate that ConQAT is potentially useful in all considered use cases, a conclusion that particularly applies to situations with considerably large rule sets such as UMLEDIT or ATLZOO. Moreover, we found that eScan and ScanQAT can be used to determine a ground truth for assessing the accuracy of heuristic techniques. However, for realistic rule sets in the magnitude of the ones used in our evaluation, it might be necessary to consider selected subsets for this purpose.

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 may represent one rule as well as multiple rules from the same or different transformation systems. An interesting border 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, that is, 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 [14]. Figure 11 shows a screenshot of the user interface: 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 more detailedly in another work [48]. Providing tool support for the ATL clone detectors created as part of this work is left to future work.

6.4 Threats to validity

Based on the classification of threats to the validity of software engineering experiments proposed by Wohlin et al. [49], our experiments are vulnerable to construct, internal and external validity threats. We discuss the implications and mitigation of these threats in what follows. Construct validity refers to the extent to which an experiment successfully measures the phenomena under investigation, in this case the performance and exhaustiveness of clone detection algorithms. A first threat to the construct validity of our study concerns exhaustiveness. We have not compared the results to a list of known clones, which would be the most reliable strategy. Unfortunately, such lists are difficult to produce manually for large rule sets. Furthermore, we only focus on the detection of the largest clones, ignoring smaller ones. While more comprehensive exhaustiveness studies are desirable, large clones are the most relevant for refactoring and performance optimizations.

Internal validity concerns an experiments capacity to highlight a causal relationship between its factors and outcomes. In our case, internal validity is ensured by the experimental design and by precautions taken while running the experiment. The adopted design does not suffer from potential confounding factors, and all trials were conducted

Table 8 Clusters of cloning-affected modules in ATLZOO, with number of modules (#M), rules (#R), source elements (#S), guards (#G), variables (#V), target elements (#T) and bindings (#B)

Cluster	#M	#R	#S	#G	#V	#T	#B
XML2RDM, R2ML2RDM, XML2MySQL, XML2Make, XML2Ant, etc.	58	1296	1299	845	41	3094	6448
ATL_WFR, ATL2BindingDebugger, ATL2Tracer, KM32SimpleClass, etc.	46	371	389	201	15	536	1890
Table2HTML, Table2TabularHTML, XML2SpreadsheetMLSimplified, etc.	14	71	105	30	19	346	935
Measure2XHTML$_2$, Measure2XHTML, KM32Measure$_2$, Measure2Table$_2$, etc.	5	29	42	0	11	36	110
TypeA2TypeB, TypeA2TypeB$_2$, TypeA2TypeB$_4$, TypeA2TypeB$_3$	4	10	10	4	0	12	16
TypeA2TypeB$_5$, TypeA2TypeB$_8$, TypeA2TypeB$_7$, TypeA2TypeB$_6$	4	9	10	0	0	11	13
UML22Measure, UML22Measure$_2$, KM32Measure$_3$, KM32Measure	4	8	8	8	0	8	24
RemovingAnAssociationClass, RedundantClassRemovable, AssertionModification	3	31	31	11	0	39	94
CloneDr2CodeClone, SimScan2CodeClone, Simian2CodeClone	3	15	16	0	0	15	30
PathExp2TextualPathExp, PathExp2PetriNet, XML2PetriNet	3	13	13	9	1	17	43
UMLCD2UMLProfile, UML2Copy	2	201	201	200	0	205	3121
Make2Ant, Maven2Ant	2	35	35	2	4	36	98
MDL2GMF, MDL2UML	2	25	25	24	2	68	131
UML2MOF, UML2MOF$_2$	2	24	24	10	6	26	228
MOF2UML, MOF2UML$_2$	2	22	22	4	0	46	342
MySQL2KM3, UML2KM3	2	22	22	14	0	27	132
UML2Transformations$_2$, UML2Transformations	2	15	15	0	0	17	80
ECore2Class, EMF2KM3	2	12	12	1	0	12	37
XML2PNML, PetriNet2PNML	2	9	9	4	0	30	55
SoftwareQualityControl2Bugzilla, SoftwareQualityControl2Mantis	2	4	4	0	1	14	72
Tree2List2, Tree2List	2	3	3	2	0	4	6
All	202	2566	2637	1524	115	5248	15445

Table 9 ATLZOO results. For each cluster, the *largest* (first row) and *broadest* (second row) reported clones are denoted with their number of modules (M), rules (R), source elements (S), guards (G), variables (V), target elements (T) and bindings (B). "–" denotes memory-related program exits or execution times longer than 1 h

Clusters			ConQAT							eScan							ScanQAT						
Name	M	R	M	R	S	G	V	T	B	M	R	S	G	V	T	B	M	R	S	G	V	T	B
XML2RDM, etc.	58	1296	1	2	1	0	0	39	88	–							–						
			15	340	0	0	0	1	0	–							–						
ATL_WFR, etc.	46	371	2	2	2	1	0	7	49	–							–						
			8	123	1	0	0	0	0	–							–						
Table2HTML, etc.	14	71	1	2	1	0	0	18	33	–							–						
			6	21	0	0	0	1	0	–							–						
Measure2XHTML2, etc.	5	29	2	2	1	0	0	4	7	2	2	1	0	0	4	7	2	2	1	0	0	4	7
			1	6	1	0	0	0	0	1	6	1	0	0	0	0	1	6	1	0	0	0	0
TypeA2TypeB, etc.	4	8	2	2	1	0	0	1	2	2	2	1	0	0	1	2	2	2	1	0	0	1	2
			3	6	1	0	0	0	0	3	6	1	0	0	0	0	3	6	1	0	0	0	0
TypeA2TypeB5, etc.	4	10	2	2	1	0	0	1	1	2	2	1	0	0	1	1	2	2	1	0	0	1	1
			4	4	1	0	0	1	0	4	4	1	0	0	1	0	4	4	1	0	0	1	0
UML22Measure, etc.	4	9	2	2	1	1	0	1	4	2	2	1	1	0	1	4	2	2	1	1	0	1	4
			4	8	0	0	0	1	0	4	8	0	0	0	1	0	4	8	0	0	0	1	0
RemovingAnAssociationClass, etc.	3	15	2	2	1	0	0	1	6	2	2	1	0	0	1	6	2	2	1	0	0	1	6
			3	3	1	0	0	0	0	3	3	1	0	0	0	0	3	3	1	0	0	0	0
CloneDr2CodeClone, etc.	3	31	3	3	1	0	0	1	2	3	3	1	0	0	1	2	3	3	1	0	0	1	2
			3	4	1	0	0	0	0	3	4	1	0	0	0	0	3	4	1	0	0	0	0
PathExp2TextualPathExp, etc.	3	13	1	2	1	0	0	1	2	1	2	1	0	0	1	2	1	2	1	0	0	1	2
			1	3	1	0	0	1	1	1	3	1	0	0	1	1	1	3	1	0	0	1	1
UMLCD2UMLProfile, etc.	2	22	1	2	0	0	0	1	14	1	2	0	0	0	1	14	1	2	0	0	0	1	14
			1	167	0	1	0	0	0	1	167	0	1	0	0	0	1	167	0	1	0	0	0
Make2Ant, etc.	2	24	1	2	1	0	0	1	2	1	2	1	0	0	1	2	1	2	1	0	0	1	2
			1	2	1	0	0	1	2	1	2	1	0	0	1	2	1	2	1	0	0	1	2
MDL2GMF, etc.	2	15	1	2	1	0	0	4	0	1	2	1	0	0	4	0	1	2	1	0	0	4	0
			2	23	1	0	0	0	0	2	23	1	0	0	0	0	2	23	1	0	0	0	0
UML2MOF, etc.	2	35	2	2	1	0	0	2	19	–							–						
			2	4	0	0	0	1	4	–							–						
MOF2UML, etc.	2	3	2	2	1	1	0	6	53	–							–						
			2	8	0	0	0	1	3	–							–						
MySQL2KM3, etc.	2	9	1	2	1	0	0	1	10	1	2	1	0	0	1	10	1	2	1	0	0	1	10
			1	5	1	0	0	0	0	1	5	1	0	0	0	0	1	5	1	0	0	0	0

Table 9 continued

Clusters			ConQAT							eScan							ScanQAT						
Name	M	R	M	R	S	G	V	T	B	M	R	S	G	V	T	B	M	R	S	G	V	T	B
UML2Transformations2, etc.	2	4	2	2	1	0	0	1	11	2	2	1	0	0	1	11	2	2	1	0	0	1	11
			2	2	1	0	0	1	11	2	2	1	0	0	1	11	2	2	1	0	0	1	11
ECore2Class, etc.	2	22	2	2	1	0	0	0	0	2	2	1	0	0	0	0	2	2	1	0	0	0	0
			2	2	1	0	0	0	0	2	2	1	0	0	0	0	2	2	1	0	0	0	0
XML2PNML, etc.	2	201	1	4	1	0	0	2	2	1	4	1	0	0	2	2	1	4	1	0	0	2	2
			2	8	0	0	0	1	0	2	8	0	0	0	1	0	2	8	0	0	0	1	0
SoftwareQualityControl2Bugzilla, etc.	2	25	2	2	1	0	0	0	0	2	2	1	0	0	0	0	2	2	1	0	0	0	0
			2	2	1	0	0	0	0	2	2	1	0	0	0	0	2	2	1	0	0	0	0
Tree2List2, etc.	2	12	2	2	1	1	0	1	1	2	2	1	1	0	1	1	2	2	1	1	0	1	1
			2	2	1	1	0	1	1	2	2	1	1	0	1	1	2	2	1	1	0	1	1
All	202	2566	1	2	1	0	0	39	88	–							–						
			15	340	0	0	0	1	0	–							–						

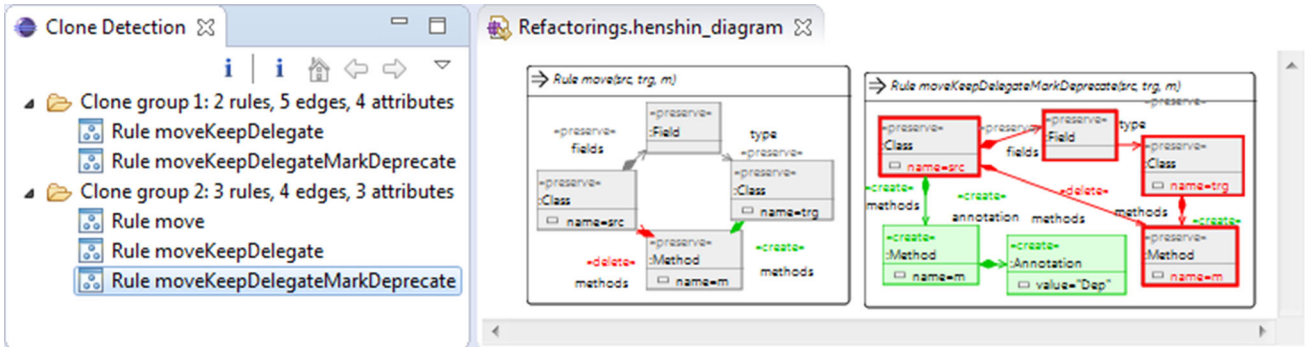

Fig. 11 Clone Detection view and Henshin editor

under similar conditions: on the same machine running the same minimal set of applications.

External validity indicates an experiments ability to produce generalizable results. We discuss the external validity of our experiments from two orthogonal perspectives: (1) the considered transformation scenarios and (2) the considered model transformation languages. From the first perspective, the external validity of our experiments is threatened by the limited sample of transformation rules sets. Although the studied graph-based scenarios are based on three heterogeneous and non-trivial rule sets, the rules were not created in an industrial context. From the second perspective, we argue that Henshin is similar in terms of features to other graph-based model transformation languages such as AGG [50], GReAT [51], VMTS [52] and Story Diagrams [53]. We therefore expect the findings of this experiment to be generalizable to such languages. Our findings are, however, not generalizable to model transformation languages representing other paradigms than graph transformation and the hybrid paradigm as embodied by ATL. Still, since we selected the graph-based and hybrid paradigms as two of the most commonly used transformation paradigms, we are confident that our results give interesting insights to both researchers and practitioners.

7 Related work

Several other techniques for model clone detection have been proposed. While the approaches by Störrle [36,54] and Ekanayake et al. [55] 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. [56] propose a clone detection technique based on identifying the longest subsequences 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.

Tairas et al. [57] propose a model-based approach for identifying clones in textual Domain Specific Languages (DSLs) and exemplify this approach by applying it to several OCL code bases, including the ATL zoo. The presented experiment suggests that clones appear in large numbers in publicly available OCL repositories. This work is complementary to our proposal; a promising combination of both approaches is to use the OCL clone detection algorithm in conjunction with our approach for detecting clones in the structure of ATL transformations.

The clones considered in this paper are Type I–II clones. Additional classes of clones are Type III or "near-miss clones," which include layout changes and additions of connections, and Type IV or "semantic" clones. The SIMONE clone detector proposed by Cordy and his colleagues [17,58,59] focuses on Type III clones in Simulink models. An evaluation of SIMONE indicates that, in addition to Type III clones, it is capable of identifying all Type I–II clones detected by ConQAT [58], suggesting that Type III clones in rule-based model transformation languages may also be reliably identified. This is an important area of future work in the direction opened by the present paper.

In our evaluation, we compared the available model clone detection techniques by using the exact detection results reported by eScan and ConQAT as a ground truth, a strategy that was suitable for small to medium rule sets, but did not scale up to the largest ones we considered. In the future, we may benefit from available works on the evaluation of model clone detection techniques. In the context of Type III clones, a suitable strategy for clone detector evaluation is mutation analysis [60,61]: This strategy involves the application of certain mutator operations to seed variations in a set of clones, which allows Type I and II clones to be turned into Type III ones.

A number of quality assurance approaches for model transformations are related. Van Amstel et al. [62] 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. [63] propose a set of quality metrics to evaluate QVT-R transformations; the number of clones is mentioned as one metric. Wimmer et al. [64] introduce a refactoring catalog to improve the quality of M2M transformations; duplicate code is mentioned as a bad smell. Gerpheide et al. [65] present a quality model for QVT-O comprising 37 quality properties and four quality goals: functionality, understandability, performance and maintainability. In line with our discussion of clone refactoring and management as use cases for clone detection (see Sect. 2), Alkhazi et al. [66] propose a search-based solution for the automatic refactoring of ATL transformations. Some of the supported refactorings (e.g., "extract superrule," "extract helper" and "merge rule") stand out as direct applications of clone detection.

8 Conclusion

Clone detection is an important static analysis for enabling the identification of duplications in software artifacts, thereby supporting the transition from an ad-hoc development style to a more systematic one. In this work, we present the first approach to address clone detection for model transformations, focusing on the rule-based transformation paradigms of *graph-based* and *hybrid* model transformations.

Our experimental evaluation features a selection of large-scale rule sets from realistic Henshin and ATL transformation scenarios. The results indicate that our adaptation of ConQAT, a technique from the domain of Simulink models, is well suited to satisfy the requirements of clone detection in rule-based model transformations: It supports the identification of overlapping patterns, which is a particularly important requirement in the context of graph-based transformations, where rule elements do not necessarily have names. In cases where we could assess its accuracy, ConQAT's accuracy was excellent, while providing favorable performance in particular for larger rule sets. Our current investigation focused on Type I and II clones.

9 Future work

Since research on clone detection for model transformation languages is still in its infancy, there are several directions for future work.

The hypothesis that transformation developers can benefit from clone detection must be validated empirically. To this end, a user experiment based on our prototypical tool support is appropriate. Moreover, we aim to broaden the scope of our work towards additional model transformation and clone detection features and paradigms.

First, in addition to transformation rules as addressed in this work, transformation languages often come with imperative features such as control flow mechanisms or even completely imperative transformation programs.

To detect clones related to these features, integrating existing clone detection approaches for imperative programming languages as well as textual DSLs is one particular development of interest. Using this approach, we aim to establish whether similar results as the ones provided for graph-based and hybrid model transformation languages can be obtained for other paradigms, such as imperative ones.

Second, in our evaluation, we saw that the number and extent of clones reported to the user can be substantial. The identification of suitable clones for a particular use cases might require to identify subsets of clones based on certain fitness criteria. This task may benefit from advances in search-based model optimization [67].

Third, a desirable clone detection feature we intend to address in the future includes support for Type III and IV clones. Since we were able to adapt existing clone detection techniques for Type I and II clones, we aim to establish whether similar results can be obtained in this case. Specifically, to detect Type III clones, we plan to adapt the SIMONE clone detector, which has shown excellent results in the Simulink domain. Moreover, an important feature to address in future work is an incremental execution mode for clone detection, which reuses detection results from earlier runs. In particular, this feature might be beneficial for the use cases of performance optimization and usability improvements.

Acknowledgements We thank the reviewers for their valuable and constructive suggestions. This research was partially supported by the research project Visual Privacy Management in User Centric Open Environments (supported by the EUs Horizon 2020 programme, Proposal Number: 653642).

References

1. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)
2. Glass, R.L.: Frequently forgotten fundamental facts about software engineering. IEEE Softw. **3**, 110–112 (2001)
3. Koschke, R.: Survey of research on software clones. In: Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software. GI 24 (2007)
4. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Boston (2002)
5. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In: ACM SIGSOFT Software Engineering Notes, vol. 30, pp. 187–196. ACM (2005)
6. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Sci. Comput. Program. **74**(7), 470–495 (2009)

7. Rattan, D., Bhatia, R., Singh, M.: Software clone detection: a systematic review. Inf. Softw. Technol. **55**(7), 1165–1199 (2013)
8. Kusel, A., Schönböck, J., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: Reuse in model-to-model transformation languages: are we there yet? Softw. Syst. Model. **14**(2), 537–572 (2013)
9. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, USA, vol. 45, pp. 1–17 (2003)
10. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
11. Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., Plöger, J.: RuleMerger: automatic construction of variability-based model transformation rules. In: International Conference on Fundamental Approaches to Software Engineering, pp. 122–140. Springer, Berlin (2016)
12. Strüber, D., Plöger, J., Acretoaie, V.: Clone detection for graph-based model transformation languages. In: Proceedings of the International Conference on the Theory and Practice of Model Transformations (ICMT), pp. 191–206. Springer (2016)
13. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Synth. Lect. Softw. Eng. **1**(1), 1–182 (2012)
14. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: International Conference on Model Driven Engineering Languages and Systems, pp. 121–135. Springer (2010)
15. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: A usability-focused framework for EMF model transformation development. In: International Conference on Graph Transformation, pp. 196–208 (2017)
16. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Satellite Events at the MoDELS 2005 Conference, Revised Selected Papers, pp. 128–138. Springer (2005)
17. Alalfi, M.H., Cordy, J.R., Dean, T.R., Stephan, M., Stevenson, A.: Models are code too: near-miss clone detection for Simulink models. In: International Conference on Software Maintenance, pp. 295–304. IEEE (2012)
18. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D.S., Paige, R.F., Lauder, M., Schürr, A., et al.: Surveying rule inheritance in model-to-model transformation languages. J. Object Technol. **11**(2), 1–46 (2012)
19. Anjorin, A., Saller, K., Lochau, M., Schürr, A.: Modularizing triple graph grammars using rule refinement. In: International Conference on Fundamental Approaches to Software Engineering, pp. 340–354. Springer (2014)
20. Strüber, D., Rubin, J., Chechik, M., Taentzer, G.: A variability-based approach to reusable and efficient model transformations. In: International Conference on Fundamental Approaches to Software Engineering, pp. 283–298. Springer (2015)
21. Nguyen, H.A., Nguyen, T.T., Pham, N.H., Al-Kofahi, J., Nguyen, T.N.: Clone management for evolving software. IEEE Trans. Softw. Eng. **38**(5), 1008–1026 (2012)
22. Narasimhan, K., Reichenbach, C.: Copy and paste redeemed. In: International Conference on Automated Software Engineering, pp. 630–640. IEEE (2015)
23. Stephan, M., Cordy, J.R.: Model-driven evaluation of software architecture quality using model clone detection. In: International Conference on Software Quality, Reliability and Security, pp. 92–99. IEEE (2016)
24. Stephan, M., Cordy, J.R.: Identification of Simulink model antipattern instances using model clone detection. In: International Conference on Model Driven Engineering Languages and Systems, pp. 276–285. IEEE (2015)
25. Stephan, M., Cordy, J.R.: Identifying instances of model design patterns and antipatterns using model clone detection. In: International Workshop on Modeling in Software Engineering, pp. 48–53. IEEE (2015)
26. Lano, K., Kolahdouz-Rahimi, S.: Model-transformation design patterns. IEEE Trans. Softw. Eng. **40**(12), 1224–1259 (2014)
27. Blouin, D., Plantec, A., Dissaux, P., Singhoff, F., Diguet, J.P.: Synchronization of models of rich languages with triple graph grammars: an experience report. In: International Conference on Model Transformation, pp. 106–121. Springer (2014)
28. Strüber, D.: Model-driven engineering in the large: refactoring techniques for models and model transformation systems. Ph.D. thesis, Philipps-Universität Marburg (2016)
29. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: International Conference on Graph Transformation, pp. 161–176. Springer (2002)
30. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26**(3/4), 287–313 (1996)
31. Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., Plöger, J.: Variability-based model transformation: formal foundation and application. Formal Aspects Comput. (2017) (accepted)
32. Ehrig, H., Golas, U., Habel, A., Lambers, L., Orejas, F.: M-adhesive transformation systems with nested application conditions. Part 1: parallelism, concurrency and amalgamation. Math. Struct. Comput. Sci. **24**(04), 240406 (2014)
33. Beller, M., Zaidman, A., Karpov, A.: The last line effect. In: International Conference on Program Comprehension, pp. 240–243. IEEE Press (2015)
34. Bauer, J., Boneva, I., Kurbán, M.E., Rensink, A.: A modal-logic based graph abstraction. In: International Conference on Graph Transformation, pp. 321–335. Springer (2008)
35. Cuadrado, J.S., Guerra, E., De Lara, J.: Generic model transformations: write once, reuse everywhere. In: International Conference on Model Transformation, pp. 62–77 (2011)
36. Störrle, H.: Towards clone detection in UML domain models. J. Softw. Syst. Model. **12**(2), 307–329 (2013)
37. Tichy, M., Krause, C., Liebel, G.: Detecting performance bad smells for Henshin model transformations. AMT workshop, vol. 1077 (2013)
38. Störrle, H.: On the impact of layout quality to understanding UML diagrams: size matters. In: International Conference on Model Driven Engineering Languages and Systems, pp. 518–534. Springer (2014)
39. Yan, X., Han, J.: gSpan: graph-based substructure pattern mining. In: ICDM'03, pp. 721–724. IEEE (2002)
40. Pham, N.H., Nguyen, H.A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N.: Complete and accurate clone detection in graph-based models. In: International Conference on Software Engineering, pp. 276–286. IEEE (2009)
41. Deissenboeck, F., Hummel, B., Jürgens, E., Schätz, B., Wagner, S., Girard, J., Teuchert, S.: Clone detection in automotive model-based development. In: International Conference on Software Engineering, pp. 603–612. ACM (2008)
42. Deissenboeck, F., Hummel, B., Jürgens, E., Pfaehler, M., Schaetz, B.: Model clone detection in practice. In: Ws. on Software Clones, pp. 57–64. ACM (2010)
43. Arendt, T., Habel, A., Radke, H., Taentzer, G.: From core OCL invariants to nested graph constraints. In: International Conference on Graph Transformation, pp. 97–112. Springer (2014)
44. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A.: Reasoning about product-line evolution using complex feature model differences. J. Autom. Softw. Eng. **23**(4), 1–47 (2015)
45. Strüber, D., Kehrer, T., Arendt, T., Pietsch, C., Reuling, D.: Scalability of model transformations: position paper and benchmark set.

In: Workshop on Scalable Model Driven Engineering, pp. 21–30 (2016)
46. Schaeffer, S.E.: Graph clustering. Comput. Sci. Rev. **1**(1), 27–64 (2007)
47. Babur, Ö., Cleophas, L., van den Brand, M.: Hierarchical clustering of metamodels for comparative analysis and visualization. In: European Conference on Modelling Foundations and Applications, pp. 3–18 (2016)
48. Strüber, D., Schulz, S.: A tool environment for managing families of model transformation rules. In: International Conference on Graph Transformation. Springer (2016)
49. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B.: Experimentation in Software Engineering. Springer, Berlin (2012)
50. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: International Workshop on Applications of Graph Transformations with Industrial Relevance, pp. 446–453. Springer (2003)
51. Balasubramanian, D., Narayanan, A., van Buskirk, C.P., Karsai, G.: The graph rewriting and transformation language: GReAT. In: ECEASST, vol. 1 (2006)
52. Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H.: A systematic approach to metamodeling environments and model transformation systems in VMTS. Electron. Notes Theor. Comput. Sci. **127**(1), 65–75 (2005)
53. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language and java. In: International Workshop on Theory and Application of Graph Transformations, pp. 296–309. Springer (1998)
54. Störrle, H.: Effective and efficient model clone detection. In: Software, Services, and Systems, pp. 440–457. Springer (2015)
55. Ekanayake, C.C., Dumas, M., García-Bañuelos, L., La Rosa, M., ter Hofstede, A.H.: Approximate clone detection in repositories of business process models. In: Business Process Management, pp. 302–318. Springer (2012)
56. Liang, Z., Cheng, Y., Chen, J.: A novel optimized path-based algorithm for model clone detection. J. Softw. **9**(7), 1810–1817 (2014)
57. Tairas, R., Cabot, J.: Cloning in DSLs: experiments with OCL. In: International Conference on Software Language Engineering, pp. 60–76. Springer (2011)
58. Cordy, J.R.: Submodel pattern extraction for simulink models. In: International Software Product Line Conference, pp. 7–10. ACM (2013)
59. Rapos, E.J., Stevenson, A., Alalfi, M.H., Cordy, J.R.: SimNav: Simulink navigation of model clone classes. In: International Working Conference on Source Code Analysis and Manipulation, pp. 241–246. IEEE Computer Society (2015)
60. Stephan, M., Alalfi, M.H., Stevenson, A., Cordy, J.R.: Towards qualitative comparison of simulink model clone detection approaches. In: International Workshop on Software Clones, pp. 84–85. IEEE (2012)
61. Stephan, M.: Model clone detector evaluation using mutation analysis. In: International Conference on Software Maintenance and Evolution, pp. 633–638. IEEE (2014)
62. Van Amstel, M.F., Van Den Brand, M.G.: Model transformation analysis: staying ahead of the maintenance nightmare. In: International Conference on Model Transformation, pp. 108–122. Springer (2011)
63. Kapová, L., Goldschmidt, T., Becker, S., Henss, J.: Evaluating maintainability with code metrics for model-to-model transformations. In: Research into Practice–Reality and Gaps, pp. 151–166. Springer (2010)
64. Wimmer, M., Perez, S.M., Jouault, F., Cabot, J.: A catalogue of refactorings for model-to-model transformations. J. Object Technol. **11**(2), 1–40 (2012)
65. Gerpheide, C.M., Schiffelers, R.R., Serebrenik, A.: Assessing and improving quality of QVTo model transformations. Softw. Qual. J. **24**(3), 1–38 (2014)
66. Alkhazi, B., Ruas, T., Kessentini, M., Wimmer, M., Grosky, W.I.: Automated refactoring of ATL model transformations: a search-based approach. In: International Conference on Model Driven Engineering Languages and Systems, pp. 295–304. ACM (2016)
67. Strüber, D.: Generating efficient mutation operators for search-based model-driven engineering. In: International Conference on Model Transformation, pp. 121–137 (2017)

Daniel Strüber received a diploma and a Ph.D. degree in Computer Science at University of Marburg in 2011 and 2016, respectively. Since 2016, he works as a post-doc at University of Koblenz-Landau. In his research, he aims to support software engineers during the development of complex systems with improved model-based languages, mechanisms, and tools. His research interests include software quality, variability management, security, and flexible, usable tools. Daniel is an Eclipse committer and the project lead of the EMF Henshin project.

Vlad Acreţoaie received an M.Sc. in Computer Science from the Technical University of Denmark in 2012, and completed his Ph.D. studies at the same institution in 2016. His doctoral dissertation provides solutions for the adoption of model transformation, query, and constraint languages and tools by end-user modelers. After graduating, Vlad worked as a software engineer in industry, and he is currently a Product Owner. His research interests are at the intersection of Model-Based and Empirical Software Engineering.

Jennifer Plöger recently received a diploma in computer science at University of Marburg, with a major focus in software technology. In her diploma thesis, she investigated the application of model clone detection techniques to model transformations, and contributed to emerging work on the variability-oriented refactoring of model transformation systems.