A Tool Environment for Managing Families of Model Transformation Rules

Daniel Strüber(✉) and Stefan Schulz

Philipps-Universität Marburg, Marburg, Germany
{strueber,schulzs}@informatik.uni-marburg.de

Abstract. Model transformation systems often contain families of rules that are substantially similar to each other. Variability-based rules are a recent approach to express such families of rules in a compact representation, enabling the convenient editing of multiple rule variants at once. On the downside, this approach gives rises to distinct maintenance drawbacks: Users are required to view and edit presence conditions. The complexity and size of the resulting rules may impair their readability.

In this paper, we propose to facilitate the editing of variability-based rules through suitable tool support. Inspired by the paradigms of *filtered editing* and *virtual seperation of concerns*, we present a tool environment that offers editable views for variants expressed in a variability-based rule. We demonstrate that our tool environment is helpful to address the identified issues, rendering variability-based rules a highly feasible reuse approach.

1 Introduction

Model transformation is a key enabling technology for Model-Driven Engineering. Algebraic graph transformation is one of the main paradigms in this field, enabling a high-level, declarative specification based on graph rewriting rules [1]. Non-trivial graph transformation systems often contain rules that are substantially similar to each other. Such rules may share a large bulk of intended actions, while differing only marginally, leading to a large amount of pattern duplications.

Several approaches can be used to capture such *families of rules* while avoiding pattern duplication. Many of these approaches embody a composition-based paradigm: rule variants are assembled from fragmentary building blocks. In the case of rule inheritance [2], the implementation of a rule family comprises a hierarchy of a base rule with sub-rules. Rule refinement [3] extends this concept by supporting multiple base rules and the capability to modify super-rules. While these approaches clearly avoid pattern duplication, they may entail managing a large number of interrelated fragments. Their semantics are often intricate; a scheduling mechanism may be required to handle conflicts during composition.

Inspired by product line engineering approaches [4], we propose *variability-based (VB) rules*, an annotative approach to managing families of rules. The key idea is to encode a family of rules as one VB rule. Portions of this VB rule are

R. Echahed and M. Minas (Eds.): ICGT 2016, LNCS 9761, pp. 89–101, 2016.
DOI: 10.1007/978-3-319-40530-8_6

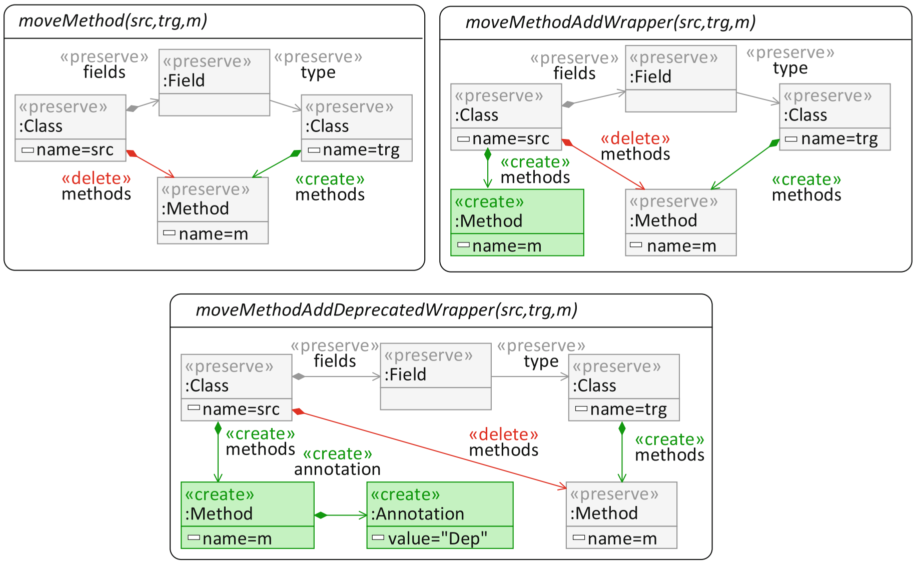


Fig. 1. Three variants of the *move method* refactoring.

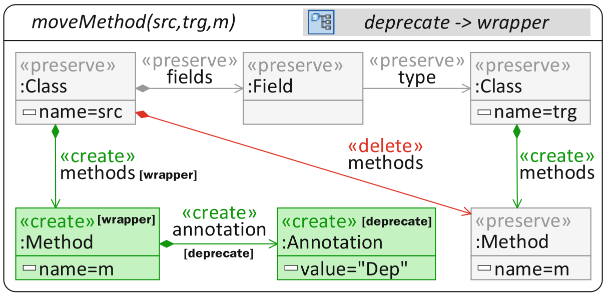


Fig. 2. Variability-based rule expressing the same three variants.

annotated with presence conditions to assign them to a subset of the encoded rules. The portion common to all rules, called the *base rule*, is not annotated.

Example. Consider a family of three in-place transformation rules. The rules, shown in Fig. 1, express variants of the *move method* refactoring for class models. The first rule specifies the relocation of a method between two classes. The second one additionally creates a wrapper method of the same name in the source class. The third one adds an annotation to mark the wrapper method as deprecated.

These three rule variants can be expressed using the VB rule shown in Fig. 2. Several elements are annotated with presence conditions over the literals `wrapper` and `deprecate`. The variants are obtained by configuring the rule, i.e., binding these literals to *true* or *false* and removing elements whose presence condition evaluates to *false*. Configuration {*wrapper=false; deprecate=false*} yields the base rule, a rule isomorphic to rule *moveMethod* in Fig. 1. The rules induced by the configurations {*wrapper=true; deprecate=false*} and {*wrapper=true; deprecate=true*} produce the additional variants. To avoid the illegal configuration {*wrapper=false; deprecate=true*}, the rule has a constraint called *variability model*, shown in the title bar, requiring `wrapper` to be true if `deprecate` is true.

This example highlights several maintainability benefits of VB rules: (i) During maintenance, all included variants are viewed and edited at once. While evidently convenient, this editing style may also be less error-prone: Fixing the same bug in multiple rules manually may lead to residual bugs not being considered. (ii) The representation is more compact, in terms of the number of rules, the total number of rule elements and the used amount of space. While compactness does not necessarily equal better readability, it is still an explicit goal of compositional approaches [3]. (iii) In contrast to compositional approaches, no additional mechanism is needed to glue fragments together, adding to the compactness of the specification. The structure of each variant is directly present; maintainers are not required to obtain a mental representation by assembling fragments.

Conversely, the example also illustrates a set of drawbacks of VB rules. (i) The use of presence conditions creates a "noisy" or "cluttered" impression, impairing readability. To make matters worse, these presence conditions are required to be edited manually, a tedious and potentially error-prone process. (ii) The rule size in terms of *average* number of elements per rule is greater. A detrimental effect of diagram size on readability is reported in [5]. (iii) To understand individual rule variants, developers are required to identify and focus on selected portions, posing a high cognitive effort. While color-coding would be helpful to mitigate this issue, it is at least complicated if not unavailable due to existing color-coding.

In this work, we address the following research question: **How can the efficient viewing and editing of variability-based rules be facilitated?** Our key idea is to provide dynamic representations suitable to the task at hand rather than one static representation – an idea inspired by the paradigms of *filtered editing* [6] and *virtual separation of concerns* [7]. We propose a tool environment that offers views on rule variants selected by the user. These views are helpful to mitigate the identified drawbacks by (i) removing the need to read and edit presence conditions, (ii) being smaller in size, and (iii) reducing the cognitive effort in deriving mental representations. In addition, we provide support for converting a legacy rule set into a VB rule with little manual effort. The basic concepts of VB rules and their automatic creation have been introduced elsewhere [8–10].

We have implemented our tool environment on top of Henshin [11], a model transformation language based on algebraic graph transformations. Lifting the concepts proposed in this work to other languages and paradigms is desirable, but left to future work. The tool and a description of its use can be found at https://www.uni-marburg.de/fb12/swt/forschung/software/varhenshin/.

2 Variability-Based Rules

In this section, we briefly revisit the main concepts of variability-based rules. We assume the reader to be familiar with double-pushout graph transformation rules, such as those shown in Fig. 1. The underlying graph kind may include typing and attributes since these concepts are orthogonal to variability. We further use the concept of subrule, a rule that can be embedded into a larger rule in an injective manner. A detailed account of these concepts is given in [10].

Definition 1 (Variability-based (VB) rule). *Given a set of atomic terms V, called* variability points*, a* VB rule $\check{r} = (r, S, vm, pc)$ *consists of a rule r, a set S of subrules of r, a propositional term $vm \in \mathcal{L}_V$ and a function $pc : S \cup \{r\} \rightarrow \mathcal{L}_V$, where $\mathcal{L}_V$ is the set of propositional terms over V. Term vm is called* variability model*. Function pc defines* presence conditions *for subrules s.t. $pc(r)$ is* true *and $\forall s \subseteq s' : pc(s') \implies pc(s)$. The* base rule *is the intersection of all subrules.*

Figure 2 shows a VB rule over variability points $\{$*wrapper, deprecate*$\}$. The rule is shown in a compact representation where subrules are not shown explicitly, but denoted using element presence conditions. Rule r is the entire rule, ignoring annotations. S comprises a subrule for each propositional term over V. Each subrule contains those elements whose presence conditions are implied by its own presence condition. For instance, subrule s with $pc(s) = wrapper \wedge \neg deprecate$ contains all elements annotated with *wrapper* and without annotations, but not those annotated with *deprecate*. The variability model is *deprecate* $\rightarrow$ *wrapper*.

Definition 2 (Configuration). *Let a VB rule $\check{r} = (r, S, vm, pc)$ over V be given. A* configuration *is a total function $c : V \rightarrow \{true, false\}$. A configuration c* satisfies *a term $t \in \mathcal{L}_V$ if t evaluates to true when each variable v in t is substituted by $c(v)$. A configuration c is* valid *if c satisfies vm.*

In the example, $\{$*wrapper=true; deprecate=false*$\}$ is a valid configuration, satisfying the presence condition *wrapper*, but not the presence condition *deprecate*.

Definition 3 (Rule variant). *For a valid configuration c, there exists a unique set of subrules $S_c \subseteq S$ s.t. $\forall s \in S : s \in S_c$ iff c satisfies $pc(s)$. Gluing together all elements contained in one of these subrules yields a rule r_c, called* rule variant induced by *c.*

The example VB rule can be used to produce three variants; details are provided in the previous description of the example. Categorically, the gluing can be expressed as a consecutive multi-pullback and multi-pushout construction [10].

There are two main application scenarios for VB rules. First, a specific user intention may lead to the selection and application of one particular rule variant. For instance, in the example, the user may configure the rule so that it produces a wrapper method. Such an *external* configuration process leads to an individual rule being applied in the classic way. Second, all rules in a rule set may be applied simultaneously. Such rule sets are found in batch transformation scenarios, such as translation or migration suites. In this case, configurations can be set *internally* by the transformation engine. This approach allows to consider the base rule of all variants at once, leading to considerable performance savings [8].

3 Main Features

In this section, we present the main features of our tool environment. The design of these features is informed by *Cognitive Dimension* [12] (CD), a framework of usability dimensions for visual programming environments. First, we give an overview of the features, relating each to the CD framework. Second, we exemplify the use of these features from the user perspective.

- **View specific rule variants**: Each variant expressed in a VB rule corresponds to a configuration, a binding of all variability points to *true* or *false*. To view specific variants, we provide a *live configuration* feature: The user performs a partial or total binding of variability points, leading to immediate feedback. Irrelevant rule elements can be either turned invisible or toned down. The former option helps the user during the comprehension of individual variants. The latter one facilitates the comparison of variants.

  This feature addresses several cognitive dimensions: The *visibility* of rule variants is increased. The need for *hard mental operations* is reduced by shielding users from the cognitive effort of deriving variants. Notational *diffuseness* is reduced as fewer different symbols are needed to capture variability.

- **Edit rule variants**: A crucial issue of editing VB rules is the requirement to have users edit presence conditions, a tedious and error-prone process. We provide features to mitigate this issue: When creating a new element, a presence condition corresponding to the currently selected configuration is assigned automatically. We also support the reassigning of elements to different variants by moving them to a more general or specific configuration (i.e., one where more or less variability points are unbound).

  By lifting the abstraction level from editing presence conditions to moving elements between variants, we aim to reduce *error-proneness*. The capability to move multiple elements also reduces *viscosity*, the resistance to change.

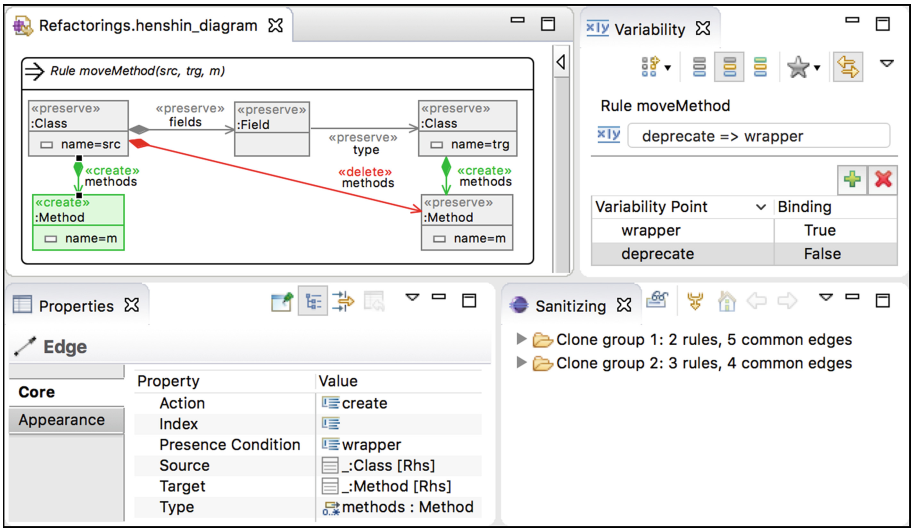
- **Explore relationships between rule variants**: We provide multiple features to support exploring multiple variants and their interrelations. First, a *favorites* feature allows rapid switching between variants. Second, a *quick access* feature provides instant access to distinguished variants such as the base rule and the maximum rule. Third, an *auto-completion feature* reduces the configuration effort of by inferring certain open bindings automatically.

  These features are key to increasing *role-expressiveness*, the ease of understanding *"how each component [...] relates to the whole"* [12].

- **Manage variability points**: We provide a dedicated viewer component for the management of variability points. Using this viewer, variability points can be created and deleted. To ensure consistency, presence conditions referring

to the deleted variability point can be updated automatically.
This dedicated component inceases the *visibility* of variability management.

- **Sanitize legacy rule sets**: Legacy rule sets may exhibit a high degree of pattern duplication, notably, if they were devised in a copy-and-paste manner. To sanitize such rule sets, we provide *clone detection* and *merge refactoring* features. Clone detection allows identifying cloned portions in a set of rules. These portions may serve as input to merge refactoring, a feature that creates VB rules automatically, including an optimization to preserve layout information from the input rules. We present this technique in [9].
  This feature shields from *premature commitment*: VB rules do not have to be devised from scratch. Users may develop rules in an ad-hoc manner and decide to use VB rules later, while retaining key layout information.



Rule moveMethod

deprecate => wrapper

Variability Point	Binding
wrapper	True
deprecate	False

Edge

Core
Appearance

Property	Value
Action	create
Index	
Presence Condition	wrapper
Source	_:Class [Rhs]
Target	_:Method [Rhs]
Type	methods : Method

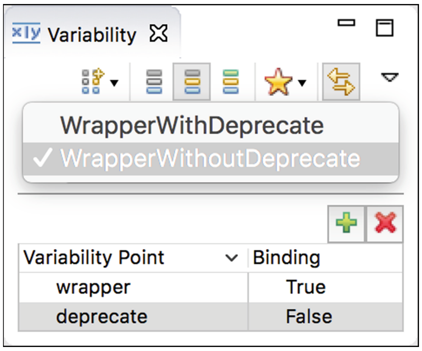
Clone group 1: 2 rules, 5 common edges
Clone group 2: 3 rules, 4 common edges

Fig. 3. Our tool environment from the user perspective.

User Perspective. Our tool environment is integrated with the default user interface of Henshin. The main components of this user interface, a graphical editor and its attached *properties* view, are shown in the left of Fig. 3. As custom components, we provide the *variability* and *sanitizing* views, shown in the right. The variability view comprises features for the definition and configuration of variability points. The variant produced from the current configuration is displayed in the editor. The sanitizing view can be used to sanitize legacy transformations.

Variability view. The variability view allows variability points in a rule to be created and deleted. To view and edit variants individually, the user configures the rule by setting the bindings for these variability points. Three literals are supported: *true*, *false*, and *unbound*. Per default, each variability point is *unbound*, yielding the maximal rule, all elements regardless of their annotations. Configurations are validated against the given variability model, *deprecate* $\rightarrow$ *wrapper* in this case. Invalid configurations and rules lead to error messages being displayed.

To navigate variants efficiently, frequently used configurations can be saved as favorites using the ☆ button in the toolbar. The star appears in yellow if a favored configuration is currently active. Each configuration has a user-specified name. In Fig. 4, the user has created two favorites, *WrapperWithDeprecate* and *WrapperWithoutDeprecate*, the latter one being active. Upon selection, the configuration is loaded and shown in the table at the bottom of the view.


Variability

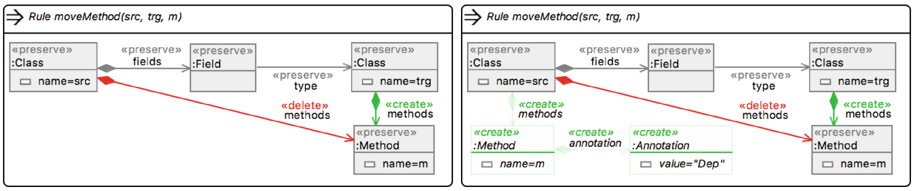
WrapperWithDeprecate
✓ WrapperWithoutDeprecate

Variability Point	Binding
wrapper	True
deprecate	False

Fig. 4. Variability view with favorites menu.

A *view mode* feature allows to access distinguished variants rapidly. In the *maximum rule* mode, represented by the ≡ icon, all elements included in the rule are shown regardless of the configuration. In the *variant mode* (≡), elements absent in the current configuration are concealed. In the *base rule* mode (≡), elements with a non-empty presence condition are concealed.

To further improve the handling of variability, the view allows the users to choose a *concealing strategy*, depicted in Fig. 5. First, elements can be turned invisible. This avoids a cluttered representation of the rule and lets users focus on the variant at hand. On the other hand, to allow the comparison of a variant with the full rule, users may choose to have the elements toned down instead.

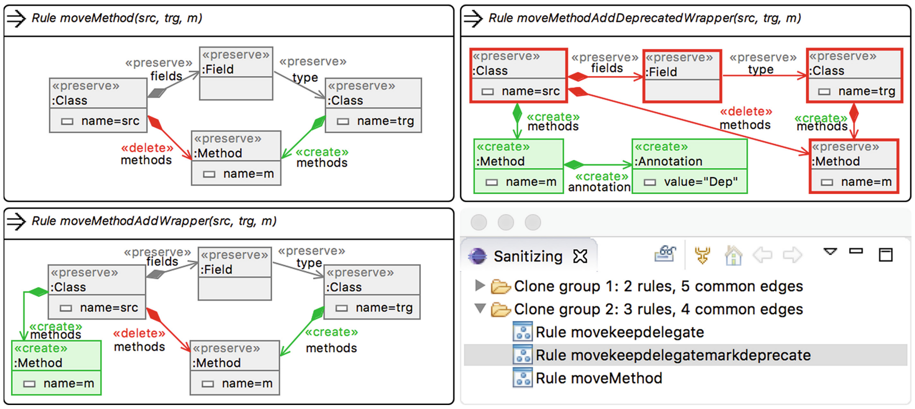


(a) Hiding unrelated elements. (b) Toning down unrelated elements.

Fig. 5. Concealing strategies.

Using the ⁝ button, users can select an *editing mode* to define which variants are affected by edits to the rule. The supported options are: all variants, variants included in the selected configuration, or variants associated to the current view mode. In particular, the editing mode determines which presence condition is assigned during the addition or deletion of elements to a rule.

Sanitizing view. The sanitizing view, shown in the lower right of Fig. 6, supports two operations for sanitizing legacy rule sets: clone detection and merging. Clone detection allows the identification of duplications in the rule set. The result is a list of *clone groups*. To display the most relevant clone groups prominently, the clone groups are ordered by their size, i.e., the number of common elements. Users can inspect the duplication interactively; when a rule is selected, the affected portions in the rule are focused and highlighted. Internally, clone detection aims to identify isomorphic sub-graphs, a computationally expensive problem in general. To ensure reasonable response times, our approach uses a heuristics provided by ConQAT [13], a clone detection technique originally introduced for Simulink models. We discuss our adaptation of this technique elsewhere [14].


Rule moveMethod(src, trg, m)
«preserve» :Class name=src
«preserve» fields
«preserve» :Field
«preserve» type
«preserve» :Class name=trg
«delete» methods
«preserve» :Method name=m
«create» methods
Rule moveMethodAddDeprecatedWrapper(src, trg, m)
«preserve» :Class name=src
«preserve» fields
«preserve» :Field
«preserve» type
«preserve» :Class name=trg
«create» methods
«delete» methods
«create» methods
«create» :Method name=m
«create» annotation
«create» :Annotation value="Dep"
«preserve» :Method name=m
Rule moveMethodAddWrapper(src, trg, m)
«preserve» :Class name=src
«preserve» fields
«preserve» :Field
«preserve» type
«preserve» :Class name=trg
«create» methods
«create» :Method name=m
«delete» methods
«preserve» :Method name=m
«create» methods
Sanitizing

Clone group 1: 2 rules, 5 common edges
Clone group 2: 3 rules, 4 common edges
Rule movekeepdelegate
Rule movekeepdelegatemarkdeprecate
Rule moveMethod

Fig. 6. Sanitizing view.

The *merge* button enables the merge refactoring feature. An algorithm is executed to construct a semantically equivalent variability-based rule automatically, using the identified duplication as base rule and annotating the variant-specific parts with their rule names [9]. The user can inspect and post-process the refactoring result using the viewing and editing features. In case that the result is not satisfactory, the process can be undone.

Context menus. Additional context menu entries allow to manage variability at the level of individual elements. Multiple nodes, edges and attributes can be selected and moved to a different configuration simultaneously.

4 Architecture and Implementation

In this section, we describe the architecture of our tool and our design principles during the implementation.

We give an overview of the architecture in Fig. 7. The novel features presented in this work are encapsulated by *Variability UI*, an integrating layer on top of the *UI*, *Clone Detection*, *Merging* and *Variability* extensions for the *Henshin* language core. To combine the Henshin UI with the variability implementation first introduced in [8], the Variability UI provides the variability view and its editor integration. Clone detection and merging are made available to users in the *Sanitizing View*. The merging component acts as a bridge between the clone detection and variability extensions: It enables the conversion of rules affected by cloning to variability-based rules. *GMF*, *GEF*, *EMF* and *Eclipse* are featured as underlying frameworks. The Henshin language is based on an EMF meta-model. The Henshin UI comprises a GMF-based editor to enable the visual viewing and editing of transformations.

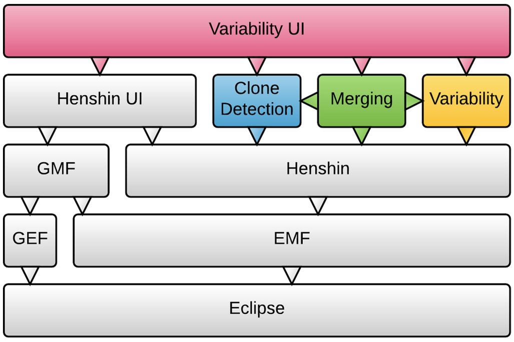


Fig. 7. Architecture.

The main design goal in our architectural design was non-intrusiveness: Changing the Henshin core and UI should be avoided where possible. The rationale for this design goal was to keep the Henshin language core, its UI and analysis functionality as simple as possible. Variability is deployed as a drop-in language extension, orthogonal to additional extensions, such as the existing support for Triple Graph Grammars [15] or possible future support for uncertainty [16]. Including multiple of these extensions might lead to feature interactions that need to be addressed explicitly by the designers of the extensions.

To define language extensions in a non-intrusive way, the Henshin meta-model provides a flexible annotation mechanism. Any element contained in a transformation may be annotated with key-value pairs of strings. The language extension at hand is responsible for processing these annotations. This concept allowed us to implement variability-based rules in the variability extension alone, without modifying the language core.

5 Related Work

Model Transformation Reuse. There are two groups of reuse approaches for model transformations. The first group focuses on *intra-transformation* reuse [17], the reuse of artifacts within the same transformation. In this group, many approaches are composition-based: Rule refinement, the modularization of a rule into a set of fragmentary rules, has been implemented in the eMoflon tool [18]. The modularization of a graph transformation rules into multiple *aspects* is another compositional approach [19]. A feature-based composition approach for the reuse of ATL transformations has been proposed by Sijtema [20]. Many of these approaches do not provide an automated tool to split a legacy rule into a set of composition fragments, an issue that might be addressed by applying a general-purpose splitting tool [21,22]. An important annotative approach is rule amalgamation. While this approach allows the specification of mandatory and optional parts in a rule, in contrast to VB rules, the optional parts are matched and applied as often as possible. VB rules provide the capability to assign one element to multiple variants, which is not supported in amalgamation. Amalgamation has been implemented in the AToM3 meta-modeling tool [23] and the eMoflon Triple Graph Grammar tool [24].

The second group focuses on *inter-transformation* reuse [17], the reuse of artifacts across multiple transformations. VCT [25] is a comprehensive toolkit that allows to accommodate variability in a chain of multiple transformations and to compose larger transformations from smaller ones. Cuadrado et al. [26] have introduced a component model to orchestrate the reuse of model transformations across multiple different modeling languages. Their Bentō [27] tool provides support for generic rules for ATL transformations. These generic rules are typed over *concepts*, abstract meta-models. To consider a new scenario, concepts are instantiated by binding them to the types of the required meta-model. To increase the flexibility of this approach, de Lara et al. propose an extension that accounts for heterogeneity between concepts and meta-models [28]. Criado et al. [29] propose to reuse existing transformations by annotating them. These works are orthogonal to ours as they address a different reuse scenario.

Implementation Approaches to Software Product Lines. We adopted the distinction of annotative and composition-based mechanisms from software product line (SPL) engineering [4], where it refers to different approaches to implementing a SPL. An important composition-based approach is *Feature-Oriented Programming* [30], in which a SPL is developed by dividing its specification into features and implementing each feature as a separate module. The AHEAD [31] tool made this approach applicable for Java. An example for an annotative approach are *#ifdef directives*: Portions of the source code are annotated with variability conditions and optionally removed during compilation. *Virtual separation of concerns* (VSoC) is a paradigm aiming to combine the benefits of both approaches by means of tool support [7]. In the CIDE tool [4], users are provided custom views, visual representations, and variability-aware type checks. Based on the VSoC paradigm, Walkingshaw et al. [32] propose an editing model for variational software based on an isolation principle: Edits to a view shall only affect the variants associated with this view. We adopt this principle in one of the editing modes of our tool. In a related work of Schwägerl et al. [33], the scope of variants affected by an edit is set using a separate configuration.

The FeatureIDE [34] framework integrates many of these approaches and makes them available during the entire development cycle. Its aim is to establish a *uniformity principle* of managing variability consistently in all design, implementation, and documentation artifacts. The integration of our approach into this framework is a promising avenue for future work.

Usability-oriented Model Transformation. As we aim to improve the maintainability of complex rules, our work is related to usability-oriented model transformation, a field of research addressed in [35]. Based on the premise that users may prefer mature model editors to experimental transformation tools, the authors provide a new modeling language that can be instantiated in any given model editor and mapped back to a host transformation language. This work is complementary to ours since we aim to contribute to the maturity of model transformation tools instead of replacing them.

6 Future Work and Additional Improvement

The most important task left to future work is a user study to validate the claim that our tool environment improves usability during editing. Such a study would substantiate our anecdotic evidence that the development of rule families without a dedicated reuse concept is a highly inconvenient and error-prone task. Furthermore, we are eliciting future usability improvements. First, the visibility of distinguished variants, such as the base rule, can be further increased by providing a "hot corner" feature. Implementing such a feature proves to be challenging due to limitations of the underlying editor framework. Second, relationships between variability points are currently expressed using a logical formula. In product line engineering, dedicated formalism have emerged to capture variability, the most important one being feature models [36]. Combining feature models with VB rules seems a promising avenue for future work.

References

1. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
2. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D.S., Paige, R.F., Lauder, M., Schürr, A., Wagelaar, D.: Surveying rule inheritance in model-to-model transformation languages. J. Object Technol. **11**(2), 3:1–3:46 (2012)
3. Anjorin, A., Saller, K., Lochau, M., Schürr, A.: Modularizing triple graph grammars using rule refinement. In: Gnesi, S., Rensink, A. (eds.) FASE 2014 (ETAPS). LNCS, vol. 8411, pp. 340–354. Springer, Heidelberg (2014)
4. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 311–320 (2008)
5. Störrle, H.: On the impact of layout quality to understanding UML diagrams: size matters. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) MODELS 2014. LNCS, vol. 8767, pp. 518–534. Springer, Heidelberg (2014)

6. Sarnak, N., Bernstein, R.L., Kruskal, V.: Creation and maintenance of multiple versions. In: SCM. Berichte des German Chapter of the ACM, vol. 30, pp. 264–275. Teubner (1988)
7. Kästner, C.: Virtual separation of concerns, Ph.D. dissertation, University of Magdeburg (2010)
8. Strüber, D., Rubin, J., Chechik, M., Taentzer, G.: A variability-based approach to reusable and efficient model transformations. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 283–298. Springer, Heidelberg (2015)
9. Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., Plöger, J.: RuleMerger: automatic construction of variability-based model transformation rules. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 122–140. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49665-7_8
10. Strüber, D.: Model-driven engineering in the large: Refactoring techniques for models and model transformation systems, Ph.D. dissertation, Philipps-Universität Marburg (2016)
11. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Rouquette, N., Haugen, Ø., Petriu, D.C. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010)
12. Green, T.R.G., Petre, M.: Usability analysis of visual programming environments: a 'cognitive dimensions' framework. J. Vis. Lang. Comput. **7**(2), 131–174 (1996)
13. Deissenboeck, F., Hummel, B., Juergens, E., Pfaehler, M., Schaetz, B.: Model clone detection in practice. In: International Workshop on Software Clones, pp. 57–64. ACM (2010)
14. Strüber, D., Plöger, J., Acreţoaie, V.: Clone detection for graph-based model transformation languages. In: International Conference on Model Transformation (ICMT). Springer, 2016
15. Hermann, F., Gottmann, S., Nachtigall, N., Braatz, B., Morelli, G., Pierre, A., Engel, T.: On an automated translation of satellite procedures using triple graph grammars. In: Duddy, K., Kappel, G. (eds.) ICMB 2013. LNCS, vol. 7909, pp. 50–51. Springer, Heidelberg (2013)
16. Famelis, M.: Managing design-time uncertainty in software models, Ph.D. dissertation, University of Toronto (2016)
17. Kusel, A., Schönböck, J., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: Reuse in model-to-model transformation languages: are we there yet? J. Softw. Syst. Model. **14**, 1–36 (2013)
18. Kulcsár, G., Leblebici, E., Anjorin, A.: A solution to the FIXML case study using triple graph grammars and eMoflon. In: TTC@STAF, pp. 71–75 (2014)
19. Machado, R., Foss, L., Ribeiro, L.: Aspects for graph grammars. Electron. Commun. EASST **18** (2009)
20. Sijtema, M.: Introducing variability rules in ATL for managing variability in MDE-based product lines. MtATL **10**, 39–49 (2010)
21. Strüber, D., Selter, M., Taentzer, G.: Tool support for clustering large meta-models. In: BigMDE Workshop on the Scalability of Model-Driven Engineering. ACM Digital Library, pp. 7.1–7.4 (2013)
22. Strüber, D., Lukaszczyk, M., Taentzer, G.: Tool support for model splitting using information retrieval and model crawling techniques. In: BigMDE: Workshop on Scalability in Model Driven Engineering, pp. 44–47. CEUR-WS.org (2014)
23. de Lara, J., Ermel, C., Taentzer, G., Ehrig, K.: Parallel graph transformation for model simulation applied to timed transition petri nets. Electron. Notes Theor. Comput. Sci. **109**, 17–29 (2004)

24. Leblebici, E., Anjorin, A., Schürr, A., Taentzer, G.: Multi-amalgamated triple graph grammars. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 87–103. Springer, Heidelberg (2015)
25. Basso, F.P., Pillat, R.M., Oliveira, T.C., Becker, L.B.: Supporting large scale model transformation reuse. In: ACM SIGPLAN Notices, vol. 49(3), pp. 169–178. ACM (2013)
26. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: A component model for model transformations. IEEE Trans. Softw. Eng. **40**(11), 1042–1060 (2014)
27. Cuadrado, J.S., Guerra, E., de Lara, J.: Reusable model transformation components with bentō. In: Kolovos, D., Wimmer, M. (eds.) ICMT 2015. LNCS, vol. 9152, pp. 59–65. Springer, Heidelberg (2015)
28. de Lara, J., Guerra, E.: Towards the flexible reuse of model transformations: a formal approach based on graph transformation. J. Logical Algebraic Methods Program. **83**(5), 427–458 (2014)
29. Criado, J., Martínez, S., Iribarne, L., Cabot, J.: Enabling the reuse of stored model transformations through annotations. In: Kolovos, D., Wimmer, M. (eds.) ICMT 2015. LNCS, vol. 9152, pp. 43–58. Springer, Heidelberg (2015)
30. Prehofer, C.: Feature-oriented programming: a fresh look at objects. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 419–433. Springer, Heidelberg (1997)
31. Batory, D.: Feature-oriented programming and the AHEAD tool suite. In: International Conference on Software Engineering (ICSE), pp. 702–703. IEEE Computer Society (2004)
32. Walkingshaw, E., Ostermann, K.: Projectional editing of variational software. In: Generative Programming: Concepts and Experiences, pp. 29–38. ACM (2014)
33. Schwägerl, F., Buchmann, T., Westfechtel, B.: SuperMod: a model-driven tool that combines version control and software product line engineering. In: International Conference on Software Paradigm Trends. SCITEPRESS, pp. 5–18 (2015)
34. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: an extensible framework for feature-oriented software development. Sci. Comput. Program. **79**, 70–85 (2014)
35. Acretoaie, V., Störrle, H., Strüber, D.: Transparent model transformation: turning your favourite model editor into a transformation tool. In: Kolovos, D., Wimmer, M. (eds.) ICMT 2015. LNCS, vol. 9152, pp. 121–130. Springer, Heidelberg (2015)
36. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document (1990)