Henshin: A Usability-Focused Framework for EMF Model Transformation Development

Daniel Strüber[1], Kristopher Born[2], Kanwal Daud Gill[1],
Raffaela Groner[3], Timo Kehrer[4], Manuel Ohrndorf[5], Matthias Tichy[3]

{strueber,daud}@uni-koblenz.de, born@mathematik.uni-marburg.de,
{raffaela.groner,matthias.tichy}@uni-ulm.de,
timo.kehrer@informatik.hu-berlin.de, mohrndorf@informatik.uni-siegen.de,
[1] Universität Koblenz-Landau, Germany,
[2] Philipps-Universität Marburg, Germany, [3] Universität Ulm, Germany,
[4] Humboldt-Universität zu Berlin, Germany, [5] Universität Siegen, Germany

Abstract. Improved usability of tools is a fundamental prerequisite for a more widespread industrial adoption of Model-Driven Engineering. We present the current state of Henshin, a model transformation language and framework based on algebraic graph transformations. Our demonstration focuses on Henshin's novel usability-oriented features, specifically: (i) a textual syntax, complementing the existing graphical one by improved support for rapid transformation development, (ii) extended static validation, including checks for correct integration with general-purpose-language code, (iii) advanced refactoring support, in particular, for splitting large transformation programs, (iv) editing utilities for facilitating recurring tasks in model transformation development. We demonstrate the usefulness of these features using a running example.

1 Introduction

Model-Driven Engineering (MDE) aims to improve the productivity of software engineers by emphasizing model transformation as a central activity during software development [1]. Still, a major roadblock to a more widespread adoption of MDE is the insufficient maturity of MDE tools [2,3]. Specifically, to make MDE tools appealing to a broader user base, it is key to increase their level of usability.

Henshin [4] is a model transformation framework for the Eclipse Modeling Framework, comprising a transformation language with a graph-transformation-based visual syntax, and a tool environment with an execution engine and analysis features, including model checking and conflict analysis support. Originally designed to offer the benefits of a solid formal foundation and efficient transformation execution, Henshin was not built with usability as an explicit goal.

In fact, based on user feedback, we identify a number of critical usability limitations: First, while its visual syntax is beneficial when *reading* a transformation program, *writing* a transformation program can be complicated due to layouting issues. Second, programs can contain subtle errors that are not caught by adequate static checks. In particular, this applies when transformations are not

specified in isolation, but embedded into a richer software infrastructure. Third, when working with large transformation programs, scalability issues occur; the performance of Henshin's visual editor may suffer to the point that it becomes unusable [5]. Fourth, users are required to perform intricate and error-prone tasks, such as creating large rules that reflect the complexity of the involved meta-models.

Therefore, in Section 2 of this paper, we present the current state of Henshin, focusing on its novel features for addressing these issues. Specifically,

- we introduce a **textual syntax** for the rapid development of transformations (Section 2.1). This syntax is not intended as a replacement for the graphical one, but as a complementary means to facilitate the initial creation of a transformation program. To support long-term maintenance, we provide a higher-order transformation that can be used to derive a graphical concrete-syntax representation of the transformation program. The design of our textual syntax was informed by a qualitative interview study.
- we provide extended **static checks** for validating the well-definedness of a transformation and its use (Section 2.2). Using this checks, one can validate if a Henshin transformation program is used correctly in general-purpose-language code, e.g., if all referred rules actually exist and their parameters are assigned correctly. Furthermore, we provide checks to see if parameters are specified and used correctly within and across particular units and rules.
- we present advanced **refactoring** support, in particular, for splitting a large transformation program into multiple sub-programs (Section 2.3). Using a wizard, the user can specify target sub-programs and assign particular units and rules to them. This splitting of programs (i.e. the abstract syntax) can also be propagated to their diagram files (i.e., the concrete syntax).
- we demonstrate a selection of **editing utilities** for complicated tasks during the development of transformations (Section 2.4), including utilities to create, simplify, generalize, or clean up Henshin rules.

Running Example. We use the following transformation program as a running example throughout this paper. The program solves one of the tasks in the classical Comb benchmark by Varró et al. [6]: It constructs a *sparse grid* in the shape of Fig. 1 for a given pair of dimensions, *width* and *height*. Note that the grid has two kinds of edges: vertical and horizontal ones (dashed and bold arrows, respectively).

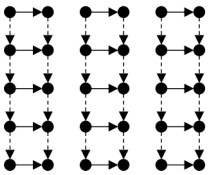

Fig. 1. Sparse grid

In Henshin, programs are specified in the form of *modules*. A module contains a set of *rules*, specifying in-place transformations, and a set of composite *units*, managing the control flow. Specifically, composite units coordinate the execution of their sub-units, which can be either rules or other composite units.

The example module shown in Fig. 2 includes three rules and four units. The entry point is the sequential unit *buildGrid* which has two input parameters, *width* and *heigth*, and one output parameter, *grid*. This unit calls two sub-units in sequential order: rule *initGrid* creates an initially empty grid to be delivered as

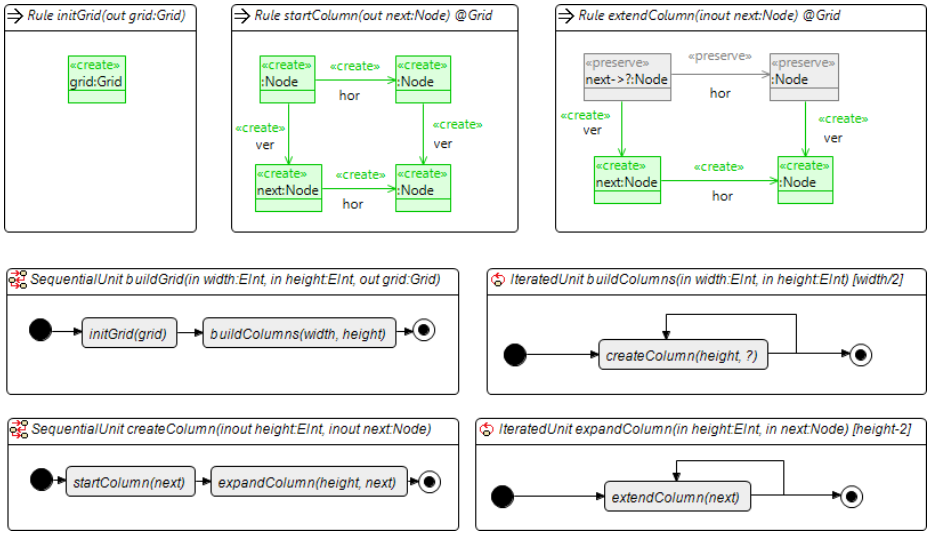

Fig. 2. Henshin module (program) for building a sparse grid.

output of the overall transformation. Iterated unit *buildColumns* has an iteration condition, specifying that its sub-unit *createColumn* is executed $width/2$ times. Sequential unit *createColumn* uses parameter *next* as a pointer to build a column of a particular height; *next* is initialized in rule *startColumn*, where the first two rows of a column are created. Note the syntactic sugar `@Grid`, which specifies the presence of a grid to be used as a container for all newly created nodes. Parameters *next* and *height* are passed to iterated unit *expandColumn*, which executes rule *extendColumn* *height-2* times. Each execution of *extendColumn* uses the *next* pointer to add another column at this node, changing the pointer to one of the newly created nodes afterwards. The program terminates after all columns have been constructed by unit *buildColumns*, yielding the sparse grid.

This graphical syntax uses a compact representation of rules, where left- and right-hand sides (LHS, RHS) are combined to one graph with annotations such as «create» for RHS nodes without a LHS counterpart. The abstract syntax maintained in the background of the visual editor captures LHS and RHS explicitly.

2 Novel Usability-oriented Features

In this section, we walk through the novel usability-oriented features.

2.1 Textual Syntax for Henshin

While much of research and practice in modeling has focused on graphical modeling languages, there has been a trend in recent years to use textual concrete syntaxes for modeling languages. The reason for that is the increasing feedback from industrial practice that graphical editors (1) require a high effort to create something usable in practice (which has been shown as a huge issue in modeling

in practice [3]), (2) require syntax correctness, resulting in cumbersome user actions to avoid intermediate incorrect models during complex editing steps, and (3) lack acceptance by many end-users, particularly, software developers.

The resurgence of modeling languages with a textual syntax is fueled by easy-to-use and feature-rich frameworks like Xtext and recent advances in projection-based modeling frameworks [7]. Finally, there is a trend to seamlessly combine graphical and textual editors to reap benefits of both worlds [8].

Henshin currently supports both a graphical editor as well as a tree-based editor. The graphical editor supports some syntactic sugar in both syntax and visualization, e.g., combined notation for LHS and RHS, NACs, container syntax. However, one needs to use the tree-based editor for more complex rules, e.g., rules with complex nesting of condition graphs. Furthermore, the usability in terms of efficiency is quite low as both the graphical editor and the tree-based editor require many steps to perform changes and high amount of focus change when moving between the graphical editor pane and the property editor.

```
rule initGridWithTwoNodes(OUT
    grid:Grid){
  graph{
    create node one:Node
    create node two:Node
    create node grid:Grid
    edges[(create one->two:ver),
      (create grid->one:nodes),
      (create grid->two:nodes)
    ]
  }
}
```

Fig. 3. Simple rule in the textual syntax

Methodology Last year, we performed a qualitative study to explore different alternatives of a potential textual syntax for Henshin. Particularly, we focused on the following variation points:

combined syntax vs. explicit LHS and RHS Shall a combined syntax using mark-ups for created and deleted elements and positive and negative applications conditions be used as in the graphical editor, or a specific LHS and RHS as in the tree-based editor?

complex application conditions How shall complex application conditions with multiple condition graphs be specified?

control flow specification Shall the control flow specification follow the current Henshin style of different units for different control flow constructs, e.g., sequences, conditions, loops, priorities or should the textual syntax resemble typical programming languages?

Furthermore, we explored other syntax variation points such as syntax for the specification of nodes, edges, and attribute assignments.

We built multiple prototypes covering the different variation points and discussed them in an interview study with 6 current and former Henshin key developers covering a diverse set of expertise in language design and experience using and developing Henshin. The interviews were based on a semi-structured questionnaire, covering demographics on the interviewees, general questions on

```
rule startNextColumn(){
   graph{
      node root:Grid
      node unnamed:Node
      create node newNode:Node
      edges[(root->unnamed:nodes),
         (create root->newNode:nodes),
         (create unnamed->newNode:hor)]
      matchingFormula{
         formula !graph1 AND !graph2
         conditionGraph graph1{
            node forbidNode:Node
            edges[(root->forbidNode:nodes),
               (forbidNode->unnamed:ver)]
         }
         conditionGraph graph2{
            node forbidNode:Node
            edges[(root->forbidNode:nodes),
               (unnamed->forbidNode:hor),
               (root->unnamed:nodes)]
         }
      }
   }
}
```

Fig. 4. Rule with complex conditions

textual vs. graphical editors and the mentioned variation points of the prototypes. They took between 1 and 1.5 hours and were executed by two of the authors of this paper. The interviews were transcribed and analyzed using thematic analysis.

Threats to Validity The external validity of our methodology is threatened by the fact that we only interviewed advanced users. Arguably, advanced users can particularly benefit from a textual syntax since they write more complicated programs, in which the limitations of graphical syntax are more obvious. Still, it yet needs to be studied if our design decisions are also useful for novice users.

Language Design The general conclusion with respect to the first variation point was that a combined syntax using markups as in the graphical editor and in various other graph transformation tools is preferable. Fig. 3 shows a transformation rule from "Full Grid", a slightly modified version of our running example.

Fig. 4 shows a transformation rule with complex condition graphs. The example describes the creation of a new initial node in a new column connected to the top node of an existing column, i.e., it neither has a horizontal incoming node nor a vertical outgoing node. Complex conditions are defined with multiple `conditionGraph`s and the `formula` keyword which contains the complex

boolean condition. Furthermore, the example contains the implicit reuse of nodes of the LHS in the `conditionGraphs` on the example of the node `unnamed`. This is similarly possible for multi rules.

Finally, Fig. 5 shows the `addColumns` transformation unit. In contrast to standard Henshin where for each type of control flow (sequence, if, loop) covering multiple rules an individual unit has to be declared, the textual syntax provides syntax constructs which are similar to imperative programming languages.

```
1 unit addColumns
2   (IN width:EInt, IN height:EInt){
3    for(width - 1){
4       startNextColumn()
5       expandNextColumn(height)
6    }
7 }
```

Fig. 5. Units

Realization We realized the textual syntax editor using Xtext with custom extensions like scoping and syntax validation. Since the language differs in syntax significantly from the Henshin meta-model, we used the Xtext generated meta-model. Instances of that meta-model are transformed by model transformations to instances of the standard Henshin meta-model. Doing so, we can reuse Henshin's visual syntax, and its interpretation and analysis plug-ins. The realized plug-ins contain automated test for the generated parser as well as automated tests for the transformation.

2.2 Static Checks

Identification and fixing of errors in the development process of a transformation program as early as possible is crucial for user satisfaction. Static checks provide an important feedback to identify such errors. Henshin supports three groups of static checks: (1) basic checks regarding the well-formedness of units and rules, (2) semantic checks regarding consistency preservation and potential mismatches between intended and specified meaning, and (3) checks for the validity of code for loading and executing units and rules. Identified violations are reported to the user using Eclipse's warning and errors markers in the respective editors.

Well-definedness checks. Violations to well-definedness constraints are detected and highlighted with an error marker. First, this applies to obvious issues such as rules and units with duplicate signatures, i.e., identical name and parameter lists. Second, parameter handling inside and between rules is checked. Parameters have a name, type, and kind, where the kinds *in, out, inout, var* specify the usage context: *in* and *inout* parameters have to be set externally; *var* and *out* parameters are set during the rule application. The value of *var* parameters is hidden to the outside world, whereas *in*, *out*, and *inout* parameters can be used to pass values between rules and units. Checks ensure that parameters are used consistently to their kind (e.g., a *var* parameter must be used in the LHS of a rule) and that parameters are passed consistently. For example, *inout* parameter *next* of rule *extendColumn* requires all units invoking the rule to specify a value, which is the case since unit *expandColumn* passes its own *next* parameter value.

Semantic checks. With semantic checks, we catch some mistakes that are frequently made by novice users. For example, if a node is to be deleted, double-pushout semantics requires that the deletion of all adjacent edges—in the case of EMF, at least the containment edge—is specified as well. Users unaware of this fact might be puzzled when an affected rule cannot be applied. Thus, for delete nodes specified without a containment edge, we show a warning (rather than an error, to support corner cases where a single root node is deleted). Moreover, we provide checks to identify rules that threaten model consistency (see [9]). For example, the application of a rule may not create containment cycles.

Integration with Java Code. Transformation programs are often used in the context of larger programs, such as Eclipse plug-ins. Henshin's Java API provides an interface for loading transformation programs, applying them, and saving the results. The API requires that certain inputs, such as unit and rule names and parameter values, are provided using method parameters. Errors such as mismatches between specified and allowed values can occur that are only discovered at runtime – a drawback resulting from Henshin's interpreter semantics. To mitigate this drawback while keeping the benefits of interpreted languages, such as flexibility w.r.t. higher-order transformations, we introduce custom checks. In Fig. 6, typos `init_grid` and `WIDTH` are detected since no suitable elements of these names exist in the input module; `"two"` yields a type error as an `int` was expected.

```
66 ruleApp = new RuleApplicationImpl(engine);
67 ruleApp.setUnit(module.getUnit("init_grid"));
68
69 unitApp = new UnitApplicationImpl(engine);
70 unitApp.setUnit(module.getUnit("buildGrid"));
71 unitApp.setParameterValue("WIDTH", 2);
72 unitApp.setParameterValue("height", "two");
```

Fig. 6. Error markers in the programmatic use.

2.3 Advanced Refactoring

Refactorings aim to improve the non-functional properties of a program, without changing its behavior. We distinguish *advanced refactorings* for achieving a higher-level goal from fine-grained *micro-refactorings*. In Henshin, typical micro-refactorings such as *rename rule* and *move rule* are supported by design: Henshin inherits EMF's features for changing models consistently in the sense that references to renamed or moved elements within the same model remain valid automatically. Therefore, in this section, we focus on advanced refactorings.

Transformation programs are typically developed iteratively. The user starts with a small module that easily fits into one screen, such as the one in Fig. 2, and end up with a large module with dozens of rules. Maintaining such a large module is difficult: Navigating the resulting diagram becomes tedious quickly; the performance of the editor may suffer to the point that it is not usable anymore.

Splitting of modules. To overcome these limitations, we provide a split refactoring that takes a module, and partitions the contained rules and units into sets that are saved into distinct modules. The splitting works in two steps: First, using the wizard shown in Fig. 7, a splitting specification is created. Users can

edit the specification by adding and removing target modules, and by reassigning rules and units using drag and drop functionality. In this example, two separate target modules are specified, one including all rules, the other including all units.

As an aid to reduce the manual specification effort, the button "Groups" produces a default suggestion based on an connected-component analysis of the call graph of units, so that each component of units and rules becomes a module. Second, the modules are created and populated with the specified rules and units. The splitting of the concrete syntax models is propagated to the diagram files, i.e., for each target module, a corresponding pair of model and diagram files is created.

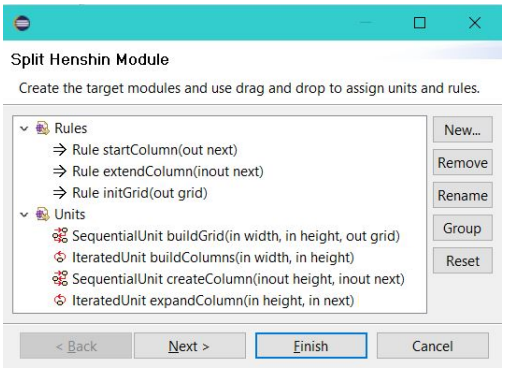

Fig. 7. Splitting wizard.

Merging of rules. A special type of complexity in transformations arises when many similar rules are required to achieve a common task. In earlier work, we extended Henshin with mechanism detecting rule clones [10] and merging them into an integrated representation that users can interact with [11,12].

2.4 Advanced Editing Support

In this section, we introduce advanced editing utilities for the development of model transformations. First, we present a basic utility that enables users to infer an initial version of a transformation rule from existing models instead of creating the rule from scratch. Second, we give a set of complex editing operations for simplifying, generalizing and cleaning up transformation programs.

Generation of Transformation Rules. We provide a facility to generate a transformation rule from a pair of models demonstrating the effect of the rule, following the principle of model transformation by-example [13]. Technically, the models serving as input of our rule generation procedure are compared with each other using EMF Compare in order to identify the corresponding elements in the original and the changed model, i.e. those elements which are considered to be the same in both models. Thereupon, a transformation rule is basically generated as follows: The original model is converted to a Henshin graph and used as the LHS of the resulting rule, while the Henshin graph obtained from the changed model is used as the RHS. Finally, LHS-RHS mappings are created for all nodes obtained from a pair of corresponding elements.

For instance, an initial version of the rule *startColumn* shown in Figure 2 can be obtained from an example where the original model contains a container element of type *Grid* and the changed model includes a *Grid* element including

the four elements of type *Node* and their respective connections. Thereupon, a Henshin module including the necessary meta-model imports and the generated transformation rule is obtained. To finally obtain the rule *startColumn*, we may first add the node identifier *next* to the lower left node, and subsequently use the advanced editing operation "Deduce Parameters" creating the out parameter *next* and adding it to the rule's signature.

Despite the simplicity of this example, this mechanism can reduce the development effort largely, particularly in the presence of accidental complexity in the sense that rules simply reflect the complexity of the involved meta-model. For instance, the UML meta-model is infamous for its size and complexity that leads to complicated rules even when expressing transformations that are simple on a conceptual level [14]. Here, examples can be provided in a much more compact form using a graphical UML editor [14]. Moreover, when transformation developers are no experts for the meta-model(s) over which the transformation rules to be developed are typed, their development from scratch is likely to be error-prone, e.g. because developers forget to specify certain edges which may lead to unexpected violations of dangling reference constraints in case of deletions.

Complex Editing Operations. In the sequel, we present a set of complex editing operations, each of them taking a structural element of an existing transformation (*Module*, *Unit* or *Rule*) as input and performing the update of the given element in an in-place fashion. All operations are made available through the "Advanced Editing" menu of the Henshin editor, their visibility depends on the selected context element. Each of these operations formalizes a recurring task during transformation development.

reduceToMinimalRule(Rule) takes a transformation rule as input and reduces it to the minimal rule yielding the same effect. Essentially, it (i) deletes all application conditions and (ii) cuts all context not needed to achieve the specified change, i.e. elements to be preserved by the rule which are not serving as boundary element of a change action are being deleted.

generalizeNodeTypes(Rule) takes a transformation rule as input and converts the types of all LHS nodes to the most general yet still valid supertype. Given a LHS node n of type T, then a supertype T_{sup} of T is a valid supertype in this context if all edges incident to n may be also incident to nodes of type T_{sup} without violating the conformace to the underlying meta-model.

cleanUp(Rule) has been introduced in an earlier Henshin release, e.g. to remove invalid LHS-RHS mappings as well as invalid multi-mappings from the given input rule. We extend this editing operation by also deleting unused rule parameters, i.e., all rule parameters which are neither mapped to a node identifier nor to an attribute variable.

3 Related Work

Usability in model transformations. Most works on addressing usability during model transformations focus on the usability in the resulting system.

Panach et al. [15] propose to map reusable usability patterns, such as *cancel*, *undo*, and *warnings*, to system models and their transformations, so that the generated systems can benefit from these features. Ammar et al. [16] investigate parametrized model transformations, where usability requirements can be used to select the most suitable among different alternatives. The paradigm of end-user model transformation [14] enables users to specify model transformations using regular model editors. This approach is orthogonal to ours, as it aims to replace specialized transformation editors, rather than to improve them.

Textual syntax. Interestingly, most model transformation languages based on the graph transformation formalism provide a graphical concrete syntax, e.g., Fujaba, VMTS, ModGraph, whereas other model transformation languages like QVT, ATL, provide a textual concrete syntax. Viatra2 [17] and GrGen [18] are the exceptions as they are based on graph transformations but provide a textual syntax. PROGRES [19] uses an hybrid syntax, whereas eMoflon [20] offers a textual syntax with a generated read-only visualisation, an interesting compromise for supporting both textual and visual notations. Arguably, in our case, having an editable graphical syntax is beneficial since users may use custom layout to give cues beyond the formal language semantics. To the best of our knowledge our work is the first empirical work on designing a textual syntax for model transformations.

Static checks. Existing works on verifying or testing model transformations focus on correctness w.r.t. a behaviour specification [21,22]. In contrast, our semantic checks can be applied when no specification is available: they represent an heuristics based on accrued experiences with users who were unfamiliar with Henshin's double-pushout semantics. Moreover, to the best of our knowledge, the validation of code that uses a transformation has not been considered before. It is worth pointing out that such validation only make sense for interpreted languages such as Henshin. Compiled ones such as PROGRES avoid the encountered issues via the type system of the target language. However, compilation has certain trade-offs, such as less flexible usage workflows. PROGRES also allows defining precisely how external code is to be integrated with the system.

Advanced refactoring. Our *split module* refactoring is inspired by an earlier tool-supported aproach to meta-model splitting [23]. While this earlier work used clustering algorithms to identify groups of related classes, our default splitting suggestion is based on a component analysis of the call graph of units and rules. Another related work allows remodularizing ATL transformations using clustering [24], where the focus was on the identification of explicit interfaces.

Editing utilities. Our most advanced editing utility is the generation of rules from existing examples. Learning model transformations from examples is highly desirable and has motivated a plethora of work, as surveyed in [13,25]. However, most approaches are not usable for our purposes since they (i) rely on the logging of editing commands demonstrating the transformation, or (ii) target model-to-model transformations. For state-based approaches targeting in-place transformations, tool support is scarcely available and, to the best of our knowledge, none of the existing tools generates Henshin rules. Our current solution is

leightweight in the sense that it abstains from using sophisticated inference algorithms, machine learning techniques or other third-party software, which was a major design decision to keep the deployment of Henshin easy to handle.

4 Conclusion and Future Work

Compared to purely textual languages, where developers have to piece together graph structures in their minds while reading a model transformation program, it is tempting to view graph-based ones as inherently user-friendly. Still, experience has shown that the devil is often in the details: while particular usability issues might not be obvious in smaller examples, the development of larger transformation program can be a tedious task. With the present work, we make a number of contributions to resolve the issues encountered during such tasks.

In the future, we are interested to study the impact of our usability-oriented features. An empirical user study would be an appropriate basis to determine the usefulness of our contributions. A key challenge for advanced features such as those introduced here is to make users aware of usage opportunities [26]. Furthermore, we plan to extend Henshin with additional features. Inferring transformation rules from a set of examples instead of a single one is an interesting goal, in which we may benefit from the existing literature on transformation-by-example. Finally, in our ongoing work, we aim to support the debugging of Henshin rules via an integration into the Eclipse Debugging infrastructure.

Acknowledgement. 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 EU's Horizon 2020 programme, Proposal number: 653642). This work was partially supported by the DFG (German Research Foundation) (grant numbers TI 803/2-2 and TI 803/4-1).

References

1. S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.
2. J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "Industrial adoption of model-driven engineering: Are the tools really the problem?" in *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. Springer, 2013, pp. 1–17.
3. G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Assessing the state-of-practice of model-based engineering in the embedded systems domain," in *International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*, 2014, pp. 166–182.
4. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: advanced concepts and tools for in-place emf model transformations," in *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. Springer, 2010, pp. 121–135.
5. D. Strüber, T. Kehrer, T. Arendt, C. Pietsch, and D. Reuling, "Scalability of model transformations: Position paper and benchmark set," in *Workshop on Scalable Model Driven Engineering (BigMDE)*, 2016, pp. 21–30.

6. G. Varró, A. Schurr, and D. Varró, "Benchmarking for graph transformation," in *Symposion on Visual Languages and Human-Centric Computing*. IEEE, 2005, pp. 79–88.
7. M. Voelter, T. Szabó, S. Lisson, B. Kolb, S. Erdweg, and T. Berger, "Efficient development of consistent projectional editors using grammar cells," in *International Conference on Software Language Engineering (SLE)*, 2016, pp. 28–40.
8. S. Maro, J. Steghöfer, A. Anjorin, M. Tichy, and L. Gelin, "On integrating graphical and textual editors for a UML profile based domain specific language: an industrial experience," in *International Conference on Software Language Engineering (SLE)*, 2015, pp. 1–12.
9. E. Biermann, C. Ermel, and G. Taentzer, "Formal foundation of consistent emf model transformations by algebraic graph transformation," *Software & Systems Modeling*, vol. 11, no. 2, pp. 227–250, 2012.
10. D. Strüber, J. Plöger, and V. Acretoaie, "Clone Detection for Graph-Based Model Transformation Languages," in *International Conference on Model Transformations (ICMT)*. Springer, 2016, pp. 191–206.
11. D. Strüber, J. Rubin, T. Arendt, M. Chechik, G. Taentzer, and J. Plöger, "Rule-merger: Automatic construction of variability-based model transformation rules," in *International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2016, pp. 122–140.
12. D. Strüber and S. Schulz, "A tool environment for managing families of model transformation rules," in *International Conference on Graph Transformation (ICGT)*. Springer, 2016, pp. 89–101.
13. G. Gerti Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Model transformation by-example: a survey of the first wave," in *Conceptual Modelling and Its Theoretical Foundations*. Springer, 2012, pp. 197–215.
14. V. Acreţoaie, H. Störrle, and D. Strüber, "VMTL: a language for end-user model transformation," *Software & Systems Modeling*, pp. 1–29, 2016.
15. J. I. Panach, S. España, A. M. Moreno, and Ó. Pastor, "Dealing with usability in model transformation technologies," in *International Conference on Conceptual Modeling*. Springer, 2008, pp. 498–511.
16. L. B. Ammar, A. Trabelsi, and A. Mahfoudhi, "Incorporating usability requirements into model transformation technologies," *Requirements Engineering*, vol. 20, no. 4, pp. 465–479, 2015.
17. D. Varró and A. Balogh, "The model transformation language of the VIATRA2 framework," *Sci. Comput. Program.*, vol. 68, no. 3, pp. 214–234, 2007.
18. R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski, "GrGen: A fast SPO-based graph rewriting tool," in *International Conference on Graph Transformation (ICGT)*. Springer, 2006, pp. 383–397.
19. A. Schürr, A. J. Winter, and A. Zündorf, "The PROGRES approach: Language and environment," in *Handbook of graph grammars and computing by graph transformation*. World Scientific Publishing Co., Inc., 1999, pp. 487–550.
20. E. Leblebici, A. Anjorin, and A. Schürr, "Developing eMoflon with eMoflon," in *International Conference on Theory and Practice of Model Transformations (ICMT)*. Springer, 2014, pp. 138–145.
21. A. Rensink, Á. Schmidt, and D. Varró, "Model checking graph transformations: A comparison of two approaches," in *International Conference on Graph Transformation (ICGT)*. Springer, 2004, pp. 226–241.
22. J. Cabot, R. Clarisó, E. Guerra, and J. De Lara, "Verification and validation of declarative model-to-model transformations through invariants," *Journal of Systems and Software*, vol. 83, no. 2, pp. 283–302, 2010.
23. D. Strüber, M. Selter, and G. Taentzer, "Tool support for clustering large metamodels," in *Workshop on Scalability in Model Driven Engineering (BigMDE)*, 2013, pp. 7:1–4.
24. A. Rentschler, D. Werle, Q. Noorshams, L. Happe, and R. H. Reussner, "Remodularizing legacy model transformations with automatic clustering techniques," in *Workshop on Analysis of Model Transformations (AMT)*, 2014, pp. 4–13.
25. I. Baki and H. Sahraoui, "Multi-step learning and adaptive search for learning complex model transformations from examples," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, pp. 20:1–20:37, 2016.
26. T. Buchmann, B. Westfechtel, and S. Winetzhammer, "The Added Value of Programmed Graph Transformations - A Case Study from Software Configuration Management," in *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. Springer, 2011, pp. 198–209.