MUTANT: Model-Driven Unit Testing using ASCII-art as Notational Text

Daniel Strüber, Felix Rieger, Gabriele Taentzer

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

Abstract. There are two established strategies to create test models: Textual notations require developers to mentally reconstruct the involved graph structures *ad hoc*; maintenance effort and time are increased. Existing visual notations compel developers to switch frequently between different editors; keeping the resulting test artifacts synchronized is complicated by insufficient tool support. In this paper, we propose to specify test models using ASCII-art – a text-based visual notation used by developers in their everyday practices. To outline our vision, we employ a motivating example and discuss challenges such as editing, collaboration, and scalability. The discussion sheds a promising light on the new idea, notably for its usefulness in collaborative and reuse-intensive settings.

1 Introduction

Model-Driven Engineering (MDE) emerged as a paradigm to combat complexity in software engineering through the use of visual notations. While its basic premise appears promising, MDE has not found broad adoption as a mainstream software development methodology, and a major cause for this failure is often seen in inadequate tool support [1]. In particular, Batory et al. found the graphical tools in a widely-known modeling framework unsuitable for teaching MDE to students, considering them unappealing and unintuitive [2].

This lack of enthusiasm is seconded by practitioners. In a recent online discussion, an industry participant observed that *"the graphical tools are bulky, lack often some features like (smart) versioning, merging, collaborating, good integration into the overall development tool chain. [...] Working with mouse, property dialogs, popup windows never gains the speed of a well configured source IDE."*[1]

In this work, we aim to provide the benefits of visual models without inducing any of these tool-related problems. The idea is to specify models using *ASCII-art*, a text-based visual notation used by developers in their everyday practices [3]. ASCII-art is widely used in specification documents, such as Internet RFCs [4]. We will explore this idea in a context where it appears particularly well-suited: Simplifying *unit testing*, a cornerstone in software quality assurance [5].

[1] *http://modeling-languages.com/failed-convince-students-benefits-code-generation* User *Det*, 2015-02-11

Often, a system under test takes as input a data structure with an intuitive visual representation: Consider (i) a routing algorithm finding shortest paths in a graph of towns, (ii) a model transformation deriving database schemas from class models, or (iii) a social network site proposing new friends based on a social graph. Adopting MDE terminology, we will call such data structures *models*.

There are two established strategies to create test models: The first is textual specification using APIs or dedicated DSLs. However, textual notations are not suited to capture the visual nature of the involved models. It has been reported that the use of visual models respecting certain layouting criteria has a positive effect on cognitive load [6]. Hence, the second option is to construct test models using visual tools. Yet this results in a complicated process, involving repeated switching between different editors and keeping test models and code synchronized – challenging tasks, given the lack of production-ready collaboration tools.

In this paper, we propose *model-driven unit testing using ASCII-art as notational text* (MUTANT): Unit tests are annotated with test models, using a visual notation based on ASCII characters. The main component of the approach is a *model compiler* that derives models from these annotations and provides these models to the test framework. By adding it to the build script, this compiler can be integrated into any given IDE. We put forth the following design rationale:

I. Diffing and merging annotated code, crucial tasks in versioning and collaboration, is supported by mature tools included in state-of-the-art IDEs.

II. Collaborators and project stakeholders, such as managers and customer-side developers, can view test models without requiring any tool setup.

III. The notation is not bound to a specific modeling platform. Alternative platforms can be supported by providing separate model compilers.

IV. Creating new test models from existing ones boils down to copying text. Sec. 2 reports on a daunting experience of performing this task using visual tools.

The rest of this paper is structured as follows: In Sec. 2, we introduce the approach by example. In Secs. 3 and 4, we discuss challenges and an implementation prototype. We discuss related work and conclude in Secs. 5 and 6.

2 Motivating Example

Consider the following example to compare the new approach against the established approaches to test model specification. The system under test is an implementation of the *pull up attribute* refactoring [7] for class models: If two classes have the same superclass and attributes of the same type and name, the attributes are moved to the superclass. The unit tests in this example involve a part where a test model is provided, a part where the refactoring is applied, and assertions to check if the attribute was pulled up.

Textual specification. Fig. 1 presents a unit test for the refactoring based on the Eclipse Modeling Framework (EMF), a modeling platform often used to create and persist models [8]. In lines 4-12, the test model is created: A *package* acting as overarching container, *classes*, and their *attributes* are created. In lines 13-14, `person` is set as common superclass for the `professor` and `student` classes. In lines 16-19, the refactoring is applied and assertions are checked.

```
public class RefactoringTest extends UnitTest {
  @Test
  public void testRefactoring() {
    EFactory fact = EcoreFactory.eINSTANCE;
    EPackage pkg = fact.createEPackage();
    EClass person = fact.createEClass(pkg);
    EClass professor = fact.createEClass(pkg);
    EClass student = fact.createEClass(pkg);
    EAttribute attr1 = fact.createEAttribute(
      "name", String.class, professor);
    EAttribute attr2 = fact.createEAttribute(
      "name", String.class, student);
    professor.setSuperClass(person);
    student.setSuperClass(person);

    assertTrue(person.getAttributes().size()==0);
    PullUpRefactoring refac = new
        PullUpRefactoring(pkg);
    refac.execute();
    assertTrue(person.getAttributes().size()==1);
  }
}
```

Fig. 1. Textual test model specification.

```
public class RefactoringTest extends UnitTest {
  @Test
  public void testRefactoring() {
    EPackage pkg = load("/test1/Test.ecore");
    EClass person = pkg.getEClass("Person");

    assertTrue(person.getAttributes().size()==0);
    PullUpRefactoring refac = new
        PullUpRefactoring(pkg);
    refac.execute();
    assertTrue(person.getAttributes().size()==1);
  }
}
```

Fig. 2. Loading a visually specified test model.

This approach has two advantages: Test model and code are maintained at the same place, eliminating the need for context and tool switching. Since the input model is specified textually, it can be easily diffed and merged, facilitating collaboration. However, the easy tractability of the code comes at a price: The specification does not reflect the graph structure of the input model. Developers have to reconstruct the graph in their minds, adding a level of complexity to the understanding process and increasing maintenance effort and time.

External visual specification. Fig. 2 shows an equivalent unit test where the test model, created using a visual editor, is loaded from the file system.

This solution has two benefits: The input model is specified visually, and the test code remains clean and simple. On the downside, to understand and maintain tests, developers are forced to switch between input models and tests. Furthermore, the absence of the actual test model in this presentation reflects the situation collaborating developers might find themselves in: The committing developer might have used an incompatible version of the visual tool, or have forgotten to include the test model in the commit. From our own experience, we report on another serious drawback: To achieve a good test coverage, it is reasonable to reuse test models by copying and modifying them. In EMF, this is a complicated process: Models and their layout information are distributed over several files, using hard-coded references that have to be adjusted manually.

MUTANT. Fig. 3 exemplifies test specification using the proposed approach. The test model is provided in the Javadoc accompanying the test method, using a custom `@InputModel` parameter. In the employed syntax, boxes indicate classes. The generalization relation is indicated by arrows: The character `A` resembles a closed arrowhead. We provide a model compiler to parse such annotations and derive models, making them accessible through a dedicated API, invoked in line 18. To specify output models, the parameter `@OutputModel` can be used.

```
1  public class RefactoringTest extends UnitTest {
2    @Test
3    /** @InputModel EPackage pkg =
4
5                    +-------------+
6                    |   Person    |
7                    +-------------+
8                        A    A
9                .-------'    '-------.
10               |                     |
11         +---------------+  +---------------+
12         | Professor     |  | Student       |
13         |---------------|  |---------------|
14         | name: String  |  | name: String  |
15         +---------------+  +---------------+
16     */
17   public void testRefactoring() {
18     EPackage pkg = Mutant.getPackage("pkg");
19     EClass person = pkg.getEClass("Person");
20
21     assertTrue(person.getAttributes().size()==0);
22     PullUpRefactoring refac = new
           PullUpRefactoring(pkg);
23     refac.execute();
24     assertTrue(person.getAttributes().size()==1);
25   }
26 }
```

Fig. 3. Specifying a test model using ASCII-art.

The outlined solution combines the advantages of both previous solutions without suffering from any of their drawbacks: It establishes locality and use of text-based collaboration tools while maintaining an intuitive, graphical layout, indicating the layout of the known graphical model notation.

3 Challenges and Vision

In this section, we outline the full vision of the proposed approach, highlighting its strengths, challenges, and strategies to tackle these challenges.

Editing. An important benefit of ASCII-art over non-textual graphical representations is that it can be created, edited and viewed using any text editor. However, not all text editors are recommendable for all of these tasks. Depending on the editor at hand, the editing process can be cumbersome: For instance, moving a box horizontally may require to add whitespace in successive lines.

Instead, our approach targets state-of-the-art code IDEs. These IDEs come with powerful text editors: Eclipse, IntelliJ and Visual Studio provide dedicated modes allowing to manipulate successive lines by a single keystroke.[2] Capabilities to add, insert or remove characters in a column-wise fashion and to select or manipulate *boxes* facilitate ASCII-art editing noticeably. To accommodate larger editing steps, we propose the use of specialized text editors, providing convenient features such as box drawing and freehand erasers.[3] For viewing and minor edits, the IDE code editor remains the designated tool – considerably reducing the need for context switching imposed by processes involving graphical editors.

Collaboration. We base our approach on the claim that employing ASCII-art enables the use of the mature collaboration tools found in present-day IDEs. More specifically, we maintain that the existing line-based *diff* and *merge* tools are sufficient in most cases. Yet, we identify two caveats:

First, text differencing considers just the syntax of the involved models and is oblivious to their semantics. As a consequence, syntactically different models are reported as different, even if they are semantically equivalent. This is analogous to code diffs, where two semantically equivalent lines of code may be highlighted as different. Second, as the only problematic case for merging, we identify merge conflicts concerning the same line of code, which have to be resolved by hand – again, a situation that is accepted in source code editing.

Scalability. The proposed idea is particularly suited for the creation of unit tests since the models involved in unit testing tend to be relatively small: Unit tests usually represent *edge cases* reflecting the core of a critical scenario. However, particular edge cases may demand large models. To address these cases, we must account for the inherent limitations of text-based notations: Zooming is not available. Space may be limited to 80 or 100 characters per line.[4]

[2] Eclipse: *block selection mode*, IntelliJ and Visual Studio: *column mode*.

[3] e.g. http://asciiflow.com/, http://www.jave.de/, http://emacswiki.org/emacs/ArtistMode. Retrieved on 2015-08-31.

[4] The example model in Fig. 3 stretches over 33 characters of horizontal dimension.

In order to address the space limitation, we provide adjustments that help to increase notational compactness. To compact multiple objects of the same type, nodes can carry a multiplicity, indicated by the character `n`. To compact multiple links of the same type, edges can be multi-edges, i.e., have multiple source or targets. To remove visual clutter in models with many edges – a potential obstacle to developer performance [9] –, we introduce *abbreviated edges*, a concept inspired by net labels in ECAD software[5]. The example model in Fig. 4 shows a school with ten teachers, nine of them named *Mary*, one named *Ed*.

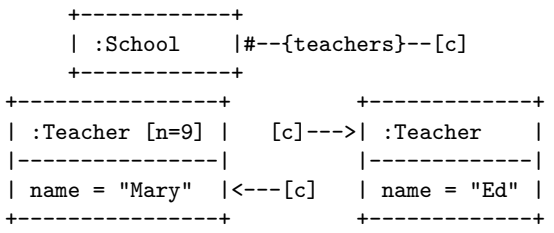

Fig. 4. Test model with multi-node and abbreviated multi-edge.

In the future, we aim to empirically investigate the scalability of the proposed approach, including the effect of these mitigation strategies. In any case, it might be argued that the advantage of unlimited space and zooming in graphical tools is debatable in the first place: It has been observed that diagrams exceeding a certain size can present an obstacle rather than an aid to comprehension [10].

Scope. To make the approach broadly applicable in various application domains, we aim to provide support for different modeling languages. All modeling languages can be supported by considering their *abstract syntax* – a representation based on boxes and arrows, exemplified in Fig. 4. Still, developers may prefer the known *concrete syntax* (CS) notations. The example in Fig. 3 presents a CS-based notation for class models. To support extensibility for arbitrary languages, we propose a framework approach, allowing customization for new languages by defining visual tokens and their mapping to the underlying meta-model.

Syntax errors. In present-day visual and textual editors, developers are supported by syntax checks, allowing them to discover specification errors early, during editing rather than at runtime. Feedback on syntax errors is part of our basic approach: Its central component, a model compiler, is able to return specific error messages in case that syntax errors are found. These error messages are forwarded to the IDE by means of the build script invoking the compiler.

Conversion. It is desirable to enable an easy transition between external specifications and the proposed notation. Sources of external specification may include regular models in custom layout formats, PowerPoint and Visio documents, and even hand-drawn sketches photographed using smartphone cameras.

[5] e.g., KiCad, http://tinyurl.com/kicad-labels/. Retrieved on 2015-08-31.

As a promising approach to address this sophisticated task, we aim to use an ASCII-art generation heuristics that accounts for line structures found inside an input graphic [11]. Postprocessing is required to ensure that valid instances of the proposed syntax are created. Furthermore, if the existing test models were generated automatically, they might not have a layout in the first place or be too large for a visualization. To support them, we intend to apply state-of-the-art layouting [12, 13] and splitting tools [14–16].

4 Implementation Prototype

We provide a prototypical implementation for the proposed approach. The main component of the implementation is a model compiler that can be plugged into any given IDE by adding an entry to the build configuration or script. During compilation of annotated test classes, the compiler parses the contained ASCII-art annotations to create models which are made available to the test framework by means of a dedicated API. The implementation and instructions for plugging it into Eclipse are found at `https://github.com/frieger/mutant-ascii`.

We support class models through a dedicated concrete syntax and arbitrary modeling languages through abstract syntax. The parser first identifies boxes and extracts contained information on names and types. Then, it identifies arrowheads, following the adjacent edges until a box or abbreviated edge label is hit. It detects and follows remaining abbreviated edges, connecting those with identical labels and converting multi-edges to multiple single edges. The information collected on objects and their relations serves as input to a model builder.

We initially planned to use Java annotations for specifying test models, which proved infeasible since multi-line String literals are not supported in Java. Instead, as shown in Fig. 3, we embed models in Javadoc. Conceptually, Javadoc is a suited place: The input and output of the code under test are documented.

5 Related Work

5.1 Model-Driven Testing

The proposed approach can be considered as a novel incarnation of model-driven testing (MDT). MDT places models as key artifacts in testing. In earlier MDT approaches, the aim was to derive test models using abstract specifications such as *coverage criteria* [17], *dedicated profiles* [18], and *visual contracts* [19]. While automated generation of test cases is a desirable and well-studied goal [20], the trade-off is a significant initial cost in adopting the associated methods and tools. From an enterprise perspective, this poses a considerable risk, notably when one takes into account that the approaches are not tailored to all relevant tasks: For instance, the approaches require to specify behavior using rules or sequence diagrams, focusing on changes of object structures and neglecting algorithmic functionality such as graph routing. Specifications are translated to plain test cases that may expose the indicated drawbacks.

Unlike previous approaches to MDT that focus on the generation of test models, the new approach takes an agile stance, allowing rapid test model creation: Developers specify test models directly, using a notation designed for easy reuse and collaboration. This allows focusing on the intuitive process of identifying *edge cases*. To our knowledge, our approach is the first to represent models in a dedicated notation to facilitate testing. To still reap the benefits of the existing MDT approaches, we aim to provide converters for the derived test models.

5.2 ASCII-art Notations in Software Engineering

Several software engineering problems have been tackled using ASCII-art. *TextTest* [21] is a tool for graphical user interface (GUI) testing based on the capture-replay paradigm: Developers interact with the GUI under test. After each interaction, a GUI snapshot is saved using ASCII-art, enabling automated regression tests. This approach is complementary to ours: Capture-replay is only available for GUIs, while our approach targets at the broad class of functionality tests that involve *models*. Furthermore, documenting each interaction leads to many text artifacts, not promoting locality and easy reuse. Another complementary approach [22] uses an ASCII-based model notation for code generation. The authors mention converters from models to ASCII-art and back; however, they do not explicate their realization strategy. They consider class models. In [23], diagram parsing is used to recover grammars for existing programming languages from reference manuals. The authors discuss an interesting solution based on *attributed multiset grammars* [24]. [25] proposes a context-free grammar for ASCII-art tables as found in network protocol RFCs.

5.3 Visualizations in Textual IDEs

The lack of visual expressiveness associated with textual notations has motivated work on visualization in code IDEs. The *JetBrains MPS* language workbench [26] supports a form of integrated textual and visual editing: Language developers can define custom box-and-arrow type diagram views that are embedded into source code editors. Such embedded views mitigate several of the problems of purely visual or textual editing, such as comprehension effort and context switching.

On the downside, they only facilitate the editing process. Diffing and merging models remains a challenge. Moreover, this approach is coupled to a specific IDE, which raises multiple problems: Developers are forced to use this IDE, which is undesirable if a particular preferred IDE exists in their domain. Besides, as in any new and experimental IDE, the embedded editors may show some of the issues reported for graphical tools. Finally, specific IDEs come with an increased business risk: It is not guaranteed that support is continued in the future. In contrast, our approach offers a drop-in solution designed to support arbitrary IDEs and their mature versioning and collaboration tools. To combine the benefits of both approaches, we consider customizing MPS to use its embedded views as front-end editors for ASCII-art model representations.

mbeddr [27], an extension of JetBrains MPS targeted at embedded software development, provides built-in visualizations for state machines. The *Xtext* language workbench [28] allows to visualize instances of textual DSLs using the ZEST visualization library [29]. Both approaches provide read-only visualizations, while the embedded views in MPS are also editable.

5.4 Tool Reuse

Our premise of using production-ready tools rather than experimental ones designed for specific purposes is inspired by recent work on usability-oriented MDE tools. The *Visual Model Transformation Language* (VMTL) [30] allows to specify model transformations using regular model editors. Similar to the new approach, VMTL uses annotations to enable the reuse of existing technology: In VMTL, models are annotated to specify transformation rules. In this work, test code is annotated to specify test models. However, while VMTL allows to reuse model editors, the current work reflects our experience that these editors are not well-suited for test model creation – an issue we avoid by using textual IDEs.

6 Conclusion and Future Work

Tests are of paramount importance in software engineering. We target the challenge of model-driven unit testing: Instead of using external editors to view and edit test models, we embed the models in the Javadoc comments accompanying the test cases. The approach is text-based and does not modify the programming language's syntax, allowing to use existing editing, versioning, and collaboration tools. The text-based visual syntax is designed to resemble the well-known graphical notations while allowing to reduce visual clutter. As in visual tools, model elements are aligned freely, supporting comprehension through spatial clues.

We address a set of challenges and solution ideas that we aim to investigate more deeply in the future. These challenges include the development of converters from external specifications to ASCII-art, the development of a framework to support multiple modeling languages through their concrete syntax, and the empirical investigation of the approach's scalability and general usefulness. Tackling these challenges will lead to a set of domain- and IDE-independent tools enabling developers to write tests more easily, combining the benefits of Test-Driven Development and Model-Driven Engineering.

References

1. J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?" in *Model-Driven Engineering Languages and Systems*. Springer, 2013, pp. 1–17.
2. D. Batory, E. Latimer, and M. Azanza, "Teaching Model Driven Engineering from a Relational Database Perspective," in *Model-Driven Engineering Languages and Systems*. Springer, 2013, pp. 121–137.
3. M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's Go to the Whiteboard: How and Why Software Developers use Drawings," in *Conf. on Human Factors in Computing Systems*. ACM, 2007, pp. 557–566.

4. L. Zhu, V. Chen, J. Malyar, S. Das, and P. McCann, "RFC 7545: Protocol to Access White-Space (PAWS) Databases," 2015.
5. G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing.* John Wiley & Sons, 2011.
6. H. Störrle, "On the Impact of Layout Quality to Understanding UML Diagrams," in *Visual Lang. and Human-Centric Comp.* IEEE, 2011, pp. 135–142.
7. M. Fowler, *Refactoring.* Addison Wesley, 2002.
8. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.
9. D. Whitney and D. M. Levi, "Visual Crowding: A Fundamental Limit on Conscious Perception and Object Recognition," *Trends in cognitive sciences*, vol. 15, no. 4, pp. 160–168, 2011.
10. H. Störrle, "On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters," in *Model-Driven Engineering Languages and Systems.* Springer, 2014, pp. 518–534.
11. X. Xu, L. Zhang, and T.-T. Wong, "Structure-based ASCII art," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, pp. (52) 1–10, 2010.
12. M. Spönemann, *Graph Layout Support for Model-Driven Engineering.* PhD diss., Uni Kiel, 2015.
13. S. Maier and M. Minas, "A Pattern-based Approach for Initial Diagram Layout," *Electronic Communications of the EASST*, vol. 58, 2013.
14. D. Strüber, M. Selter, and G. Taentzer, "Tool support for clustering large metamodels," in *Proceedings of the Workshop on Scalability in Model Driven Engineering.* ACM, 2013, p. 7.
15. D. Strüber, J. Rubin, G. Taentzer, and M. Chechik, "Splitting Models using Information Retrieval and Model Crawling Techniques," *Fundamental Approaches to Software Engineering*, pp. 47–62, 2014.
16. D. Strüber, M. Lukaszczyk, and G. Taentzer, "Tool Support for Model Splitting using Information Retrieval and Model Crawling Techniques," in *Proceedings of the Workshop on Scalability in Model Driven Engineering. ACM*, 2014.
17. R. Heckel and M. Lohmann, "Towards Model-Driven Testing," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 6, pp. 33–43, 2003.
18. P. Baker, Z. R. Dai, J. Grabowski, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile.* Springer, 2007.
19. G. Engels, B. Güldali, and M. Lohmann, "Towards Model-Driven Unit Testing," in *Models in Software Engineering.* Springer, 2007, pp. 182–192.
20. S. Anand, E. Burke, T. Chen, J. Clark, M. Cohen, W. Grieskamp, M. Harman, M. Harrold, and P. McMinn, "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.
21. E. Bache and G. Bache, "Specification by Example with GUI Tests-How Could That Work?" in *Agile Processes in Software Engineering and Extreme Programming.* Springer, 2014, pp. 320–326.
22. H. Washizaki, M. Akimoto, A. Hasebe, A. Kubo, and Y. Fukazawa, "TCD: A text-based UML class diagram notation and its model converters," in *Advances in Software Engineering.* Springer, 2010, pp. 296–302.
23. R. Lämmel and C. Verhoef, "Semi-automatic Grammar Recovery," *Software: Practice and Experience*, vol. 31, no. 15, pp. 1395–1438, 2001.
24. S.-K. Chang, "Picture Processing Grammar and its Applications," *Information Sciences*, vol. 3, no. 2, pp. 121–148, 1971.
25. A. Kay, D. Ingalls, Y. Ohshima, I. Piumarta, and A. Raab, "Steps toward the Reinvention of Programming," Technical report, National Science Foundation, Tech. Rep., 2006.
26. M. Voelter and K. Solomatov, "Language modularization and composition with projectional language workbenches illustrated with MPS," *Software Language Engineering*, vol. 16, 2010.
27. M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems," in *C. on Systems, Progr., and Apps.* ACM, 2012, pp. 121–140.
28. M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *ACM International Conf. Object-Oriented Programming Systems Languages and Applications Companion.* ACM, 2010, pp. 307–309.
29. R. I. Bull, *Model Driven Visualization: Towards a Model Driven Engineering Approach for Information Visualization.* PhD diss., University of Victoria, 2008.
30. V. Acretoaie, H. Störrle, and D. Strüber, "Transparent Model Transformation: Turning Your Favourite Model Editor into a Transformation Tool," in *International Conf. on Model Transformations.* Springer, 2015, pp. 121–130.