A Text-Based Visual Notation for the Unit Testing of Model-Driven Tools

Daniel Strüber, Felix Rieger, Gabriele Taentzer

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

Abstract. During the unit testing of model-driven tools, a large number of models and test classes needs to be managed and maintained. Typically, some of these artifacts are specified manually, some are generated automatically. Existing approaches to test management rely on the available visual and textual modeling notations. As these notations are not tailored to unit testing, distinct maintainability trade-offs arise.
In this paper, we propose a notation that aims to combine the benefits of visual and text-based approaches. The notation is at the same time visual and text-based, as it uses ASCII characters to emulate the familiar graphical notations. In our evaluation based on real models, we identify problematic model shapes challenging the scalability our notation, while finding that it is well-suited to capture typical test models.

1 Introduction

In Model-Driven Engineering, software engineers employ a large variety of tools to specify, transform, and manage models. Such *model-driven tools*, like all software artifacts, are routinely affected by software defects.

A standard practice to detect and resolve software defects is *unit testing*, the process of testing individual parts of a system to determine if they comply with its behavior specification [1]. In the context of model-driven tools, behavior is usually defined in relationship to the models processed by the tool. Therefore, a minimal unit test contains a model together with some test code that feeds the model as input to the tool. A growing body of work focuses on automating the creation of such unit tests to enable a high coverage of the involved code parts and meta-models [2–4]. As a result, a test suite typically contains a large number of models, some of them specified manually, some created automatically [5].

An important concern of unit tests that has recently attracted attention is their maintainability. Based on their findings from a broad developer study, Daka and Fraser point out that *"even when automatically generated, a unit test may be integrated into the regular code base, where it needs to be manually maintained like any other code."* [6] Several issues contribute to this concern. Foremost, to fix a bug uncovered during testing, maintainers need to understand the test case eliciting the bug. Moreover, developers use tests as usage examples to understand a system's behavior. It is noteworthy that coverage and maintainability are not necessarily conflicting goals: Daka et al. found that readable unit tests can be generated without loss of coverage [7].

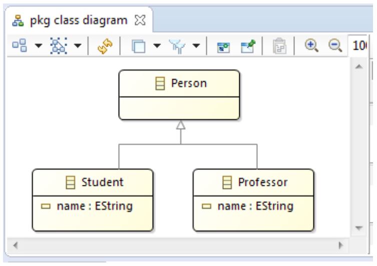
Arguably, a software artifact can only be as maintainable as the notation used to express it allows it to be [8, 9]. Therefore, in this work, we address maintainability as a function of the used specification notation. In the context of model creation for testing, developers can choose between two kinds of notation:

The first are textual model notations or APIs. At first glance, this options seems appealing as it allows developers to work with their familiar IDEs and code editors. However, during maintenance, understanding models from textual specifications can be challenging. Consider a test for the *pull up attribute* refactoring of class models (Fig. 1). In lines 2-10, a test model is created: A *package* acting as container, *classes*, and their *attributes* are created. In lines 11-12, `person` is set as common superclass for the `professor` and `student` classes. In lines 14-15, the refactoring is applied and an assertion is checked. The API used here is that of the Eclipse Modeling Framework (EMF), a modeling platform often used to create models [11]. While test cases such as this one can be generated automatically, understanding them is a necessary and time-consuming activity.

The second are graphical model notations. This option makes the benefits of visual notations available, such as the intuitive appeal to the human cognition and the use of layout to give cues [12]. In the case of unit tests, models need to

```
1  public void testRefactoring() {
2    EFactory fact = EcoreFactory.eINSTANCE;
3    EPackage pkg = fact.createEPackage();
4    EClass person = fact.createEClass(pkg);
5    EClass professor = fact.createEClass(pkg);
6    EClass student = fact.createEClass(pkg);
7    EAttribute attr1 = fact.createEAttribute(
8      "name", String.class, professor);
9    EAttribute attr2 = fact.createEAttribute(
10     "name", String.class, student);
11   professor.setSuperClass(person);
12   student.setSuperClass(person);
13
14   new PullUpRefactoring(pkg).execute();
15   assertTrue(person.getAttributes().size()==1);
16 }
```

Fig. 1: Test specification with textual API.



```
public void testRefactoring() {
  String path =
      "test1/mod/refac/pkg.ecore";
  EPackage pkg = (EPackage)
      loader.loadResource(path);
  new PullUpRefactoring(pkg)
      .execute();
  assertTrue(person.getAttributes()
      .size()==1);
}
```

Fig. 2: Test specification with a graphical notation.

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

Fig. 3: Specifying a test model using VisiText.

be viewed in the context of test code in order to be understood. However support for traceability between models and test code is widely unavailable. Maintaining models and code as separate artifacts as illustrated in Fig. 2 can be a complicated process where users need to switch repeatedly between model and code editors.

In this work, we propose a third solution that aims to offer the "best of both worlds" to developers. The key idea is to provide a notation that is visual and text-based at the same time: it uses standard ASCII characters to emulate the syntax of the modeling language at hand. This notation, called VisiText, can be used to specify models in the Javadoc comments of test methods. In the usage example in Fig. 3, boxes indicate classes. Generalization is denoted by arrows; the character `A` resembles a closed arrowhead. To specify input and output models, parameters `@InputModel` and `@OutputModel` can be used. This way, direct visibility and traceability of all models involved in the test case is established.

To support the use of such annotations as specification artifacts, the notation is machine-readable. We provide a *model compiler* that extracts models from the annotated test classes and converts them to the standard XMI format. Since this compiler is added to the IDE's build chain, it updates the XMI file automatically whenever the code is changed. The contents of the model can be referred to via a *runtime library*, using method calls such as those in lines 17–18. During test execution, this library reads the XMI file from the file system. An overview of this process is outlined in Fig. 4. Note that the only user involvement is the editing of Java files. While we currently require the user to specify the models manually, it is conceivable to generate them automatically, as we discuss later.

Benefits Our approach offers a unique combination of maintainability benefits:

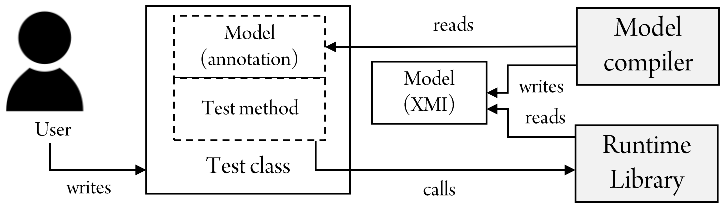


Fig. 4: VisiText: Overview

Readability. In comparison to textual notations, showing models in a two-dimensional layout, rather than as a series of statements, is a promising way to enhance their readability: The effort imposed on developers to work out things in their minds is reduced. The use of layout can give cues beyond the formal language semantics [12].

Tool independence. Visual diagrams are bound to specific tools. Requiring all current and future maintainers to use these tools can be infeasible for technical, financial, and organizational reasons. Worse, downward compatibility of these tools during evolution is not always ensured[1]. In contrast, our notation can be viewed without any tool setup. Integrating the compiler is simple: users include it into their preferred IDE by adding an entry to the build configuration. Furthermore, VisiText is not bound to a specific modeling platform. Other platforms can be supported through additional compilers.

Traceability. Models specified using visual tools are usually not directly traceable to code parts where they are used, which can lead to a complicated process when users need to view a model to understand a unit test. In turn, diagrams in our notation are directly present in the test code, thus establishing a direct traceability between models and referencing code.

Reusability. To establish a good test coverage during the unit testing of model-tools, a large set of models needs to be produced. In our experience of testing a model transformation using the default EMF tools, we found the creation of many similar, but different model diagrams highly tedious, since diagrams are maintained as separate resources with hard-wired links to the underlying models. Our notation mitigates the encountered issues since its text-based specifications can be easily copied and adapted.

Limitations Conversely, we are aware of a number of limitations of our approach. At this point, we address the two most crucial ones. The first concerns the required manual effort: clearly, using plain text editors to create diagrams as shown in Fig. 3 is infeasible; a simple action such as moving a box horizontally would require to add or remove white-space in multiple lines. To our rescue comes a relatively unknown, but highly helpful feature found in state-of-the-art

[1] An infamous example is the release of the EcoreTools 2.0 diagram editor that brought various usability improvements, but rendered EcoreTools 1.2 diagrams useless.

IDEs: in *block selection mode*[2], users can select and edit "rectangles" and "lines", which allows them to create, delete, copy, paste and move boxes and lines, the building blocks of our notation.

The seconds limitation concerns the scalability of our notation: the standard space restriction to 80–100 characters per line and the lack of zooming facilities in text editors prohibits the application of our notation to draw models with hundreds of elements. We study the effect of this limitation and suggest various mitigation strategies in our evaluation.

In our evaluation, we focus on the usefulness of our notation to specify typical test models as well as the scalability to larger models. Our overall findings indicate that our approach is suitable to specify typical test models, while the handling of large ones can be infeasible, depending on their shape. Note that we do not claim that our approach establishes optimal maintainability; our goal is to study its unique combination of maintainability trade-offs. A user study to explore these trade-offs further is left to future work and will complement our current findings.

Our notation and tool support currently support class and object diagrams. We selected these diagram types based on the following rationale: Class models are frequently reported to be the most important notation in UML [13]. Moreover, since they are used to specify meta-models, class diagrams play an important role for MDE in general. Class models are an important data structure in many different applications – for instance, code generators and quality assurance tools. Object diagrams, in turn, can be used to specify models of arbitrary modeling languages in terms of their *abstract syntax*: objects can be used to represent typed model elements, associations between objects denote links between model elements. Therefore, object diagrams allow us to establish support for arbitrary modeling languages. An interesting future challenge is to extend our notation to provide support for a larger class of concrete syntaxes.

This paper is an extended version of our earlier work [14] whereby two major parts have been added: the language description and the evaluation.

The rest of this paper is structured as follows: We present our notation and tool support in Sec. 2. We describe and discuss a preliminary evaluation in Sec. 3. We discuss related work and conclude in Secs. 4 and 5.

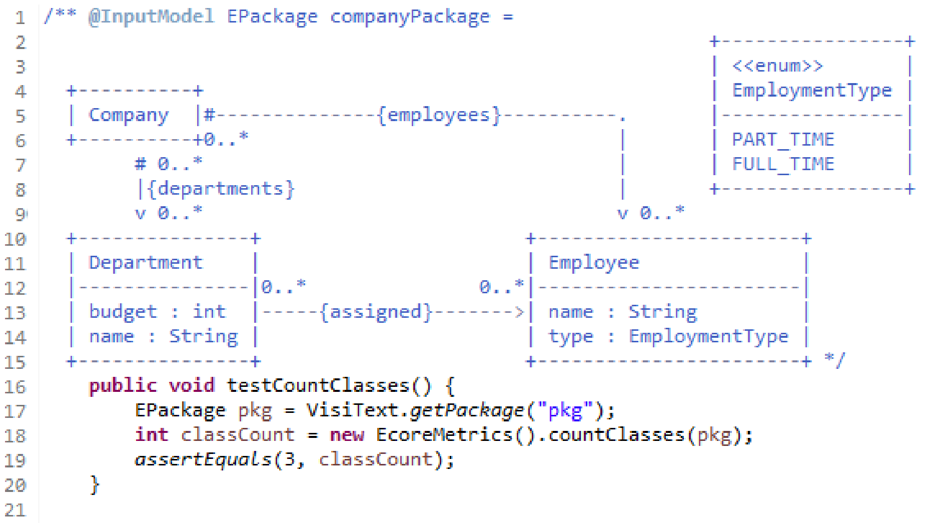
2 Language description

In this section, we present the proposed notation and our tool support to make the notation available to developers. We start with an example that illustrates its main features. Aftwards, we give a reference of the available language concepts. Finally, we give an overview of the implementation of our tool.

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

2.1 Example: Class and Object Diagrams

Our example contains an application model for a company management system and one possible instance of this model. In a testing context, such models could be used to test a model transformation or a code generator.



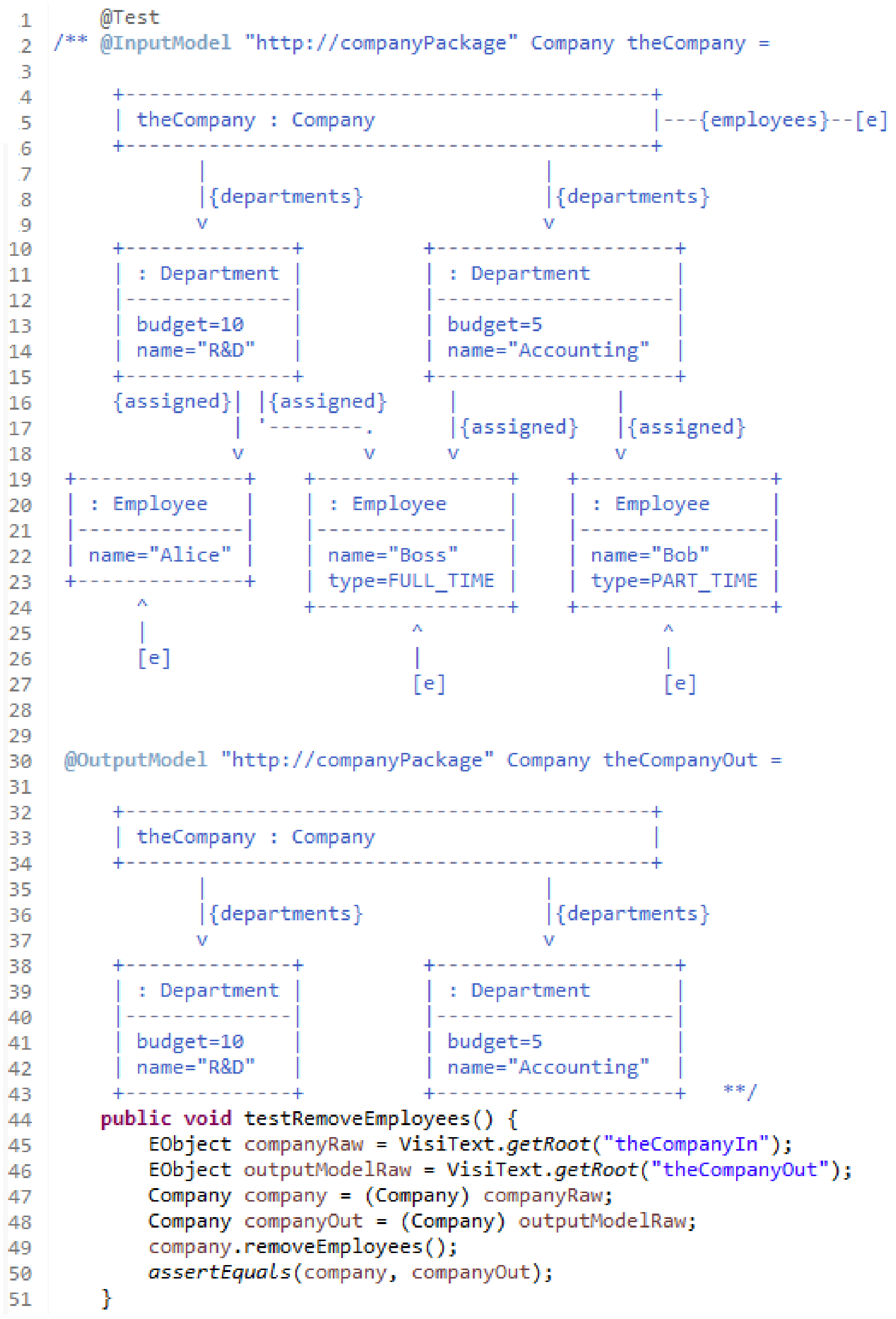
```
/** @InputModel EPackage companyPackage =
                                                          +----------------+
                                                          | <<enum>>       |
  +----------+                                            | EmploymentType |
  | Company  |#--------------{employees}----------.       |----------------|
  +----------+0..*                                |       | PART_TIME      |
        # 0..*                                    |       | FULL_TIME      |
        |{departments}                            |       +----------------+
        v 0..*                                    v 0..*
  +---------------+                       +-----------------------+
  | Department    |                       | Employee              |
  |---------------|0..*               0..*|-----------------------|
  | budget : int  |-----{assigned}-------->| name : String         |
  | name : String |                       | type : EmploymentType |
  +---------------+                       +-----------------------+ */
    public void testCountClasses() {
        EPackage pkg = VisiText.getPackage("pkg");
        int classCount = new EcoreMetrics().countClasses(pkg);
        assertEquals(3, classCount);
    }
```

Fig. 5: Example test with class diagram.

The application model, shown in Fig. 5, contains three classes: “Company”, “Department”, and “Employee”. Compositions, denoted as arrows with a dash sign, are used to assign departments and employees to a company. In addition, a plain association relates employees to departments. Each association has cardinalities and a name, denoted in curly brackets. Attributes are used to specify the budget and name of an department, and the name and type of an employee. Attributes have a name and a type, separated by a colon. There are two employment types: part-time and full-time, being specified using an enumeration.

The `@InputModel` and `@OutputModel` annotations used to specify models can have two or three parameters. The first, optional, parameter can be used to specify the namespace URI of the underlying meta-model in quotation marks. If no URI is provided, the diagram is considered a class diagram and processed using the Ecore meta-model [11]. The second parameter is the root element’s type, in this case `EPackage`. The third parameter is the root element’s name.

The company in the input model of Fig. 6 has two departments called *R&D* and *Accounting* and three employees: *Alice*, *Boss*, and *Bob*. Each model element has a (potentially empty) name and type, separated using colons. In the example, only the root element *theCompany* has a name, allowing it to be accessed from test code. To assign the employees to the company, the diagram contains *abbreviated edges*, denoted with `[e]`. Abbreviated edges are a way to



```
    @Test
/** @InputModel "http://companyPackage" Company theCompany =

     +--------------------------------------------+
     | theCompany : Company                       |---{employees}--[e]
     +--------------------------------------------+
            |                            |
            |{departments}               |{departments}
            v                            v
     +--------------+          +--------------------+
     | : Department |          | : Department       |
     |--------------|          |--------------------|
     | budget=10    |          | budget=5           |
     | name="R&D"   |          | name="Accounting"  |
     +--------------+          +--------------------+
     {assigned}| |{assigned}     |             |{assigned}
               | '--------.      |{assigned}   |{assigned}
               v          v      v             v
 +--------------+    +----------------+    +----------------+
 | : Employee   |    | : Employee     |    | : Employee     |
 |--------------|    |----------------|    |----------------|
 | name="Alice" |    | name="Boss"    |    | name="Bob"     |
 +--------------+    | type=FULL_TIME |    | type=PART_TIME |
       ^             +----------------+    +----------------+
       |                      ^                    ^
       [e]                    |                    |
                              [e]                  [e]


 @OutputModel "http://companyPackage" Company theCompanyOut =

     +--------------------------------------------+
     | theCompany : Company                       |
     +--------------------------------------------+
            |                            |
            |{departments}               |{departments}
            v                            v
     +--------------+          +--------------------+
     | : Department |          | : Department       |
     |--------------|          |--------------------|
     | budget=10    |          | budget=5           |
     | name="R&D"   |          | name="Accounting"  |
     +--------------+          +--------------------+   **/
    public void testRemoveEmployees() {
        EObject companyRaw = VisiText.getRoot("theCompanyIn");
        EObject outputModelRaw = VisiText.getRoot("theCompanyOut");
        Company company = (Company) companyRaw;
        Company companyOut = (Company) outputModelRaw;
        company.removeEmployees();
        assertEquals(company, companyOut);
    }
```

Fig. 6: Object diagrams

specify a "wormhole" in the diagram, aiming to reduce the writing and reading effort by disposing visual clutter. In the specified output model, all employees are removed, which reflects the expected behavior in this test case. In line 50, this model is compared to the one produced by the method under test. The `assertEquals()` method called here uses EMF's default method for structural equality checks, provided by the `EqualityHelper` class.

2.2 Language features

In what follows, we give a reference of available language features. Tables 1 and 2 show the available node and edge kinds for class diagrams, Table 3 shows the features available for object diagrams. For each feature, we give a small usage example and a description of the concept's correct usage. We selected these features since they represent the most important language concepts of class and object diagrams. We evaluate the usefulness of this selection in Sect. 3.

Complete coverage of class diagram features is currently not provided. Generally, additional box- and line-based features (e.g., sub-packages, interfaces, and stereotypes) are easy to include in the future. Some distinct shapes, e.g. dashed lines in the case of interface realization, appear to be more problematic.

2.3 Implementation Prototype

The two main components of our implementation are a model compiler and a runtime library. The model compiler can extract models from annotated test cases and export them as XMI files to the file system. The runtime library makes these XMI files available to be accessed from the test code. Installation of these tools is intended to be simple: the model compiler be plugged into any given IDE by adding an entry to the build configuration. The runtime library is a regular library. Instructions for plugging these tool components into Eclipse are found at `https://github.com/frieger/visitext`.

The development of these tools was straight-forward for the most part. The compiler has two main components: a scanner that collects information on objects and their relations, and a model builder using this information and EMF's model and resource APIs to create the XMI files. To read these files, the runtime library provides a set of convenience functions on top of EMF's persistence API. The main engineering effort was necessary for the development of the scanner. In what follows, we shortly describe the main idea of its implementation.

To assemble the required information, the scanner first detects all boxes, using `+` signs as cues to identify corners. The content of these boxes is read in a line-based fashion to extract information on names, types, and attributes. The scanner goes on to identify edges connected to these boxes. It follows all edges until another 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.

We initially planned to use Java annotations to specify input and output models, which proved infeasible since multi-line String literals are not supported

Feature name	Example	Description

Class	`+--------+`
	`| ClassA |`
	`+--------+`
	
	`+------------+`
	`|    ClassB  |`
	`+------------+`
	
	`+-------------+`
	`| <<abstract>> |`
	`| AClass       |`
	`+-------------+`
		A box made up of `+`, `-`, and `|` characters. For concrete classes, the class name is given in the first line. Abstract classes have the keyword `<<abstract>>` as first line, followed by the name in the second line. Can contain a section for attributes and a section for methods, separated by lines. Empty sections and their separators may be left out.

Enumeration	`+-------------+`
	`| <<enum>>    |`
	`| Enumeration |`
	`|-------------|`
	`| FIRST_LIT   |`
	`| SECOND_LIT  |`
	`| THIRD_LIT   |`
	`+-------------+`
		A box made up of `+`, `-`, and `|` characters, containing the keyword `<<enum>>` as the first line, a name as the second line, a line separator as the third line, and a list of literals (one per line).

Attribute	`+-------------------------------+`
	`| ClassA                        |`
	`|-------------------------------|`
	`| + publicAttribute : String    |`
	`| - privateAttribute : String   |`
	`| packagePrivateAttribute : String |`
	`+-------------------------------+`
		An entry in the attribute section of a class; consists of a name, followed by a colon and a type. Primitive types, String and Enumerations are supported. Object types have to be modeled as references. The characters `+`, `-`, or no character before the name denotes the visibility as public, private, or package-private.

Method	`+-----------------------+`
	`| ClassA                |`
	`|-----------------------|`
	`|-----------------------|`
	`| foo(x : int) : String |`
	`| bar(y : String)       |`
	`| baz() : Float         |`
	`+-----------------------+`
		An entry in the method section of a box; consists of the method name, method parameters in parentheses, a colon, and the return type. Multiple parameters and exactly one return type are supported.

Table 1: Class diagrams: Nodes and sections.

in Java. Instead, as shown in the previous usage examples, we embed models in the Javadoc of test methods. Conceptually, Javadoc is a suitable location for this purpose since input and output models are part of a unit test's documentation.

3 Evaluation

In this section, we present a preliminary evaluation of our approach. The goal of this evaluation is twofold: First, we intend to assess the general usefulness of

Feature name	Example	Description

Generalization	`+-------------+`
	`| Superclass |`
	`+-------------+`
	`      A`
	`      |`
	`+-------------+`
	`| Subclass   |`
	`+-------------+`
		An arrow between two class boxes, from the subclass to the superclass. Currently needs to be pointed upwards.

Association	`+----------+ 0..*          1 +----------+`
	`|  Person  |----{hasAddress}--->| Address  |`
	`+----------+                  +----------+`
	
	`+-----------+ *          * +----------+`
	`|  Company  |----{customer}-----| Customer |`
	`+-----------+              +----------+`
	
	`          .---------.`
	`          v1        |{parent}`
	`      +--------+ *  |`
	`     1| Node   |----'`
	`.---->|        |<-----.`
	`|     +--------+ 1..1 |`
	`|   0..1|   |0..1     |`
	`| {left}|   |{right} |`
	`'-------'   '--------'`
		A line between two class boxes, with a name enclosed in {} and multiplicities at each line end. Directed associations have an arrowhead of the form `<`, `>`, `^` or `v`. Multiplicity can be given in the form `x..y` or `1` or `*`, placed directly at the end of an association.

Aggregation (composite and shared)	`+---------+`
	`| ClassA |`
	`+---------+`
	`     # *`
	`     |{contents}`
	`     v *`
	`+---------+`
	`| ClassB |`
	`+---------+`
	
	`+---------+`
	`| ClassA |`
	`+---------+`
	`     @ *`
	`     |{shared}`
	`     | *`
	`+---------+`
	`| ClassB |`
	`+---------+`
		A line between two class boxes, with the end of the containing class replaced by `#` for a composite or `@` for a shared aggregation. Shared aggregations are only available for UML.

Table 2: Class diagrams: edges.

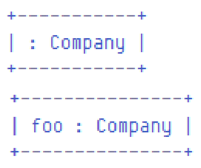
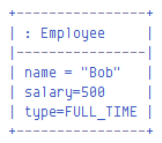
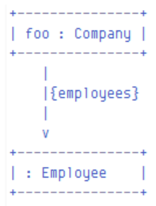
the proposed notation in the context of realistic test models. Second, we aim to explore the scalability issue, which we highlighted to be crucial due to the limited amount of space and lacking zooming capabilities. We studied the following two research questions:

- **RQ1**: How useful is our approach to specify test models in real test suites?
- **RQ2**: How well does our approach scale up to different model sizes?

We evaluated these research questions in the domain of class models, a selection that is justified at the start of Sect. 2.

3.1 Subjects

We used different types of subject models to address RQ1 and RQ2. For RQ1, to study usefulness in a realistic context, we considered existing test models. In

Feature name	Example	Description/Notes
Objects		A box containing the object name, followed by a colon, followed by the object type, with + in all 4 corners. May contain a section for attribute values, separated by a horizontal line. The name can be used to refer to this object from other parts of the model or for documentation.
Attributes		The name of the attribute, followed by =, followed by the value. The value can be a String, a number, a boolean or an enumeration literal
Reference		An arrow between two objects, with the name of the reference enclosed in . Only set semantics are supported. The order of elements in the reference is unspecified.

```
+-----------+
| : Company |
+-----------+

+---------------+
| foo : Company |
+---------------+
```

```
+---------------+
| : Employee    |
|---------------|
| name = "Bob"  |
| salary=500    |
| type=FULL_TIME |
+---------------+
```

```
+---------------+
| foo : Company |
+---------------+
     |
     |{employees}
     |
     v
+---------------+
| : Employee    |
+---------------+
```

Table 3: Objects diagrams.

the case of RQ2, to explore the scalability of our approach, we obtained class models of varying size from an online repository. The models for RQ2 represent a broader scope of contexts and purposes than just testing.

We obtained the test models for RQ1 from an existing code base. The models were taken from EMF Refactor [15], an Eclipse incubation project familiar to the authors (convenience sampling). EMF Refactor is a tool environment aiming to enable a structured quality assurance process for models. Its main components are a metrics computation tool, a smell detection tool, and a model refactoring tool. We retrieved two test suites from the metrics component, one of them constituted by UML class models, the other by Ecore class models.

In total, we obtained 11 Ecore and 26 UML class models. Table 4 gives an overview of these models. Each row represents a set of test models for one specific metric in EMF Refactor. The ID denotes the targeted metric, for instance, *nassc* for *number of associations.* Columns provide size information for the included models. We computed model size using the metric proposed by Störrle [16], counting the number of visual elements – boxes, arrows, and member labels – in the diagram. This simple metric is simple to compute on the one hand, while potentially exhibiting a strong correlation to more sophisticated metrics on the

Group	ID	#	Cl.	At.	Op.	As.	Gen.	Size
Ecore	haggec	2	3	0	0	2	0	5
	maxidtec	3	2.3	0	0	0	1.3	3.7
	dummy	3	2	3.3	0	0	0	5.3
	nepec	3	1	0	3	0	0	4
UML	cbc	4	1.8	0.5	0	0.5	0	2.8
	dnh	1	0	0	0	0	0	0
	dummy	1	1	3	0	0	0	4
	hagg	3	2.3	0	0	1.3	0	3.7
	maxditc	3	2.3	0	0	0	1.3	3.7
	nassc	2	1	0	0	0.5	0	1.5
	neatc	2	2	1.5	0	0	0	3.5
	neipo	3	1.7	0	2	0	0	3.7
	nih	2	3	0	0	0	1	4
	nsupc2	3	2.3	0	0	0	1.3	3.7
	tncp	2	1	0	0	0	0	1

Table 4: Test models used for RQ1. Each row represents a set of class models. Columns give an ID for each set, the number of models included in the set (#), their Average Number of Classes (Cl.), Attributes (At.), Operations (Op.), Associations (As.) and Generalizations (Gen.) and Average Size.

other hand [16]. In accordance with our earlier conjecture that unit test models tend to be of limited size, no model in the test suite exceeded a size of 7.

To study RQ2, scalability, we sampled class models from a comprehensive online repository of class models [17]. The repository comprises 810 class diagrams from experiments, development projects, and web resources. Our selection proceeded in two steps: First, we visually inspected 100 randomly selected models for potential scalability issues. In this process, we found that many of included models are of moderate size and did not demonstrate any specific scalability issues. In addition, we noticed that the larger models in the repository occurred in different shapes with different implications for the scalability of our approach. We found five shapes to be problematic:

– *"Panorama" Layout.* Aligning a large number of elements horizontally, thereby creating a "panorama" layout, is detrimental to meeting the horizontal space restriction in code formatting guidelines.
– *Long Member Names.* Since the width of an element is determined by the length of its longest member, layouting is inherently more difficult for models with long member names; their elements use up more horizontal space.
– *High Member-to-Class-Ratio.* The height of an element is determined by the number of its members. Despite the absence of a vertical space restriction, a high vertical extension of classes might still complicate the layouting process.

– *High Edge-to-Class Ratio.* Association arrows are usually labelled. Thus, maintaining a high number of association and generalization edges between classes may lead to complications with intersections and label alignment.
– *Large Network of Sparse Classes.* An interesting edge case is the one where a large network of sparse classes, classes with few or no members, is maintained. In this scenario, the layouting of edges may be challenging.

On the basis of these observations, we selected ten class models in total for our experiment: five deemed as "typical" and five deemed as "large" (see Table 5). The typical models were chosen as representative for average-sized class models not showing any particular scalability issues. The large models were selected to exhibit one of the problematic shapes each. This selection allowed us to explore issues for typical cases as well as edge cases in our experiments.

To ensure that the typical class models were indeed representative, we compared their size to the repository average, using the size metric from [16]. We found that the selected models, ranging between a size of 44 to 59, resembled the repository average of 48.4. Interestingly, the repository average and all typical models were in the preferable size scope for developer efficiency in comprehension tasks: according to [16], this scope ranges from 20 to 60.

Our selection was further informed by two criteria. First, we excluded class models with features not supported by our prototypical implementation: interfaces, sub-packages, and stereotypes. Second, to avoid impairing the specification process by language issues, we only used class models with English labels.

Group	**ID**	**Cl.**	**At.**	**Op.**	**As.**	**Gen.**	**Size**	**Remark**
Typical	**433**	8	15	13	2	6	44	
	564	8	22	21	2	5	59	
	159	8	35	0	5	2	50	
	262	9	35	5	8	2	58	
	794	12	19	11	6	6	54	
Large	**467**	7	84	22	7	3	123	High member-to-class ratio.
	616	10	49	45	9	22	135	High edge-to-class ratio.
	276	13	43	0	27	2	85	Long member names.
	108	29	111	0	25	3	168	Panorama layout.
	611	45	0	0	0	50	95	Large network of sparse classes.
Full Repository (n=810)	∅	10.1 ±6.4	14.4 ±20.4	14.5 ±24.3	5.7 ±5.8	3.7 ±5.0	48.4	

Table 5: Test models used for RQ2. Each of the first ten rows represents one sample model, given by its ID. The last row presents the repository average, ± denoting the standard deviation. Columns give the Number of Classes (Cl.), Attributes (At.), Operations (Op.), Associations (As.) and Generalizations (Gen.), Size and Remarks.

3.2 Set-Up

To address RQ1 and RQ2 in a qualitative and quantitative manner, we refined these questions into two sub-questions each:

- **RQ1.1**: What is the impact of our approach on the complexity of the overall test suite?
- **RQ1.2**: What is the coverage of modeling language elements included in the models?

We quantified complexity in RQ1.1 by measuring the maximum number of nested directories contained in the test suite. A high nesting depth entails an increased effort when tracing test models to test code.

For RQ1.2, we studied the coverage of language features by counting the number of test models supported by our approach.

- **RQ2.1**: What is the effect of the horizontal space restriction?
- **RQ2.2**: How much effort does the approach entail during the editing process?

To study RQ2.1, we devised each specification with the goal to keep the horizontal space limitation of 100 characters while retaining a layout closely resembling the original one. If doing so was not possible, we loosened the limitation to 120 characters. If this was still insufficient, we tried to rearrange the diagram to fit into this limitation. When this was not possible, we ignored the space limitation altogether. We measured the used amount of space in each example.

We quantified effort in RQ2.2 by measuring the time required to create each specification and tracking the perceived effort of included sub-tasks. For all sub-questions, we also gathered quantitative evidence by documenting our experiences after performing each task.

3.3 Results and Discussion

Following a pre-study to compare the applicability two kinds of editors (detailed in our later discussion), we decided to create all test models using the Eclipse IDE with its included *block selection mode* feature. We used the notation introduced in Sect. 2. All test models created for our experiments are provided online[3].

RQ1.1 We compared the complexity in terms of the maximum number of nested directories in the tests specifications.

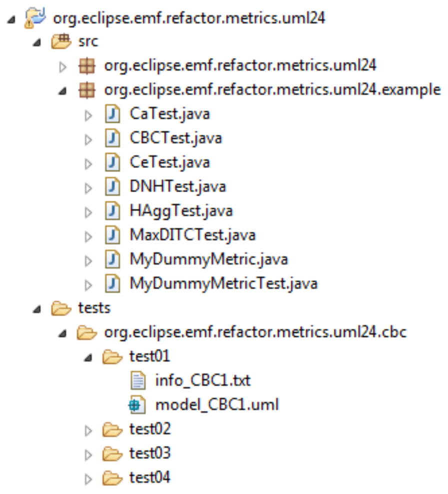
Ts illustrated in Fig. 7, the original test suite contains a Java directory and a directory for test artifacts. The Java directory has one class per test group, containing several tests. The model directory is comprised of three levels of nesting: A directory for each test group, comprising a directory for each test, comprising a pair of the test model and a text file with additional information each. In sum, the maximum nesting depth in the original specification was 3.

[3] https://frieger.github.io/visitext

Our specification comprises just the Java source directory. All test models and information relevant for the test specification are directly embedded in the source code. This results in one Java class per test group, containing several tests, their test models and test specifications. In sum, the maximum nesting depth in our specification was 1.

Consequently, our approach reduces the complexity of the test project structure considerably. Consider Fig. 8 for an illustration of the simplified structure. In the sunburst charts, innermost rings represent the top-level directory, whereas each additional ring represents a separate level of directory nesting.

Using our approach, the complexity of existing test suites in terms of artifact nesting could be considerably reduced.



```
org.eclipse.emf.refactor.metrics.uml24
  src
    org.eclipse.emf.refactor.metrics.uml24
    org.eclipse.emf.refactor.metrics.uml24.example
      CaTest.java
      CBCTest.java
      CeTest.java
      DNHTest.java
      HAggTest.java
      MaxDITCTest.java
      MyDummyMetric.java
      MyDummyMetricTest.java
  tests
    org.eclipse.emf.refactor.metrics.uml24.cbc
      test01
        info_CBC1.txt
        model_CBC1.uml
      test02
      test03
      test04
```

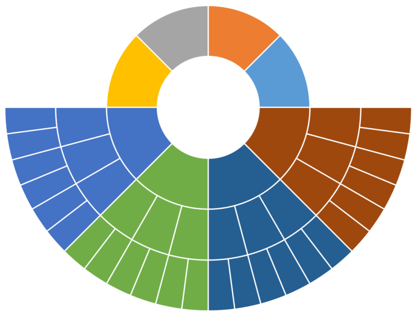
Fig. 7: Directory structure in the original test suite.

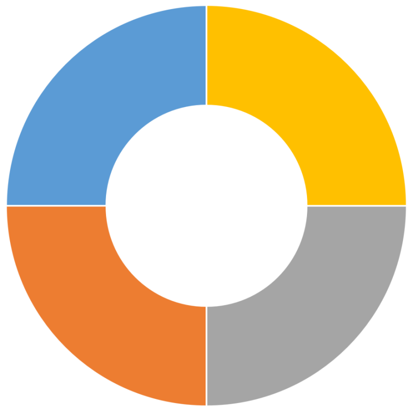
RQ1.2 We studied coverage by determining the percentage of supported test models from each test suite.

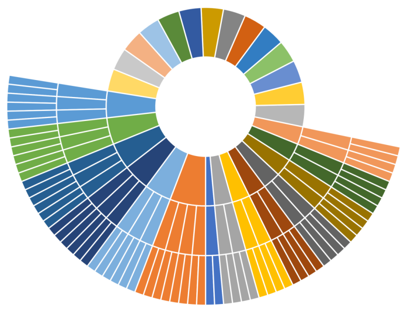
Our implementation prototype supported all 12 tests based on Ecore. In terms of modeling languages feature, these test models are constituted by classes, attributes, operations, associations and generalizations.

Our prototype was able to support 26 of the 31 UML test models. Five models are unsupported because they use nested packages, a concept currently not supported by our visual syntax. In addition, one model relies on datatypes from external models, which we do not support as well. Two models did not contain any elements except for their container package, which is supported.

In summary, the models contained in the test suited were generally minimalistic, with very few modeling language elements used. The models in the test


Existing project structure of Ecore metrics tests.


Project structure of Ecore metrics tests using our approach.


Existing project structure of UML metrics tests.

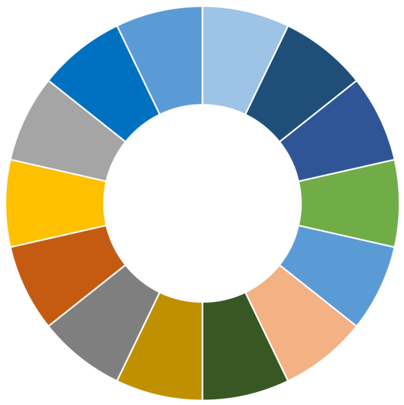
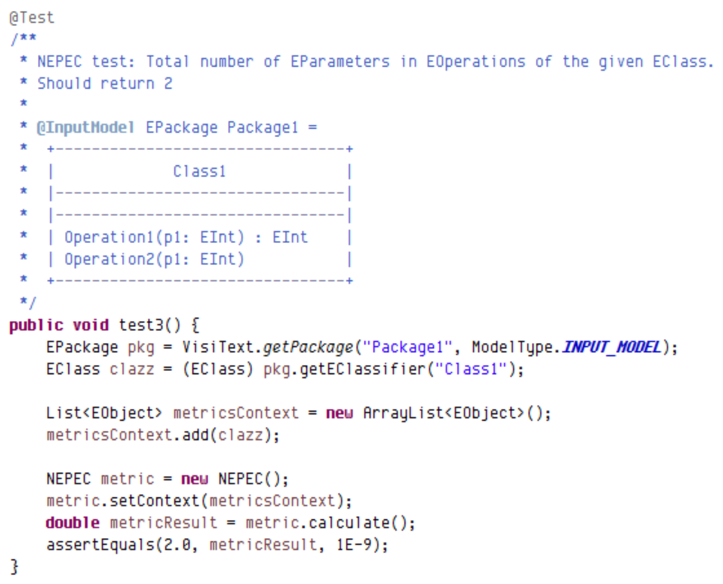

Project structure of UML metrics tests using our approach.

Fig. 8: Sunburst charts visualizing the test specifications.

suite do not contain multiplicities other than 1, associations and references are always unidirectional, and there are no compositions.

Fig. 9 shows a test case specified using our approach. The test model, comprising a class with two operations, is provided in the Javadoc comment of the test method.

Our approach was suitable to capture realistic test models that feature a limited set of language concepts.



```java
@Test
/**
 * NEPEC test: Total number of EParameters in EOperations of the given EClass.
 * Should return 2
 *
 * @InputModel EPackage Package1 =
 *  +--------------------------------+
 *  |             Class1             |
 *  |--------------------------------|
 *  |--------------------------------|
 *  | Operation1(p1: EInt) : EInt    |
 *  | Operation2(p1: EInt)           |
 *  +--------------------------------+
 */
public void test3() {
    EPackage pkg = VisiText.getPackage("Package1", ModelType.INPUT_MODEL);
    EClass clazz = (EClass) pkg.getEClassifier("Class1");

    List<EObject> metricsContext = new ArrayList<EObject>();
    metricsContext.add(clazz);

    NEPEC metric = new NEPEC();
    metric.setContext(metricsContext);
    double metricResult = metric.calculate();
    assertEquals(2.0, metricResult, 1E-9);
}
```

Fig. 9: Using our approach to specify a test case

RQ2.1: We studied the impact of the horizontal space restriction on the specification of models. Details are provided in Table 6.

The typical models fit well into the 100 or 120 character limit. At around 40 lines each, they are all of comparable vertical extent. We had to make slight changes to the layout of model #564, where generalization arrows were not all directed upwards. Our current implementation assumes a minimalistic alphabet of one-character arrowheads; hence generalization, indicated by the `A` character, is generally directed upwards.

The large models posed various problems. We could only fit one of them in 100 characters per line. Another one could be layouted to take up less than 120 characters per line. The remaining three took 154 to 168 characters of vertical dimension. In what follows, we describe our experience of deriving specifications for all considered large diagrams.

– Diagram #467 has a *high member-to-class ratio*. It is likely to be reverse-engineered from an existing code base. Each member requires a separate line, which results in a large vertical extent. The resulting diagram is 103 lines high, taking up multiple screens.
– Diagram #616 has *long member names*. Layouting this diagram was straightforward. However, we could not fulfill our line length constraints due to the very long member names. The diagram is 158 characters wide and 98 lines long, spanning multiple screens horizontally and vertically.
– Diagram #277 has a *high edge-to-class ratio*. This diagram was hard to layout. Overlapping elements are a problem with purely text-based notation.

Group	**ID**	**Longest line**	**Number of lines**	**Total editing time (min.)**
Typical	433	100	36	11
	564	95	39	18
	159	99	32	13
	262	83	37	16
	794	116	38	21
Large	467	100	103	33
	616	158	98	26
	277	154	52	46
	108	168	96	55
	611	119	48	80

Table 6: Results for RQ2

Hence, in order to fit many edges between classes, the classes themselves need to be drawn in larger boxes, increasing the overall extent of the diagram. Our diagram takes up 154 characters horizontally and is 52 lines high.

– Diagram #108 has a *panorama layout*. We had to re-layout this diagram. For this, we identified one class that connected two otherwise unconnected parts of the diagram. We placed this class in the centre of the diagram and arranged the two parts above and below this class. Nonetheless, the resulting diagram is 168 characters wide and 96 lines high. However, since it is essentially two diagrams connected by a single class in the middle, each of the two parts fits on roughly one vertical screen.
– Diagram #611 consists of a *large network of sparse classes*. This diagram was practically impossible to layout. The original diagram has many overlapping and non-straight edges. We managed to fit it into 120 characters and 48 lines. However, this took far more effort than any of the other diagrams. The resulting diagram contains multiple closely-spaced parallel edges, making it very hard to read.

Our approach scaled up to class models of average size, while showing severe limitations when used to specify larger models.

RQ2.2: We studied the effort imposed by our notation by measuring the time it took a single developer to specify each example model. These times are given in Table 6.

We could create the typical models quickly, taking around 15 minutes for each. The large diagrams from our example data set took considerably longer, ranging from 26 to 80 minutes. Diagrams with many edges, such as #277 and #611, were especially hard to layout, requiring considerable time and effort. The layouting process entailed the tasks of entering data (labels), drawing boxes, and layouting (including the drawing of edges). To study the effort for these tasks separately, we performed them in isolation for models #616 and #277. For the other diagrams, data entry, drawing boxes, and layouting were integrated.

Entering the label data and drawing boxes for model #277 took 13 minutes, yet layouting and edge drawing took 33 minutes. The reason for this exceptionally high layouting effort is a very compact layout of the original diagram. The original diagram spaces many edges close to each other. Fig. 10 illustrates one of the occurrences that required complicated re-layouting, thus increasing time needed for layouting. Diagram #611 consists of a large number of members per class, but only few classes and associations. Data entry and box-drawing took 21 minutes, while layouting and drawing edges only took another 6 minutes.

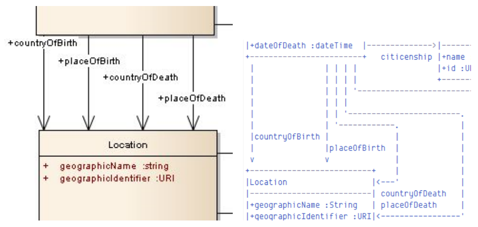


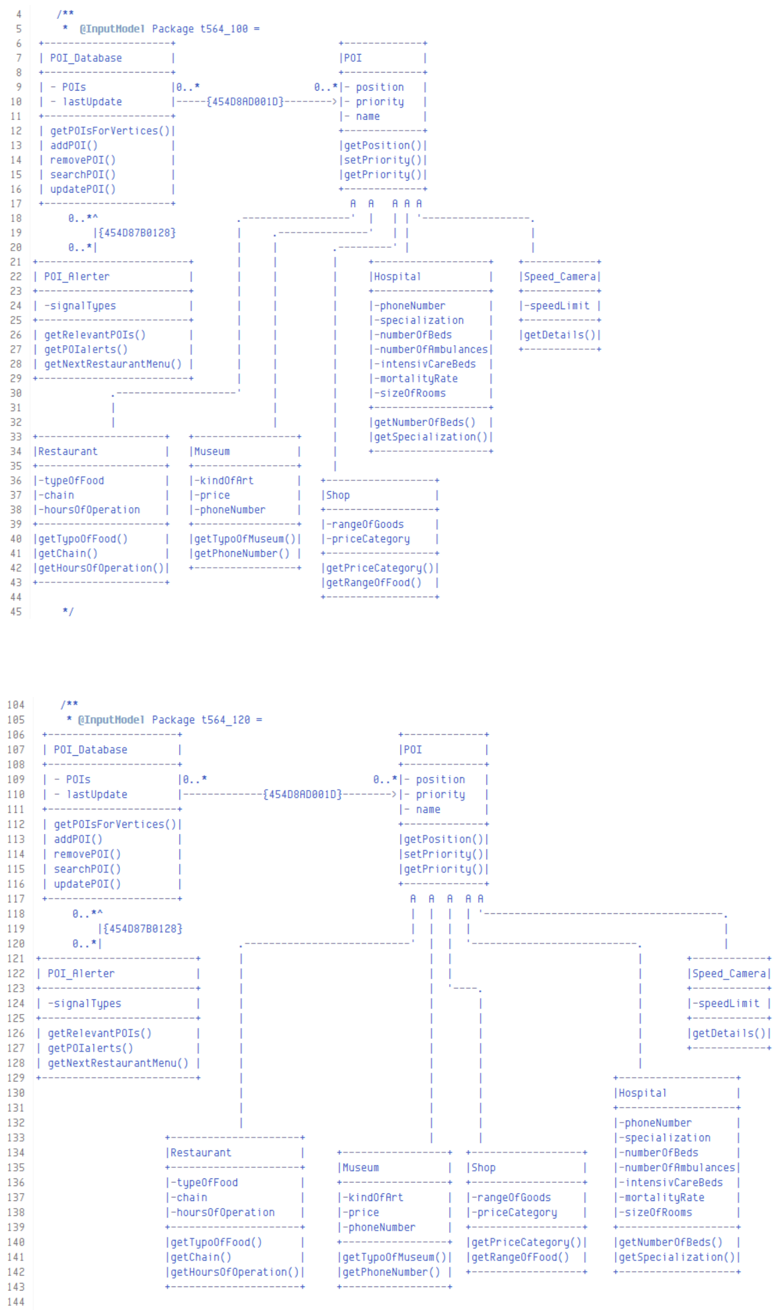
Fig. 10: Problematic group of edge agglomeration.

The effort to specify average-size class models was reasonable, while the specification of larger models partly showed to be impractical.

3.4 Discussion

Usefulness. Our approach was very suitable for expressing unit tests for computing metrics on models. We argue that it can be generalized for a larger variety of application domains where models for unit testing also tend to be small. Since our approach allows to specify a pair of input and output models, it may be especially suitable for test models for model refactorings. In addition, reusing and slightly modifying model specifications is very easy with our approach. This makes it also suitable for testing model queries, where there might be many similar input models.

Limitations for larger models. Horizontal space restrictions restrict the amount of unused space available in a diagram. This space is often used to visually group related elements, which is impossible if there is little space. Existing groupings may also be prone to be destroyed by re-layouting the diagram in order to fit into existing space constraints, leading to visual clutter. Fig. 11 shows an example of the impact of different maximum line lengths on the layout. The specification within 100 characters gives a slightly more crowded impression, while the one of larger horizontal extent reflects the grouping of sub-clases more consequently.



```
    /**
     *  @InputModel Package t564_100 =
 +---------------------+                           +-------------+
 | POI_Database        |                           |POI          |
 +---------------------+                           +-------------+
 | - POIs              |0..*                   0..*|- position   |
 | - lastUpdate        |-----{454D8AD001D}-------->|- priority   |
 +---------------------+                           |- name       |
 | getPOIsForVertices()|                           +-------------+
 | addPOI()            |                           |getPosition()|
 | removePOI()         |                           |setPriority()|
 | searchPOI()         |                           |getPriority()|
 | updatePOI()         |                           +-------------+
 +---------------------+                             A  A   A A A
      0..*^                       .------------------'  |   | | '------------------.
          |{454D87B0128}          |     .---------------'   | |                    |
      0..*|                       |     |         .---------' |                    |
+-------------------------+       |     |         |     +-------------------+    +------------+
| POI_Alerter              |       |     |         |     |Hospital           |    |Speed_Camera|
+-------------------------+       |     |         |     +-------------------+    +------------+
| -signalTypes            |       |     |         |     |-phoneNumber       |    |-speedLimit |
+-------------------------+       |     |         |     |-specialization    |    +------------+
| getRelevantPOIs()       |       |     |         |     |-numberOfBeds      |    |getDetails()|
| getPOIalerts()          |       |     |         |     |-numberOfAmbulances|    +------------+
| getNextRestaurantMenu() |       |     |         |     |-intensivCareBeds  |
+-------------------------+       |     |         |     |-mortalityRate     |
             .--------------------'     |         |     |-sizeOfRooms       |
             |                          |         |     +-------------------+
             |                          |         |     |getNumberOfBeds()  |
+---------------------+   +-----------------+     |     |getSpecialization()|
|Restaurant           |   |Museum           |     |     +-------------------+
+---------------------+   +-----------------+     |
|-typeOfFood          |   |-kindOfArt       |   +------------------+
|-chain               |   |-price           |   |Shop              |
|-hoursOfOperation    |   |-phoneNumber     |   +------------------+
+---------------------+   +-----------------+   |-rangeOfGoods     |
|getTypoOfFood()      |   |getTypoOfMuseum()|   |-priceCategory    |
|getChain()           |   |getPhoneNumber() |   +------------------+
|getHoursOfOperation()|   +-----------------+   |getPriceCategory()|
+---------------------+                         |getRangeOfFood()  |
                                                +------------------+

     */
```

```
    /**
     * @InputModel Package t564_120 =
 +---------------------+                                  +-------------+
 | POI_Database        |                                  |POI          |
 +---------------------+                                  +-------------+
 | - POIs              |0..*                          0..*|- position   |
 | - lastUpdate        |-------------{454D8AD001D}------->|- priority   |
 +---------------------+                                  |- name       |
 | getPOIsForVertices()|                                  +-------------+
 | addPOI()            |                                  |getPosition()|
 | removePOI()         |                                  |setPriority()|
 | searchPOI()         |                                  |getPriority()|
 | updatePOI()         |                                  +-------------+
 +---------------------+                                    A  A  A  A A
      0..*^                                                 |  |  |  | '---------------------------------------.
          |{454D87B0128}                                    |  |  |  |                                         |
      0..*|                      .--------------------------'  |  |  '---------------------------.             |
+-------------------------+      |                             |  |                              |       +------------+
| POI_Alerter              |      |                             |  |                              |       |Speed_Camera|
+-------------------------+      |                             |  '----.                         |       +------------+
| -signalTypes            |      |                             |       |                         |       |-speedLimit |
+-------------------------+      |                             |       |                         |       +------------+
| getRelevantPOIs()       |      |                             |       |                         |       |getDetails()|
| getPOIalerts()          |      |                             |       |                         |       +------------+
| getNextRestaurantMenu() |      |                             |       |                         |
+-------------------------+      |                             |       |                     +-------------------+
                                 |                             |       |                     |Hospital           |
                                 |                             |       |                     +-------------------+
                                 |                             |       |                     |-phoneNumber       |
                     +---------------------+                   |       |                     |-specialization    |
                     |Restaurant           |    +-----------------+  +------------------+    |-numberOfBeds      |
                     +---------------------+    |Museum           |  |Shop              |    |-numberOfAmbulances|
                     |-typeOfFood          |    +-----------------+  +------------------+    |-intensivCareBeds  |
                     |-chain               |    |-kindOfArt       |  |-rangeOfGoods     |    |-mortalityRate     |
                     |-hoursOfOperation    |    |-price           |  |-priceCategory    |    |-sizeOfRooms       |
                     +---------------------+    |-phoneNumber     |  +------------------+    +-------------------+
                     |getTypoOfFood()      |    +-----------------+  |getPriceCategory()|    |getNumberOfBeds()  |
                     |getChain()           |    |getTypoOfMuseum()|  |getRangeOfFood()  |    |getSpecialization()|
                     |getHoursOfOperation()|    |getPhoneNumber() |  +------------------+    +-------------------+
                     +---------------------+    +-----------------+
```

Fig. 11: Textual test model specification with 100 characters per line (top) and 120 characters per line (bottom)

Graphical editors can overlay elements on top of each other. In text, each position is used by exactly one character. Thus, our approach only allows very limited overlapping of elements. In our implementation, only edges can cross. This makes diagrams that make heavy use of overlapping elements difficult to layout. Long names of members or associations amplify this problem: edges need to navigate around unrelated association labels. This will result in more changes of direction, making edges hard to follow.

Layouting diagrams with a large number of edges is challenging. Each character in text is separated by a small amount of space to either side. This makes all edges appear slightly discontinuous. Edges are not an uninterrupted line, making them harder to follow. This might become a problem when very long edges or a large number of edges are used in a diagram. Figure 13 provides an example. Since edges and box outlines share the same characters in our representation, closely spaced edges and boxes will make the diagram hard to follow. Alternatively, the layout can be stretched horizontally, requiring longer line lengths.

```
     +------------+
     | :School    |#--{teachers}--[c]
     +------------+
+----------------+          +-------------+
| :Teacher [n=9] |   [c]--->| :Teacher    |
|----------------|          |-------------|
| name = "Mary"  |<---[c]   | name = "Ed" |
+----------------+          +-------------+
```

Fig. 12: Model with multi-node and abbreviated multi-edge.

Mitigation strategies. The space limitation can be addressed by two kinds of mitigation strategies. The first is to augment the notation to increase its compactness. The second is to provide a mechanism to compose a larger model from smaller ones.

To compact multiple links of the same type, edges can be multi-edges in our notation, i.e., have multiple source or targets. To further increase compactness, we have introduced *abbreviated edges*, a concept inspired by net labels in ECAD software[4]. Another option to increase notational compactness would be multiple objects of the same type by using *node multiplicity*, indicated by the character `n` in Fig. 12. The example model in Fig. 12 shows a school with ten teachers, nine of them named *Mary*, one named *Ed*.

Composing a larger model from smaller ones could be done in two ways: First, a visual base model can be extended by adding model elements programmatically. This option is already supported since models can be modified using our API. Second, a facility to split models over several fragments can be introduced. This option is left to future work.

[4] e.g., KiCad, `techdocs.altium.com/display/ADOH/Connectivity+and+Multi-Sheet+Design`. Retrieved on 2016-09-09.

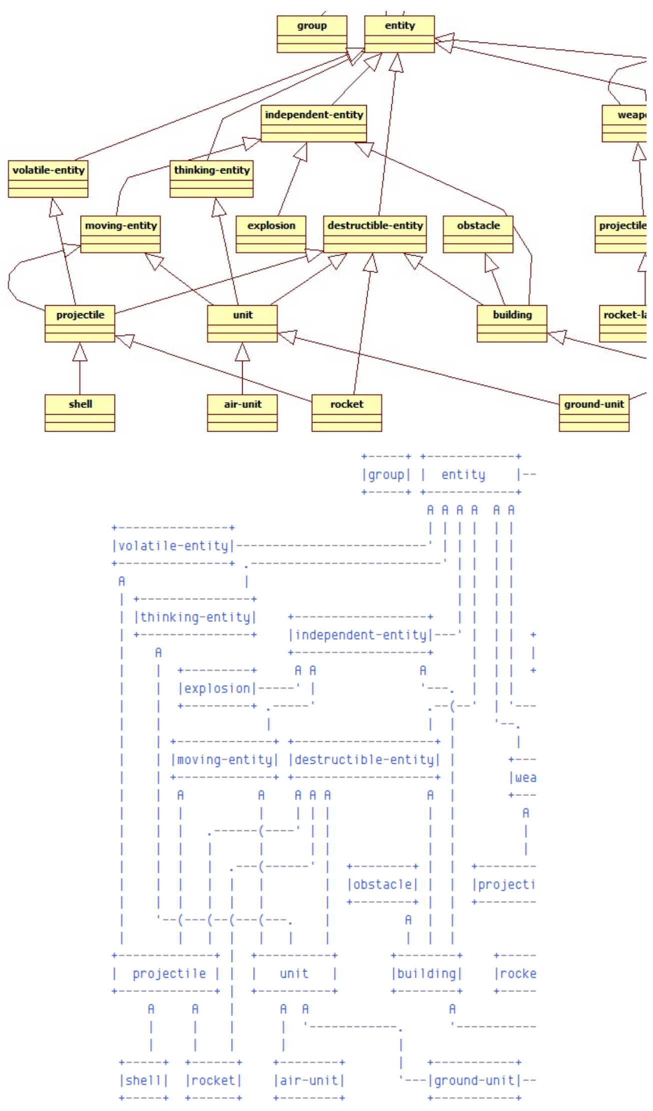
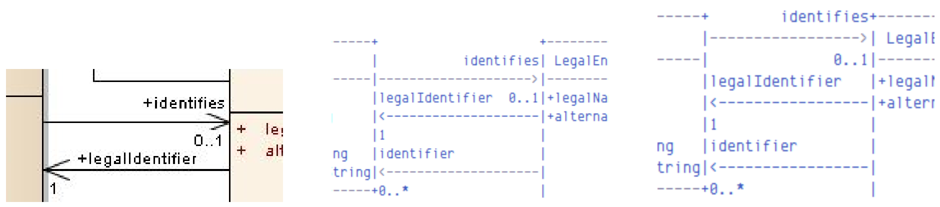


Fig. 13: Visual clutter caused by a large number of closely-spaced edges

Layout-sensitive semantics. Usually, graphical model editors provide a graphical representation of an underlying model. Conversely, in our approach, models are created from an ASCII-based specification. This leads to a number of unique challenges. Our approach relies on the proximity of elements to provide context. Role names of associations are specified by writing them directly adjacent to the association. If the name is written close to the source, it will be the role name of the source. Conversely, if it is written close to the target, it will be the role name of the target. Graphical editors allow moving labels arbitrarily if they are placed in an inconvenient location. Our approach requires proximity cues for parsing, so this is not supported. There are various substructures in existing models that are problematic when it comes to expressing them using our notation. Fig. 14 illustrates this problem. In the original diagram, role names were moved away from their association end in order to fit more associations in a small area. Fitting many associations in a small area exposes another pitfall of text-based notation. Edges can accidentally capture neighboring edges' multiplicities and role names if they are spaced to close to each other. Fig. 10 gives an example of a diagram where we encountered this problem.



graphical diagram

```
-----+                    +--------
     |          identifies| LegalEn
-----|------------------->|--------
     |legalIdentifier  0..1|+legalNa
     |<-------------------|+alterna
     |1                   |
ng   |identifier          |
tring|<-------------------|
-----+0..*                |
```

ambiguousity: `legalIdentifier` and `0..1` could belong to multiple edges

```
-----+        identifies+-------
     |----------------->| LegalE
-----|              0..1|-------
     |legalIdentifier   |+legalN
     |<-----------------|+altern
     |1                 |
ng   |identifier        |
tring|<-----------------|
-----+0..*              |
```

unambiguous solution

Fig. 14: Ambiguous specification

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.

3.5 Threats to Validity

External validity. We only considered models from two actual test suites. In our discussion, we discussed different project kinds that might benefit from the treatment proposed in our approach. Still, a larger variety of samples is required to justify extensive generality claims. Furthermore, our experiences and time measurements regarding specification effort are based on one developer and might not generalize to a larger population. In our experiments, we focused on providing preliminary evidence for capabilities and limitations of our approach, using an evaluation setup with a high internal validity. A complementary study maximizing external validity is intended as future work. Finally, we only considered class models with English identifiers. The drawbacks related to the vertical space limitations might be more relevant in languages with larger average word length.

Internal validity. Selection bias is one of the most important threats to internal validity. Since we identified the scalability issues in larger models manually and based on random samples from the repository, we cannot guarantee that the results represent the most important issues in the total population. Still, our setup allowed us to identify potential caveats and also to consider typical cases as constituted by average models from the repository.

Construct validity. Our operationalization of usefulness focuses on limitations during the creation and editing of test models. We did not study complementary facets of usefulness, such as the readability of the created diagrams. Since they employ a different aesthetics than the one familiar to developers, the created diagrams may impose an increased reading effort. To mitigate this effect, we studied if the approach is suited to capture the layout of the original diagrams. Learnability is an additional aspect not studied; the proposed notation might not be intuitive to other developers. We intend to assess the effect of our approach on readability and learnability in the future.

4 Related Work

4.1 Testing model-driven tools and artifacts

OMG's Human-Usable Textual Notation (HUTN) [18] is a textual notation for instances of MOF-based meta-models. The construction of models for tests has been pointed out as a key motivation for HUTN's implementation in [19]. Similar to our object diagram notation, HUTN provides a generic concrete syntax that can be used to support a broad range of modeling languages. HUTN's syntax is structured around the containment tree of the specified model, which is suitable to specify models with a limited number of cross-references. Conversely, our notation is particularly suitable for models where cross-references play an important role, that is, models that are "graph-like" rather than "tree-like".

A holistic approach to unit testing for model management tasks is provided by EUnit [19]. EUnit provides a custom testing framework based on the Epsilon Object Language. The involved models can be specified using a broad variety of notations, including HUTN, and be mapped to and parametrized for particular test cases. Test cases can be orchestrated using Ant tasks. Traceability between models and code, an issue addressed by our approach, is not dealt with explicitly.

Another line of work is concerned with the testing of meta-models. Sadilek and Weißleder propose an approach based on the specification of positive and negative example models using a graphical tool [20]. Test code can be generated automatically from these example models. López-Fernández et al. [21] propose two approaches for meta-model testing, the first of them being example-based as well. The examples can be specified using a textual notation or general-purpose sketching tools; converters to produce test specifications are provided. The second approach is based on specifying domain-specific invariants for the meta-model. Similar to EUnit, these approaches are based on reusing existing visual or textual languages to specify models.

4.2 ASCII-based Notations in Software Engineering

Several software engineering problems have been tackled using ASCII-based formats. *TextTest* [22] 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 model-driven tools. Another complementary approach [23] uses an ASCII-based model notation for code generation. The authors mention converters from models to ASCII-based class models and back; however, they do not explicate their realization strategy. In [24], 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* [25]. [26] proposes a context-free grammar for ASCII-art tables as found in network protocol RFCs.

4.3 Embedded 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 [27] 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. Still, this approach is coupled to a specific IDE. Developers are forced to use this IDE, which is undesirable if a particular preferred IDE exists in their domain. In addition, 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. 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 [28], an extension of JetBrains MPS targeted at embedded software development, provides built-in visualizations for state machines. The *Xtext* language workbench [29] allows to visualize instances of textual DSLs using the ZEST visualization library [30]. Both approaches provide read-only visualizations, while the embedded views in MPS are also editable.

4.4 Deriving Visual Models from Textual Specifications

The identified trade-offs between visual and textual notations are also reflected in recent approaches to derive visual models from textual notations. *PlantUML*[5] allows specifying selected UML diagrams in a simple syntax and deriving a suitable visualization automatically. While this syntax is easy to adopt, the downside of this approach is that the user cannot influence the outcome of the layouting

[5] `http://plantuml.com/`. Retrieved on 2016-09-09.

process. Gregorics et al. [31] propose a refined approach where the user also specifies layout aspects of the generated visualization. A benefit of these approaches compared to using a visual editor is their technology-independence. Still, visual models and code relying on these models are maintained separately. Maro et al. [32] investigate an approach for integrating graphical and textual editors for a given DSL. The approach addresses the relevant problems of deriving a textual editor and keeping it synchronized with the graphical editor. Yet, it does not solve the challenges of ensuring an adequate layout in the visual notation and integrating the visual notation with a version control system.

4.5 Model-driven testing

Model-driven testing (MDT) aims to derive tests from abstract specifications such as *coverage criteria* [33], *dedicated profiles* [34], and *visual contracts* [35], or *model-transformation contracts* [2]. Since these specifications are used to derive plain test cases that need to be read and understood by a maintainer, these works are orthogonal to our approach. To reap the benefits of these approaches, we aim to provide converters for the derived test models.

5 Conclusion

5.1 Summary

Tests are of paramount importance in software engineering. We target the challenge of testing model-driven tools, a scenario where the challenge lies in maintaining a large set of models and test classes. 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 IDEs and editors. 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.

Our preliminary evaluation indicates that the approach is useful to specify models of small to moderate size. Such models are common in the context of boundary analysis [36], where tests are edge cases, representing distilled essences of critical scenarios. For larger models, we distinguished five problematic models shapes and applied our approach to each of them. We experienced that our approach does not scale up well in most cases. Altogether, this finding is consistent with evidence indicating a negative correlation between diagram size and maintainability [16].

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 and the development of a framework to support multiple modeling languages through their concrete syntax. 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.

5.2 Future Work

Our object diagram notation provides support for creating models of arbitrary MOF-based modeling languages. Still, developers may prefer the known domain-specific concrete syntaxes from their application domains. To support extensibility for arbitrary languages, we envision the development of a *notation framework*, allowing customization for new languages by defining visual tokens and their mapping to the underlying meta-model.

To enable the automated translation between external specifications and our notation, converters are required. Sources of external specification may include regular models in custom layout formats, PowerPoint and Visio documents, and even photographed hand-drawn sketches. A promising technique to address this challenge is provided by an ASCII-art generation heuristics that accounts for line structures found inside an input graphic [37]. If such a technique is used, post-processing 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 to be visualized. To support them, we intend to apply layouting [38, 39] and splitting tools [40–42].

In addition to tests, a variety of further use-cases of our notation can be investigated. First, it can serve as a communication aid: developer communication over channels such as e-mail or instant messengers has been observed to contain a wide variety of informal notations, including ASCII-art [43]. Second, the notation can help as a documentation artifact. It allows usage examples and counter-examples to be specified without relying on a graphical tools, rendering it suitable for use in Javadoc and Wikis. In this area, MarkDeep[6] is another ASCII-based technique that has recently attracted developer attention[7]. Third, the tool independence of our notation makes it suitable as a neutral exchange format for conversion between proprietary diagram formats. In all of these use-cases, our provided tool set can help to bridge the gap between informal sketches and machine-readable diagrams.

References

1. G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing.* John Wiley & Sons, 2011.
2. M. Gogolla and A. Vallecillo, "Tractable model transformation testing," in *European Conference on Modelling Foundations and Applications.* Springer, 2011, pp. 221–235.
3. A. Vallecillo, M. Gogolla, L. Burgueno, M. Wimmer, and L. Hamann, "Formal specification and testing of model transformations," in *Formal Methods for Model-Driven Engineering.* Springer, 2012, pp. 399–437.

[6] MarkDeep: https://casual-effects.com/markdeep/ Retrieved on 2016-09-09.
[7] Twitter: https://twitter.com/casualeffects/status/654881908560646148 Retrieved on 2016-09-09.

4. E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Automated verification of model transformations based on visual contracts," *Automated Software Engineering*, vol. 20, no. 1, pp. 5–46, 2013.
5. A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo, "EUnit: A unit testing framework for model management tasks," in *Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 395–409.
6. E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE, 2014, pp. 201–211.
7. E. Daka, J. Campos, J. Dorn, G. Fraser, and W. Weimer, "Generating readable unit tests for guava," in *International Symposium on Search Based Software Engineering*. Springer, 2015, pp. 235–241.
8. D. Fahland, J. Mendling, H. A. Reijers, B. Weber, M. Weidlich, and S. Zugal, "Declarative versus Imperative Process Modeling Languages: The Issue of Maintainability," in *Business Process Management Workshops*, vol. 43. Springer, 2009, pp. 477–488.
9. A. van Deursen and P. Klint, "Little languages: little maintenance?" *Journal of Software Maintenance*, vol. 10, no. 2, pp. 75–92, 1998.
10. P. Klint, T. Van Der Storm, and J. Vinju, "On the impact of DSL tools on the maintainability of language implementations," in *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*. ACM, 2010, p. 10.
11. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.
12. A. F. Blackwell, C. Britton, A. Cox, T. R. Green, C. Gurr, G. Kadoda, M. Kutar, M. Loomes, C. L. Nehaniv, M. Petre *et al.*, "Cognitive dimensions of notations: Design tools for cognitive technology," in *Cognitive Technology: Instruments of Mind*. Springer, 2001, pp. 325–341.
13. P. Langer, T. Mayerhofer, M. Wimmer, and G. Kappel, "On the Usage of UML: Initial Results of Analyzing Open UML Models," in *Modellierung*, vol. 19, 2014, p. 21.
14. D. Strüber, F. Rieger, and G. Taentzer, "MUTANT: Model-Driven Unit Testing using ASCII-art as Notational Text," in *Flexible Model Driven Engineering Proceedings (FlexMDE 2015)*. Ceur-ws.org, 2015, pp. 2–11.
15. T. Arendt and G. Taentzer, "A tool environment for quality assurance based on the Eclipse Modeling Framework," *Automated Software Engineering*, vol. 20, no. 2, pp. 141–184, 2013.
16. 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.
17. B. Karasneh and M. R. Chaudron, "Online Img2UML Repository: An Online Repository for UML Models," in *EESSMOD MoDELS*, 2013, pp. 61–66.
18. O. M. Group, "Human-usable textual notation specification." [Online]. Available: http://www.omg.org/spec/HUTN/
19. A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo, "Eunit: a unit testing framework for model management tasks," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 395–409.
20. D. A. Sadilek and S. Weißleder, "Testing metamodels," in *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008,*

Berlin, Germany, June 9-13, 2008. Proceedings, I. Schieferdecker and A. Hartman, Eds., 2008, pp. 294–309.

21. J. J. López-Fernández, E. Guerra, and J. de Lara, "Meta-model validation and verification with metabest," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds., 2014, pp. 831–834.
22. 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.
23. 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.
24. R. Lämmel and C. Verhoef, "Semi-automatic Grammar Recovery," *Software: Practice and Experience*, vol. 31, no. 15, pp. 1395–1438, 2001.
25. S.-K. Chang, "Picture Processing Grammar and its Applications," *Information Sciences*, vol. 3, no. 2, pp. 121–148, 1971.
26. 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.
27. M. Voelter and K. Solomatov, "Language modularization and composition with projectional language workbenches illustrated with MPS," *Software Language Engineering*, vol. 16, 2010.
28. 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.
29. 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.
30. R. I. Bull, *Model Driven Visualization: Towards a Model Driven Engineering Approach for Information Visualization.* PhD diss., University of Victoria, 2008.
31. B. Gregorics, T. Gregorics, G. F. Kovács, A. Dobreff, and G. Dévai, "Textual diagram layout language and visualization algorithm," in *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on.* IEEE, 2015, pp. 196–205.
32. S. Maro, J.-P. 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 *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering.* ACM, 2015, pp. 1–12.
33. R. Heckel and M. Lohmann, "Towards Model-Driven Testing," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 6, pp. 33–43, 2003.
34. P. Baker, Z. R. Dai, J. Grabowski, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile.* Springer, 2007.
35. G. Engels, B. Güldali, and M. Lohmann, "Towards Model-Driven Unit Testing," in *Models in Software Engineering.* Springer, 2007, pp. 182–192.
36. B. Legeard, F. Peureux, and M. Utting, "Automated boundary testing from Z and B," in *FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, L. Eriksson and P. A. Lindsay, Eds., 2002, pp. 21–40.
37. 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.

38. M. Spönemann, *Graph Layout Support for Model-Driven Engineering.* PhD diss., Uni Kiel, 2015.
39. S. Maier and M. Minas, "A Pattern-based Approach for Initial Diagram Layout," *Electronic Communications of the EASST*, vol. 58, 2013.
40. D. Strüber, M. Selter, and G. Taentzer, "Tool support for clustering large meta-models," in *Proceedings of the Workshop on Scalability in Model Driven Engineering.* ACM, 2013, p. 7.
41. 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.
42. 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.
43. 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.