Towards Flexible Model Analysis and Constraint Development: A Small Demo Based on Large Real-Life Data

Matthias Sedlmeier and Martin Gogolla

Database Systems Group, University of Bremen, Germany
{ms|gogolla}@informatik.uni-bremen.de

Abstract. This contribution discusses the handling of a larger model on an abstract level employing the standardized modeling languages UML and OCL and an accompanying tool. We represent real-world data in form of a large object model and perform data analysis, verification, and validation tasks on the modeling level in order to obtain feedback about the original data. The tool allows to explore larger object diagrams interactively in a flexible way through a combination of visual and textual techniques. Furthermore, model invariants can be created in a versatile fashion by iteratively considering relevant object diagram fractions and evolving OCL expressions.
Keywords. Working with large models, Transformation and validation of large models, Visualization techniques for large models.

1 Introduction

Modeling and model management approaches have found their way into mainstream software production. Thus they are applied to large and complex systems, and approaches and tools have to deal with large models. In this contribution, we discuss an approach handling larger object models representing real-world data in a design tool originally elaborated for model validation and verification. The aim is to perform analysis, verification, and validation on the modeling level in terms of the languages UML [5] and OCL [8].

Working on the modeling level (in contrast to working on the code level or working with "production" systems as databases or programming languages) gives a high degree of abstraction through the available advanced modeling and validation options. UML and OCL have in contrast to traditional (relational query) languages the advantage of providing abstraction support for (a) different collection kinds (as sequences or ordered sets), (a) powerful operations (as closure or iterate), and (c) a mixture of textual and visual exploration mechanisms. Our approach gives interactive and direct feedback through the combination of textual and visual aspects, in particular for exploring model properties and model constraints. For large object models we support the exploration of queries and invariants that involve aggregation functions (that are of relevance in larger states

only) like the minimum or the average, as these functions are fully integrated into OCL.

Regarding larger object models, [4] identifies queries and expressions (e.g. in form of OCL) as an important field of activity. Our demo could further be part of an MDE tool benchmark focusing on queries [1]. Furthermore, structuring and slicing large class and objects models as in [7] is applicable in our context.

The rest of paper is structured as follows. Section 2 describes our process for obtaining our example model, in particular a large object model. Section 3 discusses the exploration of large object models by combining visual and textual techniques. Section 4 puts forwards an interactive process for constraint determination. Finally, Section 6 concludes our work and indicates further research.

2 Development Process for Class and Object Model

This section describes a real-life example for studying a large model. We wanted to obtain a model containing a UML class diagram and a large UML object diagram that can be handled interactively and flexible in *UML modeling and verification* tools. We employ the UML and OCL design tool USE (Uml-based Specification Environment) [3]. The tool supports model validation and verification for various UML diagram kinds, among them class and object diagrams, and OCL (Object Constraint Language) constraints in form of class invariants and operation contracts. Recently, the capabilities of USE have been improved in particular for handling large object diagrams (more efficient handling of object collections and evaluation of OCL expressions).

Our goal was not to compete w.r.t size with *production* tools as e.g. database systems. We aimed at an interactive development process exploring properties in the underlying data (in technical terms, in the employed object diagram) and exploring model constraints that have to be applied.

The German Federal Statistical Office (Destatis)[1] provides a municipal directory called GV100 of about 20.000 German administrative units, e.g., towns or villages, including about 60.000 structural connections based on a census in 2011. This directory can be officially obtained on the Destatis web portal as a monolithic column-oriented ASCII file together with record descriptions in natural language required to interpret the provided data[2].

The ASCII records contain information about federal states (German: Bundesland), districts (Bezirk), counties (Kreis), municipalities associations (Gemeindeverband) as well as municipalities (Gemeinde). Furthermore, the hierarchical connections between those units are given as well as the places of administration. We have complemented this data with additional geographical position information provided by the OpenGeoDB project[3] and obtain municipality records pro-

[1] https://www.destatis.de/DE/Startseite.html

[2] https://www.destatis.de/DE/ZahlenFakten/LaenderRegionen/Regionales/Gemeindeverzeichnis/Administrativ/Archiv/GV100ADQ/GV100AD3108.html

[3] http://opengeodb.org/wiki/OpenGeoDB

viding information about area, population, male population, female population, zip code as well as geographical information in form of longitude and latitude.

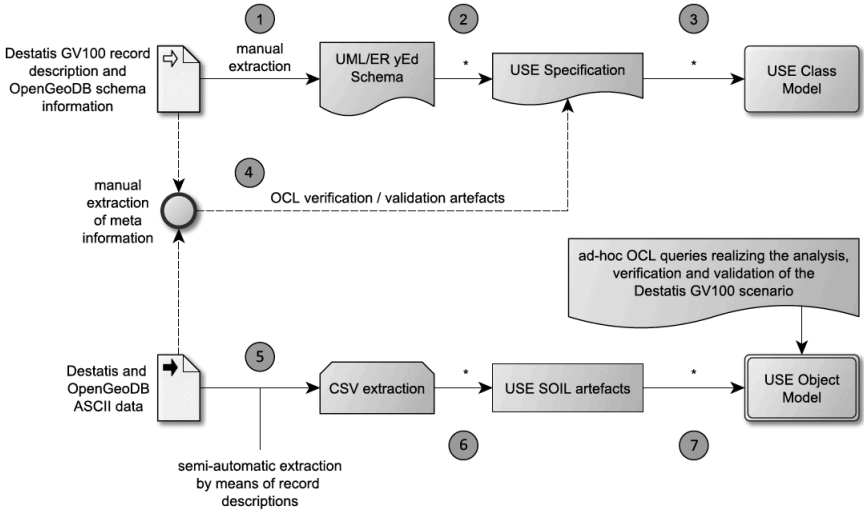

Fig. 1. Development process for UML class and object diagram

Figure 1 shows the process that we have applied in order to obtain an initial version of the class and object diagram. We started with the manual extraction of a data schema based on the Destatis GV100 record descriptions (1). As an intermediate representation, we have created this schema with the graph editor yEd in a UML/ER like modeling language (as in [6]). Moreover, we consider geographical information from the OpenGeoDB project.

In the next step, we perform an automatic transformation of the UML/ER schema into a valid USE specification (2) describing a USE class model (3). Figure 2 shows the central elements of the resulting UML class diagram. This model contains those classes and associations required for instantiating an object model. We have enriched the USE specification by OCL constraints and queries for verification and validation purposes (4), which are manually elaborated by analyzing the given record descriptions as well as the concrete ASCII records.

For the schema at hand, we semi-automatically extract the instance data as CSV files by means of Destatis record descriptions (5). The CSV data is then automatically transformed into USE data manipulation statements formulated in SOIL (6), which are interpreted by USE to instantiate the object model (7). SOIL ('Simple Ocl-like Imperative Language') is the USE "programming" language based on OCL. From this point on, we are able to interactively work in a flexible way with the class model including the object diagram employing the USE graphical interface, the command line interface and a combination of them in order to apply verification and validation tasks based on OCL constraints and queries.

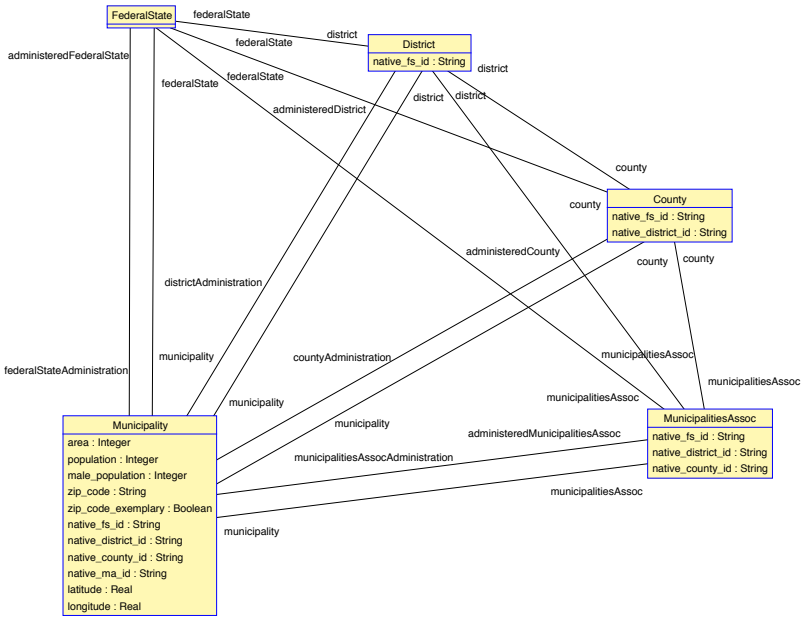

Fig. 2. Central elements of the UML class model for GV100

3 Exploring Object Models Visually and Textually

This section will explain the various interactive, but powerful options that USE offers to explore a large object diagram. The considered object diagram contains about 80.000 objects and links, as indicated in Fig. 3. Naturally, these objects do not fit on a single screen. USE allows to choose specific instances to create clearly arranged object model fractions. USE offers various object selection mechanisms [2], which effectively help to filter the desired subset. The techniques from [2] have been substantially extended in order to cope with object models showing the example's complexity. In USE, one can obtain fractions of the overall state fitting the developer's needs.

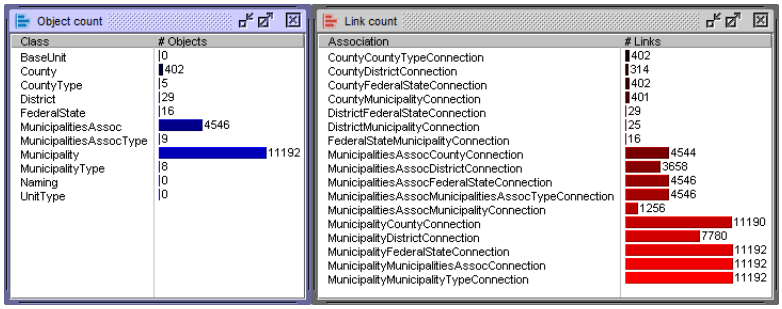

Fig. 3. Number of objects and links in an object model

First, it is possible to start with an empty object diagram to prevent long display times. We can then choose to show successively objects either by their type or by their properties.

Second, in addition to this simple filtering method, USE offers sophisticated object selection based on link path length, OCL query expressions and views. The latter method enables the developer to select objects specifically within a table view. The chosen set of objects can then be shown, hidden or cropped. A corresponding example is shown in Fig. 4, where a specific *FederalState* object (Schleswig-Holstein) and a *Municipality* object (Kiel) are manually selected by check boxes. Besides the objects, also all connecting links show up.

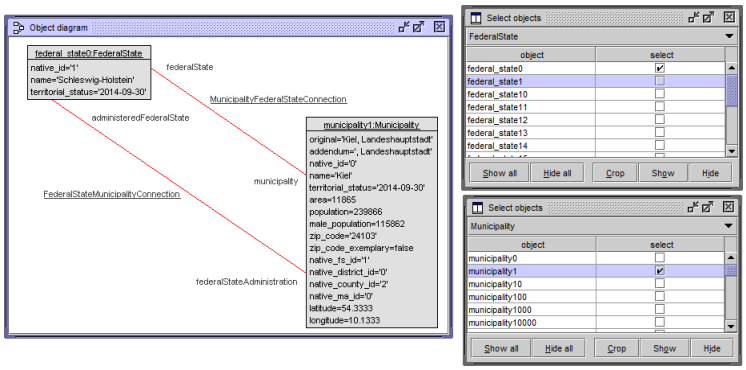

Fig. 4. Object selection based tables visually presented

Third, selection via link path length enables the designer to show only those objects, which are reachable from a given object over a specified number of link segments.

Fourth, the most flexible selection mechanism is provided via OCL query expressions, which allows the developer to show, hide or crop arbitrary sets of objects based on OCL query results being either single objects or object collections. Figure 5 shows an object model fraction containing only those federal states holding municipalities, which have a population above 100.000 and are located east of a fixed longitude and south of a fixed latitude. The example OCL expression only returns 3 objects, depending on the restrictness of the expression large object collections can be obtained.

Figure 6 illustrates a more detailed scenario, where some subordinated administrative units of the county *Merzig-Wadern* (where the Informatics meeting center "Dagstuhl" is located) are displayed. The example explores the administrative connections between all involved units (the county *Merzig-Wadern* belongs to the federal state *Saarland* and contains the municipality association *Wadern, Stadt*, which is at the same time also defined as a municipality; the county *Merzig-Wadern* also contains other municipalities like *Merzig*, *Mettlach* or *Losheim am See*).

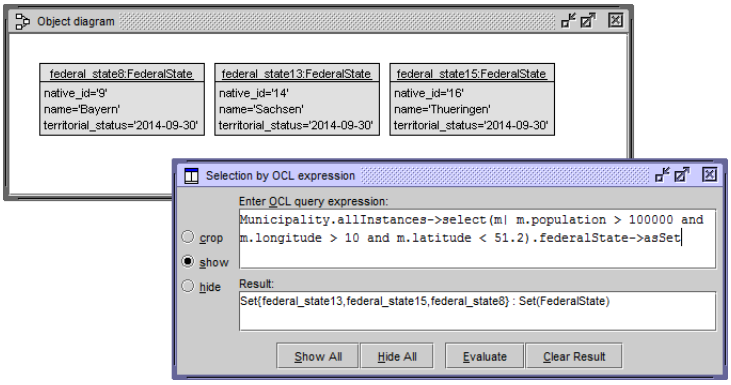

Fig. 5. Object selection by OCL expression visually represented

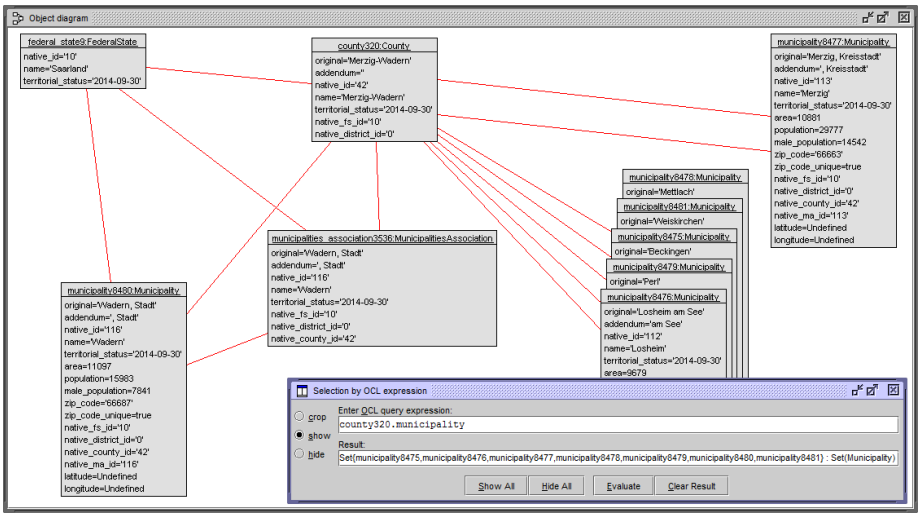

Fig. 6. Result of combining textual and visual object selection

Thus the model exploration options available in USE allow a combination of textual and visual techniques. The example object model display in Fig. 6 was created by evaluating multiple OCL expressions exploring the shown objects. USE provides the possibilities to construct object models stepwise with different OCL queries. This flexibility of switching between visual selection and evaluation of textual OCL expressions is, to the best of our knowledge, unique to USE.

4 Interactive Determination of Invariants

This section discusses how OCL model invariants can be developed interactively by considering the system state, formulating a hypothetical invariant, checking the assumed invariant against the state, and possibly revising the formulation of the invariant when the system state does not satisfy it and thus the invari-

ant cannot be validated. We follow an empirical approach, where we first make assumptions about rules, which should apply in the model application context. We therefore explore the schema information as well as the system state. Based on these assumptions we formulate an invariant, which we check by considering the system state in USE. If the invariant holds, we accept it as part of our USE model. If it does not hold, we analyze the result in order to determine, if our condition is invalid due to wrong premises or if the given object model (in the example determined by the Destatis information) is actually inconsistent. In the first case, we check if it is reasonable to adapt the invariant and start again. In the second case, we accept, that the underlying information (in the example the Destatis data) is possibly to a certain degree faulty and extend the USE specification with a modified invariant preventing the faulty information to go into the validation process (e.g., by weakening an invariant by adding an implication with a weakening premise).

In the example, the *longitude* and *latitude* attributes are not part of the original Destatis data but the information was taken from the OpenGeoDB project. This does not have any effect on the evaluation of the GV100 records. In fact, we would able to partly check the OpenGeoDB data, too.

4.1 Invariant *popGreaterThanZero*

We assume, that all municipalities have a population (pop) greater than zero:

```
context Municipality inv popGreaterThanZero:
  population > 0
```

We realize, that this invariant does not hold in the first inspection. USE allows us to check, which instances violate invariants either by the *Class Invariant View* or by the interactive shell `check -d` command. This is again an example where USE supports the developer through visual and textual techniques.

Through the analysis, we recognize, that all objects breaking our condition have the same municipality type assigned. This type classifies all assigned municipalities explicitly as uninhabited territory (German: 'gemeindefreies Gebiet, unbewohnt'). Based on this insight, we adapt our condition and accept it as part of our USE specification.

```
context Municipality inv popGreaterThanZero:
  (type.name <> 'gemeindefreies Gebiet, unbewohnt') implies
  (population > 0) and (male_population > 0)
```

This invariant holds in the system state, from which we can conclude, that the Destatis data is valid in this specific context.

4.2 Invariant *malePopLessPop*

Our premise is, that the male population must always be less than the overall population. This assumption is valid, because we explored the system state and validated, that

there are no "all-male municipalities" in Germany (the best pro-male ratio is 1:3.47 in *Freistatt, Niedersachsen* and the best pro-female ratio is 1:2.13 in *Hamm, Rheinland-Pfalz*). Computations like the best pro-male ratio can be formulated as OCL queries. In this case the ratio is determined by the following OCL expression:

```
Municipality.allInstances.collectNested(m|
  Sequence{m.federalState.name, m.name, m.population, m.male_population,
           m.population-m.male_population})
  ->collectNested(t|
    Sequence{t->at(1), t->at(2), t->at(3), t->at(4),t->at(5)
             t->at(4).oclAsType(Integer) / t->at(5).oclAsType(Integer)})
  ->asSequence()
  ->sortedBy(t|t->at(6).oclAsType(Real))
  ->select(t|t->at(3).oclAsType(Integer) > 0)->last
```

Taking the last invariant into account, we define:

```
context Municipality inv malePopLessPop:
  type.name <> 'gemeindefreies Gebiet, unbewohnt' implies
  population > male_population
```

The invariant is true in the considered system state. We accept it and conclude, that the Destatis data is correct in this context.

4.3 Invariant *zipCodeExemplary*

The *Municipality* class has a boolean attribute called `zip_code_exemplary`. This attribute indicates, that a single municipality can have multiple zip codes, but only one exemplary zip code is held in the attribute `zip_code`. However, we expect, that there must be municipalities having only one zip code, which we know from everyday life.

```
context Municipality inv notAllZipCodeExemplary:
  not Municipality.allInstances.forAll(zip_code_exemplary)
```

4.4 Invariants Based on Geographical Information

As we mentioned above, we enrich the Destatis data by geographical information. We try to assign the longitude and latitude value to each municipality by matching their names. This method is neither exact nor does it cover all instances. We also do not know, if the OpenGeoDB coordinates are always correct, since we did not compare them with third-party sources. However, we may assume, that all coordinates must be within the most northern, southern, eastern and western points of Germany. The following list explores the corresponding limits for all compass directions in the World Geodetic System 1984 format[4].

Based on this information, we postulate, that all longitude values must be in the stated interval. This should hold analogously for the latitude values. Given that, we derive 4

[4] http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf

direction	location	latitude/longitude
most northern	List, Sylt, Schleswig-Holstein	55.050000, 08.400000
most southern	Haldenwanger Eck, Oberstdorf, Bavaria	47.270108, 10.178319
most eastern	Deschka, Neieaue, Saxony	51.266667, 15.033333
most western	Isenbruch, Selfkant, North Rhine-Westphalia	51.050000, 05.866667

Table 1. Border points of Germany

similar invariants respecting the fact, that longitude and latitude values are not always present. All 4 invariants hold in the system state and we add them to the USE model.

```
context Municipality inv northLimit: -- analogously for south, east, west
  latitude <> null implies latitude <= 55.05
```

4.5 Derived Properties Employing Aggregate Functions

Instead of using longitude and latitude coordinates as constants, we can compute the bordering rectangle of a federal state with derived attributes as indicated in Fig. 7. The derived attributes for a federal state make use of the association with roles names federalState and municipality. This association indicates the municipalities that constitute the federal state. All municipalities of a federal state are considered there, and the most western (eastern, northern, southern) coordinates are computed. Another independent association is the one with role name federalStateAdministration. In a constraint one can now check that the coordinates of a federal state administration are consistent with the bordering rectangle of the federal state.

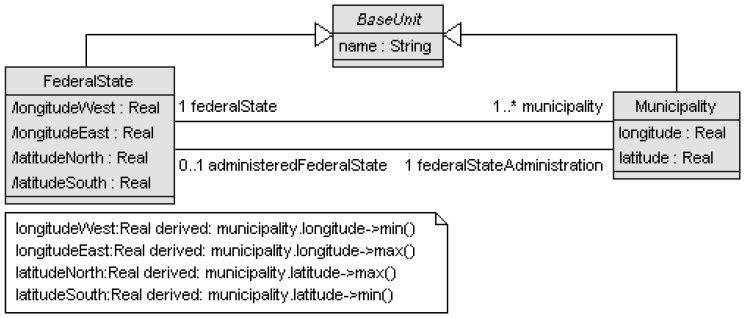

Fig. 7. Derived properties employing aggregate function in OCL

Another use of the bordering rectangle is a check about the plausibility of the bordering rectangles for different federal states. One can determine whether two federal states possess or do not possess common points (overlapping bordering rectangles). With that one can check that the actual present coordinates are such that, for example, the most northern state 'Schleswig-Holstein' and the most southern states 'Baden-Wuerttemberg' and 'Bavaria' do not have common points. We refrain from showing the detailed OCL expressions. But we emphasize that the derived attributes and the accompanying constraints only make sense in the presence of large object models as the attributes and constraints rely on aggregate function (here min() and max()) that apply in particular for large object collections.

5 Conclusion and Future Work

The practical experiences in our demo explores the capability of USE to handle larger object models with good performance. All discusssed expressions evaluate in magnitudes of at most few minutes. We have shown, how a large UML object model was employed for analyzing real-world data. The demo shows that analyzing data on the modeling level brings advantages w.r.t. to ease of formulating expressions due to the available abstraction mechanisms. We were able to browse and analyze the model and perform primary validation tasks. We used the provided textual and visual object exploration and selection mechanisms including OCL queries. In fact, our approach has an initial setup cost, but we obtain insight on a formal level with standardized modeling languages.

Further experiments will identify limitations with regard to the size of objects and links. Future work will also elaborate on fragmentation and slicing techniques for object models in order to handle representative slices of the original data. Another topic is the improvement of the visualization options for large object models, e.g., selection mechanisms by link kind (only association, binary and ternary associations, aggregation, composition). Naturally, more case studies and comparison with existing solutions must give feedback about the applicability of the proposal.

References

1. A. Benelallam, M. Tisi, I. Ráth, B. Izsó, and D. S. Kolovos. Towards an open set of real-world benchmarks for model queries and transformations. In D. S. Kolovos, D. D. Ruscio, N. D. Matragkas, J. de Lara, I. Ráth, and M. Tisi, editors, *Proc. 2nd WS Scalability in Model Driven Engineering BigMDE@STAF2014*, volume 1206 of *CEUR Workshop Proceedings*, pages 14–22. CEUR-WS.org, 2014.
2. M. Gogolla, L. Hamann, J. Xu, and J. Zhang. Exploring (Meta-)Model Snapshots by Combining Visual and Textual Techniques. In F. Gadducci and L. Mariani, editors, *Proc. Workshop Graph Transformation and Visual Modeling Techniques (GTVMT'2011)*. ECEASST, Electronic Communications, journal.ub.tu-berlin.de/eceasst/issue/view/53, 2011.
3. M. Gogolla and F. Hilken. Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool. In A. Oberweis and R. Reussner, editors, *Proc. Modellierung*, pages 203–218. GI, LNI 254, 2016.
4. D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. D. Lara, I. Rath, D. Varro, M. Tisi, and J. Cabot. A research roadmap towards achieving scalability in model driven engineering. In *Proc. 1st WS Scalability in Model Driven Engineering (BigMDE 2013)*. ACM, 2013.
5. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language 2.0 Reference Manual*. Addison-Wesley, Reading, 2003.
6. M. Sedlmeier and M. Gogolla. Model Driven ActiveRecord with yEd. In T. Welzer, H. Jaakkola, B. Thalheim, Y. Kiyoki, and N. Yoshida, editors, *Proc. Int. 25th Int. Conf. Information Modelling and Knowledge Bases (EJC'2015)*, pages 65–76. IOS Press, Amsterdam, 2015.
7. D. Strüber, M. Selter, and G. Taentzer. Tool support for clustering large metamodels. In D. D. Ruscio, D. S. Kolovos, and N. Matragkas, editors, *Proc. Workshop Scalability in Model Driven Engineering (BigMDE 2013)*. ACM, 2013.
8. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 2003. 2nd Edition.

Scalability of Model Transformations: Position Paper and Benchmark Set

Daniel Strüber[1], Timo Kehrer[2], Thorsten Arendt[1,3], Christopher Pietsch[4], Dennis Reuling[4]

[1] Philipps-Universität Marburg, Germany
[2] Politecnico di Milano, Italy
[3] GFFT Innovationsförderung GmbH, Bad Vilbel, Germany
[4] Universität Siegen, Germany

Abstract. As model transformations are often considered the "heart and soul" of Model-Driven Engineering (MDE), the scalability of model transformations is vital for the scalability of MDE approaches as a whole. The existing research on scalable transformations has largely focused on performance behavior as the involved input models grow in size. In this work, we address a second key dimension for the practical scalability of model transformations: The effect as the transformation specification itself grows larger. We outline a number of challenges related to the quality concerns of maintainability and performance. We then introduce three model transformation benchmarks and discuss how they are affected by these challenges. Our objective is to establish a community benchmark set to compare model transformation approaches with respect to the aforementioned quality concerns.

1 Introduction

Over the recent years, *Model-Driven Engineering* (MDE) has started to grow into a mature software engineering discipline. To facilitate the development of complex software systems, MDE envisions the use of abstraction and automation principles: Models are used to provide an abstract specification of a software system. This specification is automatically refined to a running software system using *model transformations*, thus enabling a reduced implementation effort. A large variety of modeling and model transformation languages has emerged to address the heterogeneity in the involved software domains and scenarios.

As MDE is increasingly applied in industrial large-scale scenarios, various limitations of the existing techniques and tools have been revealed. One of the key problem areas concerns the scalability of model transformations. In this area, existing research has mostly focused on scalability as the input models of transformations grow in size. Kolovos et al. summarize the goal of the existing works as *"advancing the state of the art in model querying and transformation tools so that they can cope with large models (in the order of millions of model elements)"* [1]. To this end, a number of performance optimizations and new execution modes, such as incremental [2,3] or concurrent [4] model transformations, have been provided to the MDE community.

Inspired by our experience of developing model transformations for several research projects, in this paper we argue that a second key dimension of scalability has been neglected so far: The scalability of the transformation specification. In particular, we point out that the performance of a model transformation is likely to deteriorate with the size and complexity of its specification. To make matters worse, input models *and* the transformation specification may be affected by scalability issues at the same time, a combination that leads to a more drastic scalability challenge. Furthermore, the size of a specification might also challenge its maintainability, even to the point that it becomes unusable when viewed and edited with the default tools.

We focus on the technical scope of rule-based model transformations. Techniques and tools in this domain are affected by the size of individual rules as well as the size of the overall rule set. Multiple factors contribute to the emergence of large rules and rule sets: The size of these artifacts can reflect the *essential complexity* of the intended transformation; model transformations are a particular kind of software, the development of which is generally complicated by the complexity of the imposed requirements. In addition, multiple factors contribute to *accidental* complexity in model transformations: The size of individual rules might reflect the size of the involved meta-models. For instance, the UML metamodel is infamous for its size and complexity that leads to complicated rules even when expressing transformations that are simple on a conceptual level [5]. Furthermore, a common situation involves *families of rules*, i.e. sets of rules that exhibit a high degree of commonalities. While some transformation languages offer built-in concepts to manage families of rules to some extent, we found these concepts insufficient in several cases as described in this paper.

The contribution of this work is twofold. First, to illustrate our position, we explore the scalability issues encountered during our past experiences and relate them to existing experiences from the literature. Second, we provide a set of benchmark scenarios of rule sets affected by these issues. We have used some of these scenarios as an evaluation basis in our recent work [6,7,8,9]. Our aim is to make them available for other researchers. By providing the rule sets with a systematic description, as part of a publicly available repository, our goal is to facilitate the comparison of model transformation approaches with respect to their scalability. We provide the benchmark set together with usage instructions at GitHub: `https://github.com/dstrueber/bigtrafo`

The rest of this paper is structured as follows. In Sect. 2, we outline the scalability challenges in more detail. Sect. 3 introduces two benchmark kinds to assess related quality aspects. We present our benchmark set in Sect. 4, discussing its limitations in Sect. 5. In Sects. 6 and 7, we discuss related work and conclude.

2 Scalability Challenges

In this section, we discuss scalability challenges related to large transformations as observed in our own work and in the literature.

Maintainability is one of the main quality goals during the development of a model transformation [10,11,12]. It refers to the capability to modify the transformation after its initial deployment when it has to be adapted, e.g., to address changing requirements or to remove defects.

A necessary prerequisite for maintainability is understandability: If the specification is difficult to understand, the time required to perform a change as well as the risk of creating defects while doing so increases. Intuitively, the understandability of a transformation system is related to two separate dimensions of size: the number of rules and the size of individual rules. In a large set of rules, the difficulty lies in pinpointing the target location for an intended change. In turn, large individual rules lead to large diagrams that may not scale to the cognitive capacities of the human mind. A detrimental effect of large diagram size on understandability has been found for the scope of class diagrams [13].

Another obstacle to maintainability concerns the scalability of the model transformation editor. The existing transformation editors were often designed and tested for transformation systems of small to medium scale. In a rule set with hundreds of rules and hundreds of elements per rule, the response time during loading, navigating, and changing rules might be substantial, thus prohibiting the transformation to be edited in an efficient manner. This obstacle can be addressed in two ways: either by improving the scalability of the editor or by providing a more compact representation of the overall rule set.

Performance is another key quality aspect of model transformations [10,11,12]. In particular, in the case of graph-based model transformation languages, the execution of a transformation entails the NP-complete sub-graph isomorphism problem, leading to substantial execution times as input models and rules grow.

In practice, the performance of a model transformation depends on its execution mode. Despite considerable progress on particular execution modes such as incremental [2,3] and parallel transformations [4], the standard mode remains *batch transformation* [2]. In batch mode, all rules in the rule set are applied as long as one of them is applicable. This mode is often found in translation, simulation, and refactoring scenarios. In a batch transformation, each rule increases the computational effort of the transformation system; the larger the rule set becomes, the harder it is to keep it tractable. For instance, Blouin et al. [14] report on a case where a transformation engine was unable to execute a transformation system with 250 rules. As language-level support for such issues is widely unavailable [15], the size of the rule set if often reduced using ad-hoc solutions.

3 Benchmark Kinds

Based on these challenges, we identify two different benchmark kinds for model transformation approaches (see Fig. 1). The distinction of these benchmark kinds allows us to study maintainability and performance disjoint from each other. A single approach to address both issues might not necessarily be most effective: Another approach is to maintain rules in a form that maximizes their maintainability and to apply a performance optimization before executing the rules [6].

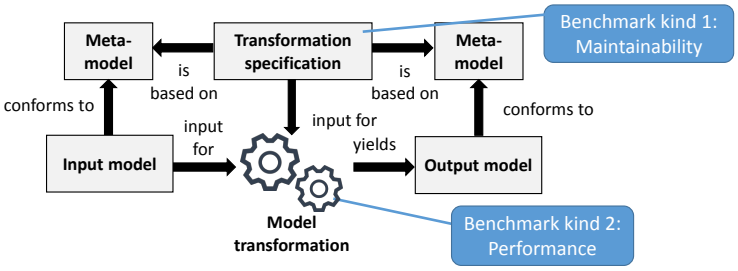

Fig. 1. Benchmark kinds: Overview

Benchmark kind 1: Benchmarking for Maintainability (M).

- **Goal.** Specifying a transformation with beneficial maintainability properties.
- **Scope.** We consider the maintainability property *compactness*.
- **Rationale.** Smaller rule sets and rules may be easier to read, entail less individual edits to perform a single change, and show better performance behavior when viewed and edited in standard editors.
- **Metrics.** Accumulative number of rule elements, number of rules, element-per-rule ratio required to specify the entire transformation.

Benchmark kind 2: Benchmarking for Performance (P).

- **Goal.** Specifying a transformation with beneficial performance properties.
- **Scope.** We consider the performance property *execution time*.
- **Rationale.** Lower execution times of the included model transformation may lead to general improvements in the existing MDE approaches.
- **Metrics.** Accumulative execution time on a set of given input models.

Usage. To use the benchmark set, a transformation tool designer needs to follow three steps: First, implement a new solution to one of the provided benchmarks. Second, validate the solution. Third, measure the mentioned quality attributes.

(1) Implementation. To guide benchmark users during the implementation of new solutions, each benchmark has three specifications: A textual *high-level specification* provided in the research paper from which the transformation was obtained. A *reference solution* based on the Henshin model transformation language [16]. A *black-box specification* based on a set of input models and the output models produced by applying the Henshin solution to these models.

(2) Validation. The provided black-box specifications can be used to validate the correctness of new solutions. To this end, the new solution needs to be applied to the provided input models. Afterwards, the produced output models can be compared against the ones provided with the benchmark. A solution is correct if the produced output models are isomorphic to the provided ones.

(3) Measurement. To measure compactness, we provide a measurement framework in our repository. This framework is currently customized for Henshin rules;

Benchmark	Kind	Scenario	#Rules	#N	#E	#A	#Models
Edit Rules	M	FM	58	803	945	272	108
	M	UML	1404	6865	2721	1433	1766
Recognition Rules	P	FM	58	3758	6580	1217	26
	P	UML	1404	26162	30777	14816	19
Translation Rules	M+P	OCL2NGC	54	2259	2389	1142	10

Table 1. Overview of the benchmark set. For each benchmark, the columns give the benchmark name, benchmark kind, scenario, number of rules, nodes, edges, and attributes.

measurements for further transformation languages can be established by implementing new instantiations. Such new instantiations of the measurement framework can account for custom language features, such as support for modularity. Execution time is measured in terms of the time required to execute the transformation on all input models. Reference times are provided in the repository.

Mailing list. To support communication about our benchmarks, we maintain a mailing list at `http://www.freelists.org/list/bigtrafo`. The purpose of this mailing list is twofold: First, it provides a platform to announce new solutions and benchmark results. Second, this platform can be used to clarify questions about the benchmarks and particular solutions.

4 Benchmarks

Our benchmark set, summarized in Table 1, comprises three benchmarks. The *Edit Rules* and *Recognition Rules* benchmarks inherently focus on the maintainability and performance concerns, respectively, while *Translation Rules* is a benchmark where these quality concerns are both relevant, thus allowing the relationship between both concerns to be studied. The statistics given in the table indicate the size of each transformation in the provided Henshin specification.

4.1 Edit Rules

Context and objectives. Edit commands as offered by visual editors and modern model refactoring tools are typical forms of edit operations on models. Many tools of an MDE tool suite must be customized to the way how models can be edited, model versioning [17,18], refactoring tools [19] as well as model mutation [20] being examples of this. Therefore, explicit specifications of the available edit operations are required. Rule-based in-place model transformations are well-suited for that purpose [21,22,23]. However, developing and maintaining a set of *edit rules* is challenging. Firstly, edit rules can become large in case of complex model restructuring operations. Secondly, the set of edit being required to specify every possible model modification becomes huge in case of comprehensive languages such as UML. Thus, the specification of edit operations is an adequate benchmark addressing the maintainability of model transformation rules. We selected two scenarios in which suitable edit operations have been specified:

FM. This scenario refers to the editing of feature models [24], a widely used variability modeling approach in software product line (SPL) engineering. The selected edit operations have been defined in [25]. Groups of edit operations are distinguished w.r.t. their semantic impact on the set of valid feature combinations. A *refactoring* leaves this set unchanged, a *generalization* (*specialization*) enlarges (shrinks) the set of valid feature combinations. Such edit operations are often complex restructurings with several non-trivial application conditions.

UML. This scenario addresses the editing of UML models. Due to the complexity of the UML meta-model, edit operations on UML models turn out to be rather complex when being specified based on the abstract syntax [5]. Restricting the navigability of an association to one end is an example of this [22]. In this benchmark, the selected UML edit operations refer to those parts of the UML which are used in [26], a case study on using a UML-based approach for modeling the Barbados Crash Management System (bCMS).

Meta-Models and Models. In the FM scenario, the selected edit rules are specified over the meta-model defined in [25]. Concerning the UML scenario, the selected edit rules are defined over the UML standard meta-model defined by the OMG [27]. The subset being relevant in this benchmark is given by the UML models of the bCMS case study which uses class diagrams, sequence diagrams, and statecharts. In sum, the relevant subset includes 83 meta-classes, together with their various relations, out of the 243 meta-classes of the standard UML meta-model. We derived test models for both rule sets. Deriving application examples from the rule structures yielded at least one test model per rule.

Rules. The FM edit rule set comprises the 58 complex restructuring operations on feature models defined in [25]. The largest rule in this rule set comprises 74 nodes, 146 edges, and 14 attributes. The UML edit rule set comprises 1404 edit rules, all of them specifying edit operations which can be considered elementary from a user's point of view: these rules cannot be split into smaller edit operations being applicable to a model in a meaningful way. Consequently, the individual rules in the UML rule set are considerably smaller than the FM rules, the largest one comprising 9 nodes, 11 edges, and 14 attributes. To support their step-wise comprehension and re-implementation by benchmark users, both rule sets exhibit an organization into groups of semantically related rules.

4.2 Edit Operation Recognition Rules

Context and objectives. To support continuous model evolution, sophisticated tool support for model version management is needed. One of the most essential tool functions is the calculation the difference between two versions of a model. Instead of reporting model differences to modelers element-wise, their grouping into semantically related change sets helps in understanding model differences [22]. Edit operations as used in our first benchmark are the concept of choice to group such change sets. [22] presents an approach which automatically translates specifications of edit operations into *recognition rules*. These rules are used by an algorithm which recognizes edit operations in a low-level difference

of two model versions. Essentially, this algorithm searches for all matches of each recognition rule in a given difference. This is a computationally expensive pattern matching problem. Thus, the optimization of a recognition rule set is a primary concern and an adequate performance benchmark.

Meta-Models and Models. Concerning the modeling languages comprised by this benchmark, we selected the same languages along with the corresponding meta-models as in our first benchmark. In general, suitable low-level model differences on which to apply the edit operation recognition rules are obtained as follows: Given (i) a pair of input models which can be considered to be revisions of each other, and (ii) a model matcher which determines the corresponding elements in both versions, the differencing engine presented in [22] creates a low-level difference which can be synthesized as a "difference model". It basically defines five types of changes which can be observed in a low-level difference, namely the insertion/deletion of objects/references as well as attribute value changes. For this benchmark, we selected the following pairs of models and matchers.

FM. For feature models, we use the synthetically created differencing scenarios presented in [25]. We have 26 revision pairs of feature models of different characteristics and varying sizes, ranging from 100 up to 500 features per model. To calculate the low-level differences, we used dedicated matcher for pairs of feature models [25] to determine corresponding elements.

UML. In the UML scenario, we selected the models provided by the bCMS case study. In fact, bCMS defines not only a single model, but an entire SPL. We used 20 models representing valid instances of this SPL, one core model and 19 additional variants. By differencing each variant with the core, we produced 19 low-level differences. To determine corresponding elements, we used a dedicated matcher that exploits persistent unique element identifiers.

Rules. As described in [22], recognition rules can be automatically generated from their corresponding edit rules. Thus, it was a natural choice to generate the rules of this benchmark from the edit rules of our first benchmark. Consequently, we have 58 recognition rules for the FM and 1404 for the UML scenario. As seen in Table 1, recognition rules are generally larger than their edit rule counterparts. The largest recognition rule comprises 348 nodes, 769 edges, and 57 attributes in the FM case and 73 nodes, 81 edges, and 40 attributes in the UML case.

4.3 Constraint Translation Rules

Context and objectives. Modeling languages are usually specified in a declarative manner, using a meta-model with well-formedness constraints expressed in the Object Constraint Language (OCL). Yet in some situations, a constructive definition of a modeling language may be required instead, e.g., to systematically enumerate all possible models for verification purposes. Such a constructive approach is provided by graph grammars [28]. To support the generation of a graph grammar from an existing meta-model, we devised a constraint translator that can transform OCL constraints to nested graph constraints (NGCs) [29], which can then be further translated to application conditions in grammar rules.

Models and Meta-Models. As input models for our benchmark, we used ten OCL constraints designed for a large coverage of applicable rules. The size of the input models, comprising meta-models with embedded constraints as well as the OCL standard library, ranges from 1832 to 1854 model elements.

OCL2NGC is an exogenous model transformation. Its implementation involves three meta-models: The OCL pivot meta-model[1] acts as source meta-model. The NGC meta-model, provided as part of the benchmark, acts as target meta-model. The trace meta-model[2] acts as a language-agnostic auxiliary meta-model to manage the correspondence between source and target model elements.

Rules. Our implementation of the transformation described in [29] comprises a model transformation system with 54 rules. In addition, a control flow to guide the rule execution is specified using Henshin's concept of transformation units. The main performance bottleneck is a subset of 36 rules that are applied in batch mode, i.e., as long as one of them can be matched.

5 Limitations and Further Extensions

Since we address the scalability of model transformations, our maintainability benchmark focuses on *size* – the size of individual rules and entire rule sets. While we discuss the detrimental effect of size to maintainability in Sec. 2, size is by no means the only relevant maintainability aspect. A promising direction for future work is to consider additional maintainability metrics from research on software quality. Moreover, we do not study the effect of rule size on the performance of the transformation editor explicitly. While it is our experience that editor performance improves as the transformation size decreases, a dedicated measurement to quantify this performance gain is needed. Finally, the performance benchmark is limited to execution time. Considering additional properties, such as memory or even energy consumption, might lead to interesting insights.

6 Related Work

A number of benchmark sets for queries and transformations has been introduced in the literature. Benelallam et al. [30] propose a benchmark set focusing on scalability to large input models. A seminal benchmark paper for graph transformation languages has been contributed by Varró et al. [31]. Bergmann et al. [32] propose a benchmark set for incremental transformations. None of these benchmarks addresses the scalability of the transformation specification.

Izsó et al. [33] propose a MDE benchmark framework targeting a large variety of use cases, such as model validation, transformation, and code generation. The main idea is to define benchmarks in a systematic way by assembling them from reusable benchmark primitives. The authors also provide a number of emerging results, some of them pointing in the direction we explore in this paper. An integration of our benchmark set with this framework is feasible as future work.

[1] `https://wiki.eclipse.org/OCL/Pivot_Model`
[2] `https://wiki.eclipse.org/Henshin_Trace_Model`

7 Conclusion

In this work, we address the scalability of model transformation specifications, focusing on the quality goals *performance* and *maintainability*. We provide a publicly available benchmark set, encouraging other researchers to compare their transformation tools and contribute additional benchmarks. In the future, we aim to explore the relationship between performance optimizations between large models and transformation specifications further.

References

1. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ráth, I., Varró, D., Tisi, M., Cabot, J.: A research roadmap towards achieving scalability in model driven engineering. In: BigMDE Workshop on Scalability in Model Driven Engineering, ACM (2013) 2
2. Johann, S., Egyed, A.: Instant and incremental transformation of models. In: Int. Conference on Automated Software Engineering, IEEE Computer Society (2004) 362–365
3. Jouault, F., Tisi, M.: Towards incremental execution of ATL transformations. In: Proc. of Int. Conference on Model Transformations. Springer (2010) 123–137
4. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: On the Concurrent Execution of Model Transformations with Linda. In: BigMDE Workshop on Scalability in Model Driven Engineering, ACM (2013) 3
5. Acretoaie, V., Störrle, H., Strüber, D.: Transparent model transformation: Turning your favourite model editor into a transformation tool. In: Int. Conference on Model Transformations, Springer (2015) 283–298
6. Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., Plöger, J.: RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules. In: Int. Conference on Fundamental Approaches to Software Engineering, Springer (2016) 122–140
7. Strüber, D., Rubin, J., Chechik, M., Taentzer, G.: A Variability-Based Approach to Reusable and Efficient Model Transformations. In: Int. Conference on Fundamental Approaches to Software Engineering, Springer (2015) 283–298
8. Strüber, D.: Model-Driven Engineering in the Large: Refactoring Techniques for Models and Model Transformation Systems. PhD thesis, Philipps-Universität Marburg, Germany (2016)
9. Strüber, D., Plöger, J., Acretoaie, V.: Clone detection for graph-based model transformation languages. In: Int. Conference on Theory and Practice of Model Transformations. (2016) 191–206
10. Syriani, E., Gray, J.: Challenges for addressing quality factors in model transformation. In: Int. Conference on Software Testing, Verification and Validation, IEEE (2012) 929–937
11. Wimmer, M., Perez, S.M., Jouault, F., Cabot, J.: A catalogue of refactorings for model-to-model transformations. Journal of Object Technology **11**(2) (2012) 1–40
12. Gerpheide, C.M., Schiffelers, R.R., Serebrenik, A.: A bottom-up quality model for QVTO. In: Int. Conference on the Quality of Information and Communications Technology, IEEE (2014) 85–94
13. Störrle, H.: On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters. In: Int. Conference on Model Driven Engineering Languages and Systems, Springer (2014) 518–534
14. Blouin, D., Plantec, A., Dissaux, P., Singhoff, F., Diguet, J.P.: Synchronization of models of rich languages with triple graph grammars: An experience report. In: Int. Conference on Model Transformation. Springer (2014) 106–121

15. Kusel, A., Schönböck, J., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: Reuse in model-to-model transformation languages: are we there yet? Software & Systems Modeling **14**(2) (2013) 537–572
16. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Model Driven Engineering Languages and Systems. Springer (2010) 121–135
17. Kehrer, T., Kelter, U., Ohrndorf, M., Sollbach, T.: Understanding model evolution through semantically lifting model differences with SiLift. In: Int. Conference on Software Maintenance, IEEE (2012) 638–641
18. Kehrer, T., Kelter, U., Reuling, D.: Workspace updates of visual models. In: Int. Conference on Automated Software Engineering, ACM (2014) 827–830
19. Arendt, T., Taentzer, G.: A tool environment for quality assurance based on the Eclipse Modeling Framework. Automated Software Engineering **20**(2) (2013) 141–184
20. Reuling, D., Bürdek, J., Rotärmel, S., Lochau, M., Kelter, U.: Fault-based Product-line Testing: Effective Sample Generation Based on Feature-diagram Mutation. In: Int. Software Product Line Conference, ACM (2015) 131–140
21. Kolovos, D.S., Paige, R.F., Polack, F.A., Rose, L.M.: Update Transformations in the Small with the Epsilon Wizard Language. Journal of Object Technology (2003)
22. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: Int. Conference on Automated Software Engineering, IEEE (2011) 163–172
23. Mens, T.: On the use of graph transformations for model refactoring. In: Generative and transformational techniques in software engineering. Springer (2006) 219–257
24. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, S.A.: Feature Oriented Domain Analysis (FODA). Technical report, CMU (1990)
25. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A.: Reasoning about product-line evolution using complex feature model differences. Journal of Automated Software Engineering (2015) 1–47
26. Capozucca, A., Cheng, B., Guelfi, N., Istoan, P.: OO-SPL modelling of the focused case study. In: Int. Workshop on Comparing Modeling Approaches. (2011)
27. Object Management Group: UML 2.4.1 Superstructure Specification. OMG Document Number: formal/2011-08-06 (2011)
28. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation: Volume 2: Applications, Languages and Tools. world Scientific (1999)
29. Arendt, T., Habel, A., Radke, H., Taentzer, G.: From Core OCL Invariants to Nested Graph Constraints. In: Int. Conference on Graph Transformation, Springer (2014) 97–112
30. Benelallam, A., Tisi, M., Ráth, I., Izso, B., Kolovos, D.: Towards an open set of real-world benchmarks for model queries and transformations. In: BigMDE Workshop on Scalability in Model Driven Engineering. (2014)
31. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. In: IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE (2005) 79–88
32. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. In: Int. Conference on Graph Transformation. Springer (2008) 396–410
33. Izsó, B., Szárnyas, G., Ráth, I., Varró, D.: MONDO-SAM: A Framework to Systematically Assess MDE Scalability. In: BigMDE Workshop on Scalability in Model Driven Engineering. (2014) 40–43

Optimized declarative transformation First Eclipse QVTc results

Edward D. Willink[1]

Willink Transformations Ltd., Reading, UK,
ed/at/willink.me.uk,

Abstract. It is over ten years since the first OMG QVT FAS[1] was made available with the aspiration to standardize the fledgling model transformation community. Since then two serious implementations of the operational QVTo language have been made available, but no implementations of the core QVTc language, and only rather preliminary implementations of the QVTr language. No significant optimization of these (or other transformation) languages has been performed. In this paper we present the first results of the new Eclipse QVTc implementation demonstrating scalability and major speedups through the use of meta-model driven scheduling and direct Java code generation.

Keywords: optimization, declarative transformation, scheduling, code generation, QVTc

1 Introduction

The OMG QVT specification [11] was planned as the standard solution to model transformation problems. The Request for Proposals in 2002 stimulated 8 responses that eventually coalesced into a revised merged submission in 2005 for three different languages.

The QVTo language provides an imperative style of transformation. It stimulated two useful implementations, SmartQVT and Eclipse QVTo.

The QVTr language provides a rich declarative style of transformation. It also stimulated two implementations. ModelMorf never progressed beyond beta releases. Medini QVT had disappointing performance and is not maintained.

The QVTc language provides a much simpler declarative capability that was intended to provide a common core for the other languages. There has never been a QVTc implementation since the Compuware prototype was never updated to track the evolution of the merged QVT submission.

The Eclipse QVTd project [5] extended and enhanced the Eclipse OCL [4] framework to provide QVTc and QVTr editors, but until now provided no execution capability. This paper describes two aspects of the architecture that promises a high performance remedy for this deficiency. In Section 2 we review

[1] Object Management Group Query/View/Transformation Final Adopted Specification.

the efficiency hazards that are addressed in Section 3. Section 4 describes the derivation of an efficient declarative schedule and Section 5 presents preliminary results demonstrating some scale-ability and code generation speed-ups. Section 6 summarizes some related work and Section 7 concludes.

2 Efficiency

The execution time of each algoritm in a computer program is the product of many factors: $W.N.R.L.C^A$

- W - the Workload - the number of data elements to be processed
- N - the Necessary computations per data element
- R - the memory Representation access overhead
- L - the programming Language overhead
- C - the Control overhead
- A - the Algorithmic overhead

We cannot optimize N or W since they define the Necessary Work.

We have limited choice in regard to L since use of a more efficient Language than Java may incur tooling, portability or cultural difficulties. We have similarly limited choice in Representation since there may be few frameworks to choose from. We should however be aware that Java may cost a factor of two or three and that unthinking use of EMF [3] may incur a factor of ten.

The Control overhead is influenced a bit by programming style, but mostly by the tooling approach, thus an interpreted form of execution may easily incur a ten-fold overhead when compared to a code generated one.

The above costs are all proportionate. Algorithmic cost is however exponential, normally with a unit exponent, but a poor algorithm may involve quadratic or worse costs. It is therefore recognized that the best way to improve performance is to discover a better algorithm or more practically to replace a stupid algorithm that performs repeated or unnecessary work.

For a model transformation using an imperative transformation language, the programmer specifies the algorithms (mappings) and the control structures that invoke them (mapping calls). We hope that the programmer selects good algorithms and that the tooling for the transformation language implements them without disproportionate overheads so that the overall execution incurs only proportionate costs relative to the given program's Necessary Work.

When we use a declarative model transformation, the control structures are no longer defined by the programmer. A control strategy must be discovered by the transformation tooling. This has the potential to give improved performance since a better control strategy may be discovered than that programmed imperatively. On the other hand, inferior declarative transformation tooling may incur Control overheads that result in disproportionate Algorithmic overheads.

In this paper we outline approaches that avoid Algorithmic overheads and since we use code generation to Java, we compound a perhaps five-fold reduction in Representation access and a perhaps five-fold reduction in Control overhead when compared to an interpreted execution using dynamic EMF.

3 Architecture

The Eclipse QVTd architecture for QVTc and QVTc 'solves' the problem of implementing a triple language specification by introducing Yet Another Three QVT Languages[10] : QVTu, QVTm and QVTi. Subsequent scheduler development has introduced two more languages QVTp and QVTs for use in the transformation chain shown in Figure 1.

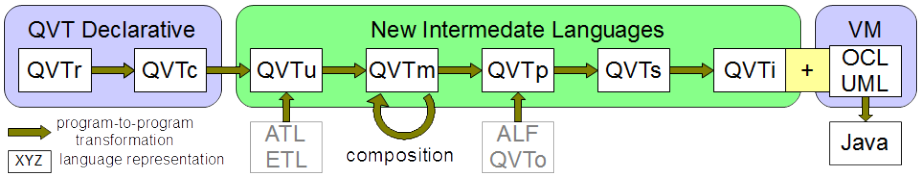

Fig. 1. Progressive transformation approach to Declarative QVT.

QVTr2QVTc implements the RelToCore transformation only partially defined by the QVT specification. This is still a work in progress.

QVTc2QVTu exploits the user's chosen direction to form a Unidirectional transformation without the bloat for the unwanted directions.

QVTu2QVTm normalizes and flattens to form a Minimal transformation free from the complexities of mapping refinement and composition.

QVTm2QVTp partitions mappings into micro-mappings that are free from deadlock hazards.

The foregoing representations use increasingly restricted semantic subsets of the QVTc language.

QVTp2QVTs creates a graphical representation suitable for dependency analysis enabling an efficient Schedule to be planned.

QVTs2QVTi serializes the graphical representation and schedule into an Imperative extension and variation of the QVTc language. This can be executed directly by the QVTi interpreter that extends the OCL Virtual Machine. Alternatively an extended form of the OCL code generator may be used to produce Java code directly. This bypasses many of the overheads of interpreted execution or dynamic EMF. The generated code comprises one outer class per-transformation. Each mapping is realized as either a nested class or a function depending on whether state is needed to handle repeated or premature execution.

4 Scheduling

A declarative transformation can always execute using the naive polling schedule

- Retry loop - loop until all work done
- Mapping loop - loop over all possible mappings
- Object loops - multi-dimensional loop for all object/argument pairings
- Compatibility guard - if object/argument pairings are type compatible
- Repetition guard - if this is not a repeated execution
- Validity guard - if all input objects are ready
- Execute mapping for given object/argument pairings
- Create a memento of the successful execution

This is hideously inefficient with speculative executions wrapped in at least three loops, guards and mementos when compared to a simple linear 'loop' nest.

The key functionality of a declarative model transformation compromises a suite of OCL expressions that relates output elements and their properties to input elements and their properties using meta-models to define the type system of the elements and properties. The suite of OCL expressions is structured as mappings that provide convenient modules of control.

The meta-model type system enables producer/consumer relationships to be established between mappings. These relationships can be used by an intelligent 'Mapping loop' to avoid many of the retries that result from careless attempted execution of consumers before producers. Similarly, considering only type compatible objects in the 'Object loops' eliminates another major naivety.

Analysis of the types alone is of limited value, since each type many have many properties. Assignments to properties must be analyzed separately to avoid deadlock hazards if, for instance, the schedule attempts to assign both forward and backward references of a circular linked list at once. Therefore a declarative mapping that appears to assign all properties at once, must be broken up into smaller micro-mappings to avoid a deadlock hazard. In practice fewer than 10% of mappings appear to need partitioning into micro-mappings, and once a valid schedule has been established some of these can be merged.

The 'Object loops' can be very inefficient even when restricted to type compatible objects, since the number of permutations grows exponentially with the number of inputs. For genuinely independent inputs this cost is fundamental and the only solution is for the programmer to provide a better algorithm that exposes a dependence. In practice many objects have a very strong relationship such as between a parent and a child object. From the parent there may be many child candidates, but from the child there is only zero or one parent object so we can replace the parent as an externally searched input by an internally derived computation. This reduces the order of the input permutation. In practice between 90% and 100% of micro-mappings require only one input.

A micro-mapping either executes successfully updating its outputs, or fails completely updating nothing until a later successful re-execution. Failure occurs whenever the computation prematurely navigates from one object to another. We therefore want to identify as many of the objects that a micro-mapping may

access in order to provide a static schedule in which the required objects are computed before use. Of course at compile-time we have no instances, just types which only give us limited producer-consumer precision. However the constraints in a declarative transformation define patterns that enable us to relate multiple types. Our scheduling analysis therefore starts with a transliteration from partitioned QVTc as shown in Figure 2 to graph form as shown in Figure 3

```
map classToTable in umlRdbms {
    check uml(p : Package, c : Class |) {}
    enforce middle(p2s : PackageToSchema|) {
        realize c2t : ClassToTable
    |}
    enforce rdbms(s : Schema|) {
        realize t : Table, realize pk : Key, realize pc : Column
    |}
    where(p2s.umlPackage = p; p2s.schema = s; c.namespace = p;) {
        c2t.owner := p2s; c2t.umlClass := c; c2t.table := t;
        c2t.name := c.name; c2t.primaryKey := pk; c2t.column := pc;
        t.kind := 'base'; t.schema := s;
        pk.owner := t; pk.kind := 'primary';
        pc.owner := t; pc.keys := Set(SimpleRDBMS::Key){pk}; pc.type := 'NUMBER';
    }
}
```

Fig. 2. Class-to-Table mapping in partitioned QVTc.

The example is taken from the ubiquitous UML to RDBMS example and shows the mapping from a Class (child of a Package) to a Table (child of a Schema) and since we are using QVTc the intermediate trace is programmed explicitly as a ClassToTable. The textual representation in Figure 2 shows a two input (p, c), two output interface (p2s, s) and four creations (c2t, t, pk, pc). The parentheses of the where clause show three pattern matching constraints. The braces of the where clause show thirteen assignments.

The graphical form is inspired in part by Henshin [1] but with additional colors. Black indicates what is constant at compile-time. Blue shows what is loaded from input models. Green indicates what is created. Additionally Cyan identifies what is required as input to this mapping after it is created elsewhere.

Solid edges show something-to-one unidirectional navigation paths between rectangular class instances and their rounded-rectangular attributes. Nodes are annotated with instance name and type, edges with property name and multiplicity. Dashed edges and ellipses show computations rather than navigations.

Bidirectional navigations are shown as pairs of unidirectional edges, but only something-to-one navigations are shown. Consequently any navigation edge may be traversed in its forward direction to the precisely one object that matches the pattern element. Following all the paths in this way identifies that all navigable objects can be reached unambiguously from the c:Class input. This is drawn with a very thick boundary to emphasize its importance. What appeared to be a 4-input mapping in textual form is actually a 1-input mapping, so what

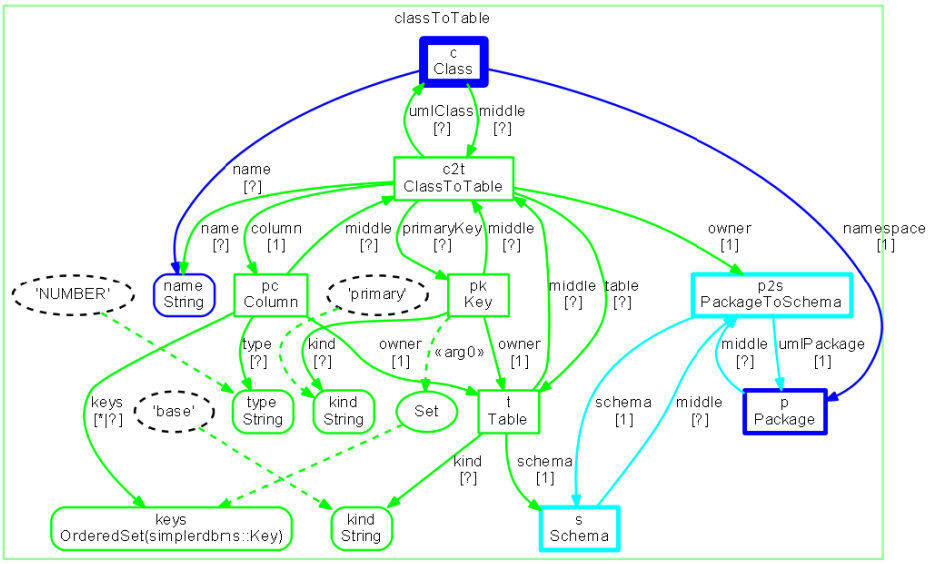

Fig. 3. Class-to-Table mapping in graphical QVTs form.

appeared to challenge the scheduler with a 4-dimensional search is realizable with a 1-dimensional loop.

The graphical form therefore clearly shows in blue that we need to match a c:Class in the loaded input model that has a Package at c.namespace and a String at c.name. Additionally the cyan shows that execution is not possible until p.middle provides p2s and p2s.schema provides s. Once these are all available, the many objects and relationships shown in green can be computed. We therefore have a very simple dependency rule, every cyan node or edge must be created by a corresponding green node or edge in another mapping.

In practice it is not quite so simple because we must account for multple producers and derived types. But in principle we can just join all the mappings up with dependencies to derive the schedule as shown in Figure 4[2].

At the top, the root pseudo mapping is a producer of input model elements, triaged as allInstances of interesting types. These are consumed by the 19 rectangles one for each merged micro-mapping of the transformation.

Solid edges pass model elements to each consuming input. Simple connections are drawn directly. Multiple producers or multiple consumers are drawn through an intermediate ellipse which corresponds to a buffer that accumulates inputs from many sources and then makes them available to many consumers.

Dashed edges show the dependencies that validate computation of the not-passed value. Thus for the ClassToTable example, a Class is passed via a solid

[2] One rectangle, two ellipses and associated lines have been removed so that the nodes and edges are discernible; the lettering is not relevant.

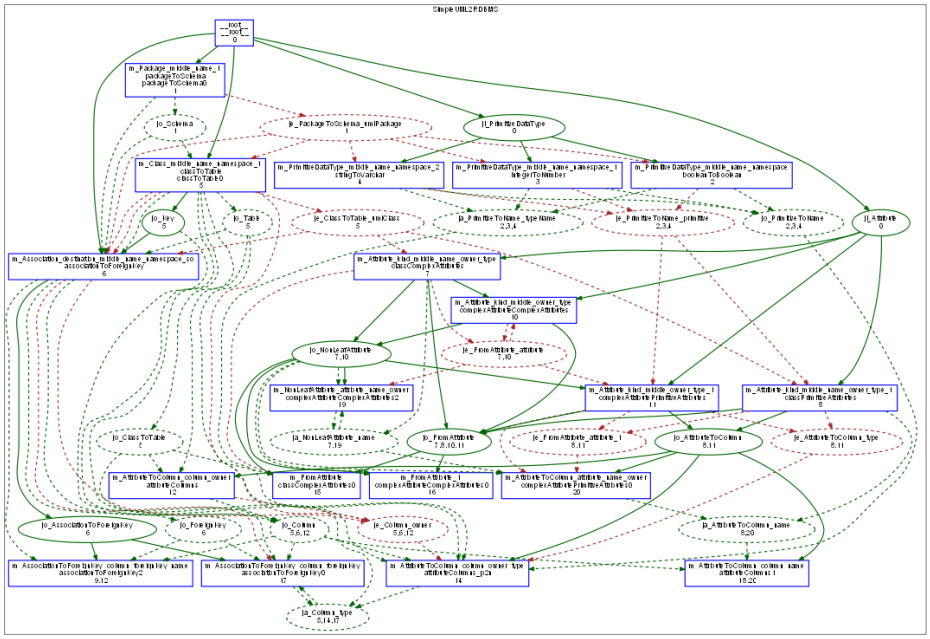

Fig. 4. UML2RDBMS schedule in graphical QVTs form.

edge; the other three inputs are computed by navigation from the one input. This computation will fail unless the dependencies have been satisfied.

The high proportion (75%) of dashed edges and ellipses shoes to a useful compile-time optimisation that eliminates parameters and intermediate buffers.

A sequential schedule is derived by allocating an integer slot index to each micro-mapping in turn such that all its predecessors have smaller indexes. In practice, a perfect static schedule can often be derived, but sometimes a recursion may introduce a cyclic dependency that can only be resolved at run-time. For this small proportion of micro-mappings additional run-time support is activated so that a failed speculative execution blocks according to the failure and retries only once the failure cause has been resolved. Few mappings incur multiple failures and so in practice the retry overhead is less than 50% and only where the data relationships exceed the compile-time analysis capability.

Three of the nineteen UML2RDBMS micro-mappings require two inputs. This incurs a quadratic tooling algorithm inefficiency that shows up when the performance is measured for input models containing associations. For just packages, classes and properties the execution scales linearly with Workload. Examination of these three mappings reveals that there is a 1:N relationship between a primary and a secondary part of the mapping. A comparatively simple optimization can therefore treat the primary input as the one input and derive the secondary input using a local loop. This local loop can iterate only over matches, whereas the current external global loop iterates over many mismatches.

This demonstrates how the combination of meta-model constraints and graph form exposition facilitate the identification of scheduling optimizations that alleviate bad algorithmic overheads.

5 Results

As just described, the performance of the QVTc variant of UML2RDBMS scales linearly when the input model contains just packages, classes and properties, but goes quadratic once associations are added. The further work to fix this was outlined.

In this section we plot the performance of the simple Familes2Persons transformation that involves a two-way guarded decision around a copy. The plot demonstrates the scale-ability and assess the underlying tooling efficiency.[3]

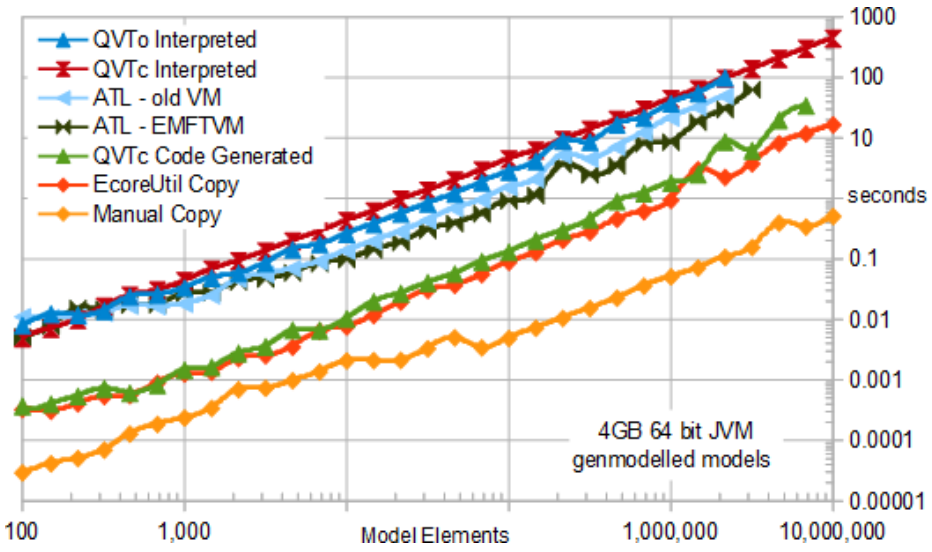

Fig. 5. Performance of the simple Familes2Persons transformation.

The top two lines of Figure 5 show the performance of the new interpreted QVTc and contrasts it with the Eclipse QVTo code performance. There is very little difference, except that QVTo has a much higher trace overhead and so fails to execute for more than 2,500,000 elements; an unexpected advantage of the explicit QVTc trace.

The two lines slightly below these show the old and new ATL VMs. Both fail to get close to 10,000,000 elements in the 4GB 64 bit VM.

[3] The plots use no averaging. Single point wobbles may be due to concurrent activity. Multiple point wobbles may be due to fortuitous cache alignment.

The next two lines are about 30 times faster and contrast the new code generated QVTc with the standard EcoreUtil copy functionality. QVTc is currently about 20% slower and just fails to reach 10,000,000 elements.

The final line is a manually coded implementation of the copy that bypasses the overheads of dynamic EMF in similar fashion to the code generated QVTc. For large numbers of model elements, this is about twenty times faster indicating that there is considerable scope for further improvement using the current approach. Using a non-EMF mode representation can give further improvements and moving to C execution yet more.

Failure of such a simple transformation at around 10,000,000 model elements in a 4 GB VM is disappointing. It suggests a 400 bytes per model element cost. Use of a modern 64 bit processor incurs an 8 byte cost for every thing, so 400 bytes is 50 things per model element. Each model element involves a String and a few pointers. For each model element, there is an input, an output and an intermediate object, pointers to them and also a surprisingly large HashMap node. Perhaps a 25% saving is available with smarter implementation. Optimizing the representation to suit the transformation can save much more. An even larger saving may be made by trading off size for speed so that one, two or four byte values are used rather than eight bytes always.

6 Related Work

The idea of generalizing the concepts of a Java Virtual Machine to a Modeling Virtual Machine is attractive and has influenced ATL [2]. However while its original representation had small byte-code like concepts, they were far from byte-sized. The newer EMFTVM is therefore able to do significantly better [6], but retains a transformation structure that supports transformation with OCL expressions as an afterthought. EMFTVM is 80% slower than its EcoreUtil 'optimum'. In contrast the OCL VM uses the OCL AS as its 'byte-code' and supports extension to the QVTi AS so that all AS elements form part of a single executable element hierarchy. The performance reported in the previous section was 20% slower than EcoreUtil but the 'optimum' target for further work is ten times faster than EcoreUtil.

Code generation has not been pursued by other transformation engines, perhaps because OCL code generation is not easy. Superficially there are similarities between OCL and Java and so a number of researchers have performed simple text template mapping [8]. This works sometimes, but fails to handle OCL in general. In contrast the Eclipse OCL Java code generator converts the OCL AS to an intermediate CG AS upon which a number of optimizations and rewrites are performed before Java code is emitted. The code generator supports all OCL constructs and its extensibility is demonstrated by its re-use for QVTi.

The Graph Transformation community has been very active in providing a rigorous foundation for graph mappings. Sadly the QVT specification ignored this important work, preferring instead to define the semantics of the QVTr transformation language using an incomplete exposition of a transformation of

QVTr written in an untested QVTr to another language (QVTc) that has at best informal semantics. The utility and power of the graphical QVTs representation may begin to bridge the gap between these two communities. The coloring in QVTs is inspired by the use of colors to denote create/delete/no-change in endogenous transformations. The reification of the QVTc traceability element mirrors the evolution operators in UMLX [9] for heterogeneous transformations.

Active Operations [7] reify also mappings at run-time so that the state necessary for incremental execution can be persisted. Micro-mappings similarly support incremental execution, but their primary rationale is to be a deadlock-free unit of computation.

7 Conclusions

We have introduced the first implementation of the QVTc specification.

We have introduced the first direct code generator for model transformations.

We have shown that the direct code generator gives a thirty fold speed-up.

We have shown how the combination of meta-model and dependency analysis aided by a graph presentation enable the naive inefficiencies of a declarative schedule to be tamed.

References

1. Biermann, E., Ermel, C., Schmidt, J., Warning, A.: Visual Modeling of Controlled EMF Model Transformation using HENSHIN Proceedings of the Fourth International Workshop on Graph-Based Tools, GraBaTs 2010.
2. Eclipse ATL Project. https://projects.eclipse.org/projects/modeling.mmt.atl
3. Eclipse EMF Project. https://projects.eclipse.org/projects/modeling.emf.emf
4. Eclipse OCL Project. https://projects.eclipse.org/projects/modeling.mdt.ocl
5. Eclipse QVT Declarative Project. https://projects.eclipse.org/projects/modeling.mmt.qvtd
6. EMFTVM performance. https://wiki.eclipse.org/ATL/EMFTVM#Performance
7. Jouault, F., Beaudoux, O.: On the Use of Active Operations for Incremental Bidirectional Evaluation of OCL 15th International Workshop on OCL and Textual Modeling, Ottowa, 2015
8. Wilke, C.: Java Code Generation for Dresden OCL2 for Eclipse, February 19, 2009. http://claaswilke.de/publications/study/beleg.pdf
9. Willink, E: UMLX : A Graphical Transformation Language for MDA Model Driven Architecture: Foundations and Applications, MDAFA 2003, Twente, June 2003. http://eclipse.org/gmt/umlx/doc/MDAFA2003-4/MDAFA2003-4.pdf
10. Willink, E.: Yet Another Three QVT Languages. ICMT 2013 (2013)
11. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3 Beta. OMG Document Number: ptc/15-10-02, March 2016.

Model-driven Video Decoding: An Application Domain for Model Transformations

Christian Schenk, Sonja Schimmler, and Uwe M. Borghoff

Computer Science Department
Universität der Bundeswehr München
85577 Neubiberg, Germany
{c.schenk,sonja.schimmler,uwe.borghoff}@unibw.de

Abstract. Modern serialization formats for digital video data are designed in a way that they allow for the combination of high compression rates, high quality as well as performant en- and decoding. As the capability for real-time decoding often is a requirement, concrete decoders are usually implemented in a way that they best fit a specific architecture and, thus, are not portable. In scenarios, where it is essential that the capability to decode existent video data can be retained, even if the underlying architecture is changed, portability is more important than performance. For such scenarios, we propose decoder specifications that serve as templates for concrete decoder implementations on different architectures. As a high level of abstraction guarantees system-independence, we use (meta)models and model transformations for that purpose. Consequently, every decoder specification is (partially) executable, which simplifies the development process. In this paper, we describe our concepts for model-driven video decoding and give an overview of a prototypical implementation.

Keywords: model driven engineering; models; model transformations; video decoding

1 Introduction

In our current work, we are focusing on the problem of preserving digital video content. One challenge is to ensure that today's data can be restored on future architectures. There are several approaches that address this problem for general content [1]. One widely used approach is to use standard file formats that are likely to be readable on future architectures (such as PDF/A for documents and JPEG 2000 for images). But, to our knowledge, there is no such standard format for digital video content that is suitable for long-term preservation. The existing formats that are widely used (e.g., H.264 [4]) are designed for another purpose: they combine high compression and high quality, and they allow for the development of efficient en- and decoders. As these formats involve the use of lossy compression techniques, it is impossible to transform an already encoded digital

video into another format without risking an additional loss of information. As this implies that we should not introduce a new standard format, we now focus on the question of how decoding capabilities can be retained. As common decoder implementations are normally system-specific and thus not portable, we propose the definition of human-readable and machine-processable decoder specifications that serve as a template for the development of suitable decoders on different systems. Our goal are specifications that are partially executable and that allow a potential developer to create code (possibly supported by tools) that serves as a basis for the implementation of a functional (but possibly inefficient) decoder prototype for any existing video compression standard. As the capability to decode a digital video can be retained this way, existing videos can be preserved without any (additional) loss of information.

In our previous work [10], we introduced the general idea of our approach. In this paper, we are focusing on details of a protypical implementation that serves as a proof of concept. We will show how we combined (meta)models, model transformations, R scripts [9] as well as a control program in order to specify essential parts of the decoding process for H.264 encoded videos, and we explain how these parts constitute the basis for the implementation of an H.264 decoder. As digital video data tend to require lots of memory, we will also introduce our approach as a potential application domain for testing MDE tools and techniques in combination with large models.

The remaining paper is structured as follows: in section 2, we give a brief overview of video coding basics that are needed for the following sections. In section 3, we give an overview of our approach and describe some aspects in more detail. In section 4, we give an overview of related work before we conclude our work and give an outlook in section 5.

2 Serialization Formats for Digital Video Data

Usually, every (digital) video consists of *frames*, which, when presented consecutively for a fixed period of time, create the feeling of movement. Each frame mainly contains the image data and some time information (called the *composition time*) enabling a video player to play the video correctly. Due to the needed memory, storing a complete video using well-known image file serialization formats (such as png, jpg or bmp) is no option. Instead, in modern video compression formats, such as the H.264 standard [4], different compression techniques are combined in order to allow for suitable file sizes. Lossy compression algorithms play an essential role. Simply said, these algorithms do not distinguish between "similar" values; when stored, they are just handled equally. Consequently, these algorithms are irreversible, i.e., video encoding generally implies a loss of information. The H.264 standard, which is widely used and which will serve as a running example in this paper, is briefly introduced in the following.

2.1 H.264 basics

The H.264 standard is used for video streaming scenarios, in combination with Blue-Ray disks and, of course, also for locally stored video files. As it can be regarded as a de facto standard, we focus on H.264 encoded videos for our considerations. As other standards are specified similarly if not identically (e.g., MPEG-4 AVC), we assume, however, that it is straightforward to use our approach for these standards.

An H.264 video track is structured in *samples*, which are (usually) grouped into *chunks*. Each sample can be seen as a (still encoded) representation of a single frame in the video, i.e., it contains every information that is needed to reconstruct a two-dimensional field of pixels. Pixel data are usually stored using the YCbCr color model, which distinguishes between one luma (Y) and two chroma channels (Cb and Cr), whereby each channel is encoded independently.

Compression basics: Principally, each sample is encoded using lossy image compression. However, in addition to that, *delta compression* is used in order to remove redundancies caused by pixel similarities between successive frames and (spatially) nearby regions within one frame. We briefly explain how it works: in case of a single frame, it is common that pixels are similar or even equal if compared with neighboring pixels. Furthermore, two successive frames often only differ in details (in order to allow for smooth frame changes). The delta compression's main principle is to just store "quantified" difference values between two similar pixels. The quantification, which is a simple integer division, just increases the desired effect: the resulting values tend to be smaller and occur more often. (By the way, the fact that the integer division is not completely reversible is one of the reasons why the complete compression is lossy.)

In an H.264 encoded video, every sample is divided into a grid of 16 x 16 blocks (called macroblocks), which are traversed row by row and column by column during the decoding process. Each macroblock may depend on one or more other macroblocks whose data already have been decoded before. When a macroblock is decoded, the data of its dependencies are used to first *predict* [4] an intermediate data representation. Afterwards, the explicitly stored difference values are added in order to restore the macroblock's actual data. Macroblocks that only depend on data of neighboring macroblocks (within the same frame) are called *intra-predicted*, whereas most of the macroblocks of an H.264 encoded video are usually *inter-predicted*, i.e., they refer to arbitrary regions of other frames using *motion vectors* (in combination with frame ids).

Inter-predicted macroblocks automatically cause inter-dependencies between different frames. In order to constrain possible inter-dependencies (which is essential when a video is not decoded from the beginning), the sequence of all frames are partitioned into groups of pictures (called GOPs), whereby two frames can only depend on each other if they belong to the same GOP. Consequently, every GOP contains one or more (intra-predicted) frames that do not depend on any other frame and thus only consists of intra-predicted macroblocks. All the

other (inter-predicted) frames contain inter-predicted macroblocks and depend on one or more other frames.

Even if the delta compression plays an essential role for the complete encoding, it is more a preparation for the actual compression: the results of the delta compression are compressed using variable length encoding (e.g., huffman encoding), which ensures that more frequent values are transformed into smaller codes than values that are less frequent.

Decoding order and composition order: An H.264 encoded video permits sequential decoding, i.e., forward jumps are generally not necessary during the decoding process. A frame can only be restored if all the frames it depends on have already been decoded. Hence, if a frame A depends on a frame B, B must be stored before A within the serialization. Nevertheless, the H.264 standard allows frames to depend on other frames that have a greater composition time, i.e., which have to be displayed later during the playback. That is why we have to distinguish between the *decoding order* and the *composition order*: frames are serialized in decoding order but have to be presented in composition order. Thus, after their restoring in decoding order, all the frames have to be put into composition order.

3 Model-driven Video Decoding

In this section, we will give an overview of our approach of model-driven video decoding. We will further give some details of the model-based parts.

3.1 Overview of the Approach

Regardless of the actual scenario and the further processing, the main task of any decoder is to transform an encoded binary representation (e.g., an H.264 serialization) into a sequence of frames. Therefore, we have decided to use a suitable abstraction of that principle as a basis for our approach.

As illustrated in Fig. 1, the model-driven decoding process is divided into 5 phases: The *modeling phase* converts the original video into a model representation that corresponds to the H.264 metamodel, which formalizes H.264 content (see Fig. 2). The *preparation phase*'s purpose is to partition the complete work into independent "parts", which we call *task chains* in the following. Each task chain contains all the information that is needed to decode one GOP. In the *execution phase*, every task chain is actually decoded, i.e., the dependencies are resolved, and the pixel data are restored. In the *finalization phase*, the result of the execution phase is sorted in accordance to the composition order and transformed into a model that corresponds to the *frame sequence metamodel*, which formalizes general video content (see Fig. 3). The *unmodeling phase*'s purpose is simple: it transforms such a model into the final result, i.e., into a sequence of concrete frames.

In order to describe the decoding process in an architecture-independent way, our decoder specifications unify different abstraction mechanisms. As the

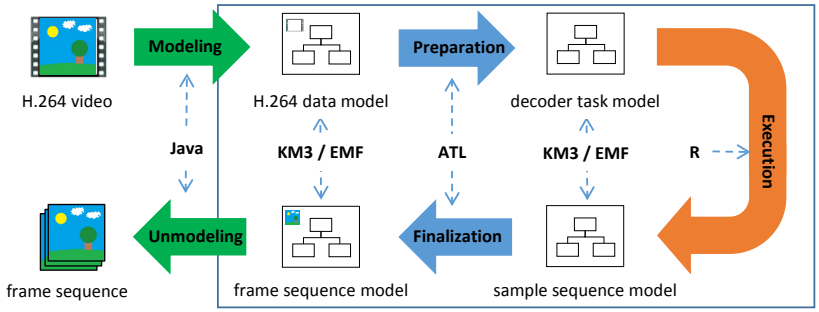

Fig. 1. Model-driven video decoding

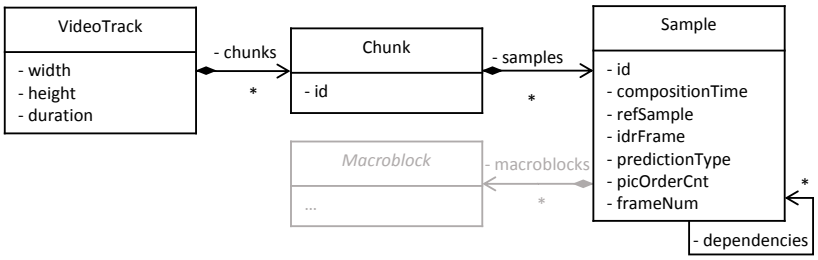

Fig. 2. H.264 input metamodel - due to clarity reasons, details of the abstract class `Macroblock` (colored in gray) and all its subclasses have been omitted.

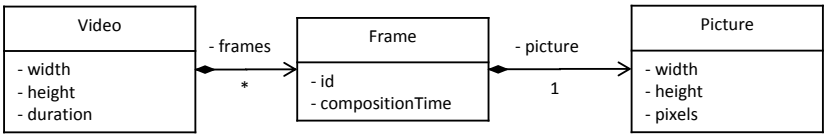

Fig. 3. Frame sequence metamodel

modeling and the unmodeling phase essentially perform pre- and post-processing steps, which depend on the underlying architecture and on the concrete scenario, they are not in the scope of such a specification.

In the next section, we give some details of how the three remaining phases have actually been implemented.

3.2 Details of the Approach

For the formalization of all the intermediate data representations, which serve as in- and output for the different phases, we use EMF-based [11] metamodels. Consequently, the complete decoding process may be regarded as a series of model transformations. Indeed, we use model transformations as the means to

formalize the preparation and the finalization phase. Within the execution phase, however, the inter- and intra-predicted values must be determined in order to restore the pixel values. As this process involves different mathematical calculations, we have decided to describe this phase using a mathematical abstraction. As we want our decoder specifications to be human-comprehensible as well as machine-processable, we have decided to use languages that are text-based. In summary, we use ATL [5] model transformations, KM3 [6] defined metamodels as well as R scripts [9]. We will give some details in the following:

KM3-based metamodels: Being convenient to define EMF metamodels using simple text, the KM3 language has been used to define all the necessary metamodels.

For the preparation phase, for example, we have defined 5 metamodels, which are used for the formalization of 7 (internal) models. The following excerpt shows the definition of the class `Frame` of the frame sequence metamodel (see Fig. 3).

```
class Frame {
  attribute frameId : Int32;
  attribute compositionTime : Int64;
  reference picture container : Picture;
}
```

ATL model transformations: The transformations of the preparation and the finalization phase are written in ATL, using mostly its declarative language features. We have decided to use ATL because it provides declarative but also imperative features, is based on EMF, is well integrated into the Eclipse IDE and can also easily be executed programmatically. Generally, every language that fulfills these requirements is an appropriate alternative for ATL (such as the Epsilon Transformation Language (ETL) [8]).

For the preparation phase, for instance, we have defined 5 ATL transformations, containing 16 rule definitions (with a total of about 190 LOCs). One example is a transformation that creates a model that explicitly stores the GOPs. The following excerpt shows the responsible ATL rule:

```
rule VideoTrack2Video {
  from
    vt : H264!VideoTrack
  to
    v : GOP!Video (
      gops <- vt.gopSmpls->collect(smpls | thisModule.createGOP(smpls))
    )
}
```

The ATL *helper* `gopSmpls` does the actual partitioning and determines a sequence of sample sequences representing the GOPs.

R scripts: For the abstraction of the mathematical operations we use R scripts. The R language was originally designed for statistical calculations. We have chosen to use it for our approach as it allows us to express and execute all

the mathematical operations that we need for the decoding (including matrix operations). Principally, every language that provides basic as well as matrix operations, e.g., Octave[1], would be a suitable alternative.

For the determination of the luma and chroma values, for instance, we have defined 25 R scripts (with a total of about 600 LOCs). The following excerpt, for example, shows how the luma values of a macroblock are determined. The variable `prediction` contains the intra-predicted values, whereas the parameter `residual` contains the (pre-processed) delta values. (The function `Clip1Y` simply ensures that the result is in the correct range).

```
for (x in 1:16) {
  for (y in 1:16) {
    values[x, y] <- Clip1Y(residual[x, y] + prediction[x, y])
  }
}
```

DSL-based control program: For coordination, we use control programs written in a DSL that we have defined with the framework Xtext [2]. Xtext is a framework that allows for the definition of DSLs (and the generation of corresponding tools) based on a grammar specification. Such a control program constitutes the frame of the specification as it defines how (meta)models, transformations and R scripts are connected. The following example shows how metamodels and models are declared within the the control program:

```
metamodel MM_H264 origin h264.km3.ecore
model H264 conforms to MM_H264 origin in.h264
```

The next excerpt shows how a transformation `H264ToGOP` is introduced, which is stored in `H264ToGOP.asm` and transforms the model `H264` into a model `GOP`.

```
transformation H264ToGOP origin H264ToGOP.asm {
  input model H264
  output model GOP
}
```

3.3 Implementation Status

As explained before, essential steps of the decoding work have already been specified using different abstraction mechanisms. Each step we have abstracted so far was originally implemented in form of a Java tool set; consequently, we have a reference implementation permitting us to generate suitable test data for every abstracted decoding step. We have developed a Java library that allows us to access and extract all information of an H.264 encoded video. This library is used for the implementation of the modeling phase. Furthermore, we have developed a Java application that is able to decode every intra-decoded frame

[1] Project URL: www.gnu.org/software/octave

of an H.264 encoded video file (stored in the mp4 file format). We have also developed a Java tool that can process control program files.

Up to now, the specification of the preparation phase is complete. In combination with the control program definition, an H.264 model can automatically be transformed into a decoder task model (containing the task chains).

We have also already defined all the necessary R scripts that are needed to decode intra-predicted macroblocks, and we have tested them with our Java application, which uses the open source library renjin[2] for interpreting the R code. In a next step, we want to integrate the execution phase into the decoder specification. The missing link is a conversion of the model-based representation into an R-compatible input format. Here, we want to check if this conversion can be implemented using languages that allow for the specification of model-to-text transformations (such as the Epsilon Generation Language (EGL), which is related to ETL [8]).

The finalization phase has also already been specified, but will certainly need an update when the execution phase is integrated.

3.4 Scalability

We have chosen abstraction mechanisms that are executable in order to simplify the development process of functional (but not necessarily efficient) decoders for arbitrary architectures. Supposing that the most effective optimization measures are system-specific and therefore decrease the level of abstraction, we generally have neglected any performance considerations within the design of the specifications.

For practical reasons, we have made one exception: all the models within the preparation phase do not contain any macroblock data as they are not needed before the execution phase. Consequently, any H.264 model (that conforms to the metamodel illustrated in Fig. 2) does not contain instances of the class `Macroblock` (and its subclasses). As these instances contain (i.e., within referenced objects) the information to restore the actual pixel data, the size of H.264 models can drastically be reduced.

For an evaluation, we generated the H.264 models for three different videos and determined their sizes before and after removing the macroblock data. As models can be regarded as graphs, we simply counted the number of nodes and edges to specify the models' size. Furthermore, we measured the time it took to execute the preparation phase and determined the percentage of this time spent on the model transformations (MTs). Finally, we measured the time it took to decode all the intra-predicted frames (I-frames) when using the renjin-based Java application. The results are listed in Table 1.[3]

Currently, the frames are decoded sequentially. But as a GOP can completely be decoded independently of other ones, decoding them simultaneously is assumed to crucially decrease the execution time of the decoding process.

[2] Project URL: www.renjin.org

[3] Used abbreviations: k for kilo ($\times 10^3$), M for mega ($\times 10^6$)

Table 1. Evaluation results for test videos (System: Debian 8, CPU: i7-4790 3.6 GHz, RAM: 32 GB)

	video 1	**video 2**	**video 3**
Frames (I-frames)	2540 (52)	8189 (273)	163327 (2373)
Resolution (w × h)	512 × 288	1920 × 1080	1280 × 720
Model graph's nodes/edges			
- original	21.6M/24.6M	1760M/1900M	14500M/15700M
- reduced	5.08k/13.0k	9.04k/24.6k	327k/646k
Preparation phase (MTs)	∼ 2 s (75%)	∼ 5 s (83%)	∼ 7 min (96%)
R-based decoding of I-frames	∼ 24 min	∼ 1 day	∼ 4 days

4 Related Work

Our approach utilizes MDE techniques in the domain of long-term preservation in order to address the issue of preserving digital video content in a way that it can be restored on future architectures. As far as we know, there are no similar approaches. In the following, we present two approaches based on image encoding standards.

Motion JPEG 2000 [3] is a part of the JPEG 2000 standard that simply uses the corresponding compression for each frame. As JPEG 2000 is already used for the preservation of digital images [7], a combination would principally allow for the preservation of digital video content. In another approach [12], digital videos are preserved using an XML-representation. Two variants are distinguished: first, video data are completely transformed into primitive XML. Second, the video frames are converted into image files that are suitable for long-term preservation (such as JPEG 2000) and referenced within the XML representation. Both variants involve the decompression of lossy compressed parts, which naturally leads to larger file sizes.

An important difference between our approach and the approaches named before is that we generally allow for delta compression, i.e., frames can have dependencies between each other. Avoiding delta compression simplifies the decoding, but results in larger file sizes. Contrary to our approach, the two approaches also imply recoding, which results in potential loss of information.

5 Conclusion and Outlook

In this paper, we have continued our presentation of a model-based approach for architecture-independent video decoding [10] with a focus on the model-based parts and its realization. At this point, our approach does not provide full support for the decoding of inter-predicted frames, and the conversion between models and R-compatible representations is still hard-coded in Java. Nevertheless, so far we have succeeded to find appropriate abstractions for all essential parts of the decoding process.

Our approach involves the processing of large models. Even if the efficiency aspect plays a subordinate role, optimizations, which do not impede the applicability, might improve the development process. Besides, the utilized models may also be used as potential test data for common MDE tools.

After finishing the specification of the H.264 decoding process, we plan to focus on the design of a serialization format that simplifies the implementation of the modeling phase, can be used to preserve H.264 encoded data without additional loss of authenticity and allows for "suitable" file sizes.

In a further step, we want to evaluate our approach by asking developers who are unfamiliar with the H.264 standard to implement a decoder based on our specification. In this context, we also want to find out whether and in which way our decoder specifications can also be used to simplify the development process by serving as a base for automatic code generation.

References

1. Borghoff, U.M., Rödig, P., Scheffczyk, J., Schmitz, L.: Long-Term Preservation of Digital Documents, Principles and Practices. Springer-Verlag Berlin Heidelberg (2006)
2. Efftinge, S., Völter, M.: oAW xText: A Framework for Textual DSLs. In: Eclipsecon Summit Europe 2006 (2006)
3. ISO/IEC: International Standard ISO/IEC 15444-3: JPEG 2000 Image Coding System - Part 3: Motion JPEG 2000. International Standard Organization (2002)
4. ITU-T: Recommendation ITU-T H.264: Advanced Video Coding for Generic Audiovisual Services. International Telecommunication Unit (2013)
5. Jouault, F., Allilaire, F., Bzivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(12), 31–39 (2008)
6. Jouault, F., Bzivin, J., Team, A.: KM3: A DSL for Metamodel Specification. In: Formal Methods for Open Object-Based Distributed Systems. pp. 171–185. Springer (2006)
7. van der Knijff, J.: JPEG 2000 for Long-term Preservation: JP2 as a Preservation Format. D-Lib Magazine 17(5/6) (2011)
8. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon Transformation Language. In: International Conference on Theory and Practice of Model Transformations. pp. 46–60. Springer-Verlag, Berlin, Heidelberg (2008)
9. Ross Ihaka, R.G.: R: A Language for Data Analysis and Graphics. Journal of Computational and Graphical Statistics 5(3), 299–314 (1996)
10. Schenk, C., Maier, S., Borghoff, U.M.: A Model-based Approach for Architecture-independent Video Decoding. In: 2015 International Conference on Collaboration Technologies and Systems. pp. 407–414 (2015)
11. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2nd edn. (2009)
12. Uherek, A., Maier, S., Borghoff, U.M.: An Approach for Long-term Preservation of Digital Videos based on the Extensible MPEG-4 Textual Format. In: 2014 International Conference on Collaboration Technologies and Systems. pp. 324–329 (2014)

Evaluation of Model Comparison for Delta-Compression in Model Persistence

Markus Scheidgen[1]

Humboldt Universität zu Berlin, Germany
{scheidge,marticke}@informatik.hu-berlin.de

Abstract. Model-based software engineering is applied to more and more complex software systems. As a result, larger and larger models with longer and longer histories have to be maintained and persisted. Already, a lot of research efforts went into model versioning, comparison, and repositories. Existing strategies either record and persist changes (change-based repositories, e.g. EMF-Store) or relay on existing text-based version control systems to persist whole model revisions (state-based repositories). Both approaches have advantages and disadvantages. We suggest a hybrid approach that infers changes via comparison to persist delta-compressed model states. Our hypothesis is that delta-compression requires a trade-off between comparison quality and execution time. Existing model comparison frameworks are tailored for comparison quality and not necessarily execution time performance. Therefore, we evaluate and compare traditional line-based comparison, an existing model comparison framework (EMF-Compare), and our own framework (EMF-Compress). We reverse engineered the Eclipse codebase and it's history with MoDisco to create a large corpus of evolving example models for our experiments.

1 Introduction

In *BigMDE*, we assume that model-based software engineering (MBSE) is applied to increasingly complex software systems and that the size of models that we need to process is getting increasingly larger. But not only the size of models is increasing, they also evolve over longer periods of time, and have longer revision histories. To cope with such evolving models, the MBSE community begun to adapt the concepts of existing text-file-based version control systems for models. Respective model repositories allow clients to navigate model history, compare revisions, and merge different branches of development [1]. We can distinguish two basic strategies for model repositories: state-based and change-based [8]. State-based systems store individual revisions of a model and whenever a user needs to compare two revisions the respective models are compared. In a change-based system, only the differences (or the operations causing these differences) are persisted and when a user requires a particular revision the respective model is re-created from prior revisions and respective difference data.

Both strategies have advantages and disadvantages. Change-based repositories, e.g. EMF-Store [8], require to record editing operations to infer and persist

changes. This requires a tight integration of editors and model repository. While this works well for certain editors like MVC-based graphical editors (e.g. GMF), it does not for background-parsing-based textual editors (e.g. Xtext), mixed-, or closed editing environments. For state-based repositories, we can use existing version control systems that were developed for managing text files [1]. Here, models are persisted in their serialized form, e.g. as XMI/XML files. While this is independent from the editing environment and works for all models that can be serialized without information loss, it imposes an unnatural organization scheme onto modeling. Clients have to organize large models in terms of directories and files. To be efficient, the respective files have to be reasonably sized.

We propose a third strategy that uses a change-based strategy internally, and expose the characteristics of a state-based system to the editing environment and the user. Git, for example, allows us to access file revisions as a whole, but stores many revisions of the same file in a delta-compressed *pack-file*. Let's persist model histories in terms of changes, but provide and take models as a whole. This requires us to create model differences through model comparison, since we can not rely on difference information provided by the editing environment. Similar to version control systems, this strategy does not force us to use the same comparison algorithms for storing models tightly (also know as delta-compression) and for presenting changes and similarities to users. We can use algorithms with good comparison quality and good performance characteristics internally for frequent delta-compression, and different algorithms with excellent comparison quality (i.e. minimal editing distance) and less performance to occasionally present changes to users.

In this paper, we want to evaluate comparison algorithms for delta-compression. We developed a framework called *EMF-Compress* [12]. We implemented signature-based matching strategies, far simpler than the similarity-based matching of existing frameworks like EMF-Compare. In contrast to existing comparison frameworks, we use a meta-model that allows us to persist differences (i.e. delta-models) between models without referencing the compared models directly. We conducted experiments with reverse engineered MoDisco [3] Java models that were taken from the Eclipse source-code history [13]. In these experiments, we compare compression quality and execution times for different matching algorithms and provide execution times for comparison (compression) and patching (de-compression) to proof the feasibility of our approach.

The paper is organized as follows. The following section 2 provides a brief introduction into model comparison. Section 3 introduces our delta-compression tailored comparison framework. The evaluation section 4 describes and discusses our experiments in compressing the reverse-engineered Eclipse code-base. We present related work, future research questions, conclusions in sections 5 and 6.

2 About Matches, Differences, Merges

In this paper, we only consider two-way comparison. We always compare a left (original) model with a right (revised) model. Algorithms for comparing two

text files (i.e. two list of comparable items) exist for a very long time. These algorithms are searching for the minimal editing distance, that is the smalles possible set of changes (adding lines, removing lines, changing lines) necessary to modify one file and get the other. One particular algorithm is *Meyer's* [11] algorithm that compares two list in $\mathcal{O}(N * D)$ where N is the sum of both sizes and D the number of changed lines.

Models are not list of items, but graphs of items. But, each model element contains lists of items, i.e. their attribute and reference values. Therefore, we could apply existing algorithms, like Meyer's, to each value-set in each model element. But in order to use this approach, we need to know which values (primitive values as well as other model elements) are supposed to be equal and which are not.

Therefore, existing model comparison frameworks use two steps. First, model elements are matched. Frameworks use different matching algorithms to establish pairs of matching model elements from both compared models. In a second step, these matches are used to find differences, comparing element by element, feature by feature. In such comparison processes, the matching strategy influences the comparison quality. Do we yield a minim set of differences or a large set of differences only depends on how good the matches are.

We can distinguish different types of matching-strategies. Among others: signature-based and similarity-based matching. Signature-based algorithms only consider local model element properties like it meta-class, name, parameters, and its position in the containment hierarchy (i.e. its parent). Similarity-based algorithms consider the larger neighborhood of an element to assess its shape in order to establish matching pairs. Finding and comparing signatures is a straightforward, linear process. Similarity-base algorithms on the other hand can be considerably different based on the used neighborhood, considered characteristics, used heuristics, etc.

Existing frameworks, like EMF-Compare, are tailored for presenting differences to users and for merging models. They use similarity-based matching for a maximum comparison quality and represent matches and differences in a comparison model. These comparison models reference both compared models. This is fine when the goal of a merge is to create a new merged model from two existing compared models. But in delta-compression, we want to re-create one model from the other and a respective difference model. This is also known as *patching* and the difference model is than called a *patch*.

3 Model Comparison for Compression

We developed the comparison framework *EMF-Compress* [12] with the goal to evaluate signature-based comparison for delta-compression. This means two things. First, we can perform matching and differentiating in one single step. In a signature-based matching algorithm, we consider the position of an element (i.e. its parent) to be part of its signature. As a consequence, a moved model element (i.e. a model element that changes its parent) is not matched with its

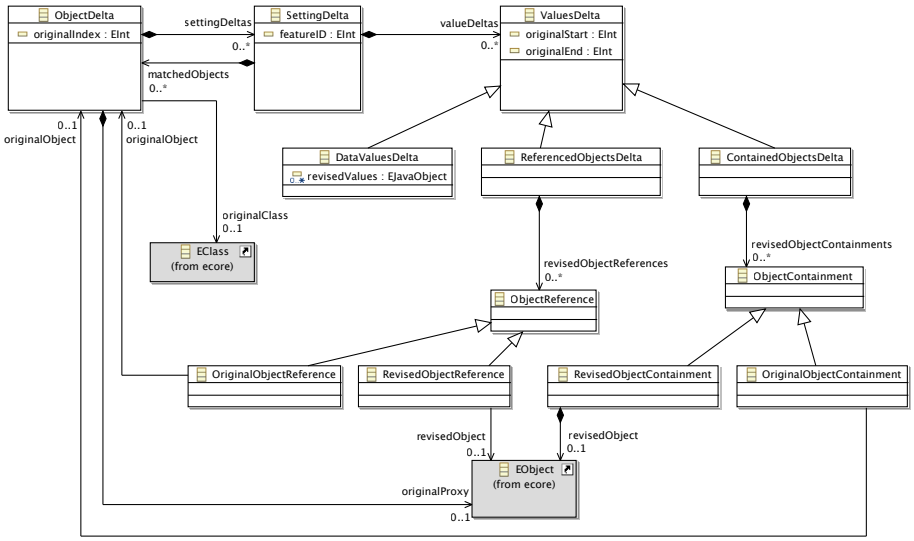

Fig. 1. Combined meta-model for matches and differences.

original. This is obviously bad for comparison quality, but we also only need to establish matches locally within individual value-sets. Therefore, we can match elements *on the fly* during differentiation. We start by assuming that the root elements match. We then apply Meyer's to all features. While Meyer's is applied, we establish matches among the values of each individual value-set. We recursively proceed for all matches in all containment references throughout the containment hierarchy.

Secondly, EMF-Compare uses a comparison meta-model that represent differences under the assumption that the right model is not available and that it needs to be re-created from the left model. This means the comparison model (depicted in Fig. 1) represents all differences as changes (i.e. deltas). An `ObjectDelta` represents all changes between two matched elements without referencing these elements directly. A `SettingDelta` is a container for differences (and matches) in a feature (via `featureID`). A `SettingDelta` can contain matching elements (`matchedObjects`) identified via index (`originalIndex`) and *ValueDeltas* that described changed ranges of values with the feature. Ranges are also identified via indices. There are different classes used to represent differences depending on the value type (primitive, contained object, cross-referenced object) and whether the value is part of the left (original) model or is a new value that is added to the right (revised) model. All object values that are added have to be contained in the comparison model, since it is supposed to be independent of the right (revised) model. All object values that are removed or moved within the model are referenced based the *ObjectDelta* instance that represent the object in the comparison model.

Therefore, the comparison model does not reference any of the compared model directly. The information in a comparison model is sufficient to recreate (i.e. patch) a right (revised) model from an left (original) model.

4 Evaluation

4.1 Setup

The subject for our experiments is a corpus that comprises a subset of the Eclipse Foundation's source code repositories. The Eclipse Foundation maintains code (and other artifacts, like documentation, web-pages, etc.) in over 600 Git repositories. We took the largest 200 of those repositories that actually contained Java code. These 200 repositories contain about 600 thousand revisions with a total of over 3 million *compilation unit* (CU) revisions that contain about 400 million SLOC (lines of code without empty lines and comments). The Git repositories take 6.6 GB of disk space. The generated MoDisco model representation of these repositories comprises over 4 billion objects and is persisted in EMF-Fragments [14] and a binary serialization format with about 230 GB of diskspace.

We run all experiments on a server computer with four 6-core Intel Xeon X74600 2.6 GHz processors and 128 GB of main memory. However, *EMF-Compress* operates mostly in a single thread with the exception of JVM and Eclipse maintenance tasks. *EMF-Compress* runs on Eclipse Mars' versions of EMF in a Java 8 virtual machine. All operations were run with a maximum of 12 GB of heap memory. The system runs on GNU/Linux with a 3.16 kernel. *EMF-Compress* operations are long running computations over thousands of revisions and CUs and respective algorithms are invoked over and over again. As usual for such macro-benchmark measures, we are therefore ignoring JIT warm-ups and other micro-benchmark related issues [16].

The use-case behind these experiments is a repository wide application of reverse-engineering (with MoDisco) for the purpose of analysis [13]. The goal of this model-based mining of software repositories (MSR) [4] approach, is an AST-level model of a large-scale software repository (e.g. Eclipse) that allows clients to derive metrics, dependencies, design structure matrices, and other data for statistical analysis over the history of many software projects. Not only would we benefit from lesser space requirements of the resulting models, delta-compression and comparison models would also allows us to omit unnecessary processing of unchanged model parts during model analysis.

4.2 Signature- v Similarity- v Line-based Comparison

As a first experiment, we want to measure and compare signature-based, similarity-based, and line-based comparison both by comparison quality and execution time. Please note, line-based comparison always refers to comparing the original source files and not any serialized representation of the models. The resulting

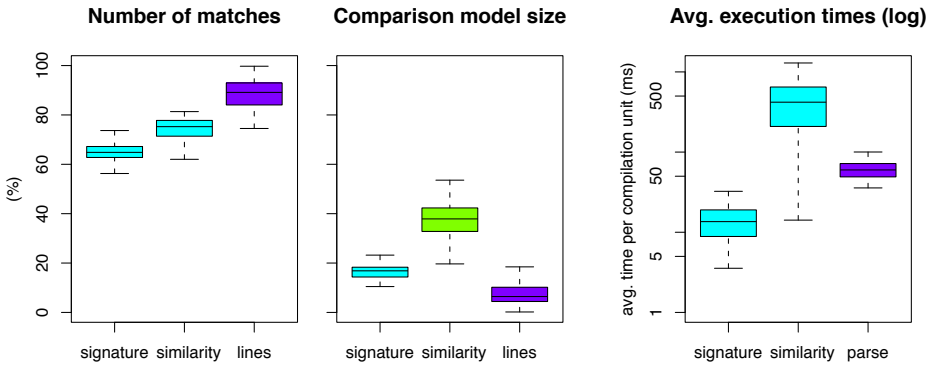

Fig. 2. Signature-based comparison with *EMF-Compress*, similarity-based comparison with EMF-Compare, and line-based comparison on the original source-code files.

line-based numbers are not really comparable, but provide a frame of reference. Due to the low runtime performance of similarity-based comparison, we had to limit our measurements to the first 1000 revisions of the 100 biggest (by Git-size) Eclipse projects. For signature-based comparison, we used our own framework *EMF-Compress*. The used signatures comprise parent, meta-class, and where applicable names. For similarity-based comparison, we used EMF-compare in its default configuration. For line-based comparison, we used the data provided by Git.

Fig 2 show the results. Different colors suggest data that is not directly comparable and that has to be read very carefully. The left chart depicts the comparison quality by means of matched elements. For models this is the number of actual matched elements. For lines this is the number of *matched* lines, i.e. lines that were not added, removed, nor changed. The chart shows the fraction of matched elements relative to the overall number of elements without the uncompressed initial revisions. The data is accumulated for each analyzed repository. As expected, similarity-based comparison produces good comparison quality, since it uses far more data to establish matches than signature-based matching. The signature-based matching quality is worse, but still of a similar magnitude. We cannot draw any sensible conclusions from comparing these model-based numbers with the number of matched lines, because lines span different numbers of model elements and shorter lines match more likely than longer lines.

The middle chart compares the size of the resulting comparison models in relation to the size of the original models. This data also does not include the uncompressed initial revisions. For models, we used the approximated size of a binary serialized representation of all models. For lines, we used the average model size per line for added lines and two bytes for removed lines to approximate sizes for a corresponding comparison model. Measuring the size of comparison models is not very informative, since the comparison models of EMF-Compare and *EMF-Compress* comprise different data. EMF-Compare expresses changes by

referencing to the differentiating model elements, while *EMF-Compress* stores changed features, indices, and includes all added values. Our line comparison model approximation does not include any overhead for organizing the differences.

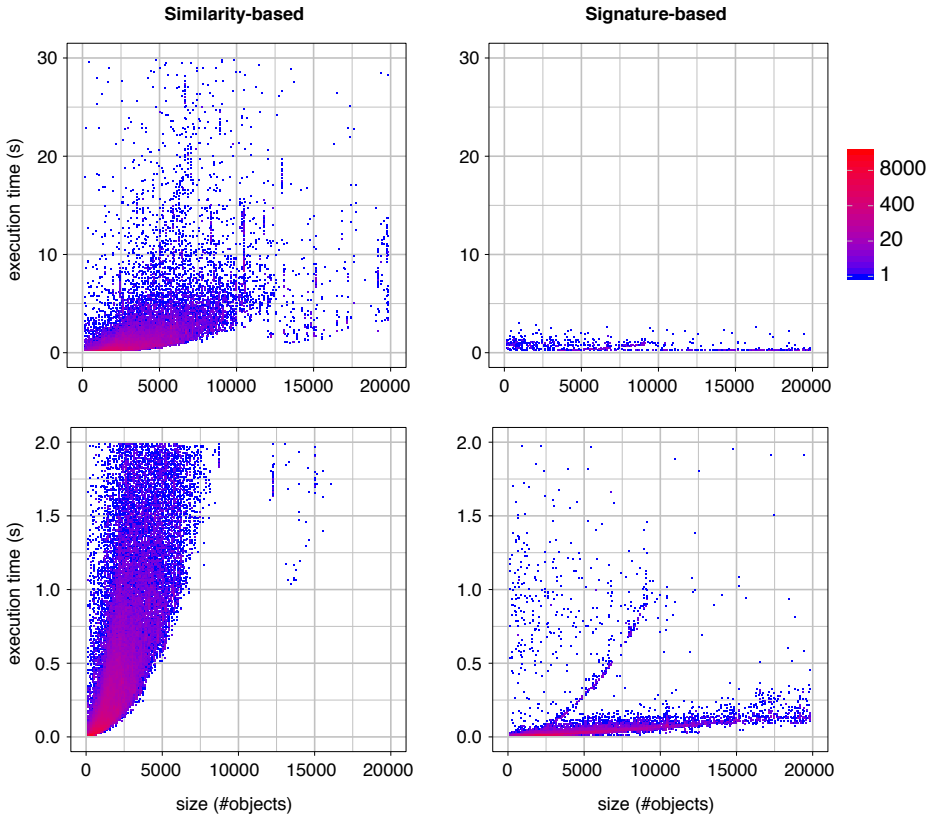

Fig. 3. Comparison execution time depending on model size for similarity- and signature-based matching. The plots show data measured for 300k compilation units taken from different Eclipse projects. Top row depicts all measurements, the lower row magnifies a smaller range.

The right chart of Fig. 4 show the average execution time spend on comparison in all repositories compared to the time that was necessary to create models for the original Java code. Similarity-based comparison is magnitudes slower than signature-based comparison. Fig. 3 provides a closer look on the execution times. This chart plots execution times spend on individual CUs in relation to their sizes in number of model elements. While signature-based comparison scales linearly for most CUs, similarity-based comparison is above linear. Addi-

tionally, the comparison times seem to depend on CU structure and not only CU size, since the measured times are particularly large for some CUs.

4.3 Top-Level Comparison v Deep Comparision

For a second experiment, we want to compare different levels of comparison. The hypothesis is that if we only try to match the first levels in a containment hierarchy and only use equality to compare the rest of the models, we can save execution time and still yield sufficient comparison quality. We devised two different matching algorithms. The first only uses top-level named elements and their signature for matching (e.g. Java packages, classes, methods). All other elements (i.e. everything contained in method bodies) only matches if their are equal to each other. The second algorithm tries to match all model elements based on their signature. We also provide line-based matching data for reference.

This time, we measure comparison for all 200 Eclipse projects, all revisions, all CUs. Fig. 4 shows the results. The top row of charts compares artifact sizes for some example Eclipse projects. The lower left shows the average size of uncompressed initial revisions and respective comparison model sizes relative to the overall size of the uncompressed models. The execution times are given as an average per revision for both compression algorithms and the times necessary to decompress (i.e. reconstruct a revision from its predecessor and the comparison model).

The comparison quality with only matching the named elements is significantly worse than with matching all elements. But more surprisingly, the execution time performance is also much worse. Even though, we do not have to match all model elements, we still have to compare them for equality.

5 Related Work

EMF-Store is an example for a change-based model version control system (Koegel and Helming [8]). An example for a model repository based on existing source-code version control systems is presented by Altmanniger et al. [1].

Different frameworks for model comparison exist: for example EMF-Compare by Brun and Pierantonio [2] or SI-Diff by Kehrer et al. [5]). Stephan and Cordy present a feature-based comparison of existing approaches and frameworks [15]. Kolovos et al. did the same for matching-algorithms [10].

In this paper, we only assumed meta-model agnostic matching-algorithms. But it is assumed that comparison quality can be increased, when matching is implemented depending on the meta-model of compared models. Kolovos et al. [9] and Kehrer et al. [5] present approaches for customization of matching algorithms.

In [6,7] Kehrer et al. research the posibilities to infer high-level editing operations from low-level comparison models. This can be valuable for presenting differences to users in a better way and to represent model differences/changes more efficiently in smaller delta-models.

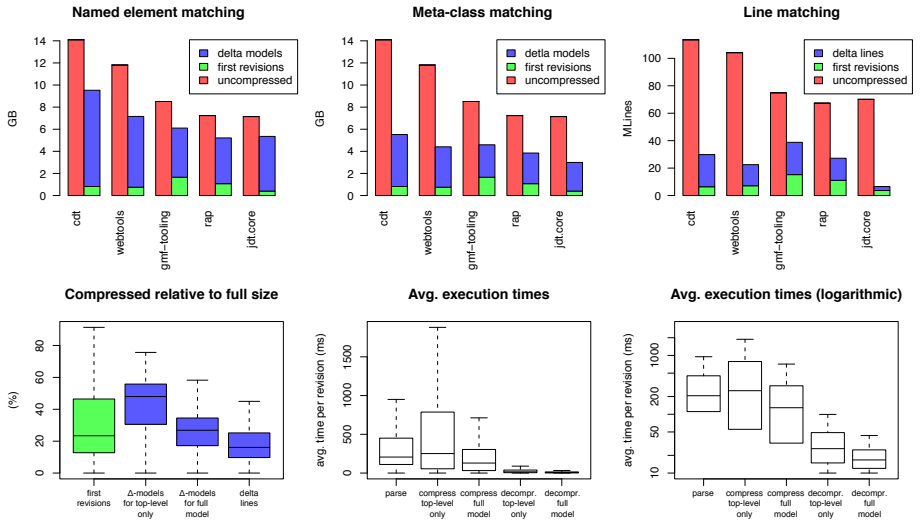

Fig. 4. Compression quality by percentage of full size and percentage of matched elements compared to line-based compression.

6 Conclusions

We evaluated the potential use of model comparison for delta-compressing state-based model repositories. We could show that signature-based matching algorithms reach sufficient comparison quality with significantly shorter execution times than the existing similarity-based model comparison in frameworks like EMF-Compare. Furthermore, we present a meta-model for comparison models that do not reference the compared models, can be persisted independently, and allows to re-create one of the compared models from the other. Therefore, we could proof the principle feasibility of time efficient delta-compression for models.

However, there are many unanswered questions. We only evaluated a single algorithm for similarity-based matching. Different algorithms, e.g. meta-model specific algorithms could yield better results. Further, we only considered model representations of source-code following a single meta-model. Different kinds of models could yield different results. What factors influence the performance of model comparison in what way? Delta-compression allows to decompress sequences of model revisions efficiently, but does not allow a random access of particular revisions. In a practical application, we need to consider to persist some intermediate revision as full model to counter that. What factors influence the right amount these intermediate models? The models that we used for evaluation have a convenient size. How can we combine delta-compression with existing approaches in fragmenting and persisting very large models [14]?

References

1. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. International Journal of Web Information Systems 5(3), 271–304 (2009)
2. Brun, C., Pierantonio, A.: Model differences in the eclipse modeling framework. UPGRADE, The European Journal for the Informatics Professional 9(2), 29–34 (2008)
3. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: Modisco: A generic and extensible framework for model driven reverse engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. pp. 173–174. ASE '10, ACM (2010)
4. Kagdi, H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of Software Maintenance and Evolution: Research and Practice 19(2), 77–131 (2007)
5. Kehrer, T., Kelter, U., Pietsch, P., Schmidt, M.: Adaptability of model comparison tools. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012 p. 306 (2012)
6. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings pp. 163–172 (2011)
7. Kehrer, T., Kelter, U., Taentzer, G.: Consistency-preserving edit scripts in model versioning. 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings pp. 191–201 (2013)
8. Koegel, M., Helming, J.: EMFStore: a model repository for EMF models. 2010 ACM/IEEE 32nd International Conference on Software Engineering 2, 307–308 (2010)
9. Kolovos, D.S.: Establishing correspondences between models with the epsilon comparison language. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 5562 LNCS, 146–157 (2009)
10. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different models for model matching: An analysis of approaches to support model differencing. Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models pp. 1–6 (2009)
11. Myers, E.W.: AnO (ND) difference algorithm and its variations. Algorithmica 1(1-4), 251–266 (1986)
12. Scheidgen, M.: EMF-Compress. https://github.com/markus1978/emf-compress (2016)
13. Scheidgen, M., Fischer, J.: Model-based mining of source code repositories. In: Amyot, D., Fonseca i Casas, P., Mussbacher, G. (eds.) System Analysis and Modeling: Models and Reusability, Lecture Notes in Computer Science, vol. 8769, pp. 239–254. Springer International Publishing (2014)
14. Scheidgen, M., Zubow, A., Fischer, J., Kolbe, T.H.: Automated and transparent model fragmentation for persisting large models. In: Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS). LNCS, vol. 7590, pp. 102–118. Springer, Innsbruck, Austria (2012)
15. Stephan, M., Cordy, J.R.: A Survey of Methods and Applications of Model Comparison (June) (2012)
16. Wilson, S., Kesselman, J.: Java Platform Performance: Strategies and Tactics. Addison-Wesley, Boston, MA (2000)

Managing Model and Meta-Model Components with Export and Import Interfaces

Daniel Strüber, Stefan Jurack, Tim Schäfer, Stefan Schulz, Gabriele Taentzer

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

Abstract. *Composite modeling* provides a modularity mechanism for models. To facilitate information hiding, it supports the declaration of export and import interfaces at the meta-model and model levels. In this paper, we present a tool set of wizards and editor extensions that makes composite modeling available to developers. The tool set is based on the Eclipse Modeling Framework. We demonstrate its use during the model-driven management of data-oriented applications.

1 Introduction

As the requirements imposed on Model-Driven Engineering (MDE) tools and techniques grow in complexity, so do the involved models, rendering their maintenance, transformation, and overall management highly challenging. To support developers during these tasks, appropriate modularity mechanisms are required [9]. Modularity is a fundamental software engineering concept with well-established principles that should ideally inform the design of such mechanisms. Two main principles are: (i) establishing a *separation of concerns* into distinct modules, (ii) facilitating *information hiding* and restricting *visibility* between modules by means of explicit interfaces.

In the MDE community, the Eclipse Modeling Framework (EMF) [12] is a widely used base technology. In EMF, a separation of concerns can be established by distributing information over a set of related models. Consider the left part of Fig. 1 for a pair of data models from operational systems for a tourism agency and an airline. The tourism agency model contains a trip assigned to a flight from the airline model. The dashed arrow denotes a *remote reference*: When the travel agency system loads the model, the flight is represented as a proxy object. In the case of data accesses on the flight, the flight model is loaded and added to the memory representation of travel model (right part). All model contents, including critical data such as flight logs and pilots, become visible.

This example highlights a lack of information hiding at the model level in EMF, in this case resulting in a situation that may compromise security and privacy. Other affected concerns include performance, analyzability, and collaborative development: During queries and transformations, considering a large set of models in union may be inefficient. Since models are not self-contained units, checking static properties of the individual models is prohibited unless certain restrictions are imposed [1]. A main issue during collaboration is proneness to inconsistencies. As an example, consider a situation where a developer deletes a model element from a model, unaware that this element is referred to from another model. This deletion results in a broken model reference.

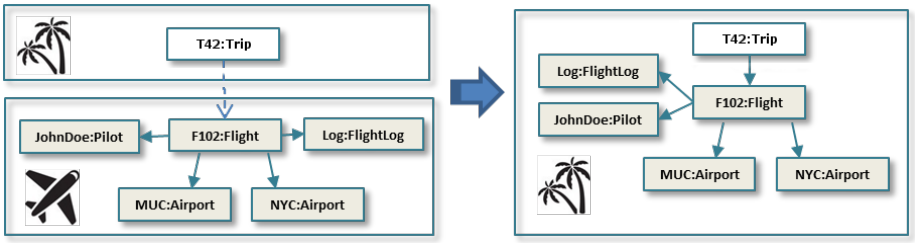

Fig. 1: Related models in EMF: *before* and *after* proxy resolution.

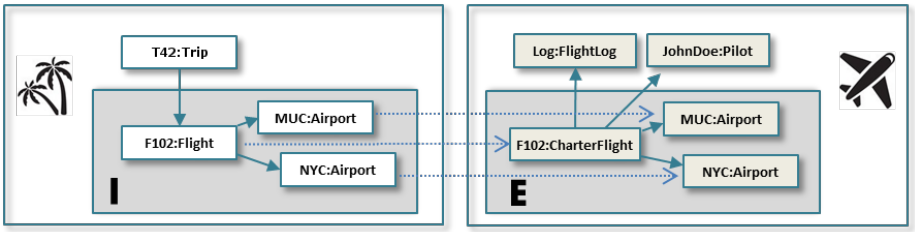

Fig. 2: Two model components with export and import interfaces.

A modularity mechanism addressing these issues is provided by *composite modeling* [6,16]. A composite model is a set of *components* where each component comprises a model with a set of export and import interfaces. Export and import interfaces declare subsets of model elements offered to and obtained from the environment. In Fig. 2, the travel agency has an import interface; the airline component has an export interface. Each model element in the import interface is mapped to a corresponding export element (dotted lines). In comparison to Fig. 1, the travel agency component now maintains the flight and its assigned airports as autonomous, but distinguished objects. We refer to these objects as *delegate objects*. Similar to a proxy, a delegate object represents an object stored somewhere else. But despite reflecting some of the remote object's features, a delegate object has an own identity. Consequently, as shown in our earlier work [16], components are self-contained units amenable to analysis and collaborative editing.

In this paper, we introduce a tool set that makes composite models available to model and meta-model developers. The tool set provides an implementation of composite models that is generic in the sense that it is applicable to any EMF-based host language. It comprises dedicated wizards plus a set of extensions of a visual meta-model editor, a generic tree-based model editor, and a model transformation tool. We provide these tools at `http://www.informatik.uni-marburg.de/~swt/compoemf/`.

2 Overview

Consider Fig. 3 for an overview of our tool set. As a starting point, we assume a set of related meta-models, e.g., for interacting systems or views on the same system. If these meta-models have not been developed as a components from scratch, they need to be extended with export and import interfaces (step 1). The resulting components can be

subject to editing (step 2). Once their development converges, the meta-model components are instantiated by model components (step 3). These components can be queried and transformed in the same manner as models in typical MDE scenarios (step 4).

Fig. 4 shows two application meta-models used to specify travel agency and airline operative systems; colors and stereotypes can be temporarily ignored. While these meta-models share a subset of common elements, such as classes `Flight` and `Airport` and their attributes, the specifications differ in the level of detail: The airline meta-model contains subclasses for charter and scheduled flights as well as additional attributes for airports. In our approach, we address such mismatches by allowing individual references and attributes to be exported and imported, and generalization to be flattened.

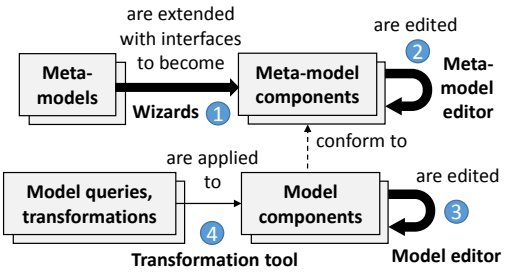

Fig. 3: Overview.

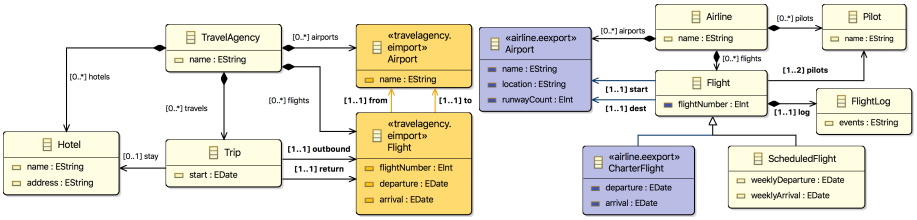

Fig. 4: Meta-models for travel agency (left half) and airline systems (right half).

(1) Extend meta-models. We provide a set of wizards to introduce export and import interfaces in a set of meta-models. Our wizards, shown in Fig. 5, allow the specification of a set of classes and features from the input model. A selection of classes and features to be exported or imported is specified using check-boxes. In the case of import interfaces, corresponding classes from an export interface have to be selected. A mapping from import to export elements is automatically derived using a name-based heuristics; missing mappings can be set by using either drag and drop or buttons.

(2) Edit meta-model components. The meta-model components can be edited using an extension of EMF's meta-model editor, the main part being shown in Fig. 4. The assignment of classes, references, and attributes to interfaces is denoted visually, using stereotypes and a custom color scheme. In addition, the editor provides dedicated functionality for the manipulation of interfaces, e.g., the (re)assigning of elements or the creation of new additional interfaces from a selection of elements. In our implementation of these features, we harnessed the view management of Sirius (`http://www.eclipse.org/sirius`) to extend the EMF meta-model editor.

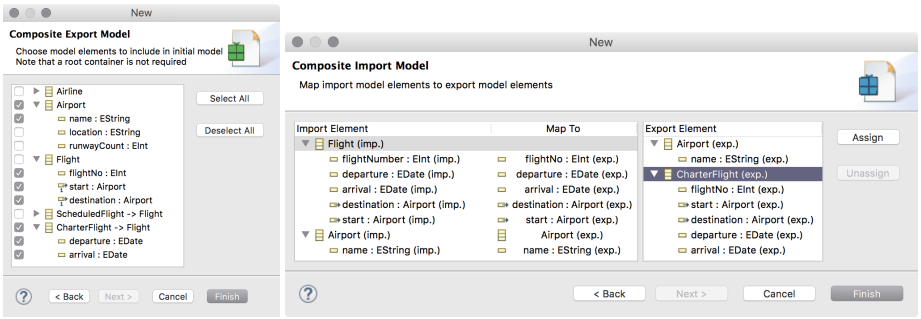

Fig. 5: Export and import creation wizards.

(3) Edit model components. The meta-models can now be instantiated. Export and import relationships can be edited using our extension of EMF's generic tree-based editor as shown in Fig. 6. Export and import interfaces, denoted *E* and *I*, are displayed as child nodes of their body models *(B)*. Their included elements are shown as distinct nodes; to help users trace their relationships, all elements related to the currently select one are highlighted. To edit the interfaces between models in an integrated visual representation (see Fig. 2), a customization of the involved editors is required. In our ongoing work, we develop a framework that allows to reduce the editor customization overhead. The Sirius view management is a promising prerequisite for this framework.

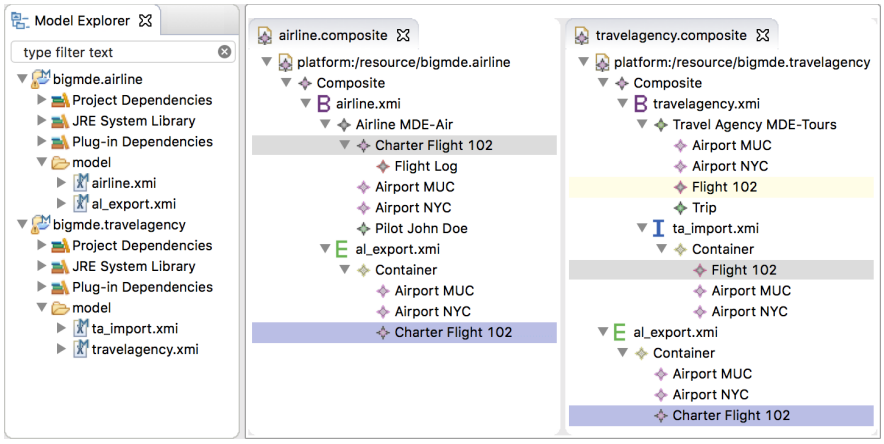

Fig. 6: Tree-based editors in EMF.

(4) Apply queries and transformations. Existing model query and transformation tools do not provide dedicated support for interfaces. Therefore, queries and transformations cannot consider export-import relations between models. To this end, we provide CompoHenshin, a model transformation tool on top of Henshin, a graph-based model transformation tool based on EMF. CompoHenshin comprises a visual editor for the

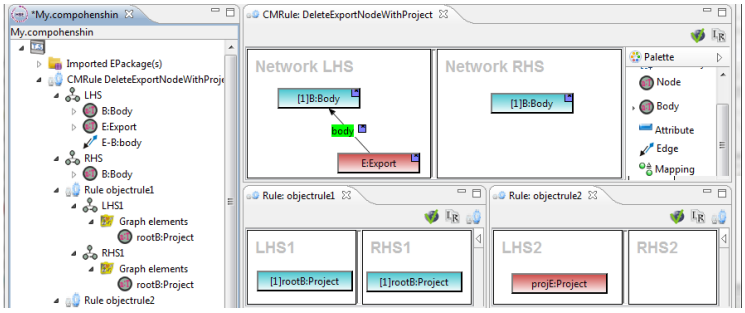

Fig. 7: CompoHenshin editor.

specification of composite rules [7] that can be applied to a set of model components. A composite rule (see Fig. 7) comprises a network rule and a set of object rules. Each object rule specifies the local change for one of the nodes in the overall set of models.

3 Discussion

Limitations. To support interoperability with existing tools, it is desirable to augment meta-models and models in a transparent manner: Export and import interfaces should be added on top of existing models, rather than changing them. In our approach, this is only possible if the meta-models contain suitable reference classes, such as `Flight` and `Airport` in our example. Otherwise, the meta-models need to be changed to contain such classes. The potential performance savings during analysis and transformations largely depend on the considered scenario. The most visible gain is to be expected if the component of interest is small, while its context is huge. For example, an OCL constraint can be expressed as a small model (the abstract syntax of the constraint) referencing a much larger one (the OCL standard library with its operators and literals).
Related work. Kelsen and Ma propose a black-box model modularity mechanism based on the fragmentation of a meta-model along some of its associations [8]. The approach by Heidenreich et al. [5] uses a component model and a composition language to express components, interfaces, and composition steps. These works assume a mandatory "weaving" step, whereas in our approach imported elements are self-contained objects reflecting features of a remote counterpart. Other works consider interfaces at the meta-model level only [17,18] or for particular DSLs [2]. Amálio et al. [1] provide a modularity approach based on *proxy nodes*, a concept emulating EMF's proxy mechanism.

A related line of work considers the splitting of a large model into multiple parts. Garmendia et al. [3] propose a tool to introduce a package structure in a model based on annotations in the meta-model. This tool can be used to explore large models [4]. Scheidgen et al. [10,11] provide a technique and tool for the fragmentation of large models for fast persistence and loading. The technique is based on annotating fragmentation points in the underlying meta-model. In our own work, we have applied clustering to optimize cohesion during splitting [13] and information retrieval techniques to consider the user intention during splitting by retrieving it from given text documents [14,15].

4 Conclusion

We have proposed a tool set that facilitates *information hiding* at the level of metamodels and models. The distinguishing feature of our tool set is that model elements imported from somewhere else are managed as distinct objects with their own identity. By maintaining modules as self-contained units, our approach may facilitate efficient static analysis, queries, and transformations, and developer independence during collaboration. In the future, we intend to evaluate this conjecture on large realistic models.

References

1. Amálio, N., de Lara, J., Guerra, E.: Fragmenta: A theory of fragmentation for MDE. In: Int. Conf. on Model Driven Engineering, Languages and Systems. pp. 106–115. IEEE (2015)
2. Arifulina, S., Mohr, F., Engels, G., Platenius, M.C., Schafer, W.: Market-specific service compositions: Specification and matching. In: Services (SERVICES), 2015 IEEE World Congress on. pp. 333–340. IEEE (2015)
3. Garmendia, A., Guerra, E., Kolovos, D.S., de Lara, J.: EMF Splitter: A Structured Approach to EMF Modularity. Workshop on Extreme Modeling pp. 22–31 (2014)
4. Garmendia, A., Jiménez-Pastor, A., de Lara, J.: Scalable model exploration through abstraction and fragmentation strategies. In: BigMDE Workshop on Scalability in Model Driven Engineering. pp. 21–31. CEUR (2015)
5. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On Language-Independent Model Modularisation. T. Aspect-Oriented Software Development VI pp. 39–82 (2009)
6. Jurack, S., Taentzer, G.: Towards Composite Model Transformations Using Distributed Graph Transformation Concepts. In: Schürr, A., Selic, B. (eds.) Int. Conf. on Model Driven Engineering Languages and Systems. pp. 226–240. Springer (2009)
7. Jurack, S., Taentzer, G.: Transformation of Typed Composite Graphs with Inheritance and Containment Structures. Fundam. Inform. 118(1-2), 97–134 (2012)
8. Kelsen, P., Ma, Q.: A Modular Model Composition Technique. In: Int. Conf. on Fundamental Approaches to Software Engineering. pp. 173–187. Springer (2010)
9. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ráth, I., Varró, D., Tisi, M., Cabot, J.: A Research Roadmap towards Achieving Scalability in Model Driven Engineering. In: BigMDE Workshop on Scalability in Model Driven Engineering. p. 2. ACM (2013)
10. Scheidgen, M.: Reference representation techniques for large models. In: BigMDE Workshop on Scalability in Model Driven Engineering. pp. 5:1–9. ACM (2013)
11. Scheidgen, M., Zubow, A., Fischer, J., Kolbe, T.H.: Automated and transparent model fragmentation for persisting large models. In: Int. Conf. on Model Driven Engineering Languages and Systems. pp. 102–118. Springer (2012)
12. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
13. Strüber, D., Lukaszczyk, M., Taentzer, G.: Tool support for model splitting using information retrieval and model crawling techniques. In: BigMDE Workshop on Scalability in Model Driven Engineering. pp. 44–47. CEUR (2014)
14. Strüber, D., Rubin, J., Taentzer, G., Chechik, M.: Splitting models using information retrieval and model crawling techniques. In: Int. Conf. on Fundamental Approaches to Software Engineering, pp. 47–62. Springer (2014)
15. Strüber, D., Selter, M., Taentzer, G.: Tool support for clustering large meta-models. In: BigMDE Workshop on Scalability in Model Driven Engineering. pp. 7:1–4. ACM (2013)
16. Strüber, D., Taentzer, G., Jurack, S., Schäfer, T.: Towards a distributed modeling process based on composite models. In: Fundamental Approaches to Software Engineering, pp. 6–20. Springer (2013)
17. Weisemöller, I., Schürr, A.: Formal definition of MOF 2.0 metamodel components and composition. In: Model Driven Engineering Languages and Systems, pp. 386–400. Springer (2008)
18. Živković, S., Karagiannis, D.: Towards metamodelling-in-the-large: Interface-based composition for modular metamodel development. In: Enterprise, Business-Process and Information Systems Modeling, pp. 413–428. Springer (2015)